Software Assurance Metrics And Tool Evaluation

Las Vegas, Nevada, USA, June 27-30, 2005

P. E. Black
Information Technology Laboratory
United States National Institute of Standards and Technology
Gaithersburg, Maryland 20899 USA

Abstract - The U.S. National Institute of Standards and Technology (NIST) is starting two ambitious projects to (1) develop a taxonomy of software security flaws and vulnerabilities, (2) develop a taxonomy of software assurance (SA) functions and techniques which detect those flaws, (3) perform and maintain a survey of SA tools implementing the functions, (4) develop testable specifications of SA functions and explicit tests, include a standard reference dataset, to evaluate how closely tools implement the functions, and (5) lead efforts to develop metrics for the effectiveness of those functions. The end result is that users will be able to choose a combination of techniques which best suits their needs and will be able to state how much confidence they have in software which has been assessed. This paper details these two projects and presents our justifications and expectations.

Keywords: metrics, security, reference dataset, software assurance

1.0 Introduction

Tools implementing software assurance (SA) metrics and checks can help software developers produce software with fewer security vulnerabilities. Such tools can also help identify malicious code and poor coding practices that lead to vulnerabilities. From requirements capture through design and acceptance to operation monitoring, we can improve results using validated metrics and well-characterized tools.

In spite of much effort, the strengths and applicability of metrics and tools is not always clear. Many are working to survey tools, e.g., [6], and to assess tools, e.g., [7] and [2]. There are more than a dozen source code scanners alone, in addition to dozens of other software security assessment tools and services.

This paper is about two related projects in which NIST is involved: the Software Assurance Metrics And Tool Evaluation (SAMATE) project and the Standard Reference Dataset (SRD) project. SAMATE, which will be funded in part by the United States Department of Homeland Security (DHS), has a broad scope. The SRD project is more narrowly focused, and the SRD goals are a subset of SAMATE goals. Since SRD is currently funded and has specific deliverables, we treat it separately.

1.1 NIST's Role

Why should NIST be involved? NIST is a non-regulatory agency and has over a century of experience in standards and measurement. NIST serves as a neutral party and a long-term repository for standard reference materials, datasets, and other items. Also, NIST already has important security and standards activities, such as developing the Advanced Encryption Standard (AES), the National Voluntary Laboratory Accreditation Program (NVLAP), and the National Information Assurance Partnership (NIAP).

We do *not* want to duplicate existing datasets, surveys, or work. Rather, we want to contribute to existing work, refer to these resources, and complement them. We do not plan to research new software assurance methods. Many are already doing that. Instead we will follow the field, perhaps pointing out areas that need work. As the methods mature, we plan to help quantify their strengths.

2.0 The Standard Reference Dataset Project

The Standard Reference Dataset (SRD) project seeks to support tool evaluation. The SRD should help answer questions like, "how can a buyer substantiate a vendor's claims?" and "how can a developer be more sure that a new method is better than an existing method?" To answer such questions we first need to understand the kinds of security problems that exist and the kinds of assessment techniques and tools that are available.

To this end, the project has the following long-term goals.

- Identify classes of security flaws and vulnerabilities.
- Develop metrics to assess software.
- Identify classes of software security assessment techniques.
- Document the state of the art in software security assessment tools.
- Develop measures to evaluate tools.
- Develop a collection of reference flawed or vulnerable programs.

Again, we plan to use existing surveys, work, findings, etc. as much as possible.

2.1 Classes of Software Security Flaws and Vulnerabilities

Before we can determine the usefulness of any reference dataset, we need to know about potential software security flaws and vulnerabilities. Examples of vulnerabilities are buffer overflows and backdoors. Rather than a complete listing of occurrences, like ICAT [1], we seek groups or classes of flaws, such as [4]. To better organize them, we will also develop a taxonomy of flaws and vulnerabilities, like [3].

Flaws can be graded by conceptual level. For instance, buffer overflows violate the semantic model of C programs, therefore they can always be considered vulnerabilities. One need not consider how the program will be used.

Injection flaws, where malicious commands in user input could reach an external resource, are a higher level. One must designate some commands as getting user input and other commands as interfaces to external resources. With that general information, tainted data flow analysis can find vulnerabilities. Many different programs can use the same designation of commands.

At an even higher level, flawed access control requires very detailed, and often quite specific, descriptions of what the access controls are, how they are implemented in the program, and the access control policy.

2.2 Software Security Assurance Functions and Tools

Complementary to classes of flaws and vulnerabilities, we need a taxonomy and list of classes of SA functions or approaches. That is, what can a program do to improve assurance. Assessment techniques range from running a product in a restricted environment (a "sandbox") to source code scanning to traceable development practices to consistent requirements, with many more in between. Possible factors in the taxonomy are: approach (code scanning vs. specification based, static analysis vs. dynamic or run time monitoring, etc.), software type (web, distributed, real time, high security, etc.), type of vulnerability detected, and rate of false positives or false negatives.

Currently we are planning to first concentrate on tools and functions that assess software directly. Although crucial to achieve high quality results, this taxonomy will not initially look at capturing specifications, making sure requirements are consistent and complete, good software engineering practices, installation, run-time audits and penetration testing, or tools for other phases of the software life cycle. The delivered software is a reasonable place to start for several reasons.

- To determine the effect on the product of a change in the development process, the resultant software must be assessed.
- Software may have an uncertain pedigree. For instance, to determine whether a commercial, off-the-shelf software package is secure enough for an application, the buyer may wish to assess the software. Contractors need to confirm the quality of delivered code.
- Developers can use software security assessment tools to find vulnerabilities or flaws in software before it is released.

2.3 State of the Art

From a practical perspective, we must understand what current tools can do, that is, which one or more of the SA functions the tool performs. A survey of tools helps ensure that the taxonomies are complete and helps users find tools which match their needs. From this survey, we will report on the state of the art in software security tools.

In addition to this a survey of tools, we will survey researchers and companies. We explicitly mention companies because some companies offer SA services, rather than tools. The companies may use proprietary tools internally, which they couple with expertise.

These surveys will be available on the web. We will maintain them, adding and changing entries as needed. If another entity fields something equivalent in coverage and availability, we will likely withdraw the redundant surveys.

These two taxonomies form a very useful matrix. The vulnerabilities are cross indexed with the SA functions that could prevent, detect, or mitigate them. For instance, a buffer overflow (a vulnerability) might be found by scanning the source code (a function) or running the program with special memory allocation aids (another function or technique).

2.4 Metrics to Evaluate Software and Tools

The outcome of running an SA tool is some report on the security, complexity, function or some other property of the software. Could we summarize that report with a number or classification? Just as the Mohs scale is a relative hardness from 1 (talc) to 10 (diamond) and the Richter scale is a logarithmic scale for earthquakes, it would be very useful if there were some way to rate software.

Any software metric must be multidimensional. There are many different properties, aspects, or facets of software that we wish to be able to summarize: security, maintainability, portability, etc.

Assuming that we have a software metric, along with a measurement method, we can ask how well a particular tool implements a metric function. That is, we also need metrics or measures for tools. Two measures that spring to mind are the number of vulnerabilities flagged and the number of false alarms. An omniscient tool would flag all vulnerabilities with no false alarms. Since this is unattainable, tools or users must trade these off.

For a user with modest security needs, very few false alarms may be important for productivity, even if a few vulnerabilities are missed. On the other hand, a high confidence application may need as many vulnerabilities caught as possible.

2.5 Standard Reference Dataset

Almost any tool measurement will require reference datasets of clean code and code with known flaws. Such standard reference datasets, used with metrics, can help advance the state of the art in software security tools by giving developers and researchers an idea of how well a new technique or tool does. These metrics and standard reference datasets can also help purchasers confirm tool vendors' claims.

The standard reference dataset (SRD) should eventually cover all classes of software security flaws and vulnerabilities. There would be versions for different languages and different platforms or environments, where applicable. Test cases will mirror the taxonomy of flaws and vulnerabilities, mentioned in Sect. 2.1. What are other attributes of an SRD? That is, what would a standard reference dataset need to be? Here are some considerations.

- Minimal test cases for each flaw

  Small test cases separate "can this be detected" from "how fast is the tool".

- Some very large test cases

  For instance, the Apache web server. This allows examination of speed and maximum size– important for industrial use.

- Test cases should be taken from actual code

  If it is actual code, nobody can dismiss it with it never happens in real life.

- Test cases should be readily usable

  That is, it shouldn't require a license or more than a nominal fee to get them or use them.

The reference dataset will not spring into being all at once. We welcome and actively seek contributions from vendors and researchers. Following the principle of being readily usable, such contributions would have to be released into the public domain or made available by, say, an open source license.

Many test cases will be developed at NIST. We plan to initially concentrate on the lowest level of flaws, that is, those that are most universally problems, such as buffer overflows. As our focus groups find gaps in high-priority areas, we will move into those areas.

Even when the standard reference dataset is complete, with tests covering all classes of flaws and vulnerabilities, it cannot be a static entity. At the very least, as new languages and environments are developed, corresponding tests will need to be developed as well. The SRD will have to adapt to new exploits and demands as well.

3.0 Software Assurance Metrics And Tool Evaluation Project

The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project began before the SRD project and has a far broader scope. The software assurance strategy of the U.S. Department of Homeland Security (DHS) had four parts:

1. People: Education and Training
2. Process: Life cycle Development Process, Best Practices, Standards
3. Technology: Tools and R&D
4. Acquisition: Sample Statement of Work (SOW) / Procurement language

The objective for part 3, Technology, is "Software Assurance tools identification, enhancement, and development". This objective has two particularly relevant parts:

- Examine software development and testing methods, and tool identification, enhancement, and development to target bugs, flaws and backdoors throughout the software development life cycle.
- Develop R&D requirements to take to DHS' office of Science and Technology. Create a set of studies and experiments to measure the effectiveness of tools (ability to produce secure software, cost effectiveness, etc.)

When the DHS became aware of NIST's Computer Forensic Tool Testing (CFTT) project [5], it asked NIST to take the lead in testing software evaluation tools and measuring their effectiveness.

To lead the testing of software evaluation tools, we determined to find or develop an organized list of software flaws and vulnerabilities, track the state of the art in tools with an on-going survey, and find or develop measures to evaluate software. With these in hand, we can develop detailed test plans. To measure the effectiveness of tools, we will find or develop metrics for SA functions and approaches. Finally we will help find deficiencies in those areas and take the lead in developing a research plan.

3.1 Taxonomy of Software Security Flaws and Assurance Functions

This is based on the work in Sect. 2.1 and 2.2, but takes a broader view. It includes assurance functions. It is also expanded to cover flaws in specifications, requirements, installation, and operation.

As one dimension, we can classify software assurance functions from automated to manual. The classification must be multivalued since there is a continuum from completely manual methods, done solely with paper and pencil, to completely automatic method that grade programs "pass" or "fail". Many approaches fall in the middle. Code reviews with a few aids, such as listings with line numbering and cross references, produced by programs fall on the human-intensive side. Toward the other end of the scale are code checkers that report occurrences which are very likely to be faults where people make the final determination.

These taxonomic classifications of functions help us contrast different functions and identify deficiencies and possibilities for research.

3.2 Survey of Tools/State of the Art

As with the taxonomy, we will expand the SRD work reviewed in Sect. 2.3 to cover tools in all phases of software, from the initial specification to secure operation.

We are currently planning to concentrate primarily on the automated methods to begin because they have low recurring cost. We narrow our focus even more to methods that have little start-up or learning cost. They have a lower threshold of assurance than highly specialized techniques, but don't require learning, say, formal specification languages. They likely don't find as many problems as code review, but the cost/benefit ratio may be better. More importantly, these techniques can dramatically reduce the occurrences of many bugs. At the beginning, it is more a matter of raising the expectation of "due diligence" so people will use appropriate and cost-effective checking tools.

3.3 Metrics to Evaluate Software and Tools

Once again we expand the metrics in Sect. 2.4 to cover more software, functions, and tools.

Users want to be able determine that tools faithfully implement the SA functions they intend to implement. Some techniques to make the determination are code scanning, running in monitored environments, tainted data flow analysis, etc. Some determination methods need a set of programs with known vulnerabilities or security flaws, along with a benchmark of programs that should be flagged by certain techniques.

Since it is easy for a technique to generate many false alarms, we must include programs which are known to be correct. More particularly (and to avoid defining the notion of "completely correct"), the programs should perform ostensibly the same function as the flawed programs, but have the flaw fixed.

3.4 Detailed Test Plans

Before going very far to support tool evaluation, we must decide on an ordering, that is, which of the tool functions are most important. It is not realistic to pretend that we will do everything first, especially when we plan to be very thorough. The importance of a function depends on whether a particular assurance function can be done in practice, how much it costs (in human and machine resources), what risks it mitigates, its cost/benefit ratio, how high a priority level vulnerabilities it find, how well studied or specified it is, and how widely it is used. For example, although not a tool, code reviews are very good for many purposes, but are quite expensive and require having source code. They are also the subject of many books, so working on how code reviews help assure software may be low on our list.

We will hold workshops to gather researchers, developers, and users to help rank the vulnerabilities and assurance functions in some order. The workshops will result in some number of top vulnerabilities or assurance functions.

How then will we support evaluation? We will follow the procedural model of the Computer Forensic Tool Testing (CFTT) project. For each of the top assurance functions, we will develop detailed requirements and testable specifications. For instance, one requirement for finding buffer overflows with a source code scanner might be to identify all memory allocations, that is, buffers that could overflow.

From the workshops and other resources, we will invite a focus group to address each of the functions. The first task of the focus groups is to review, revise, clarify, and comment on the requirements for their respective assurance functions.

From the requirements, we will derive testable specifications. Using these specifications we will develop detailed test plans, along with associated testing matter. The testing matter may include auxiliary test scripts or programs, a standard reference dataset of programs with known security vulnerabilities and corresponding "corrected" programs, software metrics and measures, and expected results.

The standard reference dataset and the software metrics and measures will be based on the SRD project work described in Sect. 2.5 and Sect. 2.4. For this project, the metrics and dataset will be more detailed so standardized evaluations can be performed.

3.5 Metrics for the Effectiveness of Approaches

The preceeding section details a plan to assess how closely a program implements a function or method. Taking one step back, we can ask ourselves, how much confidence can we have in a piece of code that has been reviewed or checked by, say, running a code scanner? Do different methods complement each other?

For instance, backdoors can only feasibly be detected by code examination: no amount of blackbox testing is likely to find a backdoor triggered by the using the name AliBaba. In contrast, code reviews for faults are unlikely to discover that a behavior is missing altogether: the code must be tested or assessed

against a specification or requirement or a use case. These examples show that we need to evaluate the effectiveness of SA security techniques themselves. In the future we will lead efforts to plan and begin studies to improve or create metrics for SA technique effectiveness.

We will also lead an effort to identify gaps in function or tools and to outline research agendas. We will assess current methods and tools in order to identify deficiencies which can lead to software failures and vulnerabilities.

4.0 Call for Participation

We finish with a call for participation from the community.

1. Help define classes of security flaws and vulnerabilities.
2. Contribute to the survey of tools, companies, and researchers.
3. Help define classes of automated software security checking functions. In the longer term, examine a broader range of techniques and methods and classify them.
4. Decide the ordering of functions, i.e., which functions should be looked at first.
5. Serve in focus group to clearly define a function and come up with testable requirements.
6. Contribute to and critique the reference dataset.
7. Critique test plans.
8. Help identify gaps in functions or tools and outline research agendas to fill them.
9. Propose metrics to evaluate security checking functions. We can begin with qualities of such a metric.

This is an ambitious program, but the world is too complex to continue developing software as we have.

References

[1] ICAT metabase: A CVE-based vulnerability database. http://icat.nist.gov/, 2005.

[2] Freeland Abbott and Joseph Saur. Comparison of code checker technologies for software vulnerability evaluation. Technical report, Joint Systems Integration Command, April 2005.

[3] Algirdas Avizienis, Jean-Clause Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, Jan-Mar 2004.

[4] Common web application vulnerabilities. http://www.aspectsecurity.com/comvuln.html, 2003.

[5] Computer forensic tool testing (CFTT) project overview. http://www.cftt.nist.gov/project_overview.htm, 2003.

[6] Arian Evan. Appsec assessment tools. http://www.owasp.org/docroot/owasp/misc/OWASP_UK_2005_Presentations/AppSec2005-Arian_Evans-AppSec_Assessment_Tools.ppt, April 2005.

[7] Misha Zitser, Richard P. Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. 12th Internt'l Symp. on Foundations of Software Engineering*, pages 97–106. ACM SIGSOFT, ACM Press, 2004.