NIST Special Publication 500-279

Static Analysis Tool Exposition (SATE) 2008

Vadim Okun
Romain Gaucher
Paul E. Black

NIST National Institute of Standards and Technology • U.S. Department of Commerce

NIST Special Publication 500-279

Static Analysis Tool Exposition (SATE) 2008

Vadim Okun
Romain Gaucher
Paul E. Black
Software and Systems Division
Information Technology Laboratory

June 2009

U.S. Department of Commerce
Gary Locke, Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Deputy Director

National Institute of Standards and Technology Special Publication 500-279
Natl. Inst. Stand. Technol. Spec. Publ. 500-279, 64 pages (June 2009)
CODEN: NSPUE2

Abstract:

The NIST SAMATE project conducted the first Static Analysis Tool Exposition (SATE) in 2008 to advance research in static analysis tools that find security defects in source code. The main goals of SATE were to enable empirical research based on large test sets and to encourage improvement and speed adoption of tools. The exposition was planned to be an annual event.

Briefly, participating tool makers ran their tool on a set of programs. Researchers led by NIST performed a partial analysis of tool reports. The results and experiences were reported at the Static Analysis Workshop in Tucson, AZ, in June, 2008. The tool reports and analysis were made publicly available in 2009.

This special publication consists of the following papers. "Review of the First Static Analysis Tool Exposition (SATE 2008)," by Vadim Okun, Romain Gaucher, and Paul E. Black, describes the SATE procedure, provides observations based on the data collected, and critiques the exposition, including the lessons learned that may help future expositions. Paul Anderson's "Commentary on CodeSonar's SATE Results" has comments by one of the participating tool makers. Steve Christey presents his experiences in analysis of tool reports and discusses the SATE issues in "Static Analysis Tool Exposition (SATE 2008) Lessons Learned: Considerations for Future Directions from the Perspective of a Third Party Analyst".

Keywords:
Software security; static analysis tools; security weaknesses; vulnerability

Table of Contents

Review of the First Static Analysis Tool Exposition (SATE 2008)

Vadim Okun vadim.okun@nist.gov

Romain Gaucher[1] rgaucher@cigital.com

Paul E. Black paul.black@nist.gov

National Institute of Standards and Technology
Gaithersburg, MD 20899

Abstract

The NIST SAMATE project conducted the first Static Analysis Tool Exposition (SATE) in 2008 to advance research in static analysis tools that find security defects in source code. The main goals of SATE were to enable empirical research based on large test sets and to encourage improvement and speed adoption of tools. The exposition was planned to be an annual event.

Briefly, participating tool makers ran their tool on a set of programs. Researchers led by NIST performed a partial analysis of tool reports. The results and experiences were reported at the Static Analysis Workshop in Tucson, AZ, in June, 2008. The tool reports and analysis were made publicly available in 2009.

This paper describes the SATE procedure, provides our observations based on the data collected, and critiques the exposition, including the lessons learned that may help future expositions. This paper also identifies several ways in which the released data and analysis are useful. First, the output from running many tools on production software can be used for empirical research. Second, the analysis of tool reports indicates weaknesses that exist in the software and that are reported by the tools. Finally, the analysis may also be used as a building block for a further study of the weaknesses and of static analysis.

Disclaimer

[1] Romain Gaucher is currently with Cigital, Inc. When SATE was conducted, he was with NIST.

Cautions on Interpreting and Using the SATE Data

SATE 2008 was the first such exposition that we conducted, and it taught us many valuable lessons. Most importantly, our analysis should NOT be used as a direct source for rating or choosing tools; this was never the goal of SATE.

There is no metric or set of metrics that is considered by the research community to indicate all aspects of tool performance. We caution readers not to apply unjustified metrics based on the SATE data.

Due to the variety and different nature of security weaknesses, defining clear and comprehensive analysis criteria is difficult. As SATE progressed, we realized that our analysis criteria were not adequate, so we adjusted the criteria during the analysis phase. As a result, the criteria were not applied consistently. For instance, we were inconsistent in marking the severity of the warnings where we disagreed with tool's assessment.

The test data and analysis procedure employed have serious limitations and may not indicate how these tools perform in practice. The results may not generalize to other software because the choice of test cases, as well as the size of test cases, can greatly influence tool performance. Also, we analyzed a small, non-random subset of tool warnings and in many cases did not associate warnings that refer to the same weakness.

The tools were used in this exposition differently from their use in practice. In practice, users write special rules, suppress false positives, and write code in certain ways to minimize tool warnings.

We did not consider the user interface, integration with the development environment, and many other aspects of the tools. In particular, the tool interface is important for a user to efficiently and correctly understand a weakness report.

Participants ran their tools against the test sets in February 2008. The tools continue to progress rapidly, so some observations from the SATE data may already be obsolete.

Because of the above limitations, SATE should not be interpreted as a tool testing exercise. The results should not be used to make conclusions regarding which tools are best for a particular application or the general benefit of using static analysis tools. In this paper, specifically Section 5, we suggest ways in which the SATE data might be used.

1 Introduction

Static Analysis Tool Exposition (SATE) was designed to advance research in static analysis tools that find security-relevant defects in source code. Briefly, participating tool makers ran their tool on a set of programs. Researchers led by NIST performed a partial analysis of tool reports. The results and experiences were reported at the Static Analysis Workshop (SAW) [20]. The tool reports and analysis were made publicly available in 2009. SATE had these goals:

- To enable empirical research based on large test sets
- To encourage improvement of tools
- To speed adoption of the tools by objectively demonstrating their use on production software

Our goal was not to evaluate nor choose the "best" tools.

SATE was aimed at exploring the following characteristics of tools: relevance of warnings to security, their correctness, and prioritization. Due to the way SATE was organized, we considered the textual report produced by the tool, not its user interface. A tool's user interface is very important for understanding weaknesses. There are many other factors in determining which tool (or tools) is appropriate in each situation

SATE was focused on static analysis tools that examine source code to detect and report weaknesses that can lead to security vulnerabilities. Tools that examine other artifacts, like requirements, byte code or binary, and tools that dynamically execute code were not included.

SATE was organized and led by the NIST SAMATE team [15]. The tool reports were analyzed by a small group of analysts, consisting, primarily, of the NIST and MITRE researchers. The supporting infrastructure for analysis was developed by the NIST researchers. Since the authors of this report were among the organizers and the analysts, we sometimes use the first person plural (we) to refer to analyst or organizer actions.

In this paper, we use the following terminology. A *vulnerability* is a property of system security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and result in a security failure [18]. A vulnerability is the result of one or more *weaknesses* in requirements, design, implementation, or operation. A *warning* is an issue (usually, a weakness) identified by a tool. A (tool) *report* is the output from a single run of a tool on a test case. A tool report consists of warnings.

Researchers have studied static analysis tools and collected test sets. Zheng et. al [23] analyzed the effectiveness of static analysis tools by looking at test and customer-reported failures for three large-scale network service software systems. They concluded that static analysis tools are effective at identifying code-level defects. Several collections of test cases with known security flaws are available [11] [24] [12] [16]. Several assessments of open-source projects by static analysis tools have been reported recently [1] [5] [9]. A number of studies have compared different static analysis tools for finding security defects, e.g., [14] [11] [24] [10] [13] [4]. SATE is different in that many participants ran their own tools on a set of open source programs. Also, SATE's goal is to accumulate test data, not to compare tools.

The rest of the paper is organized as follows. Section 2 describes the SATE 2008 procedure. Since we made considerable changes and clarifications to the SATE procedure after it started, Section 2 also describes the procedure in its final form. See Section 4 for a discussion of some of the changes to the procedure and the reasons for making them. Appendix A contains the SATE plan that participants faced early on.

Section 3 gives our observations based on the data collected. In particular, our observations on the difficulty of differentiating weakness instances are in Section 3.4. Section 4 is our review of the exposition. It describes reasons for our choices, changes to the procedure that we made, and also lists the limitations of the exposition. Section 5 summarizes conclusions and outlines future plans.

2 SATE Organization

The exposition had two language tracks: C track and Java track. At the time of registration, participants specified which track(s) they wished to enter. We performed separate analysis and reporting for each track. Also at the time of registration, participants specified the version of the tool that they intended to run on the test set(s). We required the tool version to have a release or build date that is earlier than the date when they received the test set(s).

2.1 Steps in the SATE procedure

The following summarizes the steps in the SATE procedure. Deadlines are given in parentheses.

- Step 1 Prepare
  - Step 1a Organizers choose test sets
  - Step 1b Tool makers sign up to participate (by 8 Feb 2008)
- Step 2 Organizers provide test sets via SATE web site (15 Feb 2008)
- Step 3 Participants run their tool on the test set(s) and return their report(s) (by 29 Feb 2008)
  - Step 3a (optional) Participants return their review of their tool's report(s) (by 15 Mar 2008)
- Step 4 Organizers analyze the reports, provide the analysis to the participants (by 15 April 2008)
  - Step 4a (Optional) Participants return their corrections to the analysis (by 29 April 2008)
  - Step 4b Participants receive an updated analysis (by 13 May 2008)
  - Step 4c Participants submit a report for SAW (by 30 May 2008)
- Step 5 Report comparisons at SAW (June 2008)
- Step 6 Publish results (Originally planned for Dec 2008, but delayed until June 2009)

2.2 Test Sets

We list the test cases we selected, along with some statistics for each test case, in Table 1. The last two columns give the number of files and the number of non-blank, non-comment lines of code (LOC) for the test cases. The counts for C test cases include source (.c) and header (.h) files. The counts for the Java test cases include Java (.java) and JSP (.jsp) files. The counts do not include source files of other types: make files, shell scripts, Perl, PHP, and SQL. The lines of code were counted using SLOCCount by David A. Wheeler [22].

Test case	Track	Description	Version	# Files	# LOC
Naim	C	Console instant messenger	0.11.8.3.1	44	23,210
Nagios	C	Host, service and network monitoring	2.10	73	65,133
Lighttpd	C	Web server	1.4.18	144	38,306
OpenNMS	Java	Network management system	1.2.9	1065	131,507
MvnForum	Java	Forum	1.1	839	128,784
DSpace	Java	Document management system	1.4.2	388	53,847

Table 1 Test cases

The links to the test case developer web sites, as well as links to download the versions analyzed, are available at the SATE web page [19].

2.3 Participants

Table 2 lists, alphabetically, the participating tools and the tracks in which the tools were applied. Although our focus is on automated tools, one of the participants, Aspect Security, performed a human code review. Another participant, Veracode, performed a human review of its reports to remove anomalies such as high false positives in a particular weakness category.

2.4 Tool Runs and Submissions

Participants ran their tools and submitted reports following specified conditions.

- Participants did not modify the code of the test cases.
- For each test case, participants did one or more runs and submitted the report(s). See below for more details.
- Except for Aspect Security and Veracode, the participants did not do any hand editing of tool reports. Aspect Security performed a manual review. Veracode performed a human quality review of its reports to remove anomalies such as high false positives in a particular weakness category. This quality review did not add any new results.
- Participants converted the reports to a common XML format. See Section 2.6.1 for description of the format.
- Participants specified the environment (including the operating system and version of compiler) in which they ran the tool. These details can be found in the SATE tool reports available at [19].

Tool	Version	Tracks
Aspect Security ASC[2]	2.0	Java
Checkmarx CxSuite	2.4.3	Java
Flawfinder[3]	1.27	C
Fortify SCA	5.0.0.0267	C, Java
Grammatech CodeSonar	3.0p0	C
HP DevInspect[4][5]	5.0.5612.0	Java
SofCheck Inspector for Java	2.1.2	Java
University of Maryland FindBugs	1.3.1	Java
Veracode SecurityReview[6]	As of 02/15/2008	C, Java

Table 2 Participating tools

Most participants submitted one tool report per test case for the track(s) that they participated in. HP DevInspect analyzed DSpace only. They were not able to setup analysis of the other Java test cases before the deadline.

Fortify submitted additional runs of their tool with the –findbugs option. Due to lack of time we did not analyze the output from these runs. For MvnForum, Fortify used a custom rule, which was included in their submission. No other tool used custom rules. In all, we analyzed the output from 31 tool runs: 6 each from Fortify and Veracode (each participated in 2 tracks), 1 from HP DevInspect, and 3 each from the other 6 tools.

Several participants also submitted the original reports from their tools, in addition to the reports in the SATE output format. During our analysis, we used some of the information (details of weakness paths) from some of the original reports to better understand the warnings.

Grammatech CodeSonar uses rank (a combination of severity and likelihood) instead of severity. All warnings in their submitted reports had severity 1. We changed the severity field for some warning classes in the CodeSonar reports based on the weakness names.

2.5 Analysis of Tool Reports

For selected tool warnings, we analyzed up to three of the following characteristics. First, we associated together warnings that refer to the same weakness. (See Section 3.4 for a discussion of what constitutes a weakness.) Second, we assigned severity to warnings when we disagreed with the severity assigned by the tool. Often, we gave a lower severity to indicate that in our view, the warning was not relevant to security. Third, we analyzed correctness of the warnings. During the analysis phase, we marked the warnings as true or false positive. Later, we decided not to use the true/false positive markings. Instead, we marked as "confirmed" the warnings that we determined to be correctly reporting a weakness. We marked as "unconfirmed" the rest of the warnings that we analyzed or associated. In particular, this category includes the warnings that we analyzed

[2] Performed a manual review, used only static analysis for SATE; ASC stands for Application Security Consultant – there is no actual product by that name
[3] Romain Gaucher ran David Wheeler's Flawfinder
[4] A hybrid static/dynamic analysis tool, but used only static part of the tool for SATE
[5] Analyzed one test case - DSpace
[6] A service

but were not sure whether they were correct. We discuss the reasons for using confirmed and unconfirmed in Section 4.2. Also, we included our comments about warnings.

2.5.1 Analysis Procedure

We used both human and (partially) automated analyses. Humans analyzed warnings using the following procedure. First, an analyst searched for warnings. We focused our efforts on warnings with severity 1 or 2 (as reported by the tools). We analyzed some lower severity warnings, either because they were associated with higher severity warnings or because we found them interesting. An analyst usually concentrated his efforts on a specific test case, since the knowledge of the test case that he gained enabled him to analyze other warnings for the same test case faster. Similarly, an analyst often concentrated textually, e.g., choosing warnings near by in the same source file. An analyst also tended to concentrate on warnings of the same type.

After choosing a particular warning, the analyst studied the relevant parts of the source code. If he formed an opinion, he marked correctness, severity, and/or added comments. If he was unsure about an interesting case, he may have investigated further by, for instance, extracting relevant code into a simple example and/or executing the code. Then the analyst proceeded to the next warning.

Below are two common scenarios for an analyst's work.

Search → View list of warnings → Choose a warning to work on → View source code of the file → Return to the warning → Submit an evaluation

Search → View list of warnings → Select several warnings → Associate the selected warnings

Sometimes, an analyst may have returned to a warning that had already been analyzed, either because he changed his opinion after analyzing similar warnings or for other reasons.

To save time, we used heuristics to partially automate the analysis of some similar warnings. For example, when we determined that a particular source file is executed during installation only, we downgraded severity of certain warning types referring to that source file.

Additionally, a tool to automate the analysis of buffer warnings reported by Flawfinder was developed by one of the authors [6]. The tool determined source and destination buffers, identified the lines of code involving these buffers, and analyzed several types of actions on the buffers, including allocation, reallocation, computing buffer size, comparisons, and test for NULL after allocation. The tool then made a conclusion (sometimes incorrectly) about correctness of the warning. The conclusions were reviewed manually.

2.5.2 Practical Analysis Aids

To simplify querying of tool reports, we imported all reports into a relational database designed for this purpose.

To support human analysis of warnings, we developed a web interface which allows searching the warnings based on different search criteria, viewing individual warnings, marking a warning with human analysis which includes opinion of correctness, severity, and comments, studying relevant source code files, associating warnings that refer to the same weakness, etc.

2.5.3 Optional Steps

We asked participants to review their tool reports and provide their findings (optional step 3a in Section 2.1). SofCheck submitted a review of their tool's warnings.

We also asked participants to review our analysis of their tool warnings (optional step 4a in Section 2.1). Grammatech submitted a review of our analysis. Based on Grammatech's comments, we re-examined our analysis for the relevant warnings and changed our conclusions for some of the warnings.

2.5.4 Analysis Criteria

This section describes the criteria that we used for associating warnings that refer to the same weakness and also for marking correctness and severity of the warnings. We marked severity of a warning whenever we disagreed with the tool. The limitations of the criteria are discussed in Section 4.2.

Correctness and severity are orthogonal. Confirmed means that we determined that the warning correctly reports a weakness. Severity attempts to address security relevance.

Criteria for analysis of correctness

In our analysis we assumed that

- A tool has (or should have) perfect knowledge of control/data flow that is explicitly in the code.
  - For example, if a tool reports an error caused by unfiltered input, but in fact the input is filtered correctly, mark it as false.
  - If the input is filtered, but the filtering is not complete, mark it as true. This is often the case for cross-site scripting weaknesses.
  - If a warning says that a function can be called with a bad parameter, but in the test case it is always called with safe values, mark the warning as false.
- A tool does not know about context or environment and may assume the worst case.
  - For example, if a tool reports a weakness that is caused by unfiltered input from command line or from local files, mark it as true. The reason is that the test cases are general purpose software, and we did not provide any environmental information to the participants.

Criteria for analysis of severity

We used an ordinal scale of 1 to 5, with 1 - the highest severity. We assigned severity 4 or 5 to warnings that were not likely to be security relevant.

We focused our analysis on issues with severity 1 and 2. We left the severity assigned by the tool when we agreed with the tool. We assigned severity to a warning when we disagreed with the tool.

Specifically, we downgraded severity in these cases:

- A warning applies to functionality which may or may not be used securely. If the tool does not analyze the use of the functionality in the specific case, but provides a generic warning, we downgrade the severity to 4 or 5. For example, we downgrade severity of general warnings about use of getenv.
- A weakness is unlikely to be exploitable in the usage context. Note that the tool does not know about the environment, so it is correct in reporting such issues.
  - For example, if input comes from configuration file during installation, we downgrade severity.
  - We assume that regular users cannot be trusted, so we do not downgrade severity if input comes from a user with regular login credentials.
- We believe that a class of weaknesses is less relevant to security.

Correctness and severity criteria applied to XSS

After analyzing different cross-site scripting (XSS) warnings, we realized that it is often very hard to show that an XSS warning is false (i.e., show that the filtering is complete). The following are the cases where an XSS warning can be shown to be false (based on our observations of the SATE test cases).

- Typecasting – the input string is converted to a specific type, such as Boolean, integer, or other immutable and simple type. For example, Integer::parseInt method is considered safe since it returns a value with an integer type.
- Enumerated type - a variable can have a limited set of possible values.

We used the following criteria for assigning severity.

- Severity 1 – no basic validation, e.g., the characters "<>" are not filtered.
- Severity 2 – vulnerable to common attack vectors, e.g., there is no special characters replacement (CR, LF), no extensive charset checking.
- Severity 3 – vulnerable to specific attacks, for example, exploiting the date format.
- Severity 4 – needs specific credential to inject, for example, attack assumes that the administrator inserted malicious content into the database.
- Severity 5 – not a security problem, for example, a tainted variable is never printed in XSS sensitive context, meaning, HTML, XML, CSS, JSON, etc.

Criteria for associating warnings

Tool warnings may refer to the same weakness. (The notion of distinct weaknesses may be unrealistic. See Section 3.4 for a discussion.) In this case, we associated them, so that any analysis for one warning applied to every warning.

The following criteria apply to weaknesses that can be described using source-to-sink paths. A source is where user input can enter a program. A sink is where the input is used.

- If two warnings have the same sink, but the sources are two different variables, do not associate these warnings.
- If two warnings have the same source and sink, but paths are different, associate these warnings, unless the paths involve different filters.
- If the tool reports only the sink, and two warnings refer to the same sink and use the same weakness name, associate these warnings, since we may have no way of knowing which variable they refer to.

2.6 SATE Data Format

All participants converted their tool output to the common SATE XML format. Section 2.6.1 describes this tool output format. Section 2.6.2 describes the extension of the SATE format for storing our analysis of the warnings. Section 2.6.3 describes the format for storing the lists of associations of warnings.

2.6.1 Tool Output Format

In devising the tool output format, we tried to capture aspects reported textually by most tools. In the SATE tool output format, each warning includes:

- Id - a simple counter.
- (Optional) tool specific id.
- One or more locations, where each location is line number and pathname.
- Name (class) of the weakness, e.g., "buffer overflow".
- (Optional) CWE id, where applicable.
- Weakness grade (assigned by the tool):
    - Severity on the scale 1 to 5, with 1 - the highest.
    - (Optional) probability that the problem is a true positive, from 0 to 1.
- Output - original message from the tool about the weakness, either in plain text, HTML, or XML.
- (Optional) An evaluation of the issue by a human; not considered to be part of tool output. Note that each of the following fields is optional.
    - Severity as assigned by the human; assigned by the human whenever the human disagrees with the severity assigned by tool.
    - Opinion of whether the warning is a false positive: 1 – false positive, 0 – true positive.
    - Comments.

The XML schema file for the tool output format and an example are available at the SATE web page [19].

2.6.2 Evaluated Tool Output Format

The evaluated tool output format, including our analysis of tool warnings, has several fields in addition to the tool output format above. Specifically, each warning has another id (UID), which is unique across all tool reports. Also, the evaluation section has these additional optional fields:

- Confirmed – "yes" means that the human determined that the warning is correctly reporting a weakness.
- Stage – a number that roughly corresponds to the step of the SATE procedure, in which the evaluation was added:
  - Stage 3 – (optional) participants' review of their own tool's report.
  - Stage 4 – review by the SATE analysts.
  - Stage 5 – (optional) corrections by the participants. No participant submitted corrections in the xml format at that stage; however, Grammatech submitted a detailed document with corrections to our analysis of their tool's warnings.
  - Stage 6 – updates by the SATE analysts.
- Author – author of the evaluation. For each warning, the evaluations by SATE analysts were combined together and a generic name – "evaluators" - was used.

Additionally, the evaluated tool output format allows for more than one evaluation section per warning.

2.6.3 Association List Format

The association list consists of sets of unique warning ids (UID), where each set represents a group of associated warnings. (See Section 3.4 for a discussion of the concept of unique weaknesses.) There is one list per test case. Each set occupies a single line, which is a tab separated list of UIDs. For example, if we determined that UID 441, 754, and 33201 refer to the same weakness, we associated them. They are represented as:

```
441	754	33201
```

3 Data and Observations

This section describes our observations based on our analysis of the data collected.

3.1 Warning Categories

The tool outputs contain 104 different valid CWE ids; in addition, there are 126 weakness names for warnings that do not have a valid CWE id. In all, there are 291 different weakness names. This exceeds 104+126, since tools sometimes use different weakness names for the same CWE id. In order to simplify the presentation of data in this report, we placed warnings into categories based on the CWE id and the weakness name, as assigned by tools.

Table 3 describes the weakness categories. The detailed list is part of the released data available at the SATE web page [19]. Some categories are individual weakness classes such as XSS; others are broad groups of weaknesses. We included categories based on their prevalence and severity. The categories are derived from [3], [21], and other taxonomies. We designed this list specifically for presenting the SATE data only and do not consider it to be a generally applicable classification. We use abbreviations of weakness category names (the second column of Table 3) in Sections 3.2 and 3.3.

Name	Abbre-viation	Description	Example types of weaknesses
Cross-site scripting (XSS).	xss	The software does not sufficiently validate, filter, escape, and encode user-controllable input before it is placed in output that is used as a web page that is served to other users.	Reflected XSS, stored XSS
SQL injection	sql-inj	The software dynamically generates an SQL query based on user input, but it does not sufficiently prevent the input from modifying the intended structure of the query.	Blind SQL injection, second order SQL injection
Buffer errors	buf	Buffer overflows (reading or writing data beyond the bounds of allocated memory) and use of functions that lead to buffer overflows	Buffer overflow and underflow, unchecked array indexing, improper null termination
Numeric errors	num-err	Improper calculation or conversion of numbers	Integer overflow, incorrect numeric conversion, divide by zero
Command injection	cmd-inj	The software fails to adequately filter command (control plane) syntax from user-controlled input (data plane) and then allows potentially injected commands to execute within its context.	OS command injection
Cross-site request forgery (CSRF)	csrf	The web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.	
Race condition	race	The code requires that certain state not be modified between two operations, but a timing window exists in which the state can be modified by an unexpected actor or process.	File system race condition, signal handling
Information leak	info-leak	The intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information	Verbose error reporting, system information leak
Broad categories			
Improper input validation	input-val	Absent or incorrect protection mechanism that fails to properly validate input	Log forging, LDAP injection, resource injection, file injec-tion, path manipula-tion, HTTP response splitting, uncontrolled format string
Security features	sec-feat	Security features, such as authentication, access control, confidentiality, cryptography, and privilege management	Hard-coded password, insecure randomness, least privilege violation
Improper error handling	err-handl	An application does not properly handle errors that occur during processing	Incomplete, missing error handling, missing check against null
Insufficient encapsula-tion	encaps	The software does not sufficiently encapsulate critical data or functionality	Trust boundary violation, leftover debug code

Name	Abbreviation	Description	Example types of weaknesses
API abuse	api-abuse	The software uses an API in a manner contrary to its intended use	Heap inspection, use of inherently dangerous function
Time and state	time-state	Improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads	Concurrency weaknesses, session management problems
Quality problems	quality	Features that indicate that the software has not been carefully developed or maintained	Null pointer dereference, dead code, uninitialized variable, resource management problems, incl. denial of service due to unreleased resources, use after free, double unlock, memory leak
Uncategorized	uncateg	Other issues that we could not easily assign to any category	

Table 3 Weakness categories

Some weakness categories in Table 3 are subcategories of other, broader, categories. First, Cross-site scripting (XSS), SQL injection, and Command injection are kinds of improper input validation. Second, Race condition is a kind of Time and state weakness category. Due to their prevalence, we decided to use separate categories for these weaknesses.

When a weakness type had properties of more than one weakness category, we tried to assign it to the most closely related category.

3.2 Test Case and Tool Properties

In this section, we present the division of tool warnings by test case and by severity, as well as the division of reported tool warnings and confirmed tool warnings by weakness category. We then consider which of the SANS/CWE Top 25 weakness categories [17] are reported by tools. We also discuss some qualitative properties of test cases and tools.

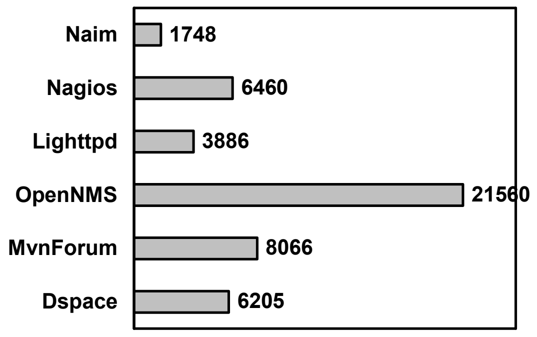


Figure 1 Warnings by test case (total 47925)

Figure 1 presents the numbers of tool warnings by test case. Almost half of the total warnings were for OpenNMS. We attribute it to the fact that the version of OpenNMS chosen for analysis was written prior to a major security overhaul [7].

Figure 2 presents the numbers of tool warnings by severity as determined by the tool. Grammatech CodeSonar uses rank (a combination of severity and likelihood) instead of severity. All warnings in their submitted reports had severity 1. We changed the severity field for some warning classes in the CodeSonar reports based on the weakness names. The numbers in Figure 2 and elsewhere in the report reflect this change.

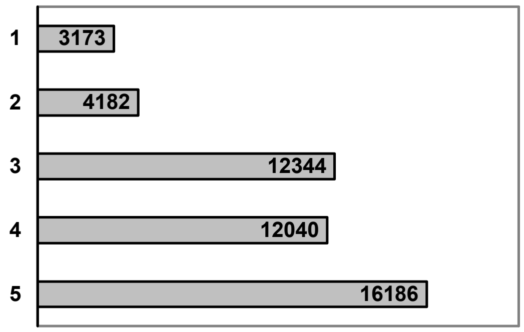


Figure 2 Warnings by severity (total 47925)

Weakness category	C track				Java track			
	All C	Naim	Nagios	Lighttpd	All Java	OpenNMS	MvnForum	DSpace
xss	0	0	0	0	2636	1748	471	417
sql-inj	0	0	0	0	715	179	483	53
buf	4525	674	2604	1247	0	0	0	0
num-err	958	155	560	243	438	174	196	68
cmd-inj	65	5	40	20	37	37	0	0
csrf	0	0	0	0	146	8	136	2
race	61	3	17	41	344	38	282	24
info-leak	1862	4	766	1092	1290	653	296	341
input-val	337	59	85	193	1851	670	303	878
sec-feat	59	30	8	21	5175	4021	333	821
quality	3030	431	1932	667	12019	8450	2380	1189
err-handl	674	113	302	259	7885	3923	2725	1237
encaps	0	0	0	0	1636	566	230	840
api-abuse	413	259	112	42	529	430	11	88
time-state	9	0	5	4	365	298	26	41
uncateg	101	15	29	57	765	365	194	206
Total	12094	1748	6460	3886	35831	21560	8066	6205

Table 4 Reported warnings by weakness category

Table 4 presents the numbers of reported tool warnings by weakness category for the C and Java tracks, as well as for individual test cases. The weakness categories are described in Table 3. Figure 3 plots the “All C” column of Table 4. Figure 4 plots the

"All Java" column. The figures do not show categories with no warnings for the corresponding track.

For the C track, there were no xss, sql-inj, csrf, and encaps warnings. In fact, Nagios has a web interface, and we found at least one instance of xss in the file cgi/status.c. However, since it is uncommon to write web applications in C, the tools tend not to look for web application vulnerabilities in the C code. For the Java track, there were no buf warnings - most buffer errors are not possible in Java.

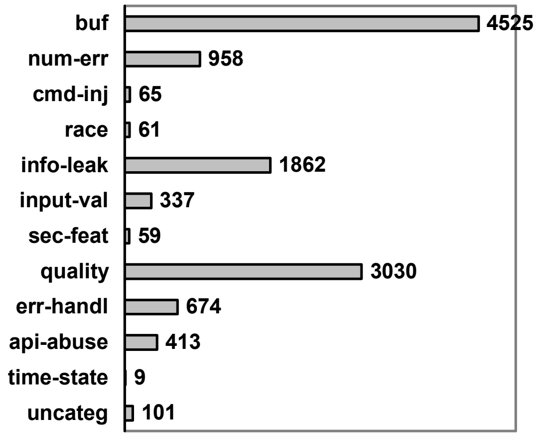


Figure 3 Reported warnings by weakness category - C track (total 12094)

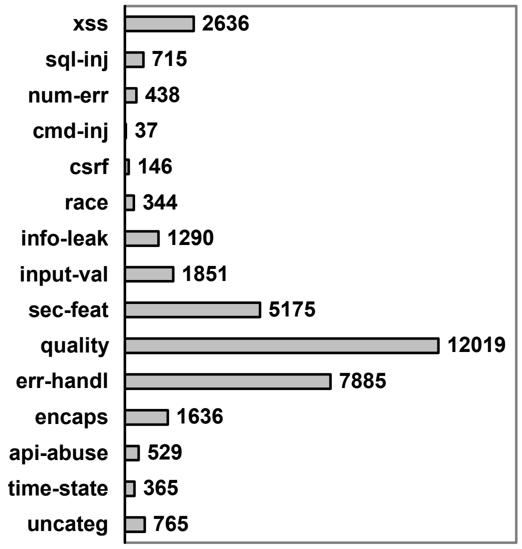


Figure 4 Reported warnings by weakness category - Java track (total 35831)

Table 5 presents the numbers of weaknesses confirmed by the analysts by weakness category for the C and Java tracks, as well as for individual test cases. Figure 5 plots the "All C" column of Table 5. Figure 6 plots the "All Java" column. The figures do not show categories with no confirmed weaknesses for the corresponding track. The numbers reflect our focus on analyzing severity 1 and 2 warnings and also the concentration of our efforts on a few weakness categories.

Weakness category	C track				Java track			
	All C	Naim	Nagios	Lighttpd	All Java	OpenNMS	MvnForum	DSpace
xss	0	0	0	0	711	167	448	96
sql-inj	0	0	0	0	57	40	6	11
buf	167	11	150	6	0	0	0	0
num-err	3	0	0	3	0	0	0	0
cmd-inj	9	3	5	1	9	9	0	0
csrf	0	0	0	0	138	1	136	1
race	21	2	6	13	24	0	0	24
info-leak	21	1	0	20	36	0	0	36
input-val	4	1	1	2	219	12	173	34
sec-feat	3	1	0	2	14	7	3	4
quality	206	40	26	140	11	10	0	1
err-handl	114	37	21	56	0	0	0	0
encaps	0	0	0	0	3	0	2	1
api-abuse	20	18	1	1	0	0	0	0
time-state	0	0	0	0	7	0	7	0
uncateg	4	1	0	3	0	0	0	0
Total	572	115	210	247	1229	246	775	208

Table 5 Confirmed weaknesses by weakness category

The tools are capable of finding weaknesses in a variety of categories. These include not just XSS, SQL injection and other input validation problems, but also some classes of authentication errors (e.g., hard-coded password, insecure randomness, and least privilege violation) and information disclosure problems.

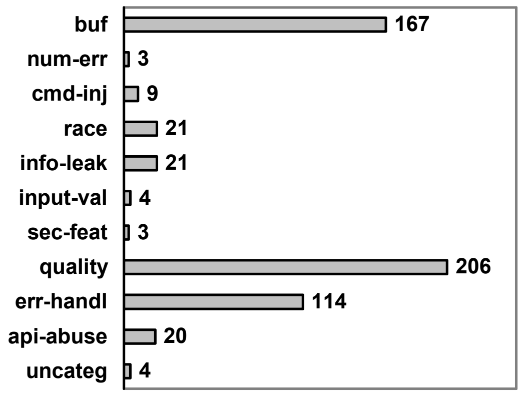


Figure 5 Confirmed weaknesses by weakness category - C track (total 572)

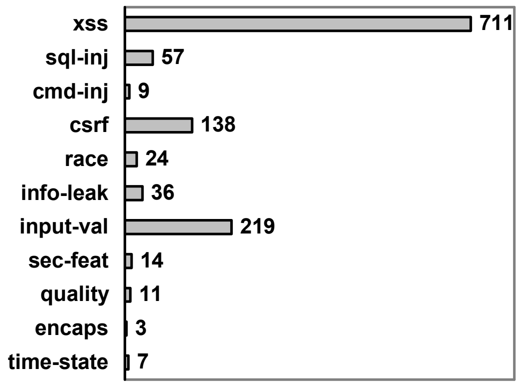


Figure 6 Confirmed weaknesses by weakness category - Java track (total 1229)

The 2009 SANS/CWE Top 25 Most Dangerous Programming Errors [17] is a list, selected by a group of software security experts, of the most significant weaknesses that can lead to serious software vulnerabilities. They organized the weaknesses into three high-level categories. They also selected some related CWE ids (not a comprehensive list) for each of the Top 25 weaknesses. Table 6 presents the CWE id and name of the weakness, and also related CWE ids.

CWE id	Weakness name	Related CWE ids
Insecure Interaction Between Components		
20	Improper Input Validation	184 74 79 89 95
116	Improper Encoding or Escaping of Output	74 78 79 88 89 93
89	Failure to Preserve SQL Query Structure (aka 'SQL Injection')	564 566 619 90
79	Failure to Preserve Web Page Structure (aka 'Cross-site Scripting')	692 82 85 87
78	Failure to Preserve OS Command Structure (aka 'OS Command Injection')	88
319	Cleartext Transmission of Sensitive Information	312 614
352	Cross-Site Request Forgery (CSRF)	346 441
362	Race Condition	364 366 367 370 421
209	Error Message Information Leak	204 210 538
Risky Resource Management		
119	Failure to Constrain Operations within the Bounds of a Memory Buffer	120 129 130 131 415 416
642	External Control of Critical State Data	472 565
73	External Control of File Name or Path	22 434 59 98
426	Untrusted Search Path	427 428
94	Failure to Control Generation of Code (aka 'Code Injection')	470 95 96 98
494	Download of Code Without Integrity Check	247 292 346 350
404	Improper Resource Shutdown or Release	14 226 262 299 401 415 416 568 590
665	Improper Initialization	453 454 456
682	Incorrect Calculation	131 135 190 193 369 467 681

CWE id	Weakness name	Related CWE ids
Porous Defenses		
285	Improper Access Control (Authorization)	425 749
327	Use of a Broken or Risky Cryptographic Algorithm	320 329 331 338
259	Hard-Coded Password	256 257 260 321
732	Insecure Permission Assignment for Critical Resource	276 277 279
330	Use of Insufficiently Random Values	329 331 334 336 337 338 341
250	Execution with Unnecessary Privileges	272 273 653
602	Client-Side Enforcement of Server-Side Security	20 642

Table 6 SANS/CWE Top 25 Weaknesses

	Warnings reported by tools	
CWE id	This CWE only	Incl. related CWEs
20	X	X
116		X
89	X	X
79	X	X
78	X	X
319		
352	X	X
362	X	X
209	X	X
119	X	X
642		X
73	X	X
426		
94		X
494		X
404	X	X
665		X
682		X
285		
327		
259	X	X
732		X
330	X	X
250	X	X
602		X

Table 7 Top 25 weaknesses reported by SATE tools

Table 7 illustrates which of the Top 25 weaknesses are reported by automated tools in SATE. The first column indicates the CWE id, the second column has a check mark if any tool reported warnings with this CWE id, the third column has a check mark if any tool reported warnings with this or related CWE id. For example, no tool reported CWE id 116, but tools reported related CWE ids. Since Aspect Security did not mark most of their warnings with CWE ids, the data in Table 7 is the same whether Aspect Security warnings are included or not.

The tools reported 13 of the Top 25 CWE ids. When related CWE ids are included, the tools reported 21 of the 25 CWE ids. Since the list of related CWE ids is not comprehensive and only about 75% of tool warnings have a CWE id, this table may underestimate the proportion of the Top 25 weaknesses reported by tools.

While some of the Top 25 weaknesses, such as Cleartext Transmission of Sensitive Information, are hard to find using static analysis tools, Table 7 suggests that the tools can help find weaknesses in most of the Top 25 weakness categories.

The human review by Aspect Security highlights the differences and synergies between human and automated reporting and analysis. While human review is needed for some types of weaknesses (e.g., some authorization problems), tools can quickly find hundreds of weaknesses. Sometimes the human describes the cause of the problem at a high level, while the tool provides the specific vulnerable paths for the instances of the problem. An example is in Section 3.4.4.

Overall, tools handled the code well, which is not an easy task for the test cases of this size. Some tools in the Java track had difficulty processing Java Server Pages (JSP) files, so they missed weaknesses in those files.

Project developers' programming style affects the ability of tools to detect problems and the ability of users to analyze the tool reports, as noted in [8]. This observation is supported by the SATE data. For example in Nagios, the return value of malloc, strdup, or other memory allocation functions is not checked for NULL immediately, instead, it is checked for NULL before each use. While this practice can produce quality code, the analysis has to account for all places where the variable is used.

Using black lists to filter input is not adequate. This observation is supported by the following example from MvnForum, which uses two types of filtering:

- For inserting data in HTML pages:
    - DisableHtmlTagFilter.filter (in myvietnam\src\net\myvietnam\mvncore\filter\DisableHtmlTagFilter.java) converts the special characters <>"& into their HTML entities. There is no check for special encoding.
    - urlResolver.encodeURL converts non-alphanumeric characters except ._- into the corresponding hex values.
- For checking file names: checkGoodFileName throws an exception if it finds any of the following characters: <>&:\0/\*?|.

Also, MvnForum sets Charset to UTF-8 using the meta tag in the JSP files.

For example, UID 27926 reports line 79 in mvnforum/srcweb/mvnplugin/mvnforum/admin/editgroupinfosuccess.jsp:

79 <td><b>&raquo;&nbsp;</b><a class="command" href="<%=**urlResolver.encodeURL**(request, response, "viewgroup?group=" + **ParamUtil.getParameterFilter(request, "group")**)%>"><fmt:message

key="mvnforum.admin.success.return_to_view_group"/></a> (<fmt:message key="mvnforum.common.success.automatic"/>)</td>

Function getParameterFilter applies DisableHtmlTagFilter.filter to its parameters. Since DisableHtmlTagFilter.filter converts only a few characters and there is no check for special encoding, we concluded that the warning is true and assigned it severity 2 (See the analysis criteria in Section 2.5.4).

3.3 On our Analysis of Tool Warnings

We analyzed (associated or marked as confirmed or unconfirmed) 5,899 warnings. This is about 12% of the total number of warnings (47,925). It is a non-random subset of tool warnings. In this section, we present data on what portions of test cases and weakness categories were analyzed. We also describe the effort that we spent on the analysis.

Figure 7 presents, by test case and for all test cases, the percentage of warnings of severity 1 and 2 (as determined by the tools) that were analyzed. It also gives, on the bars, the numbers of warnings that were analyzed/not analyzed. As the figure shows, we analyzed almost all severity 1 and 2 warnings for all test cases, except OpenNMS.

Figure 8 presents, by weakness category and for all categories, the percentage of warnings that were analyzed. It also gives, on the bars, the numbers of warnings that were analyzed/not analyzed. We use abbreviations of weakness category names from Table 3. As the figure shows, we analyzed a relatively large portion of xss, sql-inj, buf, cmd-inj, and csrf categories. These are among the most common categories of weaknesses. We were able to analyze almost all cmd-inj and csrf warnings because there were not a lot of them.

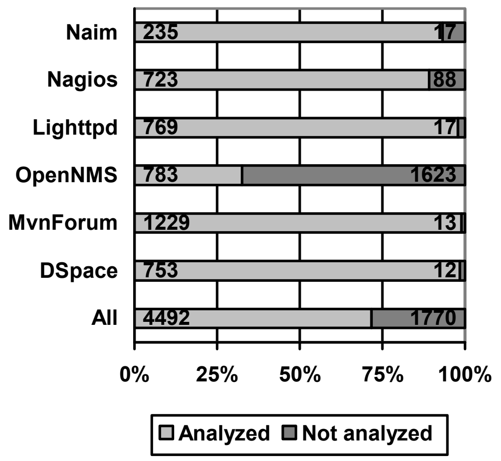


Figure 7 Severity 1, 2 warnings analyzed, by test case

Six people analyzed the tool warnings (spending anywhere from a few hours to a few weeks). All analysts were competent software engineers with knowledge of security; however, most of the analysts were only casual users of static analysis tools. 1,743 of 5,899 warnings (30%) were analyzed manually; the rest of the analysis was partially automated.

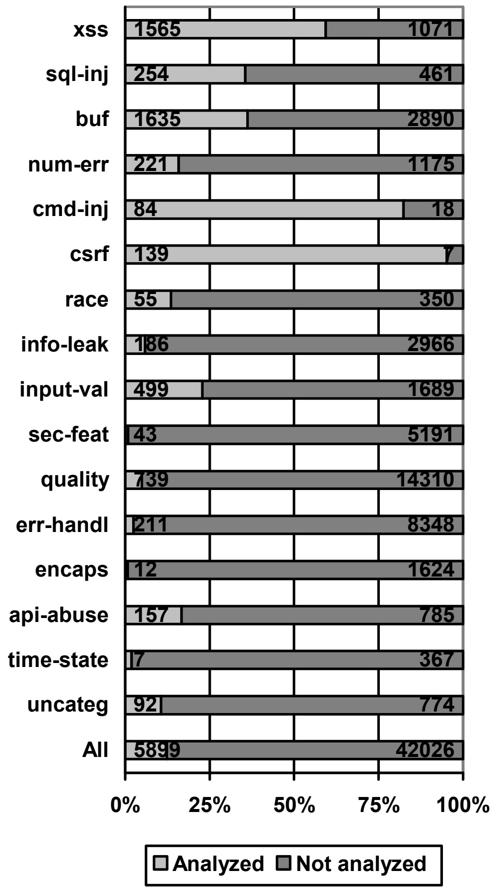


Figure 8 Warnings analyzed, by weakness category

The SATE analysis interface recorded when an analyst chose to view a warning and when he submitted an evaluation for a warning. According to these records, the analysis time for an individual warning ranged from less than 1 minute to well over 30 minutes. On average, the analysts spent between 4 and 10 minutes per warning analyzed manually.

We did not have a controlled environment for the analysis phase, so these numbers are approximate and may not reflect the actual time the analysts spent. Also, these numbers are not indicative of the time tool users can be expected to spend, because we used the tools differently and had different goals in our analysis.

3.4 On Differentiating Weakness Instances

We wanted to merge warnings that refer to the same weakness instance. Originally, we thought that each problem in the code had a unique manifestation, that it could be cleanly distinguished from other problems. In particular, we assumed that each problem could be linked with one or two particular statements. However, we found that the notion of one single distinct weakness instance is not reasonable in many cases. We also found that some weaknesses did not have well-defined locations.

The notion of distinct weakness instances breaks down when weaknesses are related as chains or composites and when data or control flows are intermingled. The notion of distinct instances is also questionable when there is a syndrome of a simple error repeated many times.

3.4.1 Chains, Composites, and Hierarchies

A single vulnerability may be the result of a chain of weaknesses or the composite effect of several weaknesses. "A 'Chain' is a sequence of two or more separate weaknesses that can be closely linked together within software." [2] For instance, an Integer Overflow CWE-190 in calculating size may lead to allocating a buffer that is smaller than needed, which can allow a Buffer Overflow CWE-120. Thus two warnings, one labeled as CWE-190 and one as CWE-120, might refer to the same vulnerability.

"A 'Composite' is a combination of two or more separate weaknesses that can create a vulnerability, but only if they all occur at the same time. For example, Symlink Following (CWE-61) is only possible through a combination of several component weaknesses, including predictability (CWE-340), inadequate permissions (CWE-275), and race conditions (CWE-362)." [2] As before, a vulnerability might give rise to a warning of CWE-61 from one tool, a warning of CWE-340 from another tool, and CWE-362 from a third tool.

Another problem is that some weakness classes are refinements of other weaknesses. In other words, some weaknesses can be organized into a hierarchy of superclasses and subclasses. For instance, UID 33888 warns of a Failure to Preserve Web Page Structure (aka Cross-Site Scripting or XSS) CWE-79 weakness in OpenNMS. But CWE-79 is a child or subclass of the more general Improper Input Validation CWE-20. Similar to the first example, two warnings labeled CWE-79 and CWE-20 might refer to the same vulnerability.

3.4.2 Intermingled Flows

Many vulnerabilities involve multiple statements, such as a flow of tainted data or one statement that frees memory and a later statement that uses that memory (Use After Free CWE-416). Because of shared routines, it is not uncommon for flows to be intermingled.

A good example is in base/events.c in Nagios. Code snippets appear in Figure 9. Two different functions, schedule_service_check() and schedule_host_check(), find an event (lines 1462 and 2603 respectively), remove the event's object from event_list, free the object, then call reschedule_event() to reschedule it. Reschedule_event() calls add_event() to add a newly allocated object to the list. The new event is added in its proper place (line 808) or added at the head with special code if necessary (line 819). This constitutes two uses of event_list after the object was freed.

One tool reported four warnings. UID 43522 cites lines 2603 and 819. UID 43523 cites lines 2603 and 808. UID 43524 cites lines 1462 and 819. UID 43525 cites lines 1462 and 808. See Figure 9 (b). Although these are not true instances of Use After Free CWE-416, the question persists: how many potential instances should be counted? There are two initial statements, two uses, and four paths all together. It does not seem correct to count every path as a separate weakness in this case nor to consider it as just a single freakish two-headed weakness. On the other hand, there is no symmetrical way to tease the flows apart into two distinct weaknesses.

```
    schedule_service_check(...){
        ...
1462    for(temp_event=event_list_low;temp_event!=NULL;
                           temp_event=temp_event->next){
            ...
            }
        ...
        remove_event(temp_event,&event_list_low);
1503    free(temp_event);
        ...
        reschedule_event(new_event,&event_list_low);



    schedule_host_check(...){
        ...
2603    for(temp_event=event_list_low;temp_event!=NULL;
                           temp_event=temp_event->next){
            ...
            }
        ...
        remove_event(temp_event,&event_list_low);
2644    free(temp_event);
        ...
        reschedule_event(new_event,&event_list_low);



    reschedule_event(...,timed_event **event_list){
        ...
        add_event(event,event_list);


    add_event(...,timed_event **event_list){
        first_event=*event_list;
        ...
 808    else if(event->run_time < first_event->run_time){
            ...
        else{
            temp_event=*event_list;
            while(temp_event!=NULL){
 819            if(temp_event->next==NULL){
```

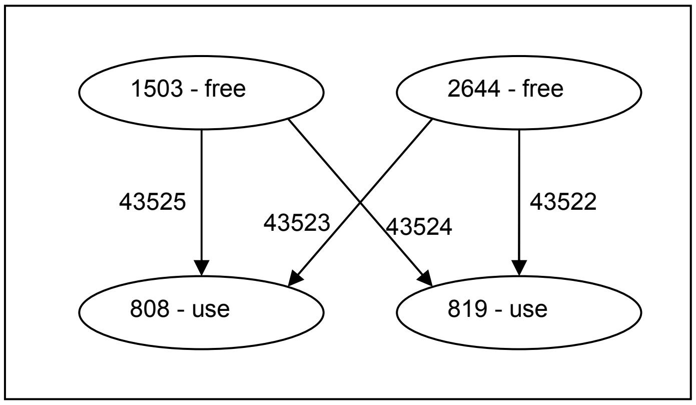


Figure 9 Two memory free locations connected to two suspected uses constituting four paths and four warnings of Use After Free CWE-416. (a) Code snippets (b) Diagram: the ovals contain line numbers; arrows are labeled with warning UIDs.

Another, more extreme case is UID 17182 which associates some 70 different warnings of Buffer Overflow CWE-120 in lighttpd, src/mod_dirlisting.c at line 638.

```
638    strcpy(path, dir->ptr);
```

The suspect statement is invoked indirectly from dozens of places. Careful analysis shows that buffer overflow is not possible. If indeed the code needed to be fixed, it might be done at that single statement, in a few immediately calling functions, or changes might be needed at many of the warning locations.

3.4.3 Repeated Error Syndrome

Sometimes the same simple mistake is repeated over and over, analogous to a word misspelled the same way throughout a document. Although there may be a weakness at every location where the error was repeated, there is some sense that it is a single syndrome, especially when the errors can be corrected with a single syntactic replacement or addition.

For instance, UIDs 33888, 40681, and 42763 all warn of Cross-Site Scripting (XSS) CWE-79 in OpenNMS. Cross-site scripting may occur when user input, such as a comment or even a name, is returned from the application to a user's web browser. If not properly sanitized, the comment may contain script code or commands, which are executed by the browser. Since OpenNMS had not been written with security in mind, it is not surprising to see the same error in many places.

3.4.4 Other Thoughts on Location

Even when weaknesses can be clearly distinguished, many types of weaknesses are the result of several statements interacting. For these it is not reasonable to say the weakness is at a single statement. An example is the just-cited UID 40681 warning of Cross-Site Scripting CWE-79. It reports the following path in web/Util.java.

1. src/web/src/org/opennms/web/Util.java (304)
2. src/web/src/org/opennms/web/Util.java (304)
3. src/web/src/org/opennms/web/Util.java (307)
4. src/web/src/org/opennms/web/Util.java (307)
5. src/web/src/org/opennms/web/Util.java (313)
6. src/web/src/org/opennms/web/Util.java (345)
7. src/web/src/org/opennms/web/Util.java (345)
8. src/web/src/org/opennms/web/Util.java (271)
9. src/web/src/org/opennms/web/Util.java (271)
10. src/web/src/org/opennms/web/Util.java (251)
11. src/web/src/org/opennms/web/Util.java (251)
12. src/web/web/performance/chooseinterface.jsp (129)
13. src/web/web/performance/chooseinterface.jsp (129)

The first lines are in the function makeHiddenTags():

```
292    makeHiddenTags(...) {
            ...
304      Enumeration names = request.getParameterNames();
             ...
307        String name = (String) names.nextElement();
                     ...
313            buffer.append(name);
```

```
             ...
345        return (buffer.toString());
      }
```

UID 189 warns of CWE-79 in web/Util.java, too, but refers to line 292, and calls out specifically makeHiddenTags(). UIDs 190, 191, and 192[7] refer to uses of makeHiddenTags(), in adhoc.jsp (line 100), chooseinterface.jsp (line 129), and choosereportanddate.jsp (line 147), respectively. UIDs 40681 and 191 are clearly different warnings about the same instance. But UID 189 is arguably the same instance as 40681, too, even though they do not refer to any of the same statements.

Another complication in assigning a location to an instance is that some weaknesses refer to a region of code, not just statements in a flow. Dead Code CWE-561 or Leftover Debug Code CWE-489 may relate to large pieces of code, not just a few statements or even a path. Improper Input Validation CWE-20 might be mitigated anywhere along the path that the input is accessible, not necessarily immediately after the input or before its first use.

If a function is missing entirely, no statement or even group of statements is wrong. Unimplemented or Unsupported Feature in UI CWE-447 might point out some location in the user interface processing where the flow might be diverted, but again no existing statement is wrong.

Even when particular statements are involved, it is not obvious what statement should be reported. Certainly a path may be helpful, but typically a particular statement is indicated for summary or initial report purposes. Should it be the source statement, where the data or state first appears? The statement where a fault can occur? The statement where it is best mitigated? Likely the answer is different for different weaknesses and possibly depends on many details of the code and context.

To summarize, a simple weakness can be attributed to one or two specific statements and associated with a specific CWE. In contrast, a non-simple weakness has one or more of these properties:

- Can be associated with more than one CWE (e.g., chains and composites).
- Can be attributed to many different statements.
- Has intermingled flows.

We estimate that only between 1/8 and 1/3 of all weaknesses are simple weaknesses.

4 Critique of SATE

This section explains the reasons for choices we made in organizing SATE, changes from our original plan, and lessons we learned from the exposition. These are grouped into three areas: test case selection, analysis of tool reports, and the data format. Finally, Section 4.4 presents our findings from a reanalysis of a small subset of warnings that we did to get an idea of the types and frequency of errors made in the initial analysis.

[7] UIDs 189, 190, 191, and 192 were reported by Aspect Security as a single warning. In general, Aspect Security reported different warnings of the same kind as a single warning. For our analysis, we divided these into separate warnings.

4.1 Test Cases

We chose C and Java due to their popularity and the many static analysis tools which support them. We looked for test cases of moderate size: thousands to hundreds of thousands lines of code. We believed that this was a reasonable compromise between representing real life software and our ability to analyze the tool output. We also looked for code that is representative of today's state of practice. Since SATE is focused on security, we chose software with aspects relevant to security.

The test cases have a number of properties that may or may not be representative of today's software. First, all test cases are open source programs. Second, all three Java test cases, as well as Nagios, have a web interface. Web applications have exploitable, relevant errors; however, these types of errors are not the focus of some tools.

Third, the code quality and design of all test cases were decent. However, some test cases were developed without much attention to security, although that is changing [7].

We chose stable versions of the software as test cases. Sometimes, that meant taking a version released a year or more prior. Some project developers told us that they would have preferred us to analyze the latest version instead.

The stable version of OpenNMS was difficult for some participants to compile and run. Choosing a later version would have avoided this problem. Additionally, the version of OpenNMS chosen was written prior to a major security overhaul and had a very large number of security issues. It had many more warnings than the other test cases.

4.2 Analysis of Tool Reports

Originally, we planned to create a master reference list of security weaknesses found in the test cases and to compare each tool's security warnings against this master reference, thus marking all warnings as true positives or false positives. We planned to accomplish this by grouping (associating) tool warnings that refer to the same weakness and analyzing correctness of the warnings. The two main adjustments that we made, as detailed below, were (1) not producing the master reference list and (2) replacing the true/false markings with the confirmed/unconfirmed markings.

For the following three reasons, we had to abandon the goal of producing a master list of security weaknesses. First, we received much more tool warnings than we anticipated. Second, we found that analyzing tool warnings (both analyzing correctness and determining when warnings refer to the same weakness) was more difficult than expected. Third, we had hoped for more resources to analyze the warnings. Therefore, we chose to analyze a small subset of tool warnings (mostly those assigned high severity by the tool) and did not associate many warnings that refer to the same weakness.

Due to the variety and different nature of security weaknesses, defining clear and comprehensive analysis criteria is difficult. As SATE progressed, we realized that our analysis criteria were not adequate. We adjusted the criteria during the analysis phase, but did not have resources to go back and reanalyze all warnings. As a result, the criteria were not applied consistently.

The determination of correctness turned out to be more complicated than simply a binary decision for these three reasons. First, there are warnings that are true, but are not useful.

An example is a warning that describes properties of a standard library function without regard to its use in the code. Second, there are warnings that require a very large amount of time to analyze for correctness exhaustively. Third, there are true warnings for which we do not see any relevance to security. Similarly, determining whether a weakness is relevant to security is often difficult.

These limitations of analysis make computing rates of true and false positives from the analysis data meaningless and counter-productive. For these reasons, and also to minimize the chance of people making unsupported conclusions, after the analysis phase was completed, we decided not to use the true/false positive markings, but rather the confirmed/unconfirmed markings. "True" became "confirmed," all other warnings that we analyzed or associated became "unconfirmed."

4.3 SATE Output Format

In order to have a consistent analysis of output from different tools, we defined a common tool output format. Our goal was to preserve as much content from the original tool report as possible.

In devising the tool output format, we tried to capture aspects reported textually by most tools. In particular, we included a field for CWE id and encouraged participants to include this information, because the CWE [3] provides a comprehensive set of weakness classes. A consistent weakness designation scheme is important for having a consistent view of different tools. Also, we included one or more code locations in order to represent a trace through the code. Some weaknesses, such as use of dangerous function, can be represented with a single location. Others are best explained by providing a control flow or data flow path.

In retrospect, the SATE output format has several limitations. First, the format is not expressive enough; in particular, it cannot represent multiple traces. Often, the same weakness manifests itself via multiple paths through the code where, for example, the sink and the source and the affected variables may be the same. To help the user, a tool may combine such paths into a single warning.

To better understand the warnings, we used some of the information (details of weakness paths) from Grammatech CodeSonar and Fortify SCA reports in their original formats. Other tools also use expressive output formats. This suggests that the SATE output format omits important information compared to the original tool output.

Second, the format was not specified with sufficient detail making it necessary for us to spend a considerable time working with the participants in modifying the submitted tool reports so that the reports conform to the expected format. The problem areas included inconsistent weakness paths and incorrect conversions from the original tool formats.

Third, the participants often had difficulty mapping their warning types to the CWE ids. This is due to a large number of (and complex relationships between) different weakness types. CWE has hundreds of entries organized into a hierarchy. Depending on perspective, some tool warnings may be mapped to different CWE entries (in particular, entries at different levels of the CWE hierarchy). Some tool warning types do not match any CWE entry precisely.

Finally, we did not provide enough guidance to the participants on mapping their tool's severity classification to the severity field in the SATE format.

4.4 Reanalysis Findings

After the analysis phase was completed, we reanalyzed a small (and not statistically significant) subset of 30 SATE warnings. For this reanalysis, from each of the 6 test cases, we randomly chose 5 warnings that were marked as either true or false and were assigned severity 1 or 2.

The goal of this reanalysis was to understand better the types and frequency of errors that we made during the original analysis (steps 3-5 of the SATE procedure). In particular, we looked for cases where we incorrectly marked a warning as true or false and where we failed to combine a warning with other warnings that refer to the same weakness. We used the same analysis criteria as during the original analysis. We did not consider any severity annotations we made.

The main data findings are as follows.

1. We incorrectly marked 5 warnings (1/6) as true or false. Since this involved changes in both directions (2 from true to false, 3 from false to true), the change in the overall proportion of true and false positives is small.
2. We failed to combine 13 warnings (about 2/5) with other warnings, with a large effect on simplistic overlap rates.

Other significant observations include:

1. The original analysis underestimated the tool overlap.
2. We found additional evidence that analysis criteria (e.g., criteria for combining warnings) can impact the quantitative results.

This reanalysis was one of the motivating factors for replacing true/false positive markings with confirmed/unconfirmed categories.

5 Conclusions and Future Plans

We conducted the Static Analysis Tool Exposition (SATE) in order to enable empirical research on large data sets and encourage improvement and adoption of tools. Participants ran their tools on 6 test cases - open source programs from 23k to 131k lines of code. Nine participants returned 31 tool reports with a total of 47,925 tool warnings. We analyzed approximately 12% of the tool warnings, mostly those rated high severity by tools. Several participants improved their tools based on their SATE experience.

The released data is useful in several ways. First, the output from running many tools on production software is available for empirical research. Second, our analysis of tool reports indicates weaknesses that exist in the software and that are reported by the tools. The analysis may also be used as a building block for a further study of the weaknesses in the code and of static analysis.

We observed that while human analysis is best for some types of weaknesses, tools find weaknesses in many important weakness categories and can quickly identify and describe in detail many weakness instances. In particular, the tools can help find weaknesses in most of the SANS/CWE Top 25 weakness categories.

We identified shortcomings in the SATE procedure, including test case selection, analysis of tool reports, and the data format. In particular, the tool interface is important in understanding most weaknesses – a simple format with line numbers and a few other fields is not enough. Also, binary true/false positive verdict on tool warnings is not enough.

The tools' philosophies about static analysis and reporting are often very different, so they produce substantially different warnings. For example, tools report weaknesses at different granularity levels. The SATE experience suggests that the notion that weaknesses occur as distinct, separate instances is not reasonable in most cases. We also found that some weaknesses did not have well-defined locations. We hope that these and other lessons from SATE will help improve further similar studies.

Due to complexity of the task and limited resources, our analysis of the tool reports is imperfect, including lapses in analyzing correctness of tool warnings. Also, in many cases, we did not associate warnings that refer to the same weakness. For these and other reasons, our analysis must not be used as a direct source for rating or choosing tools or even in making a decision whether or not to use tools.

For the next SATE, analysis of tool reports must be improved. First, since complete analysis is impractical, a better scheme for choosing a representative subset of warnings for analysis is needed. Second, involving more people in the analysis of tool reports would allow us to cover more warnings as well as get different views on the more complex warnings. Since project developers know the most about their software, involving them in the analysis of tool reports may improve the accuracy of analysis and provide a different measure of tool utility, i.e., what warnings actually lead to fixes.

The following are some additional ideas for making SATE easier for participants and more useful to the community.

- Provide participants with a virtual machine image containing the test cases properly configured and ready for analysis by the tools.
- Provide participants with a more precise definition of the SATE tool output format or scripts to check the output for format errors.
- Select the latest, beta versions of the open source software as test cases instead of the stable earlier versions.
- Include a track for another tool class, such as web application security scanners.

6 Acknowledgements

Bill Pugh came up with the idea of SATE, participated in the design of the output format, and provided valuable critique. SATE is modeled on the NIST Text REtrieval Conference (TREC): http://trec.nist.gov/. We thank Steve Christey, Bob Schmeichel and Bob Martin for help with organization and playing an important role in the analysis. We thank other members of the NIST SAMATE team for their help during all phases of the exposition. We especially thank the participants – Paul Anderson, Fletcher Hirtle, Tucker

Taft, Maty Siman, Brian Chess, Katrina O'Neil, Fredrick DeQuan Lee, Chris Wysopal, Bill Pugh, Jeff Williams, Nidhi Shah - for their effort, valuable input, and courage.

7 References

[1] Accelerating Open Source Quality, http://scan.coverity.com/.
[2] Chains and Composites, The MITRE Corporation, http://cwe.mitre.org/data/reports/chains_and_composites.html.
[3] Common Weakness Enumeration, The MITRE Corporation, http://cwe.mitre.org/.
[4] Emanuelsson, Par, and Ulf Nilsson, A Comparative Study of Industrial Static Analysis Tools (Extended Version), Linkoping University, Technical report 2008:3, 2008.
[5] Frye, C., Klocwork static analysis tool proves its worth, finds bugs in open source projects, SearchSoftwareQuality.com, June 2006.
[6] Gaucher, Romain, SATE 2008: Automated Evaluation, Presentation, PLDI 2008, Static Analysis Workshop, June 12, 2008, Tucson, AZ
[7] Gehlbach, Jeff, Injections and reflections and web apps, oh my! http://jeffgehlbach.com/?p=13.
[8] Holzmann, G.J., The Power of Ten: Rules for Developing Safety Critical Code, IEEE Computer, June 2006.
[9] Java Open Review Project, Fortify Software, http://opensource.fortifysoftware.com/
[10] Johns, Martin, Moritz Jodeit, Wolfgang Koeppl and Martin Wimmer, Scanstud - Evaluating Static Analysis Tools, OWASP Europe 2008.
[11] Kratkiewicz, K., and Lippmann, R., Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools, In Workshop on the Evaluation of Software Defect Tools, 2005.
[12] Livshits, Benjamin, Stanford SecuriBench, http://suif.stanford.edu/~livshits/securibench/.
[13] Michaud, F., and R. Carbone, Practical verification & safeguard tools for C/C++, DRDC Canada – Valcartier, TR 2006-735, 2007.
[14] Rutar, N., C. B. Almazan and J. S. Foster, A Comparison of Bug Finding Tools for Java, 15th IEEE Int. Symp. on Software Reliability Eng. (ISSRE'04), France, Nov 2004.
[15] SAMATE project, https://samate.nist.gov/.
[16] SAMATE Reference Dataset (SRD), http://samate.nist.gov/SRD/.
[17] SANS/CWE Top 25 Most Dangerous Programming Errors, http://cwe.mitre.org/top25/.
[18] Source Code Security Analysis Tool Functional Specification Version 1.0, NIST Special Publication 500-268. May 2007, http://samate.nist.gov/docs/source_code_ security_analysis_spec_SP500-268.pdf.
[19] Static Analysis Tool Exposition (SATE), http://samate.nist.gov/index.php/SATE.html
[20] Static Analysis Workshop 2008, Co-located with Conference on Programming Language Design and Implementation, Tucson, AZ, June 2008.
[21] Tsipenyuk, K., B. Chess, and G. McGraw, "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," to be published in *Proc. NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM)*, US Nat'l Inst. Standards and Technology, 2005.
[22] Wheeler, David A., SLOCCount, http://www.dwheeler.com/sloccount/.
[23] Zheng, J., L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. Vouk, On the Value of Static Analysis for Fault Detection in Software, IEEE Trans. on Software Engineering, v. 32, n. 4, Apr. 2006.
[24] Zitser, M., Lippmann, R., Leek, T., Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code. In IGSOFT Software Engineering Notes, 29(6):97-106, ACM Press, New York (2004).

Appendix A SATE Plan (as of February 2008)

Introduction

Goals

- To enable empirical research based on large test sets
- To encourage improvement of tools
- To speed adoption of the tools by objectively demonstrating their use on real software

Our goal is *not* to choose the "best" tools: there are many other factors in determining which tool (or tools) is appropriate in each situation.

Characteristics to be considered

- Relevance of warnings to security
- Correctness of warnings (true positive or false positive)
- Prioritization of warnings (high, medium, ...)

Note. A warning is an issue identified by a tool. A (Tool) report is the output from a single run of a tool on a test case. A tool report consists of warnings.

Language tracks

- The exposition consists of 2 language tracks:
    - C track
    - Java track
- Participants can enter either track or both
- Separate analysis and reporting for each track

Call for participation

We invite participation from makers of static analysis tools that find weaknesses relevant to security. We welcome commercial, research, and open source tools.
If you would like to participate in the exposition, please email Vadim Okun (vadim.okun@nist.gov).

Protocol

Briefly, organizers provide test sets of programs to participants. Participants run their own tool on the test cases and return the tool reports. Organizers perform a limited analysis of the results and watch for interesting aspects. Participants and organizers report their experience running tools and their results during a workshop. Organizers make the test sets, tool reports, and results publicly available 6 months after the workshop.
Here is the protocol in detail.

Step 1 Prepare

Step 1a Organizers choose test sets

- A test set for each language track
- A test set consists of up to 3 open source programs (or program components)
- Size of each program is at least several thousand lines of code
    - We anticipate some of the test cases to be tens or hundreds of thousands lines of code
- Each program has aspects relevant to security
- We expect programs to have various kinds of security defects
- We expect the code to be representative of today's state of practice
- Compilable on a Linux OS using a commonly available compiler

Step 1b Tool makers sign up to participate (8 Feb 2008)

- Participants specify which track(s) they wish to enter
- For each track, participants specify the exact version(s) of the tool(s) that they will run on the test set. The version must have release or build date that is earlier than the date when they receive the test set.

Step 2 Organizers provide test set(s) (15 Feb 2008)

- Organizers will specify the method of distribution in advance

Step 3 Participants run their tool on the test set(s) and return their report(s) (by 29 Feb 2008)

- Participants cannot modify the code of the test cases, except possibly for comments (e.g. annotations).
    - If annotations were manually added, note this and send back the modified test case with annotations.
- For each test case, participants can do one or more runs and submit the report(s)
    - participants are encouraged to do a run that uses the tool in default configuration.
    - participants may do custom runs (e.g., the tool is configured with custom rules). For a custom run, specify the affected settings (e.g., custom rules) in enough detail so that the run can be reproduced independently.
- Participants specify the environment (including the OS, version of compiler, etc.) in which they ran the tool
- Hand editing of the tool reports (e.g., manually removing false positives or adding true positives) is not allowed.
- The reports are in common format (in XML). See Tool output format.
- Participants can withdraw from any language track or from the exposition prior to this deadline. In that case, their intention to participate and decision to withdraw will not be disclosed.

Step 3a (optional) Participants return their review of their tool's report(s) (by 15 Mar 2008)

Step 4 Organizers analyze the reports (by 15 April 2008)

- For each test case, combine all submitted reports and information from other analysis
- Come up with a master reference list, that is, true positives, of security relevant weaknesses
- Compare each tool's security warnings against the master reference: true positives, false positives
- Participants receive the master reference list, comparison of their report with the master reference list, reports from other tools

Note. We do not expect (and will emphasize this in our report) that the master reference list will be perfect. Participants are welcome to submit a critique of the master reference list, either items missing or incorrectly included.

Step 4a (Optional) Participants return their corrections to the master reference list (by 29 April 2008)

Step 4b Participants receive an updated master reference list and an updated comparison of their report with the master reference list (by 13 May 2008)

Step 4c Participants submit a report for SAW (by 30 May 2008)

- The participant's report presents experience running the tool, discussion of their tool's results, etc.
- The report is a paper up to 10 pages long

Step 5 Report Comparisons at SAW (June 2008)

- Organizers report comparisons and any interesting observations.
- Participants receive the detailed comparisons for *all* participating tools (see next step for what these include)
- Participants report their experience running the tools and discuss their results
- Discuss comments, suggestions, plans for the next exposition

Step 5a Participants submit final version of report (from Step 4c) (by June 30 2008)

- To be published as NIST special publication or NIST technical report

Step 6 Publish Results (Dec 2008)

- Organizers publish test sets, master reference list, and detailed comparisons, including
    - tool version and any configuration parameters (e.g., custom rule set) used

    - verbatim textual report from the tool
    - warning by warning comparison with the master list

Future plans

We plan for the exposition to become an annual event. Our future plans include the following.

- Multiple tracks for different domains
- More languages
- Other tool classes
    - Web Application Security Scanners
    - Binary analysis
- Static analysis for other purposes
- Consider metrics (program size, assurance level, size of security problem, etc.)
- Consider inserting back doors in application
- Interactive track: to measure the way the tool is used by the programmers

Tool output format

The tool output format is an annotation for the original tool report. We would like to preserve all content of the original tool report.
Each warning includes

- weakness id - a simple counter
- (optional) tool specific unique id
- one or more locations, including line number and pathname
- name of the weakness
- (optional) CWE id, where applicable
- weakness grade
    - severity on the scale 1 to 5, with 1 - the highest
    - (optional) probability that the problem is a true positive, from 0 to 1
- output - original message from the tool about the weakness, either in plain text, HTML, or XML

Commentary on CodeSonar's SATE Results

Paul Anderson, GrammaTech, Inc.

This document provides a commentary on the results of running CodeSonar on the example C programs for the SATE experiment. These programs were:

- Lighttpd-1.4.18
- Nagios-2.10
- Naim-0.11.8.3.1

The following sections describe issues with the configuration and use of CodeSonar that may help with the interpretation of the results that it generated.

Domain of Application

CodeSonar is designed to be a general purpose tool for finding programming errors in C and C++ programs. It can be used to uncover security weaknesses, violations of safety-critical programming rules, inconsistencies, as well as generic programming errors. As such some of the warnings it reports have little impact on security, except in that they may indicate programmer confusion. One such warning class is *Unused Value*. This is reported when a variable is assigned a value, but that value is never used. It is very unlikely that such a weakness could be used to construct a security exploit, but this class of weaknesses is prohibited in much safety-critical code.

Similar Warnings

CodeSonar may report different instances of the same underlying flaw as multiple warnings. For example, warnings 483.629 through 483.631. The CodeSonar user interface treats these warnings as one in many respects, but in the report submitted to SATE, they are not aggregated. The warning IDs can be used to tell which warnings are treated as one. Given a warning with id *x.y*, all other warnings whose ID begins with *x* are considered identical.

A second effect can also give rise to similar warnings. If there are multiple interprocedural paths from different callers to the same program point, then these warnings may seem the same because they are reported as terminating on the same line. The warnings reported on line 46 of buffer.c in project lighttpd are good examples of this. These may or may not be the same "bug". The only way to effectively tell is by looking at the path to the point. This information is available through the CodeSonar user interface, but may not be evident from the report submitted to SATE.

Rank

CodeSonar associates a single *rank* attribute with each warning. This is designed to suggest an order in which a user ought to review the warnings. As such it is a combination of the severity of the problem and the confidence the tool has that the warning is a true positive. It is not meant to be a measure of the risk of the offending code.

Malloc

The default configuration of CodeSonar assumes that calls to memory-allocating functions, especially *malloc* can return NULL if the request cannot be satisfied. As a result, this gives rise to a large number of null pointer dereference and related warnings being issued in all of the applications. We consider these to be true positives because this is a genuine error condition that the code should handle. The error can occur not only if the application runs out of memory, but also if the size requested is unreasonable or if there is excessive fragmentation on the heap.

However some users may consider these warnings extraneous because their applications are deployed in environments where this condition is unlikely to occur, and the risk of harm if it does occur is low. There is a parameter to CodeSonar, named MALLOC_FAILURE_BEHAVIOR that can be used to disregard the possibility that the allocation can fail. In order to explore the effect of this, we ran the analysis again on all the projects with this parameter set to DOESNT_FAIL.

Doing this generally has two effects on the results. First, the number of null pointer dereferences usually goes down. A second effect is that the number of redundant condition warnings goes up. This latter warning is issued when a condition is being checked that is known to be either always true or always false. A reduction in these usually indicates that the return value of malloc *is* being checked.

Table 1. The change in the number of warnings reported for each project when the configuration parameter MALLOC_FAILURE_BEHAVIOR is changed from RETURN_NULL (the default) to DOESNT_FAIL.

Program	Null pointer dereference	Redundant condition
Lighttpd	-167	+27
Nagios	-3	+117
Naim	-49	+32

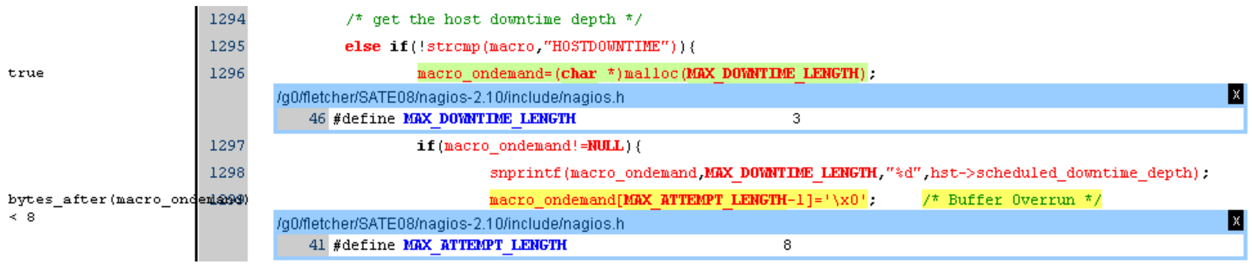
Table 1 above shows the effect on the number of each warning when this parameter is changed. In addition, the number of *unreachable code* warnings for nagios increased for the same reason. The number of other warnings did not change significantly.

It is evident from this that both lighttpd and naim are not written to carefully consider what should happen when allocators fail. However, nagios is written to take account of this possibility. This indicates that the value of that parameter should be set differently depending on the different styles of programming.

Use of Complex Macros

The naim source code uses macros very heavily. For example, see the definition of HOOK_ADD in the file modutil.h. This code is written in a way that makes it difficult for CodeSonar to report meaningful information about the location of the warning. In order to be precise, CodeSonar runs the preprocessor on the code, which means that those constructs are modeled correctly. However, the entire expansion of the macro is to a single line in the code. This means that multiple warnings are reported for the same line, even though they are caused by different parts of the body of the macro. For example, the function fireio_hook_init in file fireio.c shows several warnings on line 902. Each warning corresponds to a separate program point in the expansion of the macro, but all those points end up being associated with the same line in the compilation unit. This makes them difficult to diagnose and categorize.

CodeSonar provides an interface to help users understand what macros expand to. This can help diagnose warnings as illustrated below in Figure 1, but it has its limitations, especially for complex multi-line macros.



```
                 1294         /* get the host downtime depth */
                 1295         else if(!strcmp(macro,"HOSTDOWNTIME")){
true             1296                 macro_ondemand=(char *)malloc(MAX_DOWNTIME_LENGTH);
     /g0/fletcher/SATE08/nagios-2.10/include/nagios.h                                          X
       46 #define MAX_DOWNTIME_LENGTH                    3
                 1297                 if(macro_ondemand!=NULL){
                 1298                         snprintf(macro_ondemand,MAX_DOWNTIME_LENGTH,"%d",hst->scheduled_downtime_depth);
bytes_after(macro_ondemand)
< 8              1299                         macro_ondemand[MAX_ATTEMPT_LENGTH-1]='\x0';     /* Buffer Overrun */
     /g0/fletcher/SATE08/nagios-2.10/include/nagios.h                                          X
       41 #define MAX_ATTEMPT_LENGTH                     8
```

Figure 1. A screenshot from the CodeSonar user interface showing how the user can see how macros have been expanded. This is an excerpt from warning 794.988, on line 1299 of file utils.c.

Static Analysis Tool Exposition (SATE 2008) Lessons Learned: Considerations for Future Directions from the Perspective of a Third Party Analyst

Steve Christey
The MITRE Corporation
coley@mitre.org

Abstract

In 2008, the NIST SAMATE project conducted a Static Analysis Tool Exposition (SATE) to understand the capabilities of software-based tools that automatically identify security weaknesses in source code. Organizers selected real-world open source programs and used a simplified exchange format to obtain automatically-generated warnings from the tools of participating vendors. These warnings were then collected into a single database. Experienced analysts selected a subset of these warnings and manually evaluated the warnings for accuracy. Preliminary results were published in 2008 before the release of the final report and data set in 2009.

This paper highlights several important considerations for evaluating and understanding the capabilities of static analysis tools. While SATE's web-based interface demonstrates powerful features to assist in the evaluation, the lack of direct access to the tools can adversely impact the speed and accuracy of the human analysts. Appropriate sampling techniques may need to be devised if there are not enough resources to evaluate all the warnings that have been generated. To consistently evaluate the warnings, clearly specified definitions of true positives and false positives are required, and there may be variation in how human analysts will interpret warnings. Quantitative and comparative methods should also account for differences in how security weaknesses are counted and described by the tools. Finally, this paper provides specific suggestions for future efforts in the evaluation of security-related software analysis tools.

Table of Contents

Introduction

The first Static Analysis Tool Exposition (SATE[1]) was conducted in 2008, led by the NIST SAMATE team [1]. SATE 2008 has demonstrated the powerful capabilities of modern static analysis tools and provides valuable lessons for future analyses. This paper captures some of the experiences of a third-party analyst, Steve Christey of MITRE, who supported the SAMATE team during the exposition. The primary task was to evaluate the accuracy of security warnings that were generated by the participating tools.

It is assumed that the reader is already familiar with the goals, organization, and execution of SATE 2008 as documented in "Review of the First Static Analysis Tool Exposition (SATE 2008)" by Vadim Okun, Romain Gaucher, and Paul E. Black [2].

Throughout the text, references to the Common Weakness Enumeration (CWE) are included. Each CWE [3] identifier provides additional information for the given weakness.

Review Procedure

During the review period, six people analyzed 5,899 warnings from a total of 47,925 warnings that were generated by all of the tools. Over a period of several weeks, a MITRE analyst evaluated approximately 500 warnings, or 9% of the warnings that were evaluated. Often, the warnings were annotated with comments that briefly explained the conclusion that was reached. Many of these 500 findings only produced a "not sure" assessment. Note that the exact number is not being reported to discourage readers from improperly inferring a false-positive rate using data that has not been independently verified.

[1] http://samate.nist.gov/index.php/SATE.html

Time Required for Warning Analysis

The MITRE analyst required 1 to 30 minutes to analyze each warning, which was consistent with the experiences of other participants. Ultimately, the effort took approximately 40 hours of labor, or 5 minutes per report, to evaluate the 500 warnings.

If a rate of 5 minutes per warning is typical for an evaluation, then it could require approximately one staff year to analyze 25,000 warnings, or almost two staff years to analyze all of the 47,925 warnings in SATE. This was a problem because only six calendar weeks were scheduled for the evaluation. It must be emphasized that the tools were not used directly, so the manual process likely had several inefficiencies. In addition, it is not necessary to evaluate every warning in order to understand tool performance, which will be elaborated in later sections.

Selection of Warnings to Evaluate

With the large number of warnings generated by the tools, it was not possible for the SATE evaluators to review them all. A later section outlines various approaches for the selection and prioritization of tool warnings. The method for selecting warnings was not structured or formal, although there was an emphasis on the warnings with the highest severity. However, sometimes tools would assign lower severities (e.g., one tool used severity-3 for format string vulnerabilities). Some tools did not assign severities at all, and CodeSonar used a single rank that combined severity and confidence. In scenarios in which a consumer may want to combine two or more tools, this variance in assessment of severity should be considered.

Most of the MITRE analyst's evaluations were conducted on the C code test cases, often focusing on a single source file or certain product functionality. In other cases, warnings were selected based on the individual bug type, especially anything that seemed new or interesting. In yet another method, the results from a single tool would be reviewed. Focusing on a single tool made it easier to learn what types of weaknesses the tool would find and how much detail it used when explaining a warning.

The web interface was used as the primary tool for navigation and analysis. Two search modes from the web GUI were particularly useful: "Find other issues on the same line of code," and "Find other issues with the same weakness type." Manual code review was occasionally performed. Some of the most critical data was the backtrace information that was provided by some tools, in which a single warning listed the call tree and important sources of input. As a result of the limited exchange format, this backtrace information was not always easy to read, so custom routines were developed to provide additional context that simplified analysis.

Some tools provided less contextual information and appeared to have high false-positive rates, and as a result, these results would often be skipped. This lack of contextual information shifted the workload from the tool to the human analyst. This difficulty was most notable with Flawfinder [4]. Note that Flawfinder is an older-generation tool that does not use modern data flow analysis techniques, which can significantly reduce the number of false positives by limiting results to externally controllable inputs. Therefore, Flawfinder's performance is not necessarily indicative of the state of the art.

Definitions and Interpretation of True and False Positives

The central question for much of the analysis was whether an individual warning was correct ("true positive") or incorrect ("false positive"). After the exposition was underway, the SATE evaluators realized that each analyst had different criteria for evaluating correctness. It became necessary to provide clear definitions of true positives and false positives. For example, consider a warning for a buffer overflow in a command line argument to an application that is only intended to be run by the administrator. Even if two analysts determine that the warning is correct – i.e., that a long input could trigger the overflow – one analyst might evaluate the warning as a true positive, and another might treat it as a false positive. The warning's security relevance may be questionable, given the expected operational environment of the application.

Other warnings might be technically correct, but only tangentially related to security even in privileged code, such as code quality or conformance to coding standards. For example, the failure to use symbolic names in security-critical constants (CWE-547[2]) may be important for many people, but this weakness might be disregarded if the goal is to evaluate the current security posture of the application.

Eventually, the evaluator team agreed to guidance for the evaluation of true positives and false positives.

To determine if a bug report is a false positive:

- Assume the tool should have perfect knowledge of all potential control and data flow paths.
- It is a false positive if there are no possible conditions, as reflected in the source code, under which the claimed weakness could lead to a vulnerability.
- Considerations of the software's environment or configuration should not be a factor in the assessment.

Using these criteria, an evaluator would label a warning as a true positive in cases in which the reported weakness was technically correct, even when the issue was of low severity or low utility. As already noted, the criteria were not always followed consistently during the exposition.

In addition to clearly vulnerable code, some unexpected constructs would also be labeled as true positives, such as:

- Buffer overflow in a command-line argument to an unprivileged program
- Off-by-one buffer overflow in which adjacent variables were not used after the overflow occurs

[2] http://cwe.mitre.org/data/definitions/547.html

- Use of a dangerous function (e.g., `strcpy`) if the tool was simply stating that the function itself should be avoided
- Missing default case in switch statement
- Free of a NULL pointer

Since more-consistent labeling of true and false positives only took place after the warning analysis period had begun, not all previously-evaluated results were reexamined, due to the scale of the effort. Consequently, there are important inconsistencies in the data. This is one reason why all analyst-assigned false positives were changed to "unconfirmed" in the final SATE 2008 data.

Role of Severity in Warning Evaluation

The interpretation of severity was sometimes important for assessing whether a result was a true positive or a false positive.

Sometimes when a true positive was assessed, an analyst would reduce the severity of the issue to reflect the common environment under which the program would be executed. For example, a buffer overflow in a command-line argument to an unprivileged program would be reported by a tool as high severity (1 or 2), which could be lowered to 4 or 5, since a buffer overflow could be a problem if the program is invoked from a networked component, through privilege-granting routines such as sudo, or when the user does not have full access to a shell, such as a kiosk or hosted environment. Another example is a bug that can only be triggered by an administrator who provides a malformed configuration file. This practice was only adopted by SATE analysts after the review period had already begun.

A more difficult distinction between low-severity findings and false positives occurred in the area of information leaks. If a numeric process identifier (PID) is included in an error message through syslog, then this is not an issue if syslog is only recording to a log file that is only readable by the administrator. Even if the file is readable by all local users, the PID is already available to unprivileged users through legitimate means, since most UNIX-based systems allow local, unprivileged users to list the processes of other users. In this case, there is no leak. However, if syslog sends data across the network, then the PID is effectively private information, although the severity might be extremely low since this information might only be useful in exploits of other vulnerabilities that require knowledge of a PID. At least one tool reported a logging message that included a file descriptor number, and in other cases, the address of a pointer was leaked. It is difficult to envision an attack in which this knowledge would be useful, so these were generally marked as false positives. With respect to logging, it was assumed that both sniffing attacks and reading of a log file were viable attacks.

Other Factors

Some tools provided a confidence value for each generated warning. This value captured the likelihood that the warning was correct. While the exchange format and web interface captured this information, it was not a major consideration when interpreting the correctness of the reports.

The evaluation of true and false positives could also depend on the analyst's understanding of the application's intended behavior. For example, a tool might report a NULL pointer dereference in a C program in which the developer intentionally dereferenced NULL in order to force an immediate program exit. In a separate well-known example [5] [6], OpenSSL intentionally avoided initializing a certain memory location in order to use it as an additional source of entropy for random number generation; this was flagged as a use of uninitialized data by an automated tool, causing the programmer to "fix" the problem in the Debian GNU/Linux distribution. As a result, the entropy was reduced, and only a small number of possible encryption keys could ever be generated.

Examples of False Positives

Following are some interesting examples of false positives that were reported during SATE. Unfortunately, SATE warning IDs (UIDs) are not available for these examples.

- Some coding constructs were labeled as input validation errors even when validation was performed or not needed.
- Some functions contained potential vulnerabilities if called incorrectly, but all current callers invoked the functions safely.
- In the lighttpd test case, the memory management routines would exit the entire program when `malloc` failed. Sometimes a tool would recognize that these routines could return memory from `malloc`, but the tool would generate a warning for a NULL pointer dereference in their callers.
- Pointer aliasing sometimes caused initialized data to appear to be uninitialized.
- Log information leaks were reported for memory addresses or file descriptor numbers.

Difficulties in Interpretation of Warnings

There were several factors that sometimes made warnings difficult to interpret and evaluate. Since the SATE analysts were knowledgeable about security, developers might also encounter similar problems when handling tool warnings.

Efficiency Challenges in Results Interpretation

Since the design of SATE relied on the database and web GUI instead of the native tool interfaces, analysts did not have access to capabilities that could have helped to interpret results, such as navigation and visualization, or linkage with an Integrated Development Environment (IDE). The backtrace information is essential for interpreting warnings, but the support was limited in the web interface. So, SATE's design probably led to increased labor and more ambiguous results.

The source code browser in the web-based GUI did not always link to the proper location, which required manual lookups. This became especially problematic with respect to callback functions. A single source code page might take several minutes to load, due to the large number of links and images for call trees.

There was significant overhead while learning the underlying implementation of the test cases, which was not necessarily well-documented. For example, it required significant effort to understand how lighttpd's memory management routines worked, but this could not be avoided because they were listed in a large number of results for high-severity issues such as buffer overflows. Note that many of these problems were originally labeled as false positives or "not sure."

Many reports remained ambiguous, even accounting for the lack of direct access to the tools. Sometimes, analysts needed to know more about the test case's operating context than was readily available. For example, one reported issue was a false positive on every platform but Solaris. For Solaris, the correctness of the warning depended on whether the size of a particular data structure could exceed a certain number of bytes on any of various Solaris operating systems and hardware configurations. This required investigation into low-level include files that was more comprehensive than most programmers would ever need to perform in normal operations. Not surprisingly, this report took at least 30 minutes to analyze and only produced a "not sure" answer. Such an analysis would not even be relevant if the tool was assuming an operating system other than Solaris.

Errors and Disagreements in Interpretation

In multiple cases, there was disagreement between the analyst and the tool vendor about whether the report was a true positive or a false positive. This disagreement was not always resolved. Multiple vendors did not have enough time to provide comprehensive feedback.

To evaluate each bug report, the analyst needs to fully understand the warning that has been generated by the tool. In one case, analysts mislabeled several bug reports as false positives because they misinterpreted the tool's description of the weakness that it had found. Use of CWE names could help to reduce the risk of confusion, but CWE does not currently cover all issues that tools report, and some CWE entries have vague descriptions that could lead to misinterpretation and improper or unexpected mappings by the tool vendor.

Other interpretation problems probably stemmed from the lack of access to a tool's native interface and its navigation capabilities. While supplementary tool output was used as much as possible, sometimes the evaluator would believe that the tool was reporting an issue in one part of a code expression, when in fact the tool was reporting an issue in another part of the expression. For example, the analyst might believe that the tool was reporting a NULL pointer dereference for a pointer `p` when in fact the tool was concentrating on `p->field`. In other cases, such as the statement `sprintf(str, "%s%s", var1, var2)`, the analyst might believe that the tool was reporting a buffer overflow through `var1` when in fact it was reporting it through `var2`.

In other cases, the analyst made improper assumptions about which compile-time options were set, causing the analyst to analyze the wrong code within an `#ifdef` block.

In the lighttpd test case, analysis sometimes required investigation of a backtrace of 20 or more code points that did not lead to a clear result. These long flow paths were costly to investigate, likely prone to analyst error, and often produced only “not sure” evaluations. Direct access to the tool environment would simplify navigation, but it might not always help the analyst to mentally build the logic chain to assess the accuracy of the bug report.

Finally, to interpret some warnings, an analyst would require significant expertise of the test case’s environment that might not be readily available, even to the original developer. For example, low-level operating system constructs in include files rarely need to be understood at the application layer, but byte-sizes and alignment can be critical in buffer overflow analysis.

Context-Based Interpretation of Tool Warnings

Analysts may reach different conclusions based on the context in which a warning is evaluated. The following coding constructs exemplify some of the warnings that led to inconsistent evaluations.

Code such as the following was labeled as improper null termination (CWE-170[3]):

```
n = sizeof(dst);
strncpy(dst, src, n);
dst[n-1] = ‘\0’;
```

The tool in question identified the `strncpy` line as being improperly terminated. With respect to this line, the finding was technically correct, since `dst` might not be terminated when the `strncpy` completes. But since the next line ensures that null termination occurs, the warning was treated as a false positive.

Another example is for a coding construct that could be labeled as an assignment instead of comparison (CWE-481):

```
if (res = ParseMessage(a, b)) {
  printf("Command is %s\n", res[0]);
}
```

This construct appears in many programs. It could be treated as a false positive, since it is fairly clear that `ParseMessage()` is expected to return a non-NULL result when it succeeds, so the assignment is intentional. However, some analysts might treat the warning as a true positive with very low severity, since it described an assignment within a conditional.

[3] http://cwe.mitre.org/data/definitions/170.html

In other cases, a solid understanding of the developer's intentions may be considered when evaluating a warning.

Consider the labeling of stylistic preferences that may appear to be coding errors. For example, the logic in the following segment produces an unreachable block, i.e., dead code (CWE-561):

```
state = 1;
if (SpecialCondition) {
  state = 2;
}
if (state == 1) {
  DoThis();
}
else if (state == 2) {
  DoThat();
}
else {
  error("How did I get here?");
}
```

This coding construct could be argued as defensive programming to catch unexpected situations, or to improve readability of the code. As a result, a tool warning might be treated as a false positive. In a similar case, a programmer might intentionally leave an empty block in an "if-then-else" conditional, using the block to contain comments that improve code readability. The presence of an empty block might be flagged as a warning by some tools but treated as a false positive by an analyst.

Incorrectly-Derived True Positives

Both tools and human analysts could declare a result to be a true positive, but for incorrect reasons. This issue arose during feedback with tool vendors.

A simple example of an incorrectly-derived true positive might be:

```
home = "/home/www/l18n/";
input = GetWebParameter("lang");
if (! strcmp(input, "../") {
  die("Bad! ../ in pathname"); }
input = URLdecode(input);
/* ignore the buffer overflow */
sprintf(fname, "%s%s.txt", home, input);
ReadFileAndDumpToUser(fname);
```

A tool might report this problem as a relative path traversal weakness (CWE-22[4]), but its data flow analysis might not consider that there is a sanity check for a "../" sequence that would prevent one common attack against relative path traversal. The core problem arises because the programmer validates the input before decoding it (CWE-179), or alternately, there is

[4] http://cwe.mitre.org/data/definitions/22.html

insufficient input validation (CWE-20) that does not protect against additional variants of path traversal.

While this is an esoteric problem, it might be possible to detect when tools accidentally produce the correct results. The success of such an effort would depend on how well a tool explains how it obtained its findings.

Causes of Naming Differences in Warnings

In addition to the inconsistent interpretations of warnings, some differences could arise when tools used different weakness names, even when they were really describing the same underlying issue. For this reason, there are probably many associations that were missed within SATE. Many tools mapped their warnings to CWE identifiers, but some tools did not. Even the use of CWE did not fully resolve these differences.

The differences in weakness names or descriptions would arise from:

1) Presence of chains. Some vulnerabilities occur when the introduction of one weakness directly leads to another, called a *chain* [7] in CWE. One tool might report the first weakness, while another tool reports the second weakness. These would appear to be different results when the tools are only reporting different aspects of the same problem. This happened in at least one case when one tool reported a failure to check the return value from a `malloc` call, and another tool reported the NULL pointer dereference that would result from that call. The failure to detect chains can cause tool results to appear more diverse than they actually are, which makes it difficult to combine results from multiple tools.
2) Presence of composites. Some vulnerabilities might only occur when several weaknesses are present simultaneously, called a *composite* [7] in CWE. Tools might report different parts of the composite. For example, in a symbolic link following attack (CWE-59), one tool might report poor randomness in file names (CWE-330), whereas another might report a race condition between time-of-check and time-of-use (CWE-367). However, both of these weaknesses are needed to make a symbolic link following attack feasible. For SATE, it is not clear how often tools were reporting different aspects of the same composite.
3) Alternate perspectives. Tools may have different perspectives when generating warnings, producing inconsistent results. For example, one tool might categorize buffer overflows depending on whether they are heap-based or stack-based, while a separate tool might categorize overflows based on root causes such as improper trust in a length parameter, unchecked array indexing, and failure to check the length of the source buffer. A bug report for a heap-based buffer overflow may well be the same as a bug report for an improperly trusted length parameter. The SATE evaluation team explored ways to normalize this data, but there was limited success due to insufficient time.
4) Varying levels of abstraction. The level of abstraction can vary between tools. For example, one tool might produce a bug report for a relative path traversal, whereas another tool might produce a bug report for its parent, path traversal. Since CWE and

other weakness naming schemes are hierarchically organized, this difference probably occurs often. However, the variance in abstraction was not investigated closely within SATE.

5) Approximate CWE mappings. In some cases, there might not be a one-to-one mapping between what the tool reports and the closest-matching CWE identifier. For example, CWE distinguishes between integer overflows and integer underflows, whereas some tools do not make such a distinction. An access of a C++ object after it has been deleted might be mapped to a use after free (CWE-416), which was the closest available CWE identifier in 2008[5].
6) Mapping errors or inconsistencies. Some mapping errors may be due to typos. Others may occur because of the lack of clarity within CWE entries themselves. For example, in Draft 5 (released December 2006), CWE-242 had the name "Dangerous Functions." In Draft 8 (released January 2008), the name was changed to "Use of Inherently Dangerous Function," and in Draft 9 (released April 2008), a new entry CWE-676 was created to handle "Use of Potentially Dangerous Function." A tool might map the use of `strcpy` to CWE-242, when in Draft 9, the more appropriate map would be CWE-676 (since it is possible to use `strcpy` safely, albeit not necessarily recommended). As CWE entries become more stable and more precisely described, these mapping inconsistencies should occur less frequently.

Note that the CWE project is working to resolve these problems, such as developing analytical methods to normalize results. The CWE team has published a working draft [8] that provides more details on challenges in weakness naming and mapping. CWE has matured significantly since the initial SATE analysis, so it may be easier to develop guidance for mapping that also includes normalization of abstraction and possibly perspective.

Chain/Perspective Example - Same code, 4 CWEs

To further demonstrate how perspectives and chains may cause different warnings to be generated, consider the following pseudo-code:

```
InputBuf = GetUntrustedBuffer();
height = GetUntrustedInteger();
width = GetUntrustedInteger();
SZ = 512;
size = height * width;
buf = malloc(size);
memmove(buf, InputBuf, SZ);
```

One chain in this example involves an integer overflow (CWE-190[6]) in the `size` calculation, which leads to less memory being allocated than expected, triggering a heap-based buffer overflow (CWE-122) in the `memmove` operation. A different perspective of the overflow might label this as a failure to properly detect and handle a length parameter inconsistency (CWE-130).

[5] In 2009, a more abstract identifier was created, "Use of a Resource after Expiration or Release" (CWE-672)

[6] http://cwe.mitre.org/data/definitions/190.html

A second chain exists because certain height and width values would cause `malloc` to return NULL due to out-of-memory conditions, but this return value is not checked (CWE-252). As a result, the `memmove` triggers a NULL pointer dereference (CWE-476).

In this example, there are five different CWEs that could be listed, but arguably there is only one bug ("trusting input for height/width"), two bugs based on vectors ("trusting height and trusting width"), a different set of two bugs based on type ("failure to prevent buffer overflow" and "NULL pointer dereference"), or five bugs (each distinct CWE).

The variations in warning names and CWE mappings are likely to skew any comparison between tools in unpredictable ways, since tools can describe the same fundamental problem differently. SATE has provided some evidence that tools do not overlap significantly with respect to the issues they find, but chains and composites may be causing this overlap to be underestimated.

Counting Differences in Warnings

SATE revealed how raw results from tools should not be directly compared. In some cases, two tools would detect the same problem in the same portion of the code, but generate different numbers of warnings. This may have been due to different interpretations of the location of the bug, variations in detection techniques, or an intentional omission of results that had a high likelihood of being false positives. The difference could be radical, in which one tool reported a single bug, and another reported 20 bugs. This was especially the case in utility or API functions that were reachable via multiple vulnerable paths. The counting problem is reflected to some degree in the associations that were recorded between warnings.

Consider the following example code:

```
char * CopyString(char *src) {
  char *dst;
  dst = malloc(512);
  /* ignore NULL dereference */
  strcpy(dst, src);
  return(dst);
}

void HandleInputs() {
  char *p1, *p2;
  p1 = GetUntrustedInput("param1");
  p2 = GetUntrustedInput("param2");
  p1 = CopyString(p1);
  p2 = CopyString(p2);
}
```

One tool might flag a single bug report, the unbounded `strcpy` in the `CopyString function`. A different tool might create two bug reports, one for each invocation of

`CopyString`. This variation is likely to occur depending on whether a tool produces separate bug reports based on where input enters the program (the `GetUntrustedInput` calls) or where the error occurs (the `strcpy` call). A tool that produces different warnings for each separate location of input may generate more results than other tools, which increases the amount of labor needed to evaluate the warnings. Similar variations can occur when there is more than one control flow path that reaches the same erroneous code. Better support for finding and recording associations would minimize the extra cost. Ideally, different tools could produce results that are more consistent for counting.

There were also differences with respect to abstraction that further complicate comparative analysis. A manual review by a software security auditor might list high-level, systemic issues such as "lack of a centralized input validation mechanism." Aspect Security, a SATE participant, often reported warnings at this systemic level. The design-level omission of input validation could translate into numerous XSS or SQL injection errors that would be found by a code-auditing tool, causing the number of warnings to be radically different.

In sum, code analysis tools should not be compared using raw numbers of warnings, unless the warnings are normalized to account for variations in quantity, names, and perspective.

Example Warnings

The following table contains some of the warnings that demonstrate the analytical questions and discrepancies that faced the SATE evaluators. The warnings are identified by their UID in the SATE data set.

UID	Description
26588	Chain: the tool reports a buffer overflow, which does not seem to be possible unless an integer overflow occurs in the call to malloc. The tool may have reported a true positive for the wrong reason, but it is not clear if the integer overflow was considered.
43048, 46909	Chain: one tool reports an unchecked error condition; a separate tool identifies the NULL pointer dereference that occurs when the error condition is not checked.
33348	Chain: an unchecked return value could be flagged as a NULL pointer dereference by others.
43023	The program intentionally checks for the existence of a file for better error-handling. The affected file is expected to be under full control of the administrator, thus low severity.
24929	An information leak is reported, but the leak is only a standard SSL error number.
24443	An information leak in a logging function is reported, but arguably the information is not sensitive.
43012	A "stat after open" weakness is labeled as a false positive, probably due to the perception that the issue is low severity or irrelevant.
43765	The warning is labeled as a false positive, probably due to the perception that the tool's claim of high severity is incorrect. The severity label was incorrect, possibly due to a data conversion error when the vendor provided results in XML.
42958, 42965	Variables appear to be uninitialized, but they are initialized indirectly through dereferencing of other variables.
43308	A variable appears to be indirectly null-terminated. The tool may have thought that

UID	Description
	pointer arithmetic in a loop might lead to a boundary error.
43932, 43933	An issue is classified as a memory leak by the tool, but it might be better described with the more-abstract CWE-404 (Improper Resource Shutdown or Release).
42923	Naming confusion occurs due to abstraction. An issue is labeled as a "leak" but the analyst labels it as a false positive, perhaps believing it is a memory leak. The tool is actually reporting a file descriptor leak.
44026	Separate problems occur in heavily-used macros – is this one bug or many? Also see 33346.
44040	Different input paths generate separate warnings – is this one bug or many? Also see 44042 and 44043.
44071	A NULL pointer dereference is reported, but this code would not be reached because another statement would trigger the dereference first. It is still worth noting since fixing this NULL dereference would expose the weakness in the later statement.
42908	The analyst may have been examining the wrong data/control flow path when evaluating the warning. Also see 43522, 43013, and 43218.
43367, 43439	The analyst may have been examining code that was excluded by the preprocessor, leading to a disputed evaluation.
43992	The root cause of the issue is obscured due to macro expansion. Also see 43999, 43426, 43433, and 43450.

To conduct a more extensive investigation into these types of interesting results, the SATE data could be examined for warnings that were labeled as "not sure," or warnings that were labeled as both true and false positives.

Strengths and Limitations of the SATE Web Interface

There were several important features in the SATE web interface and the underlying database that simplified analysis.

The common XML exchange format made it possible for analysts to view warnings from different tools within a single interface.

Throughout the exposition, various modifications were made to the sorting and searching capabilities to provide important flexibility. Some of the most effective aspects of the interface were:

- View multiple tool results in a single interface
- View comments for a single warning
- Use a "bookmarking" feature for warnings that the analyst will revisit at a later time
- Search by test case
- Search for bug reports within "X" lines of code of the current bug
- Search for bug reports with an uncertain evaluation status, such as a "true positive" assessment from one analyst and a "false positive" from another analyst
- Search by severity
- Search by CWE ID

- Search by tool vendor

The web interface supported different modes for casual browsing versus focused searching. The search for bugs within a certain distance of an existing bug was used heavily. The interface was effective for labeling associations as well as evaluating all relevant bugs once the relevant portion of the code was understood by the analyst.

Some additional improvements to the interface would simplify the analyst's job. Being able to apply the same evaluation to multiple weaknesses at once, instead of entering evaluations one-by-one, would be more efficient when evaluating utility functions or other code that may have many callers. Note that associations effectively provide this functionality, but not all multi-weakness actions would necessarily involve associations. The method for recording associations was also cumbersome.

If the analyst fully understands a function, then it would be useful to conduct a search for any bugs within that function; the closest capability, searching within a specified number of lines of the location in code, was too coarse. It would also be useful to search for any weaknesses within the entire call tree of a selected code point. A search for similar CWEs, or potential chain/composite relationships, would help in finding associations. Richer displays of search results would also be useful. For example, when searching for bugs that do not have a conclusive evaluation, it would be useful to include the comments for each bug report that is listed, along with its status.

The package that was used to display and navigate the source code was powerful and effective, although sometimes it had usability limitations. For example, in large source files with many functions or callers, the web browser's load time would be excessive, sometimes generating a large number of graphic images for call trees. This experience was not shared by all analysts, however, so it might be a reflection of one particular mode of usage. In other cases, the source file was not displayed with proper links to other locations in the code, forcing manual searches for the appropriate line of code. This was most likely a bug in the source-navigation software itself.

For long flow paths, it was sometimes difficult to navigate the source code interface. The individual bug report often contained more detailed information from the tool that displayed the flow paths, but it did not always have the desired details.

The web-based source code navigation feature was not closely integrated with the evaluation interface. The analyst could move from the bug reports to the code, but not the other way around. Closer integration would be extremely powerful, such as the ability to annotate the source code itself with all the relevant warnings. While this capability is provided in the native interface for some tools, it would be useful to be able to view the results of multiple tools in this way.

The ability to save preferences or queries would make navigation more efficient.

Lessons Learned: Considerations for Future SATE Efforts

For future SATE efforts, or for any project that seeks to understand the capabilities of static code analysis tools, the following lessons may be useful.

Use Longer Review Periods

The review period was too brief. During the evaluation stage, the SATE evaluation team received more warnings than expected, so there was not enough time to evaluate all the warnings that were generated. Important factors included the unexpected large number of warnings from the tools, and the amount of time required to analyze each individual warning. In addition, tool vendors did not have enough time to provide detailed feedback or corrections to the original evaluations.

Link Directly with Tools

It is highly likely that using the native interfaces of the tools, instead of static web-based results, would improve the efficiency and accuracy of an analysis. It would also create opportunities to conduct usability studies of the tools. Such an effort would need to account for the overhead of learning the tool combined with licensing costs or restrictions.

Select Warnings More Consistently

Since this was an exploratory investigation and not a formal experiment, the variations across analysts for warning selection did not necessarily pose a problem. However, because the sample was not random, this restricts the ability to conduct certain studies that would have been useful for fully understanding the strengths and limitations of the tools. A later section will identify some approaches for improving warning selection.

Provide Clear Criteria for True Positives and False Positives

As described in a previous section, there was significant variation between analysts in the assessments of true and false positives, especially in the early stages of the exposition. Future efforts should establish guidelines up front, discuss detailed scenarios, and hold regular review sessions to ensure consistency.

Criteria should be established to determine if a “not sure” rating is a satisfactory result. While the usage of “not sure” might be useful for developers who have a need to prioritize fixes, it is not necessarily appropriate for third-party analysts who seek to understand the capabilities of tools, or for consumers who want to have high confidence that software is free of all reported bugs. Chess and West [9] have recommended three types of results: “Obviously exploitable,” “Ambiguous,” and “Obviously secure.” These roughly align with the SATE values of “true positive,” “not sure,” and “false positive.” One participating SATE vendor suggested that a “Don’t care” result could also be used; this concept is effectively captured within the SATE data, in which analysts declared warnings as true positives but reduced the severity to 5.

The criteria for evaluation of true and false positives may need to be adjusted when tools provide confidence values for individual warnings, especially if quantitative analysis is conducted.

Improve Cross-Analyst Coordination

The primary method of communication between analysts was e-mail. It would be useful to develop an environment in which analysts could centrally share derived information as the investigation progresses. This is especially important as analysts learn the design and coding practices of the test case. In some cases, duplicate efforts were not detected until after the fact. For example, two SATE analysts separately performed some research into the memory management routines of lighttpd without being aware of each other's efforts.

Hold Regular Review Sessions

The analyst team was geographically diverse. Much of the interaction was through e-mail. Occasionally teleconferences would be conducted, but these were usually for deciding how to handle an important design question for the exposition. Regularly-scheduled review sessions would help to ensure consistency, detect potential problems sooner, improve cross-analyst coordination, and share derived information.

Conduct Qualitative Assessment of Analyst Disagreements

It might be informative to perform a qualitative analysis of the types of analysis errors or disagreements that occur from humans versus automated tools. A thorough study could compare the determinations from SATE analysts, tool vendors, and developers. It could potentially expose opportunities for improvements in analyst training and in how tools present their warnings. However, such a study would be resource-intensive. Because of the large number of results that were produced, there were very few warnings that had multiple reviewers, so the SATE 2008 data set cannot be used to perform this analysis.

Support Assessment of False Positives and False Negatives

Based on feedback from attendees of various SATE-related talks before publication of this document, there is a strong desire to understand how tools perform with respect to false negatives, as well as false positives. This should be seriously considered in future tool analysis methodologies. As previously described, the SATE 2008 data cannot be used to determine false positive rates, because there were disagreements about the accuracy of some of the assessments. The data cannot be used to estimate false negative rates either, since only 12% of all generated warnings were evaluated. There was some evidence of false negatives that may have been caused by the use of callbacks and pointer aliasing, but this was not examined closely.

If future efforts can address the limitations of SATE 2008 with respect to data completeness, then the calculation of false positive rates might seem to be straightforward. However, due to previously-discussed counting differences between tools, the raw rates might not be comparable. For example, a tool that generates a false positive for each of 20 different inputs would be over-penalized with respect to another tool that combines those inputs into a single warning.

A multiple-tool evaluation may be a mechanism for estimating false negative rates for tools. This could be performed by examining warnings that are confirmed as true positives and determining which tools did not produce an equivalent warning. When a tool does not report the same weakness, it can be treated as a false negative for that tool. Automating the calculation of false negatives could be difficult, however, due to differences in perspective or abstraction, as described elsewhere. Analysts would also need to decide whether a tool should be flagged for a false negative if it does not even claim to find the related type of weakness.

Characterize Tool Overlap

Using associations and other data, one could count how many tools found each individual weakness that was determined to be a true positive. Overlap between tools could be visualized using mechanisms such as pie charts [10], although there are some limitations of this approach [11]. The pie chart could also be used to determine overall coverage of all the tools with respect to a set of weaknesses such as the CWE/SANS Top 25 [12] or the entire CWE itself. The same underlying data could be used to identify capabilities that are unique to particular tools.

For real-world test cases, there are currently several significant limitations to such an approach. As mentioned in other sections, the tool warnings often differ with respect to perspective, abstraction, counting, and mapping inconsistencies. Also, such a study would require that all associations have been identified and all results have been evaluated, which would be labor-intensive. Finally, the test cases are not likely to have every type of weakness of interest, so the pie chart might not include all the weaknesses that each tool is capable of finding.

Note that this usage of pie charts would not cover the false-positive or false-negative rates, which would be important decision factors for many consumers.

Use Iterative Analyses or non-Default Configurations

SATE 2008 used default configurations wherever possible. However, some tools may provide capabilities in which rules could be tailored (e.g. to identify functions that perform correct input validation), or in which the analyst could label a warning as a false positive, which might cause related warnings to be omitted (e.g., if the false positive occurs in a frequently-invoked library function). Thus, analysis could be iterative in nature. Use of such capabilities, when available, might more closely reflect the real-world usage of these tools in day-to-day analysis. However, it could pose special challenges for the interpretation of results, since the set of warnings might change with each iteration.

Tool Capabilities that may Assist Future Evaluations

The following capabilities would be very useful in conducting source code analysis. They are not mutually exclusive. Note that some of these capabilities may already be offered by some tools.

1) Offer separate tools based on two different capabilities: an “analyzer” that generates the results, and a weakness “browser” that can be used to review the results. This would be useful for distributing the interpretation of results to multiple parties, while there is a centralized analyzer that generates the initial warnings. One consideration for this is cost, since third-party analysts and security consultants might wish to use multiple tools.
2) Adopt a common output format. The simple format used by SATE was insufficient for conducting full analysis. As of this writing, the raw output results from tools are typically in XML but use complex representations in which the information for a single tool result may be scattered throughout the XML document.
3) Support an output mode that normalizes which level of abstraction is used for reporting individual results. For example, “SQL Injection” and “XSS” could be used instead of “incorrect input validation” (which may be too general) or “XSS in Cascading Style Sheet definitions” (which may be too specific). In normal operations, the tool could still use any abstraction that it chooses.
4) Provide output modes that normalize the number of reports for each core problem. For example, if a buffer overflow occurs in an API function, all tools could report that problem as a single issue, instead of listing each vulnerable call as a separate result. Tools with similar analytical capabilities (e.g. data flow analysis) might be able to identify such “choke points” in a predictable fashion.
5) Use consistent severity metrics for results. It may be possible to leverage external efforts such as the Common Weakness Scoring System (CWSS), although CWSS is still in the early stages of development [13].
6) Interpret or filter results based on the code’s execution context and environment. This could be an important step for reducing the amount of warnings to review. For example, the severity of results could be reduced automatically for non-setuid programs that read input from an environment variable or configuration file.
7) Improve the explanations of weaknesses. This may have been available in the native interfaces for some tools, but even with extensive backtraces and labeling, it can be difficult to understand how a tool reached a particular conclusion.
8) Expand the support for third-party analysis. Many tools are focused on integration within a development environment. Third-party analysis may have different requirements.

While it may not be easy for vendors to adopt any of these suggestions quickly, it will be difficult to fully understand tool capabilities in a cost-effective manner without industry-wide enhancements in how tool outputs are generated. Ideally, there could be a balance between integration of results across multiple tools, along with the ability to navigate through the results from an individual tool.

Alternate Criteria for Test Case Selection and Analysis

The following criteria might improve how test cases are selected and analyzed.

Test Case Selection Options

Depending on the nature of the tool analysis, future efforts could use different criteria for selecting test cases. These criteria are not mutually exclusive.

1) Select well-understood code. This would reduce the "rampup" time needed to learn the application. For example, lighttpd's centralized memory management capability required some effort to understand, but this knowledge was required to properly evaluate many overflow-related reports by tools.
2) Conduct a differential analysis. One could select open source software with known vulnerabilities that were later fixed, and run the tools against both the earlier version and the fixed version. By focusing on the weaknesses reported in the original version, the analyst could determine if the tool detected those weaknesses in the original version (true positives) and avoided reporting them in the later version (true negatives). However, this can have a limited scale in terms of the number of weaknesses that are analyzed.
3) Develop custom test cases. One benefit of this approach is that the weaknesses are known ahead of time, thus reducing the need for true-positive analysis. The test cases can also be tailored to represent the weaknesses of interest. However, the code can be expensive to develop, and incidental weaknesses could arise due to unanticipated errors in the code. Finally, it can be difficult to represent real-world complexity and variations in programming style.
4) Use weakness seeding. This technique is a hybrid of differential analysis and custom test case development. Existing code could be modified or "seeded" to intentionally introduce weaknesses. The analysis would only focus on the tool's performance with respect to the seeded weaknesses. This approach would preserve any inherent features of complexity in the code and may require less time to develop than custom test cases, but it still might have a limited scale.
5) Select smaller applications. While larger applications might have more complexity and demonstrate the ability of tools against large codebases, the reported weaknesses could be easier to evaluate due to fewer interactions and less complex code paths. However, this would not expose any tool limitations that are related to code size, which is an important consideration for developers of large software systems.
6) Select fewer test cases. This could support the selection of applications with a reasonably large size, but since each test case can differ based on programming style, functionality, and the developer's attention to security, there is a risk that the test cases will only contain a small or biased set of weaknesses.
7) Conduct weakness-focused analyses. Select which weaknesses are the most important, then analyze software that is likely to have them. Configure the tools so that they only report those weaknesses. This may be a viable option when there is a goal to reduce specific weaknesses or if there is limited time.

8) Use previously-developed test cases. The SAMATE project has developed the SAMATE reference dataset [14] containing vulnerable code. Other projects such as OWASP WebGoat [15] are fully-functional applications that have been developed with insecurity in mind. However, it is likely that tool vendors have already tested their products against those test cases, so the tool performance may not be representative. In addition, test case libraries typically contain small chunks of code without the complexity that is found in real-world programs.
9) Include closed-source code. While this might be more difficult to obtain, it is possible that open source code has different properties than closed source.

Selection of Warnings within a Test Case

If tools generate more warnings than the human analysts have time to process, then there are several ways of consistently deciding which warnings to evaluate.

1) Random sampling. This has a few important benefits. It avoids any analyst bias, whether conscious or unconscious. Also, the sample can be scaled to match the amount of labor that is available. Finally, the confidence in the results could be estimated (assuming a well-understood statistical distribution, although this might not be knowable). Depending on the sample size relative to the total number of findings, the sample might not be large enough to provide the desired confidence.
2) Use tool-reported severity. This was the main approach taken by SATE analysts, but it had a few limitations. First, tools varied in how they labeled the severity of an issue, so associations were less likely to be discovered. For example, one tool might label a format string vulnerability as high severity, while another tool would label it as medium severity, causing the latter tool's report to be missed. Integer overflows were given a severity of 2 by one tool and 3 by another. Second, selection based on severity still resulted in so many warnings that the SATE team struggled to complete analysis by the deadline.
3) Focus on selected source files or the most-exposed code. This was periodically undertaken by some analysts for SATE. This introduces an unknown bias, since the selected file may contain code that is only susceptible to a limited set of weaknesses. It may not be feasible in a third-party analysis context, since the code would not be well-understood.
4) Select weakness-focused warnings. Select a limited set of weaknesses that are the most important. Then analyze the warnings related to those weaknesses. This may be useful when analyzing the tools for specific capabilities, but not when trying to understand the general capabilities of those tools. Differences in tool perspectives and abstraction, or the presence of chains and composites, might cause some related warnings to be missed because they do not have the expected weakness type.

Conclusions

While SATE 2008 was not designed to compare the performance of participating tools, it was successful in understanding some of their capabilities in a wide variety of weaknesses. SATE demonstrated that results from multiple tools can be combined into a single database from which further analysis is possible. While the backtrace explanations were extremely useful, the evaluation might have been more efficient and less error-prone by closely integrating with the navigation and visualization capabilities of the tools.

Future studies should plan for the possibility that the tools may generate more warnings than they can evaluate. Consistent criteria for warning selection are needed to address any analytical resource limitations in a way that produces cleaner data. It is important to use a clear definition of true positives and false positives from the beginning, although there may still be subtle difficulties in producing consistent evaluations. Finally, if any comparative analysis is to be performed, warnings will need to be normalized to account for tool-specific differences in how warnings are reported and quantified.

Acknowledgements

SATE benefited greatly from the active involvement of many tool vendors. Thanks to them for their willingness to participate in this effort to understand the state of the art.

At NIST, Paul Black and Vadim Okun were responsive to many issues that were raised during the operation of SATE. Romain Gaucher rapidly fulfilled several requests for features in the web interface, which directly increased the efficiency of warning evaluation.

MITRE's participation in SATE was sponsored by Joe Jarzombek of the US Department of Homeland Security's National Cyber Security Division, under the CWE project. Bob Martin and Bob Schmeichel (both of MITRE) also provided valuable support.

References

1. **NIST.** SAMATE - Software Assurance Metrics And Tool Evaluation. [Online] [Cited: June 15, 2009.] http://samate.nist.gov/index.php/Main_Page.html.
2. **Vadim Okun, Romain Gaucher, and Paul E. Black.** *Review of the First Static Analysis Tool Exposition (SATE 2008).* 2009.
3. Common Weakness Enumeration. [Online] [Cited: June 16, 2009.] http://cwe.mitre.org/.
4. **Wheeler, David.** Flawfinder. [Online] [Cited: June 15, 2009.] http://www.dwheeler.com/flawfinder/.
5. CVE-2008-0166. *CVE Web Site.* [Online] [Cited: June 16, 2009.] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166.

6. **Laurie, Ben.** Vendors Are Bad For Security. [Online] [Cited: June 17, 2009.] http://www.links.org/?p=327.
7. Chains and Composites. [Online] [Cited: June 16, 2009.] http://cwe.mitre.org/data/reports/chains_and_composites.html.
8. **Loveless, Mark.** CWE Mapping Analysis. [Online] [Cited: June 16, 2009.] http://cwe.mitre.org/documents/mapping_analysis/index.html.
9. **Chess, Brian and West, Jacob.** *Secure Programming with Static Analysis.* s.l. : Addison-Wesley, 2007.
10. **Goldsmith, Dave.** Random Thoughts On OWASP. *Matasano Chargen.* [Online] July 6, 2007. [Cited: June 16, 2009.] http://www.matasano.com/log/906/random-thoughts-on-owasp/.
11. **Christey, Steve and Martin, Robert A.** Finger 79/tcp # Christey/Martin: Evolution of the CWE Pie Chart. *Matasano Chargen.* [Online] July 10, 2007. [Cited: June 15, 2009.] http://www.matasano.com/log/912/finger-79tcp-christeymartin-evolution-of-the-cwe-pie-chart/.
12. **Martin, Bob, et al.** 2009 CWE/SANS Top 25 Most Dangerous Programming Errors. [Online] May 27, 2009. [Cited: June 15, 2009.] http://cwe.mitre.org/top25/index.html.
13. CWSS - Common Weakness Scoring System. [Online] [Cited: June 15, 2009.] http://cwe.mitre.org/cwss/index.html.
14. **NIST.** SAMATE Reference Dataset. [Online] [Cited: June 17, 2009.] http://samate.nist.gov/SRD/.
15. **OWASP.** OWASP WebGoat Project. [Online] [Cited: June 17, 2009.] http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.