Building a Test Suite for Web Application Scanners

Elizabeth Fong Romain Gaucher Vadim Okun Paul E. Black
National Institute of Standards and Technology
Gaithersburg, MD 20899-8970
{ efong,romain.gaucher,vadim.okun,paul.black}@nist.gov

Eric Dalci
Cigital, Inc
Dulles, VA 20166
edalci@cigital.com

Abstract

This paper describes the design of a test suite for thorough evaluation of web application scanners. Web application scanners are automated, black-box testing tools that examine web applications for security vulnerabilities. For several common vulnerability types, we classify defense mechanisms that can be implemented to prevent corresponding attacks. We combine the defense mechanisms into "levels of defense" of increasing strength. This approach allows us to develop an extensive test suite that can be easily configured to switch on and off vulnerability types and select a level of defense. We evaluate the test suite experimentally using several web application scanners, both open-source and proprietary. The experiments suggest that the test suite is effective at distinguishing the tools based on their vulnerability detection rate; in addition, its use can suggest areas for tool improvement.

Keywords: Black box testing; software assurance; software security; web application; web application scanners; vulnerability.

Disclaimer: Any commercial product mentioned is for information only; it does not imply recommendation or endorsement by NIST nor does it imply that the products mentioned are necessarily the best available for the purpose.

1. Motivation

New security vulnerabilities are discovered every day in today's system, networking, and application software. In the recent years, web applications have become primary targets of cyber-attacks. Analysis of the National Vulnerability Database (NVD) [15] maintained by the National Institute of Standards and Technology (NIST) shows the rapid increase of vulnerabilities that occur mostly in web-based applications (Cross-Site Scripting (XSS), SQL Injection, and File Inclusion) as percent of the total vulnerabilities. This is shown in Figure 1 (updated from [10]).

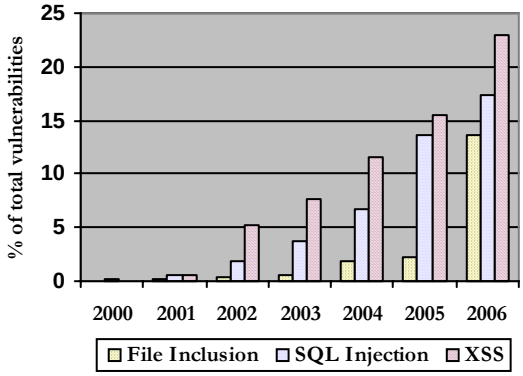

Figure 1: File Inclusion, SQL injection, and XSS as percent of total NVD vulnerabilities (as of January, 2007)

Web application security is a difficult task because these applications are, by definition, exposed to the general public, including malicious users. Additionally, input to web applications comes through HTTP requests. Correctly processing this input is difficult. According to the OWASP Top 10, the incorrect or missing input validation is the most frequent vulnerability type in web applications [17].

Network firewalls, network vulnerability scanners, and the use of Secure Socket Layer (SSL) do not, by themselves, make a web site secure [11]. The Gartner

Group estimates that over 70% of attacks against a company's web site or web application come at the application level, not the network or system layer [21].

One type of tools being employed to address these application-level vulnerabilities is web application scanners [10]. Briefly, web application scanners are automated, black-box testing tools that examine web applications for security vulnerabilities.

Web application scanners have reached a certain level of maturity and are becoming widespread; they find a myriad of vulnerabilities in web applications. Our goal, as part of the NIST Software Assurance Metrics and Tool Evaluation (SAMATE) project [23], is to enable thorough testing of web application scanners. This will help tool users understand tool capabilities and stimulate tool improvement.

1.1. Definitions

Often, different terms are used to refer to the same concept in security literature. Different authors may use the same term to refer to different concepts. For clarity we give our definitions.

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 [16]. In our model the source of any failure is a latent vulnerability. If there is a failure, there must have been vulnerability. Vulnerability is the result of one or more weaknesses in requirements, design, implementation, or operation.

An *exploit* is a technique that takes advantage of a vulnerability to cause a failure. An *attack* is a specific application of an exploit [5]. In other words, an attack is an action (or sequence of actions) that takes advantage of vulnerability.

1.2. Web Application Scanners

A *web application scanner* is an automated program that examines web applications for security vulnerabilities [23]. In addition to searching for web application specific vulnerabilities, the tools also look for evidence of software coding errors, such as unchecked input strings and buffer overflows.

There are many web application scanners available today. Some commercial web application scanners are AppScan [26], WebInspect [24], Hailstorm [6], Acunetix WVS [2]. Some open source web application scanners, such as Paros [7] and Pantera [18], are also popular.

A web application scanner explores an application by crawling through its web pages and performs penetration testing – an active analysis of a web application by attacking. This involves generation of probing inputs and subsequent evaluation of application's response. Web application scanners perform different types of attack. For instance one type of attack, called *fuzzing*, is submitting random inputs of various sizes.

Web application scanners have their strengths and limitations. Since a web application scanner cannot examine source code, it is unlikely to detect specialized vulnerabilities such as back doors. However, it is well suited for detecting input validation problems. Additionally, client-side code (JavaScript, etc.) is available to the web application scanner and can provide important information about the inner workings of a web application.

While web application scanners can find many vulnerability instances, they alone cannot provide evidence that an application is secure. Web application scanners are applied late in the software development life cycle. Security must be designed and built in. Different types of tools and best practices must be applied throughout the development life cycle [14].

1.3. Testing Web Application Scanners

There are several benchmarks, with vulnerabilities of different types, which can be used for evaluation of web application scanners. Foundstone has a series of "Hackme" web applications written in different languages [12]. OWASP SiteGenerator Project [19] enables the user to create web pages with vulnerabilities and test them against a web scanner. OWASP has also produced the WebGoat Project [20] which embeds vulnerabilities in its web application.

A test suite must be useful for differentiating web application scanners based on their vulnerability detection capabilities. An appropriate choice of vulnerability types, while very important, is not sufficient for such a test suite. Vulnerabilities within one type differ significantly in terms of difficulty of exploiting them and types of attacks that are effective against them. A web application scanner may be able to find one SQL injection vulnerability, but fail to detect another. The reason is that web application

developers implement different defense measures that make attacks more difficult.

Therefore, for each type of vulnerability, a test suite should attempt to implement multiple instances ranging from easily exploitable (and thus easily detectable by web application scanners) to the unbreakable, that is, tests for false alarms. This range includes the vulnerabilities hidden behind a series of defense walls. None of the benchmarks cited earlier follow this path in their implementation.

There are many different defenses. We structure and organize them as follows. For several common vulnerability types, we classify the defense mechanisms that can be implemented to prevent various attacks. We combine the defense mechanisms into levels of defense of increasing strength. This allows us to develop a test suite that can be easily configured to select different levels of defense.
We explain the defense mechanisms and levels of defense, and the related concept of attack, in Section 2. We describe the test suite in Section 3. We present the results of experimental evaluation of the test suite in Section 4. Our conclusions and plans for future work are in Section 5.

2. Interplay between Vulnerabilities, Attacks and Defense Mechanisms

This section defines the most common vulnerability types, describes attacks performed by malicious users and web application scanners, and details the defense mechanisms that prevent some attacks and make other attacks more difficult. Finally, it combines the defense mechanisms into levels of defense of increasing strength.

2.1. Vulnerability Types

Researchers and practitioners identified different types of web application vulnerabilities [17, 25]. Based on our analysis of vulnerability reports in the NVD (Figure 1), as well as the analysis in [8], the most common web application vulnerabilities are:

- Cross-Site Scripting (XSS) - A web application accepts user input (such as client-side scripts and hyperlinks to an attacker's site) and displays it within its generated web pages without proper validation.
- SQL Injection - Unvalidated input is used in an argument to a function that calls an SQL query.
- File Inclusion - Unvalidated input is used in an argument to file or stream functions.

2.2. Attacks

The following is an example of an attack that takes advantage of SQL Injection vulnerability. Assume an application contains an embedded SQL query that retrieves user name for an integer value of the input variable *userid:*

SELECT name FROM users WHERE userid = value(*userid*);

Here, "value(userid)" is a pseudo-code for passing the content of the variable userid into the SQL query. If *userid* comes from user input (e.g., GET or POST variable) without filtering, a malicious user can inject a non integer value which contains SQL code, for example:
1; UPDATE users SET password = 'foo' WHERE name LIKE '%admin%'

The resulting SQL query replaces the administrator password.

Different attacks exploit different vulnerabilities. For a particular defense, an attack A may fail, while its variation, attack B, may succeed.

In order to clearly show the diversity of attack variants, we give several injection strings that can be used to probe for Cross-Site Scripting vulnerabilities.

- <script>alert('XSS');</script>
-
-
- </a style="foo:expression(alert('xss'))">

When an attacker looks for XSS vulnerability in a web application, he typically tries different variants of attack in order to bypass the defenses used by the target web application. Many more XSS attacks are described in [13].

The CAPEC Project describes over 100 attack patterns [9] with their associated mitigation techniques. An *attack pattern* [5] is a general framework for carrying out a particular type of attack, such as a method for exploiting a buffer overflow or an interposition attack that leverages certain kinds of architectural weaknesses.

2.3. Defense Mechanisms

Developers must defend against attacks. For example, to prevent the SQL injection attack described in Section 2.2, the developer can use one of the relevant defense mechanisms described in this section.

We classified common defense mechanisms that can be used to make various attacks more difficult to succeed. The following list presents, with examples, the defense mechanisms implemented in our test suite.

- **Typecasting** - convert the input string to specific type, such as integer, Boolean, double.
  Cast to integer transforms input value "8<script>" into the integer 8

- **Meta-character replacement** - encode characters from a blacklist.
  "<" is replaced with "<" in HTML documents. For XSS, replace these characters: ‘, “, <, >, &, %, #, (,)

- **Restricted input range** - restrict the range of integers, the type of an entry (only alphanumeric), length of a string, etc.
  For HTML injection, use a regular expression such as [a-zA-Z0-9_]+ to restrict the input to alphanumeric characters and the underscore character. The developer can either ignore the whole string or remove all invalid characters. For SQL queries use a data binding such as for prepared statements.

- **Restricted user management** - use a restricted account for performing data manipulation, SQL queries, etc.
  If user is not logged in, use a read-only SQL account that only allows SELECT and EXECUTE.

- **Use of stronger function** - use a stronger function for performing a secure action.
  Use SHA-256 instead of SHA-1 or MD5, salt the passwords, HttpOnly in cookies…

- **Character encoding handling** - canonicalize resource names and other input that may contain encoded characters.
  Determine whether an input string contains encoded characters that may be interpreted as malicious content. Always convert these encoded characters into a “standard” representation before filtering.

- **Information hiding** – do not give internal information such as errors, Session ID, etc. to the user

2.4. Levels of Defense

Having different levels of defense for the application’s core functions allows the application to have many instances of different vulnerability types; the higher the level, the harder it is to break the application. For example, filters are a defense mechanism commonly used for input and output validation. The simplest filters may prevent only the crudest attacks, while more comprehensive filters are very hard to bypass.

The following list presents the levels of defense implemented in our test suite for the three selected vulnerabilities. Each level includes the mechanisms of the previous levels. Note that level 3 is not guaranteed to be unbreakable.

Cross-Site Scripting

Level 0. No input filtering.
Level 1. Level 0 + Typecasting
Level 2. Level 1 + Meta-character replacement
  Use PHP function htmlentities to escape all special HTML characters and the equivalent ones for other languages
Level 3. Level 2 + Use a special function which checks for possible nested JavaScript
Level 4. Level 3 + Probing and decoding the input string charset

SQL Injection

Level 0. No filtering of SQL query parameters
Level 1. Level 0 + Information hiding
  Hide the MySQL errors
Level 2. Level 1 + Typecasting
Level 3. Level 2 + Meta-character replacement
  Escape potential MySQL characters: \x00, \n, \r, \, ', " and \x1a.
Level 4. Level 3 + Restrict the SQL user rights
Level 5. Level 4 + Using prepared statements

File Inclusion

Level 0. Include input file name concatenated with '.inc'
Level 1. Level 0 + Test that file exists on the server (prevents inclusion of remote files)
Level 2. Level 1 + Meta-character replacement
Check that file name does not contain special characters, such as /etc/..., /.../..., so file is restricted to a certain directory.
Level 3. Level 2 + Test that file is in the Apache DOCUMENT_ROOT

Such ordering may not be possible for some other vulnerability types, such as session management problems and weak hash functions. There, the level of defense corresponds to the level of security in the configuration, for example, the function used to hash the session ID.

2.5. An Illustrative Example: File Inclusion

There are many variants of attack for file inclusion vulnerability type: from server-side code execution to content spoofing.

We describe a vulnerability involving the PHP include function. Conceptually, a PHP page gets a portion of a file name (e.g., *file1* or */dir1/file1* or even *http://site.com/dir/file1*) without a file extension via an input parameter (POST or GET), appends an extension ".inc" to it, and then evaluates the file.

The goal of the attacker is to supply a file of his choosing.

For level of defense 0, there is no input validation. The attacker can create a malicious script, e.g., badfile.inc, on his server and pass an appropriate portion of its URL, e.g., http://badsite.com/badfile, as an input parameter. The server will execute the malicious script.

For level of defense 1, an attempt to include a remote file fails. However, if the attacker has an account on the same server, he can upload a malicious file on the server and pass an appropriate portion of the file pathname, e.g., /users/eve/badfile, as an input parameter. Again, the server will execute the malicious script.

For level of defense 2, path manipulation is prevented. In addition, for level of defense 3, only files in the Apache document root can be included. Therefore, the above attacks fail.

3. The Test Suite

The test suite is an imitation of an online banking application. A user can create an account with fake social security number and other information. He can also search the website and perform an imitation of money transfers between accounts.

The application contains the following vulnerabilities:

- Cross-Site Scripting
- SQL Injection/ Blind SQL Injection
- File Inclusion
- Cookies poisoning
- Sessions Management problems
- Weak hash function
- Cross-Site Request Forgeries

For testing in the static configuration mode, the user selects all, or only one, vulnerability type. In the dynamic configuration mode, the tester reads a configuration file and then the test suite is dynamically modified. The dynamic configuration allows the user to interactively change the level of defense and the vulnerability type. The following configuration modes are available:

- The interactivity of the website
- The type of vulnerability present in the application (all or only a single vulnerability)
- The level of defense
- Whether the login page is bypassed by the application itself (*auto-login*).
- Whether the application uses Ajax.

3.1. Test Suite Environment

Since the web is rich in the use of technologies, the test suite includes many technologies commonly used in the modern web applications. The test suite uses PHP, MySQL, HTML, CSS, JavaScript, and Ajax. According to [1], PHP is the most common server-side scripting language. We chose these technologies because they are the most commonly used technologies for developing web applications.

Why did we include Ajax? Asynchronous JavaScript and XML (Ajax) [3] is a development technique utilizing the combination of JavaScript, XML, XHTML, DOM, and XMLHTTPRequest for creating interactive web applications. Basically, the XMLHTTPRequest object is used in the JavaScript code to perform asynchronous calls to the server. There

are several reasons for including this technology in our test suite.

First, many "Web 2.0" developers feel secure and do not perform enough input validation on the remote scripts when the calls are hidden. However, nothing is really hidden because everybody can access the JavaScript source code.

Second, Ajax affected the ways in which people are using the Internet, increasing interactivity and, unfortunately, the number of vulnerabilities. We are in the early days of Ajax worms [4].

Third, Ajax-based applications are harder for tools to analyze than classical web applications because the website crawlers (spiders) have to parse the JavaScript code in order to retrieve the server-side script names. In addition, retrieving the parameter names may require execution of the JavaScript code. Therefore, use of Ajax represents a challenging test for web application scanners.

4. Experiments

We ran experiments in order to find out whether the test suite described in Section 3 is useful for differentiating tools based on detection capabilities. For the experiments, we limited the tests to the top three of OWASP Top Ten 2007 vulnerabilities [17]:

- Cross-Site Scripting
- SQL Injection / Blind SQL Injection
- File Inclusion

We used 4 commercial and open source web application scanners to evaluate the test suite. We designate them Tool A, B, C, and D.

Since some of the tools do not support password authentication, we always configured the test application to use auto-login. Additionally, we configured it to use Ajax.

4.1. Test Procedure

The test procedure consists of the following steps:

1. Clean and initialize the test database.
2. Configure the test application by selecting a specific vulnerability type and selecting a level of defense.
3. Run a selected web application scanner to evaluate the test suite.
4. Count and classify vulnerabilities in the tool output from the test run.

We repeated these steps for every web application scanner, selecting each individual vulnerability type and testing with every level of defense we implemented. We performed a total of 4 * 3 * 4 test runs.

4.2. Results and Analysis

The results of the test runs are shown in Table 1 in the Appendix. The tools did not detect any vulnerabilities at level 2 or above, so we only present the data for levels 0 and 1. The table contains the following data:

- Total number of vulnerabilities seeded in the test suite.
- Number of detections, i.e., true vulnerabilities reported by the tool.
- Number of false positives. A false positive is a report of a vulnerability instance by a tool where no vulnerability is present.

Web application scanners usually report many attacks for each vulnerability instance. In the experiments, we count only unique vulnerability instances.

There was only one case where an instance of vulnerability was misclassified by a tool.

For each level of defense, there are 21 seeded vulnerability instances: 8 XSS, 2 file inclusion, and 11 SQL injection vulnerabilities.

In Figures 2 and 3, we present the following metrics for the tools:

- Detection rate – the number of vulnerabilities detected by the tool divided by the total number of seeded vulnerabilities.
- False positive rate – the number of false positives divided by the sum of the number of false positives and the number of detected vulnerabilities.

As shown in Fig. 2, there is a noticeable decrease in the tools' detection ability as level of defense increases. This suggests that, first, the levels of defense indeed have increasing strength and, second, the test suite is effective at distinguishing tools based on their vulnerability detection rate across different levels of defense.

Fig. 3 shows false positive rates for the tools. Tool A had no false positives. Tool C had very high false positive rate. With an increase in level of defense, false positive rate increased for tools B and C, but it decreased for tool D.

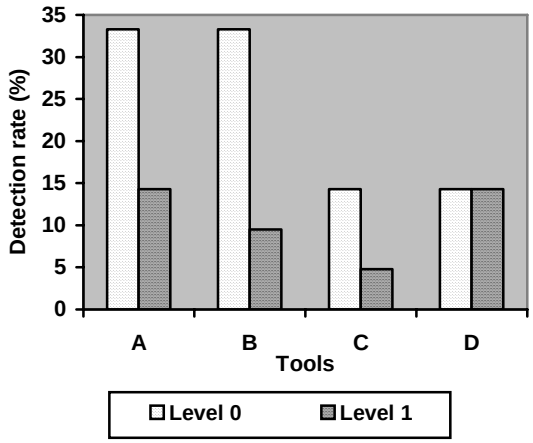

Figure 2: Detection rates for different levels of defense

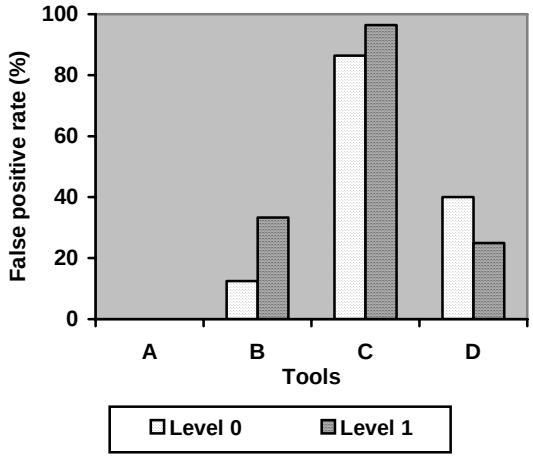

Figure 3: False positive rates for different levels of defense

5. Conclusion and Future Work

We described the design of a test suite for web application scanners. The design is based on a novel idea of combining different defense mechanisms into levels of defense of increasing strength. Our experiments suggest that the test suite is effective for distinguishing tools based on their vulnerability detection rates.

Tools in the experiments were unable to detect any vulnerability at level 2 or above. These levels use strong defense mechanisms, so they are difficult to overcome even for a sophisticated human attacker.

Our plans for future work include developing several other test suites, using different web technologies, for evaluation of web application scanners. We also plan to define more levels of defense to enable more fine grained evaluation of web applications.

6. Acknowledgement

We thank Will Guthrie for many helpful suggestions on this paper, Stephano Di Paola and Anurag Agarwal for their technical review.

References

[1] 4th language in the TPCI, March 2007 http://www.tiobe.com/tpci.htm
[2] Acunetix Web Vulnerability Scanner, http://www.acunetix.com/
[3] Ajax Technologies, http://adaptivepath.com/publications/essays/archives/000385.php
[4] Ajax Worms, http://www.whitehatsec.com/downloads/WHXSSThreats.pdf
[5] Sean Barnum, Amit Sethi, Attack Pattern Glossary, in Build Security In. https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack/590.pdf
[6] Cenzic Hailstorm http://www.cenzic.com/products_services/cenzic_hailstorm.php
[7] Chinotec Technology Company, Paros for Web Application Security Assessment, http://parosproxy.org/index.shtml
[8] Steve Christey, "Vulnerability Type Distributions in CVE," http://cwe.mitre.org/documents/vuln-trends.html, Oct. 2006
[9] Common Attack Pattern Enumeration and Classification (CAPEC) http://capec.mitre.org/
[10] E. Fong and V. Okun, “Web Application Scanners: Definitions and Functions,” in Proceedings of HICSS-40 Conference, Jan 3-6, 2007, Hawaii, USA.
[11] Jeremiah Grossman, The Five Myths of Web Application Security, WhiteHat Security, Inc, 2005.
[12] Shanit Gupta, Foundstone Hacme Bank v. 2.0 Software Security Training Application, April 2006, http://www.foundstone.com/resources/whitepapers/hacmebank_userguide2.pdf

[13] Robert Hansen, Cross Site Scripting Cheating Sheet, http://ha.ckers.org/xss.html
[14] G. McGraw, "Software Security: Building Security In", Addison-Wesley Software Security Series, 2006
[15] National Vulnerability Database (NVD), http://nvd.nist.gov/
[16] National Institute of Standards and Technology (NIST), “Engineering Principles for Information Technology Security (A Baseline for Achieving Security)”, NIST SP 800-27, Revision A, June 2004, http://csrc.nist.gov/publications/nistpubs/
[17] OWASP, Top Ten Project, http://www.owasp.org/index.php/OWASP_Top_Ten_Project
[18] OWASP, Pantera Web Assessment Studio Project, http://www.owasp.org/index.php/Category:OWASP_Pantera_Web_Assessment_Studio_Project
[19] OWASP Site Generator Project, http://www.owasp.org/index.php/Owasp_SiteGenerator
[20] OWASP, WebGoat Project, http://www.owasp.org/software/webgoat.html.
[21] Prescatore, John, Gartner, quoted in Computerworld, Feb 25, 2005. http://www.computerworld.com/printhis/2005/0,4814,99981,00.html
[22] SAMATE Reference Dataset, http://samate.nist.gov/SRD/
[23] SAMATE project Web Application Scanners, http://samate.nist.gov/index.php/Web_Application_Vulnerability_Scanners
[24] SpiDynamics, WebInspect http://www.spidynamics.com/products/webinspect/index.html
[25] Web Application Security Consortium, WASC, “Threat Classification,” http://www.webappsec.org/projects/threat/
[26] Watchfire, AppScan http://www.watchfire.com/products/appscan/default.aspx

Appendix

Table 1. Number of detections and false positives for three vulnerability types

Vuln. type	Def. level	Tool	Detections	False pos.	Total vuln.
XSS	0	A	2	0	8
		B	1	0	
		C	1	5	
		D	1	1	
	1	A	2	0	8
		B	1	0	
		C	1	5	
		D	1	0	
File incl.	0	A	1	0	2
		B	1	0	
		C	1	5	
		D	1	0	
	1	A	1	0	2
		B	0	0	
		C	0	5	
		D	1	0	
SQL Inj.	0	A	4	0	11
		B	5	1	
		C	1	9	
		D	1	1	
	1	A	0	0	11
		B	1	1	
		C	0	17	
		D	1	1	