Web Application Scanners: Definitions and Functions

Elizabeth Fong and Vadim Okun
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8970
{efong,vadim.okun}@nist.gov

Abstract

There are many commercial software security assurance tools that claim to detect and prevent vulnerabilities in application software. However, a closer look at the tools often leaves one wondering which tools find what vulnerabilities. This paper identifies a taxonomy of software security assurance tools and defines one type of tool: web application scanner, i.e., an automated program that examines web applications for security vulnerabilities. We describe the types of functions that are generally found in a web application scanner and how to test it.

Keywords: Software assurance; software security; software security assurance tool; web application; 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. Introduction and motivation

New security vulnerabilities are discovered every day in commonly used applications. In the recent years, web applications have become primary targets of attacks. The National Vulnerability Database (NVD) [14] maintained by the National Institute of Standards and Technology (NIST) has over 18,500 vulnerabilities (as of August 18, 2006). These include 2,757 buffer overflow, 2,147 cross-site scripting (XSS), and 1,600 SQL injection vulnerabilities. XSS and SQL injection vulnerabilities occur mostly in web-based applications.

Figure 1 shows the percentages of the total vulnerabilities reported in the NVD represented by cross-site scripting and SQL injection vulnerabilities. The NVD contains no reports for XSS and SQL injection vulnerabilities prior to year 2000. The share of these vulnerabilities is large and rapidly growing. On the other hand, the share of the buffer overflows, a widely studied security weakness, has not increased in the last several years.

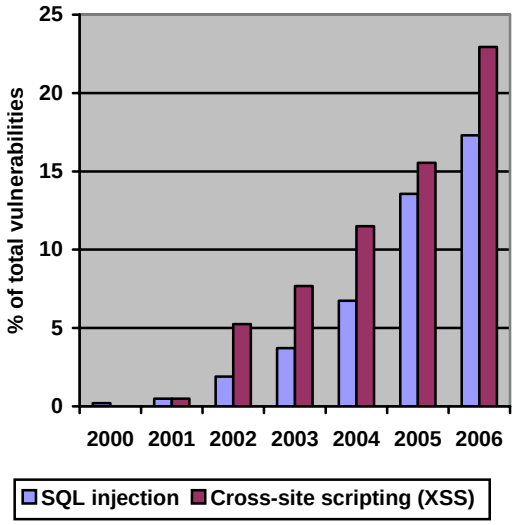

Figure 1. SQL injection and cross-site scripting as percent of total vulnerabilities (as of August 18, 2006)

Web application security is difficult because these applications are, by definition, exposed to the general public, including malicious users. Additionally, input to web applications comes from within HTTP requests. Correctly processing this input is difficult. The incorrect or missing input validation causes most vulnerabilities in web applications.

Network firewalls, network vulnerability scanners, and the use of Secure Socket Layer (SSL) do not make a web site secure [7]. The Gartner Group estimates that over 70% of attacks against a company's web site or web application come at the application layer, not the network or system layer [22].

Web application scanners help reduce the number of vulnerabilities in web applications. Briefly, web application scanners crawl through a web application's pages and search the application for vulnerabilities by simulating attacks on it.

While web application scanners can find many vulnerabilities, 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 [11].

Currently, there is no agreement about what a web application scanner is. To enable objective comparison of different tools, the required functionality of web application scanner must be clearly identified.

We define "web application scanner" and present some vulnerabilities that this tool class should detect. This work is a part of the NIST SAMATE project.

1.1. The SAMATE project

The Software Assurance Metrics and Tool Evaluation (SAMATE) [23] project intends to provide a measure of confidence in the software tools used for software assurance. Part of the SAMATE project is the identification and measurement of software security assurance tools, including web application scanners.

When we have chosen a particular class of tools to work on, we begin by writing a specification. The specification typically consists of an informal list of features, and then more formally worded requirements for features, both mandatory and optional. For each tool class, we recruit a focus group to review and advise on specifications. We also develop a test plan and test sets to check that the tool is indeed capable of satisfying a set of mandatory requirements.

Currently, we are developing a specification and test plan for source code analyzers. We also plan to develop a specification for web application scanners.

1.2. 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.

Software assurance [13] is the planned and systematic set of activities that ensures that software processes and products conform to requirements, standards and procedures in order to help achieve:

- Trustworthiness – no exploitable vulnerabilities exist either of malicious or unintended origin, and
- Predictable execution – justifiable confidence that software, when executed, functions as intended.

In general, a *software security assurance (SSA) tool* is an automated piece of software that detects or prevents security weaknesses and vulnerabilities.

Weaknesses in requirements, design, implementation, or operation may have either direct or indirect impact on security. In what follows, we use the terms "weakness" and "security weakness" interchangeably.

A weakness may result in a *vulnerability*, that is, a possibility of harming the system. A weakness may be the lack of program instructions, for example, lack of a check for buffer size. Since a weakness may or may not result in a vulnerability, we use the term "weakness" instead of "flaw" or "defect". Often, vulnerability is caused by a combination of weaknesses.

A *false positive* is a situation where a tool reports correct behavior as vulnerability.

To accurately determine how well a tool checks for weaknesses, one must begin with a taxonomy of weaknesses. Several security weakness classification schemes have been proposed [1,2,10,21,28,8]. The latest attempt at unifying the schemes is the Common Weakness Enumeration (CWE) [4].

1.3. A taxonomy of SSA tool classes

As the first step in identification of SSA tools, we need a taxonomy, or classification, of SSA tools and techniques in order to prioritize our effort.

We started by asking what classes of tools are currently used to identify potential vulnerabilities in software. We then asked what capabilities a tool should have to be placed into a particular class of tools. A taxonomy, proposed in [24], is organized around four facets: software development life cycle phase (from requirements to operation), automation level (from manual to fully automated), approach (preclude, detect, mitigate, react), and viewpoint (external vs. internal). The classification of SSA tools is based on [3,5,9].

2. What is a web application?

The Web Application Security Consortium (WASC) [31] defines a *web application* as "a software application, executed by a web server, which responds to dynamic web page requests over HTTP."

A web application is comprised of a collection of scripts, which reside on a web server and interact with databases or other sources of dynamic content. Using the infrastructure of the Internet, web applications allow service providers and clients to share and manipulate information in a platform-independent manner. For a good introduction to web application from the penetration tester's perspective, see [12].

The technologies used to build web applications include PHP, Active Server Pages (ASP), Perl, Common Gateway Interface (CGI), Java Server Pages (JSP), JavaScript, VBScript, etc. Some of the broad categories of web application technologies are communication protocols, formats, server-side and client-side scripting languages, browser plug-ins, and web server API.

A web application has a distributed n-tiered architecture. Typically, there is a client (web browser), a web server, an application server (or several application servers), and a persistence (database) server. Figure 2 presents a simplified view of a web application. There may be a firewall between web client and web server.

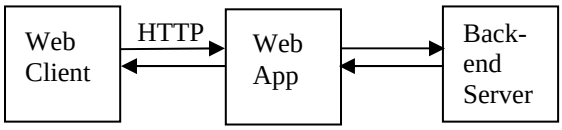

Figure 2. Environment for Web Application

2.1. Sources of vulnerabilities in web applications

Web applications typically interact with the user via FORM (buttons, text boxes, etc.) elements and GET or POST variables. The incorrect processing of data elements within the HTTP requests causes most critical vulnerabilities in the web applications. While SSL ensures secure data transfer, it does not prevent these vulnerabilities because it transmits HTTP requests without scrutiny.

Web applications are a gateway to databases that hold critical application data and assets. Some of the main threats to the database server tier include SQL injection, unauthorized server access and password cracking. Most SQL injection vulnerabilities result from poor input validation.

Most web applications store sensitive information in databases or on a file system. Developers often make mistakes in the use of cryptographic techniques to protect this information.

Since HTTP is a stateless protocol, web applications use separate mechanisms to maintain session state. A session is a series of interactions between user and web application during a single visit to the web site. Typically, session management is done through the use of a pseudo-unique string called Session ID, which gets transmitted to the web server with every request. Most web scripting languages support sessions via GET variables and/or cookies. If an attacker can guess or steal a session ID, he can manipulate another user's session.

We provide a list of vulnerabilities in Section 4.1.

3. What is a web application scanner?

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

Web application scanner explores an application by crawling through its web pages and performs penetration testing - an active analysis of a web application by simulating attacks on it. This involves generation of malicious inputs and subsequent evaluation of application's response. Web application scanner performs different types of attack. A generally useful attack, called *fuzzing*, is submitting random inputs of various sizes to the application.

Penetration testing is a black-box testing approach. The limitation of this approach is its inability to examine source code, thus it is unlikely to detect such vulnerabilities as back doors. However, it is well suited for detecting input validation problems. Additionally, client-side code (JavaScript, etc.) is available to the penetration tester and can provide important information about the inner workings of a Web application.

Some instances of commercial web application scanners are listed below. This list is obtained from references [5,25,6] and web sites.

- AppScan [29]
- WebKing [20]
- WebInspect [26]
- NTOspider [16]

3.1. Other web application security tool types

We contrast web application scanner with some other approaches and point out their differences.

A *web application firewall*, sometimes called wrapper, is a tool that examines HTTP requests and responses for application specific vulnerabilities. It is used primarily during system operation phase, whereas

web application scanners are used primarily during testing phase. Also, web application scanner performs active detection by simulating attacks, whereas web application firewall mitigates vulnerabilities.
Although web application firewall can be used to detect vulnerabilities by examining saved attack information, the detection is passive. That is, nothing is detected until and unless an attack triggers a response indicating a vulnerability.

Source code analysis is a white-box testing approach that scans the application source code for security weaknesses. Source code scanners are primarily used during the implementation phase of the software development life cycle. Some source code scanners can detect web application specific vulnerabilities.

Using a framework is another approach. Frameworks assist coders and security analysts in the process of testing their Web applications, either by providing an interface that exposes the internals of the HTTP traffic, or by helping create automated tests for custom Web applications.

No single approach is sufficient to make web applications secure: different types of tools must be used at different stages of the development life cycle, starting with the early phases. Below are some instances of web security tools which are not web application scanners.

- NC2000 [15] is an application gateway. It is a physical box that is placed in front of a web server and examines the traffic to/from the web application.
- Nessus [27] is an open source scanner that supports a plugin architecture allowing users to develop security checks with the NASL (Nessus Attack Scripting Language).
- WebScarab [19] is a framework for analyzing applications that communicate using the HTTP and HTTPS protocols. It observes the conversations (requests and responses) and allows the operator to review them. It provides a number of plugins, mainly aimed at security functionality. Plugins perform one of two tasks: generate requests or analyze conversations.

3.2. Other types of information security tools

SANS Institute [25] classifies the information security tools into the following five categories:

1. Blocking attacks: Network based (includes secure web filtering)
2. Blocking attacks: Host based
3. Eliminating security vulnerabilities (includes penetration testing and application security testing)
4. Safely supporting authorized users
5. Tools to minimize business losses and maximize effectiveness

Web application scanners are in category 3. The class of web application scanners consists of tools that detect potential vulnerabilities in the web applications only, and not on the network. In addition to web application scanners, the overall security defense should include tools for web services, database scanners, network firewalls, anti-virus gateways, routers, intrusion detection/protection systems, and other tools.

4. Functional requirements for web application scanner

To develop a specification for web application scanners, we must clearly define a set of functions that a tool must successfully perform. A web application scanner must:

- Identify a selected set of software security vulnerabilities in a web application.
- Generate a text report indicating an action (or a sequence of actions) that leads to vulnerability.
- Generate an acceptably low ratio of false positives.

4.1. Some web application vulnerabilities

In this section, we identify a list of vulnerabilities that a web application scanner should detect. This list will form the basis for a formally worded requirement for mandatory features for a web application scanner. An extensive classification of web security threats can be found in [30]. The Open Web Application Security Project (OWASP) publishes the list of the most critical web application vulnerabilities [17]. These and other efforts are being incorporated into CWE [4].

Input validation weaknesses cause most web application vulnerabilities. Other types of weaknesses include use of poor authentication mechanisms, logic weaknesses, unintentional disclosure of content and environment information, and low-level coding weaknesses (such as buffer overflows). Often, vulnerability is caused by a combination of weaknesses. Some common vulnerabilities and attacks are:

- Cross-site scripting (XSS) vulnerabilities. The vulnerability occurs when an attacker submits

malicious data to a web application. Examples of such data are client-side scripts and hyperlinks to an attacker's site. If the application gathers the data without proper validation and dynamically displays it within its generated web pages, it will display the malicious data in a legitimate user's browser. As a result, the attacker can manipulate or steal the credentials of the legitimate user, impersonate the user, or execute malicious scripts on the user's machine.

- Injection vulnerabilities. This includes data injection, command injection, resource injection, and SQL injection. SQL Injection occurs when a web application does not properly filter user input and places it directly into a SQL statement. This can allow disclosure and/or modification of data in the database. Another possible object of injection is executable scripts, which can be coerced into doing things that their authors did not anticipate.

- Cookie poisoning is a technique mainly for achieving impersonation and breach of privacy through manipulation of session cookies, which maintain the identity of the client. By forging these cookies, an attacker can impersonate a valid client, and thus gain information and perform actions on behalf of the victim.

- Unvalidated input. XSS, SQL Injection, and cookie poisoning vulnerabilities are some of the specific instances of this problem. In addition, it includes tainted data and forms, improper use of hidden fields, use of unvalidated data in array index, in function call, in a format string, in loop condition, in memory allocation and array allocation.

- Authentication, authorization and access control vulnerabilities could allow malicious user to gain control of the application or backend servers. This includes weak password management, use of poor encryption methods, use of privilege elevation, use of insecure macro for dangerous functions, use of unintended copy, authentication errors, and cryptographic errors.

- Incorrect error handling and reporting may reveal information thus opening doors for malicious users to guess sensitive information. This includes catch NullPointerException, empty catch block, overly-broad catch block and overly-broad "throws" declaration.

Some other vulnerabilities are:

- Denial of service
- Path manipulation
- Broken session management
- Synchronization timing problems

More work is needed to refine the list of vulnerabilities that the web application scanners must support.

5. Issues in testing web application scanners

In addition to a functional specification, we need a test plan and a suite (or several suites) of test cases to check that a web application scanner satisfies the specification.

A test plan details how a tool is tested, how to interpret test results, and how to summarize or report tests. Currently, tools produce reports in a variety of formats. A common reporting format would make it easier to automate comparison of different tools.

We measure conformance of a tool to the specification by running it against a variety of test cases. In choosing test cases, it is important to understand the ways in which an attacker exploits vulnerabilities.

In normal operation, a user submits a request to the web application and gets a response back. An attacker submits an unexpected request to an application in hopes of exploiting an existing vulnerability. The goal of an attacker is to violate application's security policy. The attacker recognizes the existence of vulnerability either by examining application's response or indirectly, by noticing changes in application's behavior (this may include probing different parts of the application). Web application scanner works by simulating attacker's action.

To test web application scanners, we need web applications with vulnerabilities. For each vulnerability class, there must be at least one test application that exhibits it. Small test cases with a single vulnerability can be used to precisely test tools' ability to detect specific vulnerabilities. Large applications with a variety of vulnerabilities, such as WebGoat [18], will test scalability of a tool for real life applications. It is also important to test tools' ability to detect vulnerabilities in web applications built using different web technologies.

A basic test suite may contain only applications with easily exploitable vulnerabilities. For instance, if an application does no input validation at all, there are many ways to exploit the vulnerability and most tools

can find it. However, to thoroughly test a scanner, we need programs with subtle vulnerabilities.

Different types of SQL injection represent another example. An attacker typically sends a request to cause the application to generate a SQL query that can induce unexpected behavior. Then the attacker examines the error message returned to the web client. A typical mitigation approach is to prevent the application from displaying any database error messages. The vulnerability, though harder to detect, still exists – it is called "blind SQL injection".

In order to check for false positives, we need test cases that are free of vulnerabilities but have some features that cause difficulty for web application scanners. Generation of such test cases is an interesting research problem that requires understanding the way the tools work.

While developing test suites, we collect much larger numbers of candidate test cases. This collection, the SAMATE Reference Dataset (SRD) [23], is freely accessible on-line. We intend the database to support empirical research of software assurance. It contains over 1,600 test cases for source code analysis tools (as of August 18, 2006). We intend to add many test cases for web application scanners. We welcome participation from researchers and companies.

6. Summary

We defined web application scanners and presented some vulnerabilities that this class of tools should detect. We plan to develop a specification for web application scanners. The specification will give a precise definition of functions that the tools in this class must perform. We will develop suites of test cases to measure conformance of tools to the specification. This will enable more objective comparison of web application scanners and stimulate their improvement.

7. Acknowledgments

We thank Jeffrey Meister, Paul E. Black, and Eric Dalci for improving our understanding of web application scanners and many helpful suggestions on this paper. We also thank the anonymous reviewers for their insightful comments.

8. References

[1] A. Avizienis, J-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. on Dependable and Secure Computing*, 1(1):11-33, Jan-Mar 2004.

[2] M. Bishop and D. Bailey, "A Critical Analysis of Vulnerability Taxonomies," Technical Report 96-11, Department of Computer Science, University of California at Davis, Sep. 1996.

[3] Black, Paul E. and Fong, Elizabeth, "Proceedings of Defining the State of the Art in Software Security Tool Workshop," NIST Special Publication 500-264, September 2005.

[4] Common Weakness Enumeration (CWE), MITRE, http://cve.mitre.org/cwe/

[5] DISA, Application Security Tool Assessment Survey, V3.0, July 29, 2004. (To be published as STIG)

[6] Arian J. Evans, "Software Security Quality: Testing Taxonomy and Testing Tools Classification," Presentation viewgraph for OWASP APPSec DC, October 2005.

[7] Jeremiah Grossman, *The Five Myths of Web Application Security*, WhiteHat Security, Inc, 2005.

[8] Michael Howard, David LeBlanc, and John Viega, *19 Deadly Sins of Software Security*. McGraw-Hill Osborne Media, July 2005.

[9] Andrew J. Kornecki and Janusz Zalewski, *The Qualification of Software Development Tools From the DO-178B Certification Perspective*, CrossTalk, pages 19-23, April 2006

[10] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A Taxonomy of Computer Program Security Flaws," Information Technology Division, Naval Research Laboratory, Washington, D. C., September 1994.

[11] G. McGraw, *Software Security: Building Security In*, Addison-Wesley Software Security Series, 2006.

[12] Jody Melbourne and David Jorm, Penetration Testing for Web Applications, in SecurityFocus, 2003.

[13] NASA Software Assurance Guidebook and Standard, http://satc.gsfc.nasa.gov/assure/assurepage.html

[14] National Vulnerability Database (NVD), http://nvd.nist.gov/

[15] Netcontinuum, NC2000, http://netcontinuum.com/products/

[16] NT Objectives, NTOSpider, http://www.ntobjectives.com/products/ntospider.php

[17] OWASP, "The Ten Most Critical Web Application Security Vulnerabilities," http://www.owasp.org/index.php/OWASP_Top_Ten_Project

[18] OWASP, WebGoat Project, http://www.owasp.org/software/webgoat.html.

[19] OWASP, WebScarab
http://www.owasp.org/software/webscarab/

[20] Parasoft, WebKing, http://www.parasoft.com/webking.

[21] F. Piessens. “A taxonomy (with examples) of software vulnerabilities in Internet software,” Report CW 346, Katholieke University Leuven, 2002.

[22] Prescatore, John, Gartner, quoted in Computerworld, Feb. 25, 2005,
http://www.computerworld.com/printthis/2005/0,4814,99981,00.html

[23] SAMATE project, http://samate.nist.gov/

[24] SAMATE Tool Taxonomy,
http://samate.nist.gov/index.php/Tool_Taxonomy

[25] SANS Institute, http://www.sans.org/whatworks

[26] SPI Dynamics, WebInspect,
http://www.spidynamics.com/products/webinspect/

[27] Tenable Network Security, Nessus,
http://www.nessus.org/about/

[28] K. Tsipenyuk, B. Chess, and G. McGraw, “Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors,” *Proc. NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM),* US National Institute of Standards and Technology, 2005.

[29] Watchfire, AppScan,
http://www.watchfire.com/products/appscan/

[30] Web Application Security Consortium, “Threat Classification,” http://www.webappsec.org/projects/threat/

[31] Web Application Security Consortium Glossary,
http://www.webappsec.org/projects/glossary/