Implementation of a Web Application for Evaluation of Web Application Security Scanners

L. Ertaul, Y.Martirosyan,
Mathematics and Computer Science, CSU East Bay, Hayward, CA, USA

***Abstract*-With more and more people becoming Internet users there have been great increase in using Web in all areas of life, including communication, education and shopping. And as a result of these changes the security concerns have also grown. The web application vulnerability scanners help reduce these security concerns in Web-based applications. In today's market a large number of web application-scanning tools are available, e.g. QualysGuard WAS, Acunetix, Hailstorm, Appscan, WebInspect, etc. Although these tools are available in the market but question becomes how efficient they are to address security concerns in WEB applications? To compare vulnerability detection rate of different scanners, it is important to have an independent test suite. This paper describes a web application, which is intended to be used to evaluate the efficiency of QualysGuard WAS and Acunetix web application vulnerability scanners. The application implements real life scenarios for imitation of OWASPs Top Ten Security Risks that are presented in the wild. For several vulnerabilities presented in this application, we also explain defense measures, which secure the application significantly. The results of web application evaluation identifies the most challenging vulnerabilities for scanner to detect, and compare the effectiveness of scanners as penetration testing tools for exploiting OWASP Top Ten vulnerabilities. The assessment results can suggest areas that require further research to improve scanner's detection rate.**

***Index Terms*—Black box testing, web security scanners, web security, web security vulnerabilities.**

I. Introduction

In today's world many of the most dangerous security risks are based on vulnerabilities in web applications. ISO 27005 defines vulnerability as "*a weakness of an asset or group of assets that can be exploited by one or more threats where an asset is anything that can has value to the organization, its business operations and their continuity, including information resources that support the organization's mission*" [1]. According to National Vulnerability Database (NVD) [2] the number of vulnerabilities has become lower since 2009, which means that security measures has been incremented over last year. This is shown in Figure1.

In spite of this fact, percentage likelihood that at least one vulnerability will appear in a website remains very high. During 2010 every day almost every websites were exposed to at least one of high, critical, or urgent severity vulnerability, 64% of which had at least one Information Leakage vulnerability [2]. These web application vulnerabilities may cause attacks to exploit weaknesses on any tier or layer of web-based applications.

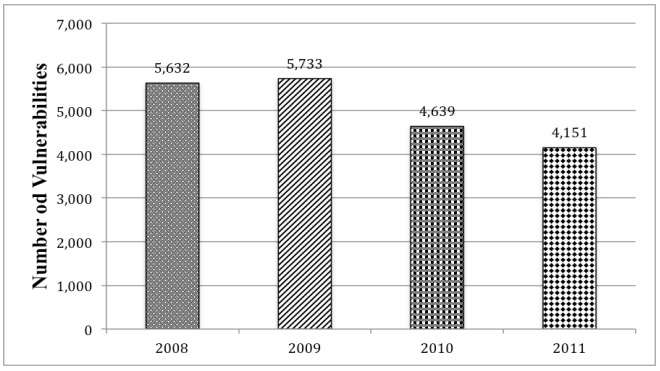

Fig. 1. Vulnerability distribution over years (2008-2011)

Most applications deployed on the Web implement a 3-tier architecture: presentation tier, business tier and data tier. Presentation tier is a web browser and dynamic web pages containing various types of markup language; business tier is a web application server; data tier is a database server. All tiers communicate with each other using strings to process input data. Web Application Server processes the inputs it receives from the clients and interacts with the database as shown in Figure 2.

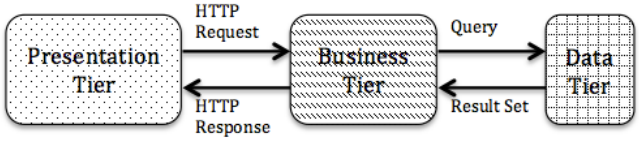

Fig. 2. The interaction between Client Tier (Web Client), Application Tier (Web Application) and Database Tier (Database Server)

Because web application server must validate and/or modify incoming strings before processing them or passing to database tier, in this paper we discuss input validation from client tier problem along with other most popular security flaws. Client Tier technologies include HyperText Markup Language (HTML) [53], Extensible Markup Language (XML) [57], JavaServer Pages (JSP) [58], JavaScript [55], and web applications continue to become

more feature-rich and more dynamic, in particular with the advent of Asynchronous JavaScript and XML (AJAX) client tier technology. In the Web Application used to evaluate web application scanners we implement modern features such as JavaScript and AJAX to present more complex tasks for security scanners [3]. Another challenge that can result in limitations for security scanners presented in the Web Application is the difference of vulnerabilities within one class in terms of types of attacks vector. For example, exploiting Persistent XSS is more complex task than Non-Persistent XSS vulnerability [26]. Our goal is to assess the strengths and limitations of QualysGuard WAS [47] and Acunetix [56] tools and to report the test results. In the first part of our experiments we create a tested, the Web Application (MusicStore) that contains The Open Web Application Security Project (OWASP) Top Ten [4] most critical security risks. In the second phase we test QualysGuard WAS and Acunetix security scanners for vulnerability detection.

In Section II we present OWASP Top Ten web application security risks of 2010. Section III describes the technical characteristics, functionality and vulnerabilities of Web Application, which is implemented as a test suit for assessment of scanners. We explain defense mechanisms against web application attacks in Section IV. Section V contains web application assessment results. In Section VI we present conclusions.

II. OWASP Web Application Security Risks

The OWASP security community has released its annual report in 2010 capturing the top risks in web application development as a combination of the probability of an event and its consequence. Following is the list of the top risks in web applications:

1. Injection
2. Cross-Site Scripting (XSS)
3. Broken Authentication and Session Management
4. Insecure Direct Object References
5. Cross-Site Request Forgery (CSRF)
6. Security Misconfiguration
7. Insecure Cryptographic Storage
8. Failure to Restrict URL Access
9. Insufficient Transport Layer Protection
10. Unvalidated Redirect and Forward

In web application described in this paper, we implement vulnerabilities 1 to 10, presenting them as real-life scenarios.

III. Web Application (MusicStore)

There are several existing web applications to demonstrate common web application vulnerabilities such as “HacMe” series [41]. Those applications are well known by users and scanner developers. These applications may be used by scanner developers to optimize their performance. Other concern is the unavailability of the source code to estimate the rate of positive and false negative results of security scanner’s findings. In addition to that these applications do not implement all the vulnerabilities from OWASP Top Ten report. Another well-known application is “WebGoat”[42] which is very complex web application. It is mainly used in educational purposes and not all of its test cases replicate real-life scenarios. Because of these drawbacks of available applications, there is a need to have an independent Web Application, which has real life scenarios and implements OWASP Top Ten vulnerabilities, to be used to test these web scanners. The Web Application (MusicStore) we present in this paper is designed to realistically simulate the steps a regular user goes through while using a dynamic web page and replicates the behavior of online store. The availability of source code and the control over server results in better evaluation of web application scanners.

Now let’s have a look at functionality of the application. First a user creates an account, providing his/her personal data, including credit card number and shipping address. Second he/she selects the product and stores his selection in personal shopping cart. Later when the user decides to make the purchase an invoice is placed in queue for further processing. In addition to that the user can add reviews to products and read other customers’ opinions, check partners’ newsletters and subscribe to mailing list. Figure 3 illustrates the interface of the web application.

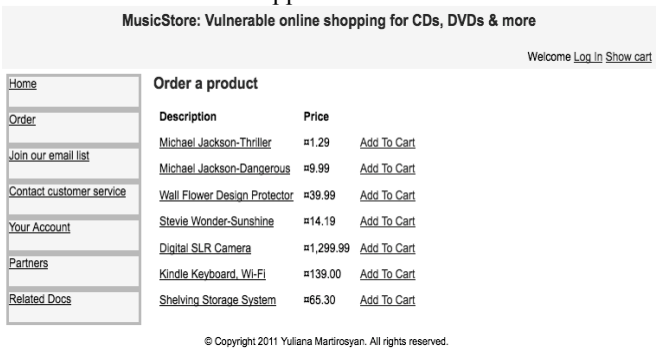

Fig. 3. Web Application User Interface

The MusicStore Web Application is Java [50] based application, which is deployed on Tomcat Server [51]. It uses database on Oracle database management server [52] to store the data for the web site in its tables. Apache is a web server with Tomcat servlet/JSP engine. The application uses JSPs to present the user interface. It also uses HTML, CSS [54], JavaScript, and AJAX technologies. The presence of such technologies as AJAX and JavaScript in our web application gives additional opportunities. JavaScript is widely used in modern web applications and it is important to analyze the behavior of tools and their ability to parse JavaScript code.

The web application was developed based on OWASP Top Ten report of 2010. In this section we go over the characteristic vulnerabilities presented in the Web Application. The full list of the flaws designed in the project is available in Vulnerability Report [43]. As seen in the report we implement fifty-five variations of OWASP Top Ten Security Risks (see Table1 ‘Total’ column).

A. First Order SQL Injection: ***recoverPassword*** function is intended to recover user’s password based on her answer to security question.

```
String query = "SELECT Password FROM v_UserPass WHERE (v_UserPass.EmailAddress = '" + emailAddress + "' AND v_UserPass.Answer = '" + answer + "') ";
```

```
Payload:
emailAddress=test%40test.com%27%29--&answer=anycolor
```

In ***recoverPassword*** function concatenation is used to create dynamic SQL query. Attacker can easily impersonate site user and recover victims password by commenting out the part of the query using '--' single-line comment indicator [6].

B. Blind SQL Injection: ***updatePassword*** function is intended to update user's password based on her emailAddress.

```
String query = "UPDATE v UserPass SET Password = ?, Answer =
'"+ answer+ "' WHERE EmailAddress = '"+ emailAddress + "'";
```

Manipulating 'answer' query parameter attacker can verify if email address he is interested in is stored in application database.

```
True payload:
password=test11&answer=red%27+WHERE+EmailAddress%3D%28%27e
xistedEmail%40test.com%27%29--
```

If there is user with existedEmail@test.com email address in application database then query will be executed.

```
False payload:
password=test11&answer=red%27+WHERE+EmailAddress%3D%28%27n
otExistedEmail%40test.com%27%29--
```

If there is not any user with notExistedEmail@test.com email address in application database then query will fail.

C. SQL Injection Using Database constant: ***insertReview*** function adds customer product reviews database in online store.

```
String query = "INSERT INTO v Reviews (Title, Message) VALUES
('"+title + "', '"+ message+ "')";
Payload:
title=%27%7C%7CSYSDATE%7C%7C%27&message=%27%7C%7CSYS
DATE%7C%7C%27
```

SYSDATE is Oracle function that returns date and time on a local database. This way attacker receives additional information about SQL Server.

D. Non-Persistent XSS: In this JSP Expression Language and Java example user registration information is stored in online store database after creditCardNumber parameter is validated on server side. No input inspection for firstName parameter is performed.

```
<form action="registrationServlet" method=post>
First Name <input type="text" name="firstName"
value="${newUser.firstName}">
Card number <input type="text" name="creditCardNumber">
<input type="button" value="Continue">
</form>
Payload:
firstName=John'"><script>alert ("firstName parameter is
vulnerable")</script>&creditCardNumber=1234
```

If credit card number is incorrect firstName value be reflected on web page.

E. Persistent XSS: ***insertReview*** function adds customer product reviews database in online store.

```
String query = "INSERT INTO v_Reviews (Message) VALUES ('"+
message+ "')";
Payload:
message=message+%3Cscript%3Ealert%280%29%3C%2Fscript%3E&SU
BMIT=Submit
```

F. DOM Based XSS: web page uses firstName parameter in URL to greet the user. Web browser parses this HTML, which is received from server, into DOM. Parser executes the JavaScript code and as a result the XSS vulnerability is exploited.

```
<div id="greeting">
Hello
<SCRIPT>
var url = window.location.href;
var pos = url.indexOf("firstName=") + 10;
var firstName_string = url.substring(pos);
document.write(unescape(firstName string));
</SCRIPT>
</div>
Payload in URL:
http://www.vulnerablewebapp.com/join_email_list.jsp?firstName=%3Cscri
pt%3Ealert%28%22DOM%20XSS%22%29%3C/script%3E
```

G. Broken Authentication: web application uses password recovery function, when you need to answer the security question. Using social engineering attacker can guess the country. Then using brute force dictionary method attacker can find the city and obtain victim's credentials [7], [8], [9], [10].

```
Question: Where were you born?
Payload is list of cities.
```

H. Insecure Direct Object Reference: web application receives reference to a file as form parameter 'letter', reads and displays the text. An attacker manipulates 'letter' parameter to access other objects.

```
Form parameter: letter=SomePartner.html&SUBMIT=View+Letter
Java code:
File f = new File(path + "/" + request.getParameter("letter"));
String text = getFileText (new BufferedReader (new FileReader (f)), false);
Payload: ../../../../../../../apps/java/apache-tomcat-6.0.16/conf/server.xmls
```

I. CSRF: the victim is authenticated at vulnerable online store. Attacker has placed malicious CSRF code on a web site. The browser will submit the request to vulnerable online store.

```
Malicious CSRF code:
<img src=
"http://www.vulnerablewebapp.com/updateUserPassword?password=false
pass" width="1" height="1" border="0">
```

J. Security Misconfiguration: Web application server is vulnerable to slow HTTP headers DDoS attack. Using slowhttptest [11] tool attacker can get denial of service by slowing down requests.

K. Failure to Restrict URL: web application protects all data under "/user" directory. After user is authenticated web application makes possible to access /userAccess.jsp link. But it is not under /user directory and attacker can guess that hidden link and take advantage of it.

```
<% if (request.isUserInRole("user")) {%>
<a href= "https://www.vulnerablewebapp.com/userAccess.jsp">User
Only</a>
```

L. Insufficient Transport Layer Protection: any data under "/user" directory should be protected using SSL.

```
https://www.vulnerablewebapp.com/user
```

M. Unvaidated Redirect and Forward: Redirect and Forward functionality is very common in many web applications. But insecure implementation of it can result in tricking the user by an attacked into clicking the link that will navigate to unsafe destination. This is an example of Java code that demonstrates implementation of redirect function where site parameter value is the URL.

```
String site = request.getParameter("site");
if(site!=null && site!=""){
response.setStatus(response.SC_MOVED_TEMPORARILY);
response.setHeader("Location", site); }
Payload: ="http://www.vulnerablewebapp.com
/partners/displayParnerLetter?site=http://www.attackerDestination.com
```

With all these threats widely available in web it is important

to secure web application against them. In the next section we explain defense mechanisms and we show the implementation of several most important techniques in our web application.

IV. Defense Mechanisms

Preventing vulnerabilities in applications extremely important due to high number of attacks (see Fig 1). In this section we describe several defense techniques against web application attacks used in our application.

• *SQL Injection and Cross-Site Scripting (XSS) Defense.* Server side defense using Prepared Statement [12] is the most effective way to protect from SQL Injection, because it ensures that intent of query is not changed. It is very important to lockdown database server and to follow the Principle of Least Privilege [13], [14]. Modern web applications also rely heavily on caching and database schema design to improve performance [40].

For prevention code injection attacks, including SQL Injection and XSS all user data should be validated. Input validation can be performed client side using JavaScript, but from security prospective it is not effective, because it doesn't provide protection for server-side code.

```
JavaScript Example:
    var emailexp = /^([A-Za-z0-9_\-\.])+\@([A-Za-z0-9_\-\.])+\.([A-Za-z]{2,4})$/
    if (!isValid(emailexp,form.emailAddress.value)){
    return false}
```

Despite rule that input must be validated server-side sometimes validation should be performed client-side [15][16]. Web frameworks and filters that offer automate sanitization to prevent XSS in web applications are gaining popularity, because manual implementation of input sanitization in web application is prone to errors [17-25]. Unfortunately input filters can be circumvented with various attack vectors [26] [27].

• *Broken Authentication Defense, Session Management and Transport Layer Protection.* Authentication and session security is critically important because compromised credentials leads to impersonation and loss of confidentiality. To protect user's session ID strong efforts should be made to avoid XSS flaws as described in Injection Defense Section. Authentication key points are Password Strength and Password Use, including number of possible attempts and storage; and Password Recovery mechanism [28].

```
Example:
<security-constraint>
      <web-resource-collection>
         <web-resource-name>User</web-resource-name>
         <url-pattern>/user/*</url-pattern>
      </web-resource-collection>
      <auth-constraint>
         <role-name>user</role-name>
      </auth-constraint>
      <user-data-constraint>
         <transport-guarantee> CONFIDENTIAL
       </transport-guarantee>
      </user-data-constraint>
   </security-constraint>
```

Authentication relies on secure communication, so it is important to maintain Transport Layer Protection [29].

In this example data under /user/ directory will be transferred using secure connection. Also session cookie used to identify authenticated user should contain the "secure" or "HTTPOnly" attribute.

• *Insecure Direct Object Reference Defense.* This attack represents a serious threat to parameter-driven site if parameter is modified to point to a local file on the Web server. It is a good practice to use a reference map to prevent parameter manipulation.

• *Cross-Site Request Forgery Defense.* Main defense technique is using authorization token, generated web application on server side. The Anti-CSRF token should be a randomly generated value, specific to the user's current session [29-32].

• *Security Misconfiguration Defense.* Maintaining security settings of the application, frameworks, application server, web server, database server, and platform is very complex problem. Web servers are frequent target of attacks so trying to secure web servers the following aspects should be taken into account: Configuration, Web content and server-side applications, Operating System, Documentation [33].

Example:

HTTP server is subject to Slow type HTTP Attack [34].

There is number of steps to protect against this attack [35]. The RequestReadTimeout directive value should be set to limit the time a client may take to send the request [36].

• *Insecure Cryptographic Storage.* Sensitive data should not be displayed in clear form. The data should be stored encrypted with strong encryption algorithms, such as AES [44], RSA [45], and SHA-256 [46], [37] in database and decrypt it on server side upon request, or store hash of the data.

• *Failure to Restrict URL access.* Hidden pages are difficult to find, but sometimes it is possible to guess the URL, which is not intended for presence to unauthorized users. It is important to use an effective and trusted access control mechanism [38] and access control matrix that is carefully planned [39].

• *Unvalidated Redirect and Forward.* As for many previous discussed attacks parameter value validation should be performed before redirection. It can be done by ensure that the URL parameter is indeed a valid URL.

With all described flaws and defense mechanisms we need to find out whether the Web Application presented in this paper is useful to identify weak and strong points of a security scanner. In next section we examine the experimental results of running web application vulnerability scanners against our MusicStore web application.

V. Evaluation of Web Application Vulnerability Scanners

Two web application vulnerability scanners, QualysGuard WAS (scanner Q) and Acunetix (scanner A), were tested using our MusicStore web application in order to find out whether those tools are actually successful at finding existent vulnerabilities. The results discuss the challenging vulnerabilities to detect, the possibility of false positive reports and the variation of vulnerabilities detection between

different types. Both scanners support identification of web application vulnerabilities in the OWASP Top Ten approach, including dynamic and static search lists, links crawling, brute force and authentication.

Before the testing procedure Web Application is restored to original state. The setup consist of following steps:

1) Count and classify vulnerabilities in web application before the initial test.

2) The database server and web server are put in an initial state. This state includes seven products, two regular users and one administrator user in database and seven images for each product on web server.

3) Run web application scanner in initial mode.

4) Count the vulnerabilities found by web application scanner and compare to actual vulnerabilities report in step 1. The details of analysis presented next in this section.

5) Count False Positive/Maybe/Duplicate results.

The results of running Scanner A and Scanner Q against web application are shown in Table 1. The Table contains the following data:

- First column represents the vulnerabilities presented in the test suit. (Top 10 OWASP Vulnerabilities)
- Second column shows the different types of a vulnerability presented in first column.
- Third column contains the total number of vulnerabilities of each type existing in the web application MusicStore.
- Forth column contains the number of vulnerabilities detected by scanners.
- Fifth column is named False Positive (FP) results, which are reported by scanners but are not actually presented in the Web Application. The list included the findings of vulnerabilities marked as 'possible', which we will consider as 'maybe'; or vulnerabilities, which were reported, previously in the same type but with different description.
- The last column represents False Negative (FN) results, those are the vulnerabilities missed by the scanners.

Full report of running QualysGuard WAS and Acunetix against Web Application can be found in QualysGuard WAS Evaluation [48] and Acunetix Evaluation [59].

The Table 1 reports the vulnerabilities that were detected by web application scanners. As seen from the Table 1 both tools missed some weaknesses. Here we present the analysis of why the scanners missed certain vulnerabilities.

1) SQL Injection. Scanner A was able to discover all First Order SQL Injection vulnerabilities. But both scanners failed to find second order SQL Injection vulnerabilities, which are not executed immediately. The result of the injection is displayed on a page that should be navigated by user after the payload was submitted. Scanners fails to follow this logic thus interprets it as a negative response.

2) Cross-Site Scripting. Scanner Q discovered all Non-Persistent XSS vulnerabilities. Scanner A's results were very impressive too, but as a group most Persistent multi-step XSS and DOM XSS vulnerabilities were missed by both scanners.

TABLE 1
RESULTS OF WEB APPLICATION VULNARABILITY SCANNERS ASSESSMENT

Vulnerabilities	Vuln. Type	Total	Detected		FP		FN	
			A	Q	A	Q	A	Q
SQL Injection	First Order	2	0	2	1	0	2	0
	Second Order	4	0	0	0	0	4	4
XSS	Non-Persistent XSS	10	9	10	36	10	1	0
	Persistent XSS	4	1	3	1	3	3	1
	DOM XSS	4	3	1	0	0	1	3
Broken Authentication		2	1	1	0	0	1	1
Insecure Direct Obj. Ref.		1	1	1	0	0	0	0
CSRF		11	0	4	0	8	11	7
Security Misconfiguration	Password sent via GET method	2	0	0	0	0	2	2
	Web Server DDoS	2	0	2	0	2	2	0
	Sensitive Data display	1	0	0	0	0	1	1
Insecure Cryptographic Storage		7	2	4	0	0	5	3
Failure to Restrict URL Access		1	0	0	0	0	1	1
Insufficient Transport Layer Protection	Insecure session cookie	2	2	2	0	0	0	0
	Insecure Login (no SSL)	1	1	1	0	0	0	0
Unvalidated Redirect and Forward		1	0	1	0	0	1	0
		55						

3) Broken Authentication and Session Management. In our web application we present two vulnerabilities of this type. The first one is vulnerability with weak password recovery model. The weakness is easily exploited by guessing. So scanners were not able to find the flow, which is not surprising. Both scanners easily discovered the second vulnerability because it had plain brute force attack possibility.

4) Insecure Direct Object Reference. Both security scanners were able to detect this type of vulnerability.

5) CSRF. Scanner Q found only 4 CSRF vulnerable links. Scanner A didn't show any results for this type of vulnerability. We relate this to the fact that during the information gathering phase the link crawling did not enumerate all the reachable pages. For those links presented in crawling report CSRF vulnerability was detected. For full

information on links presented in our web application see Full Crawling Report [49].

6) Security Misconfiguration. The 2 vulnerabilities missed by the tool Q in this type are based on insecure data handling by web server, which is able to process requests sent by GET method. Scanners missed this vulnerability because the form with sensitive data was submitted by POST method although it was possible to send the request by adding the parameters in URL and process it as GET method. Scanner A didn't find any of the presented flows.

7) Insecure Cryptographic Storage. Both scanners discovered all session flaws. Although Scanner Q tested the possibility of sending credit card information securely, but it missed the same type of vulnerability: secure processing password and the answer to secret question. Those are application specific vulnerability.

8) Failure to Restrict URL Access. Both scanners did not detect the hidden link. The link is accessible by registered user only. Another way to reach the hidden link is force browsing which has failed for scanner specific testing.

9) Insufficient Transport Layer Protection. The scanners were able to detect all insecure cookie and session processing vulnerabilities.

10) *Unvalidated Redirect and Forward.* Scanner Q detected this vulnerability, while Scanner A didn't report any findings.

In Figure4 and Figure 5 we present the following details on our findings:

- False Negative Rate (FN)- The rate is calculated as the number of FN vulnerabilities of each type over total number of vulnerabilities of each type.
- False Positive/Duplicate/Maybe Rate (FP) – percentage of vulnerabilities reported by scanner, but not the actual weaknesses. The rate is calculated as the number of FP vulnerabilities of each type over total number of vulnerabilities of each type.

The interesting result for Scanner Q was found for CSRF vulnerability type as shown in Fig. 4. False Positive rate is higher than false Negative. This means that despite the fact that scanner is very attentive to this type of weaknesses and suspected many web pages to be vulnerable it wasn't able to reach all possible web pages to try there the attacks as a result of complex multi-step application design.

Fig. 5 shows that Scanner A has very high FP results for XSS vulnerability. Almost all FP reports were duplicates.

VI. Conclusion

We described OWASP Top 10 Security Risks implemented in the independent web application, which was designed and used as a testbed for evaluation of effectiveness of QualysGuard WAS and Acunetix web application vulnerability scanners. Each vulnerability type, presented in the web application was implemented as separate real life scenarios, including the popular coding mistakes and possible defense mechanisms.

Web Application vulnerability scanners failed to crawl the entire web application, which resulted in missing vulnerabilities. The other challenge was the difficulty to exploit stored and multi-step vulnerabilities. This also resulted in high rate of False Negative results. False Positive report was mostly the result of duplicates and 'possible' vulnerabilities. The tools showed very good results on detecting straightforward vulnerabilities as Non-Persistent XSS, Transport Layer Protection and Insecure Direct Object Reference.

Our plans for future work include evaluation of another two well-known web application vulnerabilities scanners using MusicStore web application with a purpose to get more extensive independent scanner evaluation report.

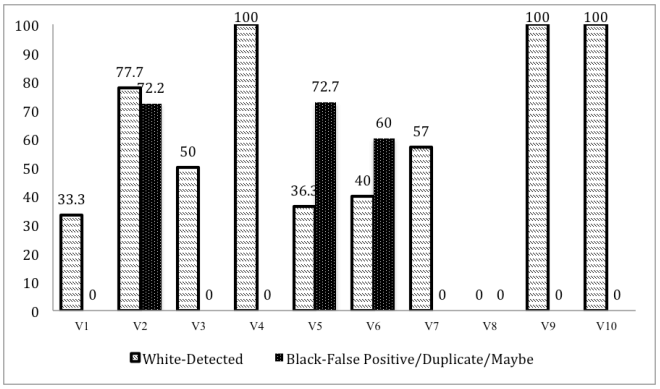

Fig. 4. QualysGuard. False Negative and False Positive/Duplicate/Maybe.

V1- SQL Injection, V2-Cross-Site Scripting, V3-Broken Authentication, V4-Insecure Direct Object Reference, V5-Cross-Site Request Forgery, V6-Security Misconfiguration, V7-Insecure Cryptographic Storage, V8- Failure to Restrict URL Access, V9- Insufficient Transport Layer Protection, V10-Unvalidated Redirect and Forward.

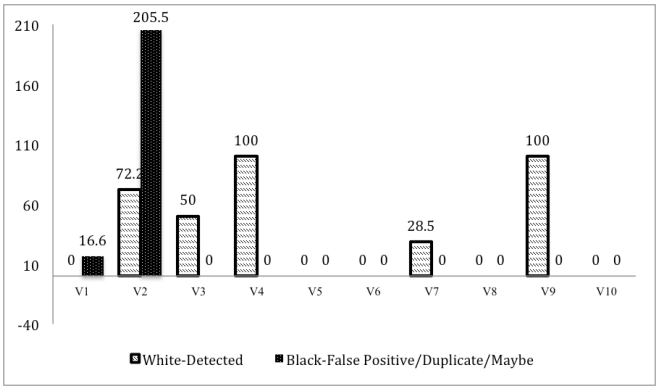

Fig. 5. Acunetix. False Negative and False Positive/Duplicate/Maybe.

V1- SQL Injection, V2-Cross-Site Scripting, V3-Broken Authentication, V4-Insecure Direct Object Reference, V5-Cross-Site Request Forgery, V6-Security Misconfiguration, V7-Insecure Cryptographic Storage, V8- Failure to Restrict URL Access, V9- Insufficient Transport Layer Protection, V10-Unvalidated Redirect and Forward.

References

[1] International Organization for Standardization and International Electrotechnical Commission. ISO/IEC 27001:2005, Information technology – security techniques – information security management systems – requirements, 2005.

[2] National Vulnerability Database, http://nvd.nist.gov.

[3] Anthony T. Holdener III, "Ajax: The Definitive Guide Interactive Applications for the Web", O'Reilly Media, 2008.

[4] The OWASP Foundation, "OWASP Top Ten Web Application Security Risks", http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2011.

[5] Oracle Learning Library. Defending Against SQL Injection Attacks, http://apex.oracle.com/pls/apex/f?p=44785:1:4073230388602787::NO

[6] Oracle PL/SQL Tutorial. http://www.java2s.com/Tutorial/Oracle/CatalogOracle.htm.

[7] THC Hydra, thc releases, thc-hydra v. 7.1, http://www.thc.org/thc-hydra/, 2011.

[8] John the Ripper password cracker, http://www.openwall.com, 2011.

[9] B. -S. Huang. "Brutus Project Groups Technical Report", Brutus Project. http://www.hoobie.net/brutus/.

[10] Massimiliano Montoro, Cain& Abel, http://www.oxid.it/cain.html.

[11] slowhttptest. Application Layer DoS attack simulator. http://code.google.com/p/slowhttptest, 2011.

[12] Java SE Technical Documentation. JDBC(TM) Database Access, Using Prepared Statements, http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html, 2011.

[13] Jerome H. Saltzer and Michael D. Schroeder, The protection of information in computer systems. Proceedings of the IEEE, 63(9): 1278-1308, 1975.

[14] Fred B. Schneider, Least Privilege and More. IEEE Security & Privacy, pp. 55-59, September 2003.

[15] Matt Johansen and Kyle Osborn, "Hacking Google Chrome OS". Black Hat USA, Briefings and Trainings, August 2011

[16] Jeremiah Grossman, "Sometimes Input MUST be Validated Client-Side". WhiteHat Security. https://blog.whitehatsec.com/sometimes-input-must-be-validated-client-side-o_o/, September 1, 2011.

[17] E. Athanasopoulos, V.Pappas, A. Krithinakis, S.Ligouras, E. P. Markatos, "xJS: practical XSS prevention for web application development", Proceedings of the 2010 USENIX Conference on Web Application Development, 2010.

[18] P. Bisht, V. Venkatakrishnan, "XSS-GUARD: precise dynamic prevention of cross-site scripting attacks", Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 23–43, 2008.

[19] W. Robertson, G. Vigna, "Static enforcement of web application integrity through strong typing", Proceedings of the 18th Conference on USENIX Security Symposium, SSYM 2009. USENIX Association, Berkeley (2009).

[20] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, D. Song, "A symbolic execution framework for JavaScript", Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010. IEEE Computer Society, Washington, DC, USA 2010.

[21] P. Saxena, D. Molnar, B. Livshits, "Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization", Tech. rep., Microsoft Research. September 2010.

[22] Zend Framework. Zend Filter. http://framework.zend.com/manual/en/zend.filter.set.html.

[23] Yii Framework. Special Topics. Security. http://www.yiiframework.com/doc/guide/1.1/en/, 2010.

[24] Template Toolkit.Manual. http://template-toolkit.org/docs/manual/Filters, January 2012.

[25] P. Saxena, S. Hanna, P. Poosankam, D. Song, "FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications", 17th Annual Network & Distributed System Security Symposium NDSS, 2010.

[26] K. K. Mookhey, Nilesh Burghate, Detection of SQL Injection and Cross-site Scripting Attacks, Symantec Connect Community, 02 November 2010.

[27] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A Systematic Analysis of XSS Sanitization in Web Application Frameworks", University of California, Berkeley, 2011.

[28] C. Miller, "Password Recovery". http://fishbowl.pastiche.org/archives/docs/PasswordRecovery.pdf, October 20 2002.

[29] K. Jaggi, "Securing Web Apps on Tomcat with SSL". Sun Developer Network, August 2006.

[30] Sun Microsystems. Mojarra Project. Mojarra JavaServerTM Faces JSF 2.0, 2011.

[31] Apache Struts. Class Token. http://struts.apache.org/2.0.14/struts2-core/apidocs/org/apache/struts2/components/Token.html.

[32] E. Sheridan. OWASP CSRFGuard Project.https://www.owasp.org/index.php/CSRF_Guard, 2010.

[33] N. Mendes, A. A. Neto, J. a. Dura˜es, M. Vieira, and H. Madeira, "Assessing and Comparing Security of Web Servers," Proceedings of the 2008 14th IEEE Pacific International Symposium on Dependable Computing. IEEE Computer Society, 2008.

[34] S. Shekyan. "Identifying Slow HTTP Attack Vulnerabilities on Web Applications". Qualys Community, July 7, 2011.

[35] S. Shekyan, "How to Protect Against Slow HTTP Attacks". Qualys Community, November 2, 2011.

[36] The Apache Software Foundation. Security Tips. Apache HTTP Server Version 2.5. http://httpd.apache.org/docs/2.3/misc/security_tips.html, 2012.

[37] Oracle Database Documentation Library. Developing Applications Using Data Encryption. Oracle® Database Security Guide 10g Release 1 (10.1). Part Number B10773-0.

[38] Vincent C. Hu David F. Ferraiolo D. Rick Kuhn, "Assessment of Access Control Systems". National Institute for Standards and Technology (NIST), September 2006.

[39] The Apache Software Foundation. The Apache Tomcat 5.5 Servlet/JSP Container Realm Configuration HOW-TO. http://tomcat.apache.org/tomcat-5.5-doc/realm-howto.html.

[40] M.Shema. "Seven Deadliest Web Application Attacks", Syngress, 2010.

[41] Foundstone Hacme Series. McAfee Corp.

[42] WebGoat Project. OWASP. http://www.owasp.org/index.php/Category:OWASP WebGoat Project .

[43] L.Ertaul, Y.Martirosyan, *"Vulnerability Report"*, http://www.mcs.csueastbay.edu/~lertaul/WEBSEC/VulnerabilityReport.pdf, January 2012.

[44] NIST, "Advanced encryption standard (AES)," Nov. 2001, http://csrc.nist.gov/publications/fips/fips197/fips- 197.pdf.

[45] R. L. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems,"Commu- nications of the ACM, vol. 21, pp. 120-126, 1978

[46] NIST/NSA, "FIPS 180-2: Secure Hash Standard (SHS)", August 2002 (change notice: February 2004).

[47] QualysGuard Web Application Scanning (WAS), Qualys Inc., http://www.qualys.com/products/qg_suite/was/

[48] L.Ertaul, Y.Martirosyan, *"QualysGuard WAS Evaluation"*, http://www.mcs.csueastbay.edu/~lertaul/WEBSEC/QualysGuardWASEvaluation.pdf, January 2012.

[49] L.Ertaul, Y.Martirosyan, "*Full Crawling Report*", http://www.mcs.csueastbay.edu/~lertaul/WEBSEC/FullCrawlingReport.pdf, January 2012.

[50] Java. Oracle Corporation, 1995.

[51] Tomcat Server. Apache Software Foundation.

[52] Oracle database management Server, Oracle Corporation.

[53] HyperText Markup Language (HTML), World Wide Web Consortium and Web Hypertext Application Technology Working Group (WHATWG).

[54] Cascading Style Sheets (CSS). World Wide Web Consortium.

[55] JavaScript. Brendan Eich. Netscape Communications Corporation. Mozilla Foundation.

[56] Acunetix Web Vulnerability Scanner. Acunetix. http://www.acunetix.com/vulnerability-scanner/

[57] Extensible Markup Language (XML), World Wide Web Consortium.

[58] JavaServer Pages Technologies (JSP), Sun Microsystems.

[59] L.Ertaul, Y.Martirosyan, *"Acunetix Evaluation"*, http://www.mcs.csueastbay.edu/~lertaul/WEBSEC/AcunetixEvaluation.pdf, February 2012.