CROSS-SITE SCRIPTING AND SQL INJECTION ATTACK LAB

- ARVIND PONNARASSERY JAYAN
& SHRIYA KUNERIYA

PART 1: CROSS-SITE SCRIPTING ATTACK LAB

TASK 0: Initial Setup

For this task we will be using Ubuntu version 16.04 with the preconfigured specifications. We will be using Elgg web application, that is a web-based social networking application, which have several accounts in the server.

TASK 1: Posting a Malicious Message to Display an Alert Window

For this task we will be embedding a JavaScript program to cause an alert action in an user's account. For this we initially select the user "Samy". We embed a JavaScript program, that alerts "XSS", in the brief description field of Samy, as shown in Figure 1. We save this profile and we logout of Samy's account.

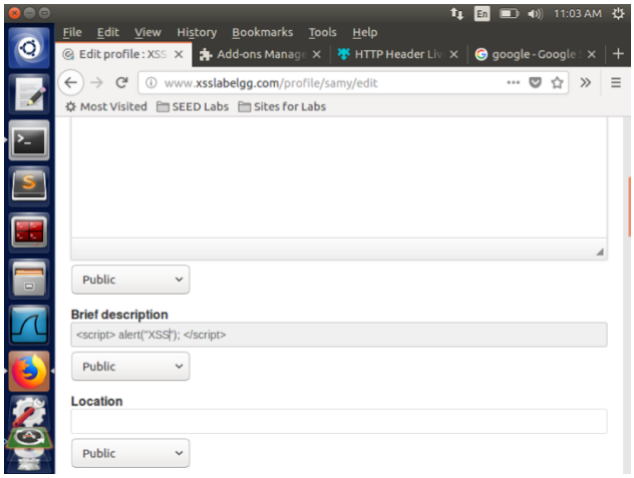

Figure 1: Embedding the JavaScript Program

Now we login to Alice's account to demonstrate victim behaviour. Now when Alice visits Samy's profile, the JavaScript program gets triggered and an alert saying XSS is displayed on the page as shown in Fig 2.

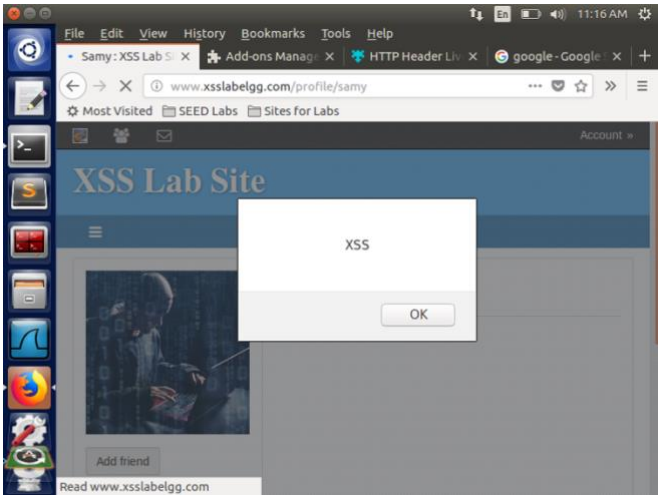

Figure 2: Pop-up Alert Window for a different User

TASK 2: Posting a Malicious Message to Display Cookies

For this task we will be embedding the JavaScript program to alert the cookie of the user in the document. Initially we go to Samy's Profile and add the JavaScript tag as shown in Figure 3. Once the Brief description field contains the JavaScript program the profile is saved and we logout from the user account.

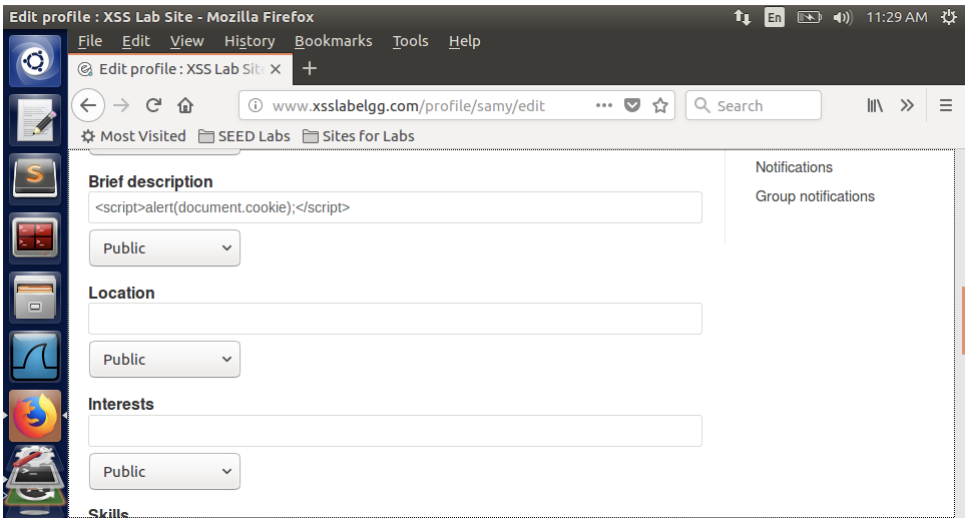

Figure 3: Embedding the JavaScript program to display the cookie

Now we login into Alice's account and go to Samy's profile. Due to the JavaScript program embedded in the Brief description in Samy's profile, alert gets triggered and the user cookie is displayed, as shown in Figure 4. But this cookie will only be displayed for the user, the attacker will not be able to see this value.

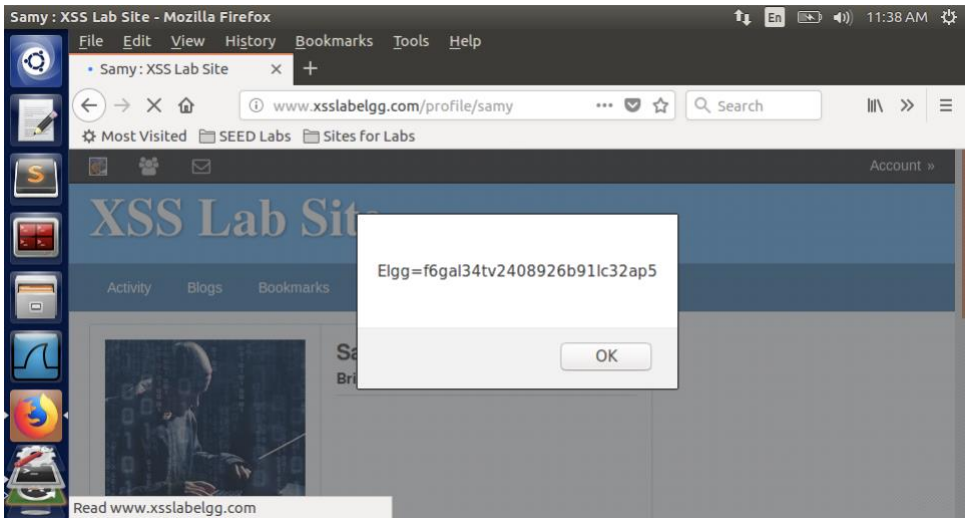

Figure 4: Alert displaying the user cookie value

TASK 3: Stealing Cookies from the Victim's Machine

For this task we will be embedding the JavaScript program to send an HTTP request with the victim's cookie value to the attacker's system. Now we go to Samy's Profile and add the JavaScript tag as shown in Figure 5. The Brief description field contains the JavaScript program that loads an image from the URL specified, this results in a HTTP GET request sent to the attacker's machine, which includes the victim's cookie value, to the port 5555 as specified.

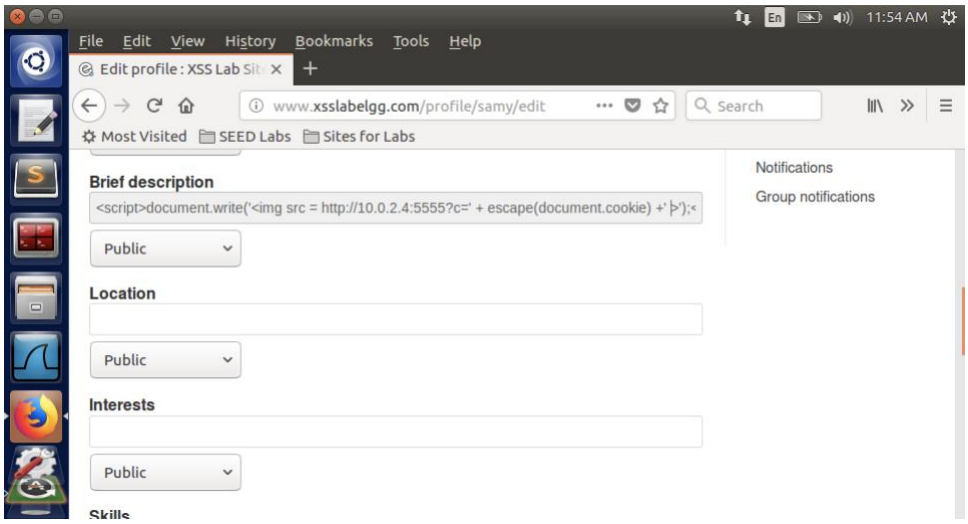

Figure 5: Embedding the JavaScript Program to steal cookie information

We will start listening for this HTTP GET request in the attacker's machine using netcat command. Now when the victim Alice visits Samy's profile the JavaScript embedded program gets triggered and the cookie value is sent to the attacker's machine as shown in Figure 6.

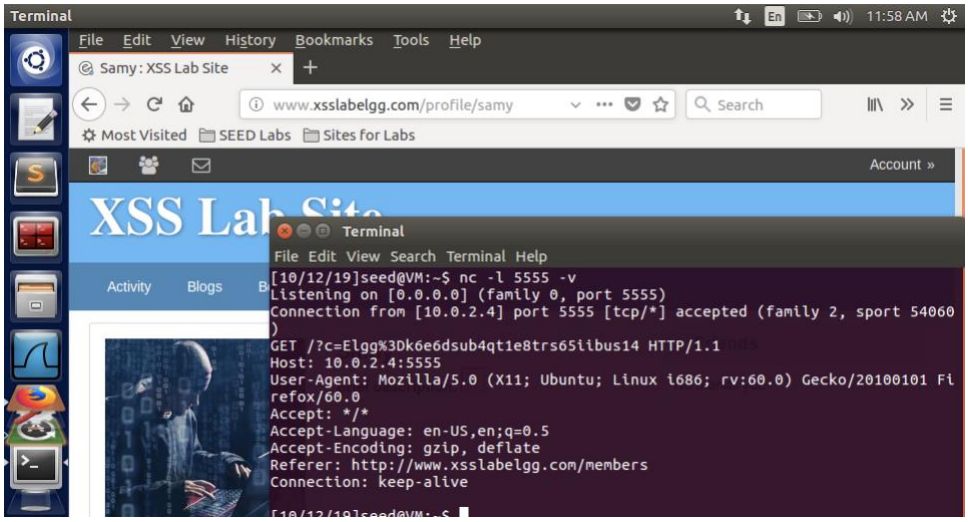

Figure 6: Obtaining the cookie value using netcat

Now when learn the mechanism to manipulate and steal data from a victim's machine.

TASK 4: Becoming the Victim's Friend

For this task we will try to understand how the server Elgg adds a friend to another user. For this we will be using Mozilla Firefox extension HTTP Header Live to check the HTTP request that is made while adding a new friend, as shown in Figure 7. Here Samy uses Charlie's account to add himself as a friend.

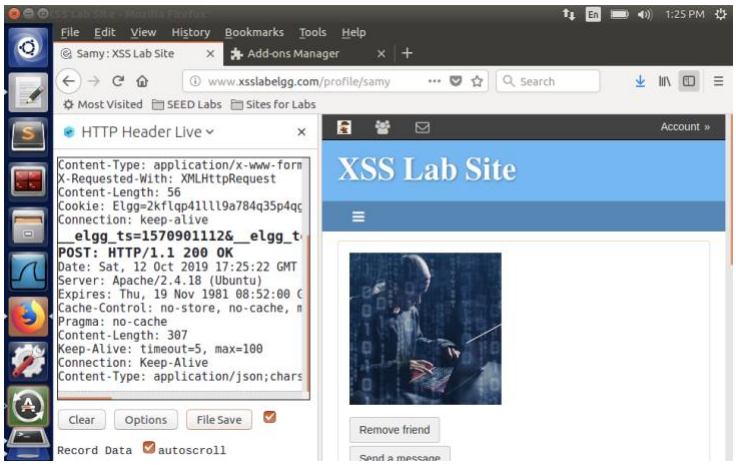

Figure 7: Using HTTP Header Live to understand the HTTP request

Now we will use header information we got from the HTTP request, as shown in Figure 8, to construct a malicious JavaScript program. For this we identify that the request URL takes the parameters friend, and 2 secret values __elgg_ts and __elgg_token. We observe that friend parameter acts as group id of the user Samy and it has a value "47". We also observe that Elgg uses session cookies that prevents third-party pages to send the request.

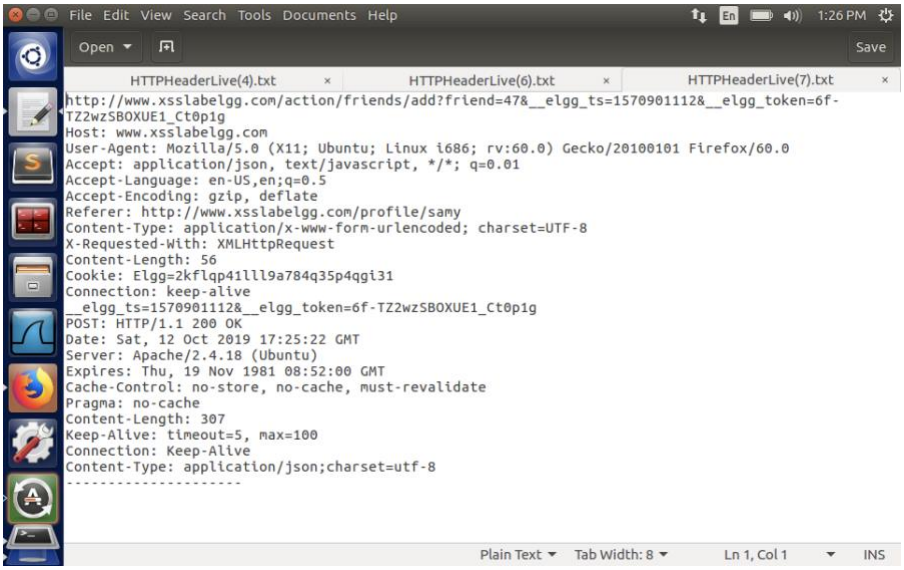

Figure 8: HTTP request content

We can now start constructing the JavaScript program with what we have observed. We should construct the HTTP request with the URL from the Now we will use header information we got from the HTTP request, as shown in Figure 8, to construct a malicious JavaScript program. For this we use the same URL for HTTP request that was send to Elgg, with the friend parameter as "47", where 47 is the group ID used by Elgg used to identify the Samy uniquely. The values of __elgg_ts and __elgg_token are set by Elgg as a countermeasure for CSRF attacks. These values cannot be hardcoded in our malicious JavaScript program as they are page specific. The cookie for the request will be automatically set by the browser helping us in the exploit.

Question 1: The values we set for __elgg_ts and __elgg_token is page specific and we can grab these values from the page itself using JavaScript as shown in Figure 9. The secret tokens that counters CSRF set by the server are assigned to elgg. Hence we can access these values by using elgg.security.token. __elgg_ts and elgg.security.token.__elgg_token respectively and then inject it into the URL instead of hardcoding them.

```
Open                                                                    Save
<script type="text/javascript">
window.onload = function () {
var Ajax=null;
var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
var token="&__elgg_token="+elgg.security.token.__elgg_token;
//Construct the HTTP request to add Samy as a friend.
var sendurl="http://www.xsslabelgg.com/action/friends/add" + "?friend=47" + token + ts; //FILL IN
//Create and send Ajax request to add friend
Ajax=new XMLHttpRequest();
Ajax.open("GET",sendurl,true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
Ajax.send();
}
</script>
```

Figure 9: Constructing Malicious JavaScript Program

Now we embed this malicious JavaScript program in About me field of Samy's profile. We observe that this field has additional formatting to the data we add, which can cause problems for our JavaScript program. Hence we initially remove the editor and add then embed the JavaScript code as shown in Figure 10.

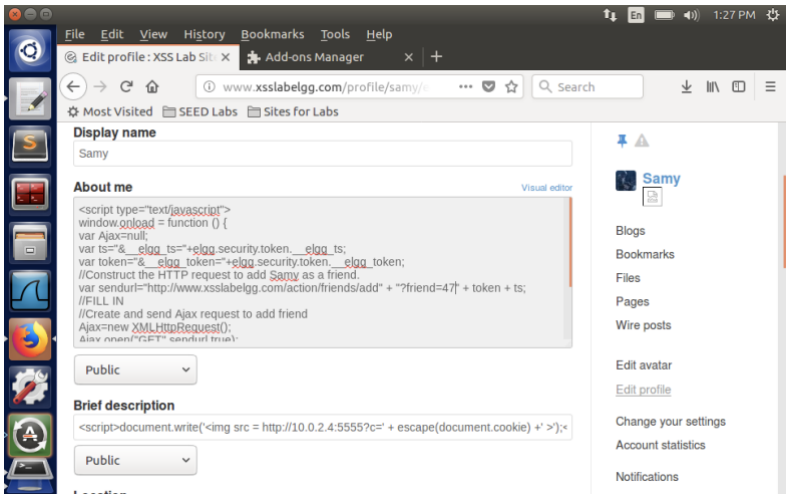

Figure 10: Embedding JavaScript in About me Field of Attacker

Question 2: In the case that Elgg does not define a plaintext editor, resulting in us not being able to insert Javascript in the About Me section directly, we can still launch the attacks. One way would be to use a browser extension to eliminate the excessive formatting data from correspondingly generated HTTP request. Also, the attacker could use a customised client instead of regular browsers to send out the requests. In both the cases, the attacker would be able to execute the attacks.

We demonstrate the attack as follows. Logging into Alice's account, we initially look at her profile. Note, as per Fig 11, Alice is not friends with anyone, specifically she is not friends with Samy.

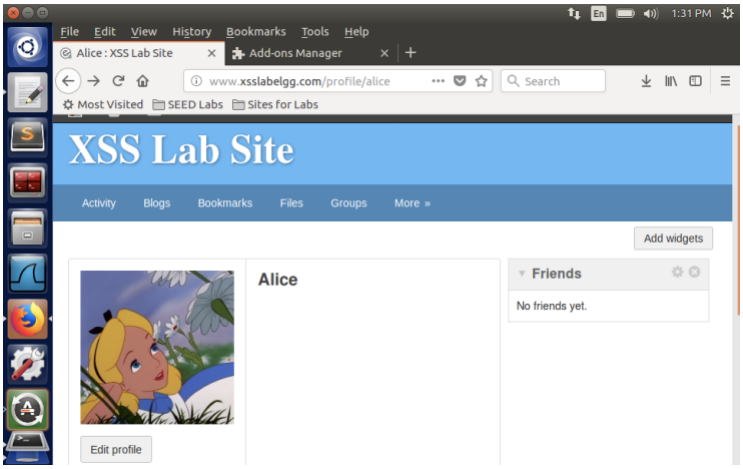

Figure 11: Alice has No Friends Initially

Now, we visit Samy's page from Alice's profile as shown in Fig 12. We have already placed the buggy JavaScript code in Samy's "About Me" which should trigger Alice to be friends with Samy without her knowing it.

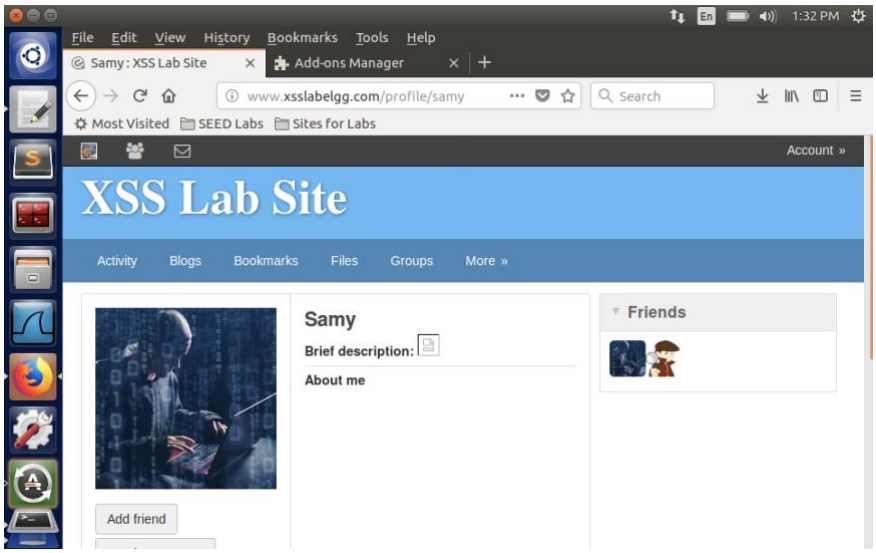

Figure 12: Alice visits Samy's profile

Going back to Alice's page. As indicated in Fig 13, we see that she and Samy are friends now. This is without Alice physically requesting it as a user. Thus, our bug performs.

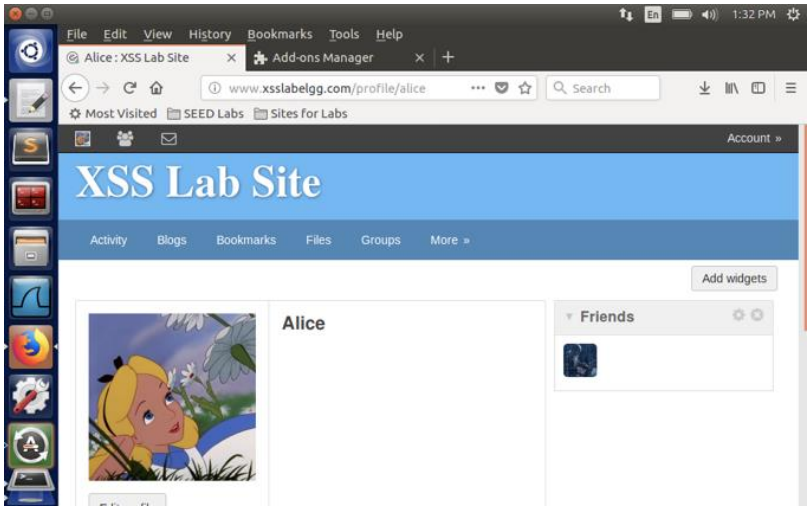

Figure 13: Malicious code makes Alice Samy's friend automatically

TASK 5: Modifying Victim's Profile

For this task we need to identify the HTTP request send to the Elgg server for editing a profile. We will try to do this using the HTTP Header Live extension of Firefox browser, as shown in Figure 14. We add some values to the fields in the profile page of Samy and then click the save button.

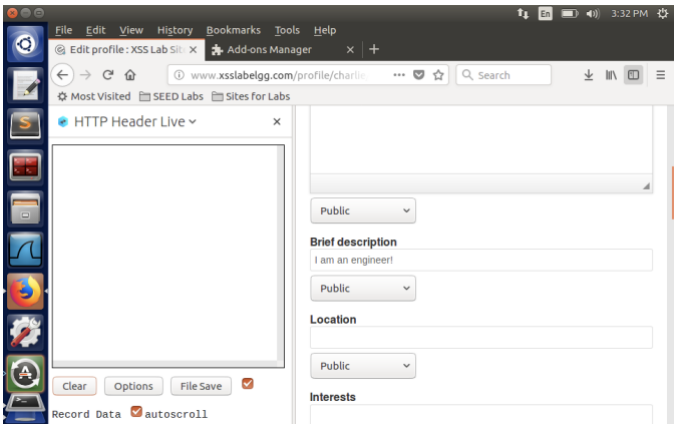

Figure 14: Using the HTTP Header Live and editing Profile

After saving, we obtain the header information in the extension, as shown in Figure 15.

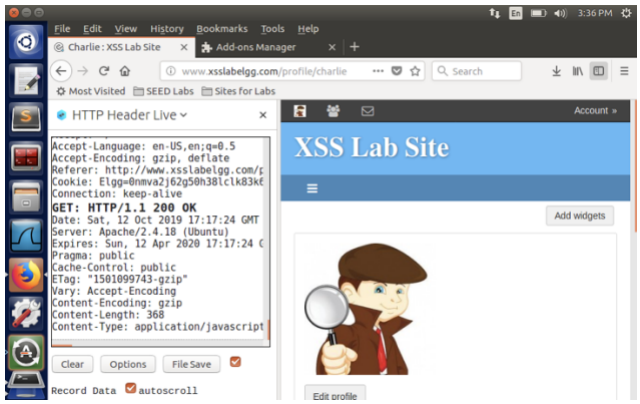

Figure 15: Obtaining the Header Information in the Extension

We observe that there are several fields, the main ones include the guid, description, the secret tokens and the user name as shown in the Figure 16. We also observe that the URL for editing profile is "www.xsslabelgg.com/action/profile/edit".

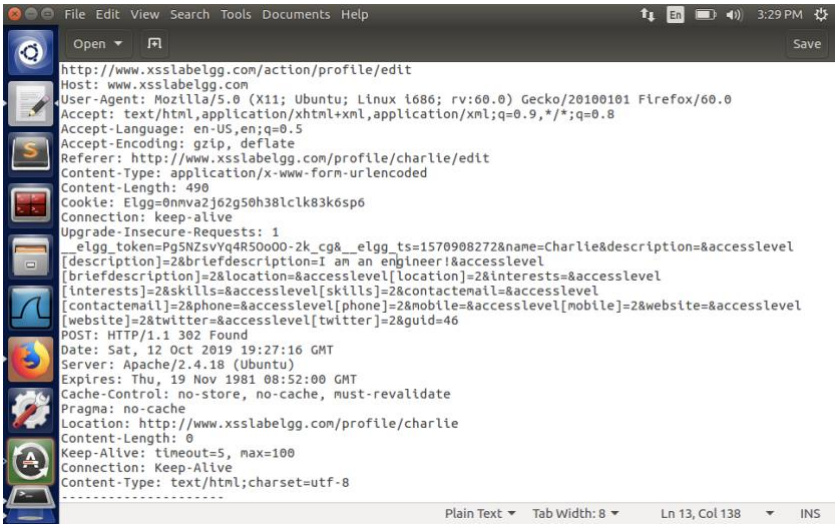

Figure 16: The content of HTTP POST request

We can now use this information to construct our JavaScript program. The JavaScript program should include the description that we will be adding which is ‘SAMY is MY HERO’, the secret tokens which we will obtain from the elgg like in the previous task, the POST action for the URL “www.xsslabelgg.com/action/profile/edit”, the Group id of the victim and the username of the victim will also be obtained from the elgg object. Hence we can construct our JavaScript program as shown in Figure 17.

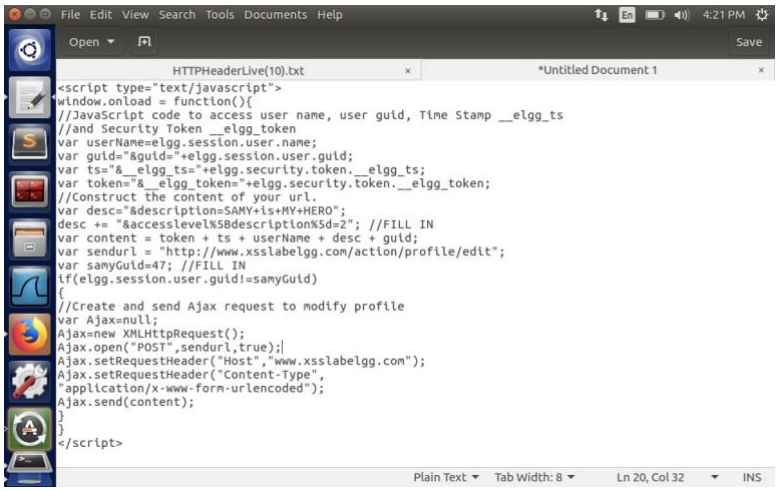

Figure 17: Malicious JavaScript Program

Now we will embed this JavaScript program in Samy’s profile page in the About me field as shown in Figure 18. The exploit succeeds when others check Samy’s profile and their About me changes to ‘SAMY is MY HERO’, as we have designed in the malicious JavaScript program.

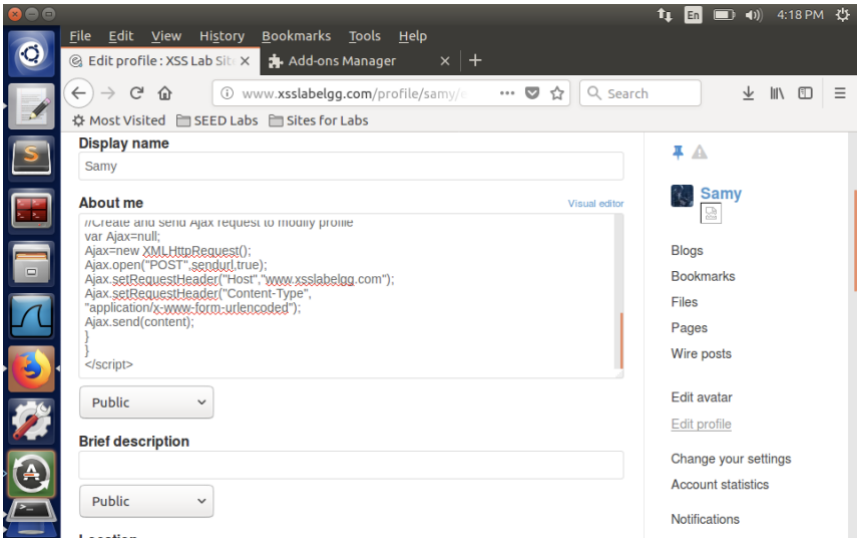

Figure 18

To demonstrate the attack, we login to Alice and use her account as a victim. Initially visiting her profile, we see, that her About me section does not display anything, as indicated in Fig 19.

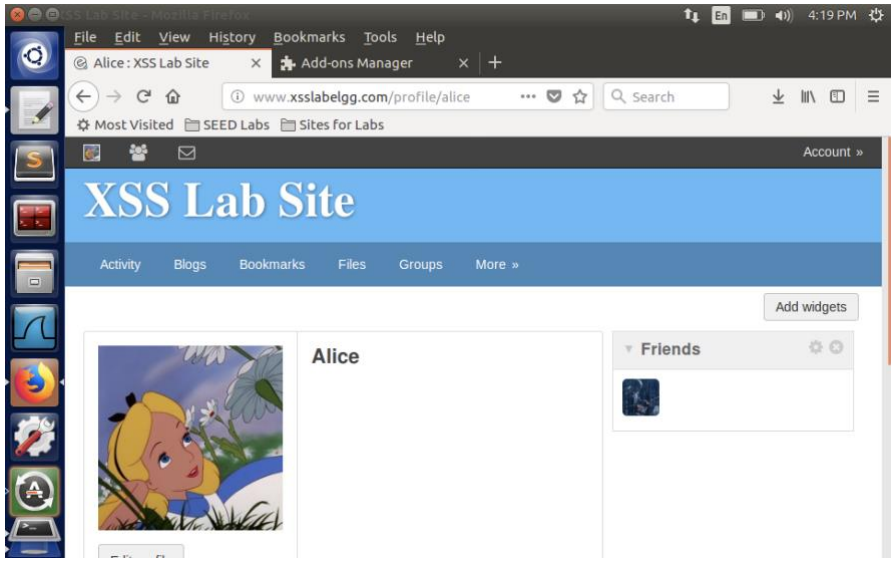

Figure 19: Nothing is present in Alice's profile

Now, we visit Samy's page from Alice's account as shown in Fig 20. Samy's About Me section has the attack to change the profile for Alice. The exploit that we have written is triggered and JavaScript program gets executed.

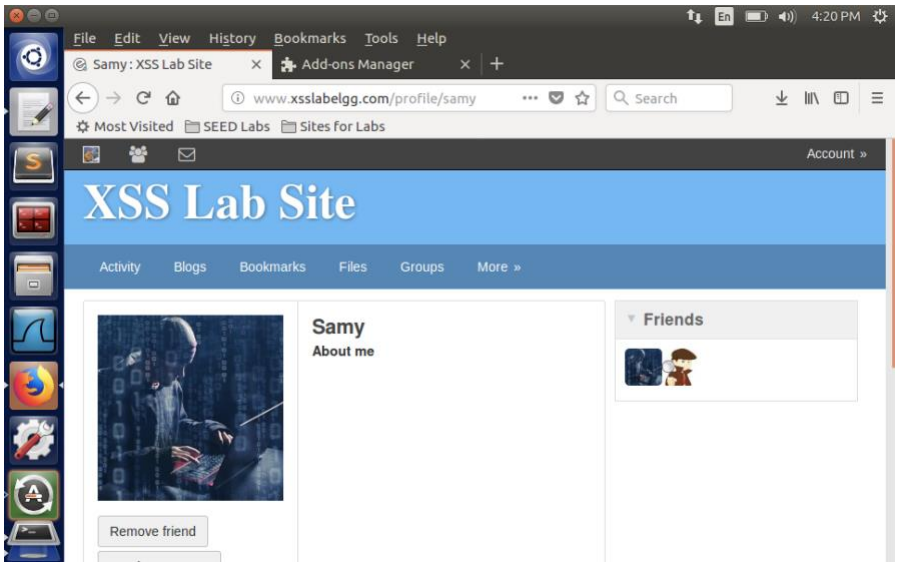

Figure 20: Visiting Samy's profile

Hence, looking at Alice's profile page again as indicated in Fig 21, we see that the About Me section is changed to display "SAMY is MY HERO"!

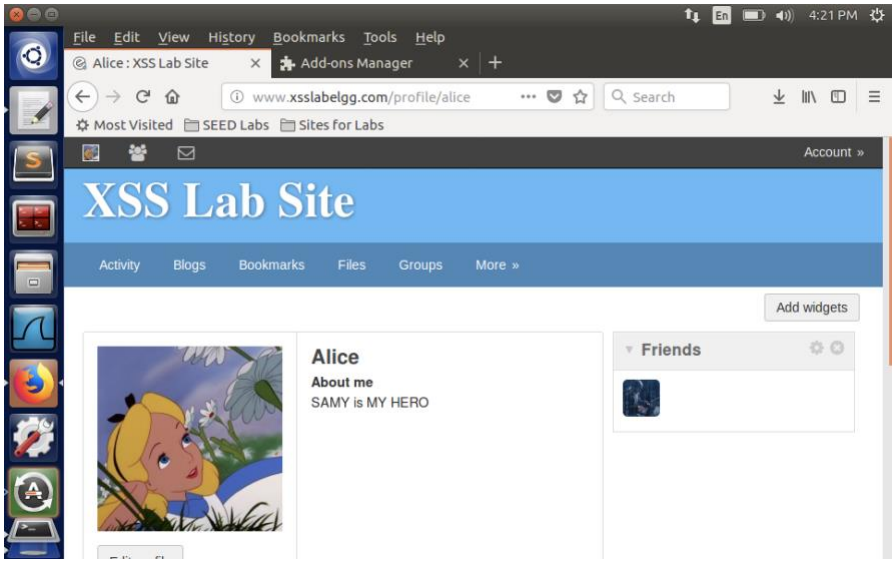

Figure 21: Alice webpage is hacked

Question 3: The if statement on the given line is needed to ensure Samy does not attack himself. The condition prevents Samy from overwriting the bug introduced in his About Me page. If overwritten, new users visiting Samy's page would no longer be susceptible to the attack as the attacking script is overwritten. We remove the conditional if statement and observe the changes from Fig 22.

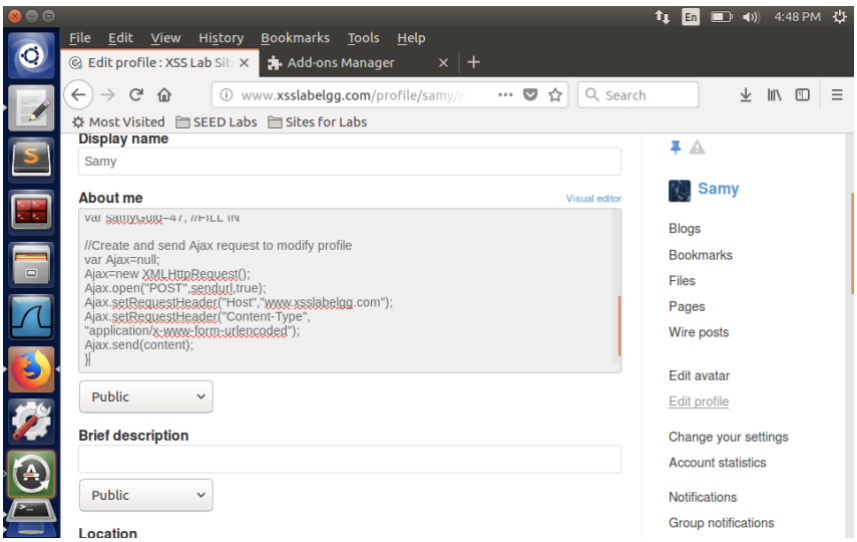

Figure 22:Removing the condition

Fig 23 indicates Samy's profile when we save the changes in his profile (eliminate the if condition that checks whether the attacked user id is Samy's). Note that the About Me section is blank.

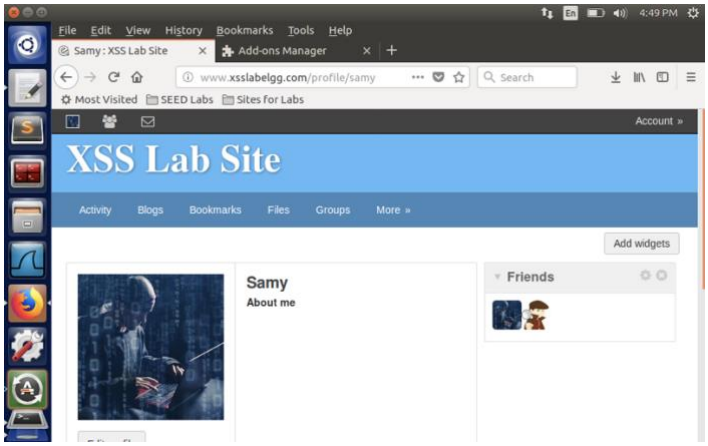

Figure 23: Samy's profile is empty

When Samy views his own profile, we observe that Samy's says "SAMY is MY HERO", the script that the victim's About Me section should display. This can be seen in Fig 24.

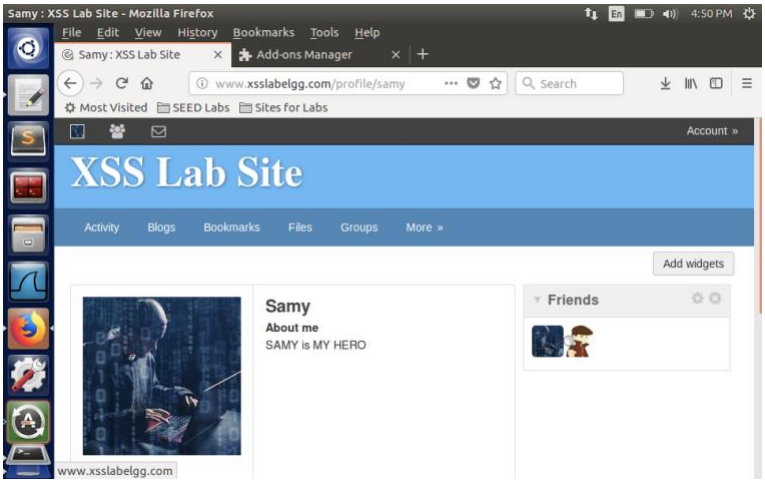

Figure 24: Samy viewing Samy's profile

Furthermore, when we open Samy's profile to edit details, we see that the JavaScript attack in Samy's About Me is overwritten by the attacking script. Basically, Samy attacked himself. Henceforth, all users visiting Samy's page would not be attacked by the vulnerability, as the This can be seen in Fig 25.

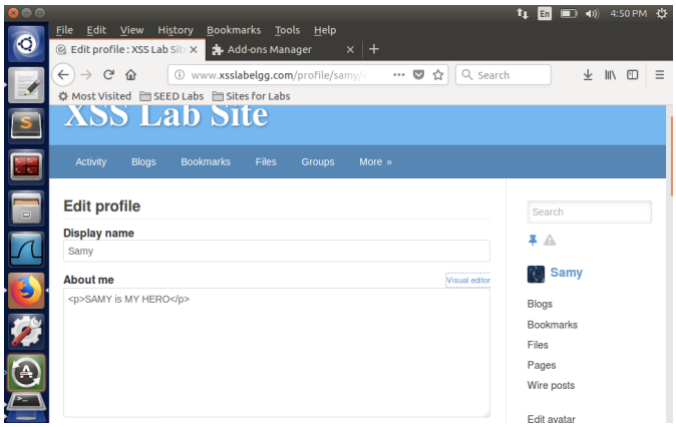

Figure 25: About me of Samy changed

TASK 6: Writing a Self-Propagating XSS Worm

To infect a visitor's profile on a malicious user, we have previously changed the "About Me" description field of the visitor. In order to self-propagate this worm, we design our code to replicate the attacking JavaScript code (initially located on the attacker's profile) on the "About Me" section of the victim.

Figure 26 depicts the changes we implement to the original attacking code. We observe that in the code, we create the opening and closing script tags in the variables "headerTag" and "tailTag" respectively, the header tag is given the id "worm" just like to original enclosing JavaScript code. The code then takes the value present inside the enclosing script referenced through the id "worm" and places it in the variable "jsCode". This is then appended to the end of the description of the HTTP request as "wormcode". We can also notice that "wormcode" is the encoded format of the combination of the "headerTag", "jsCode" and "tailTag". The encoding scheme used here is to replace the non-alphanumeric characters in the data with ASCII code of the character. This URL encoding is required as many characters like "?","/","#" etc have special meaning in a URL and hence they have to be encoded before transmitting over the URL in the HTTP request. The semantic logic for the implementation is to generate a copy of the attacking code and paste it in the About Me section of the victim.

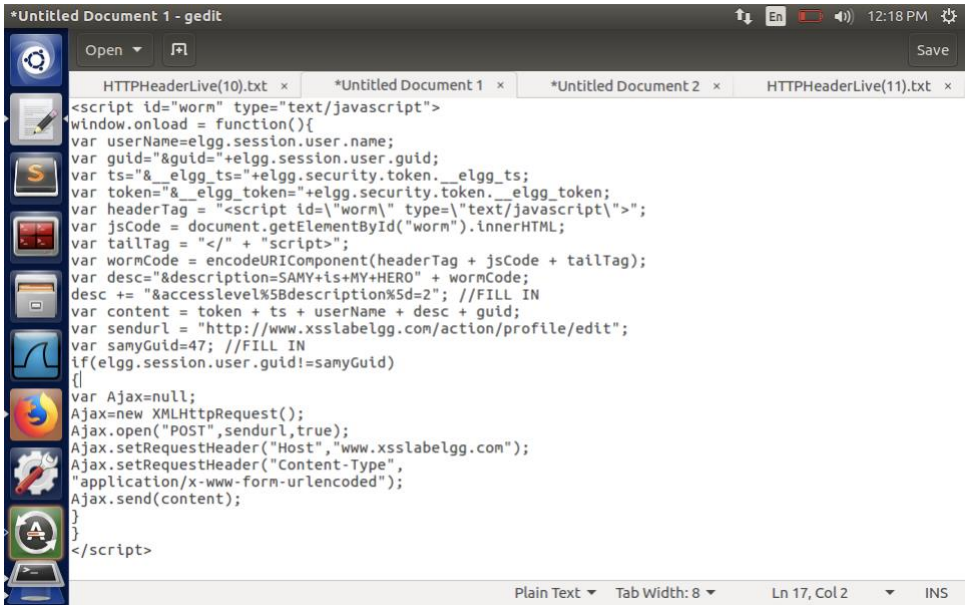

Figure 26: Self-propagating XSS worm – DOM approach program

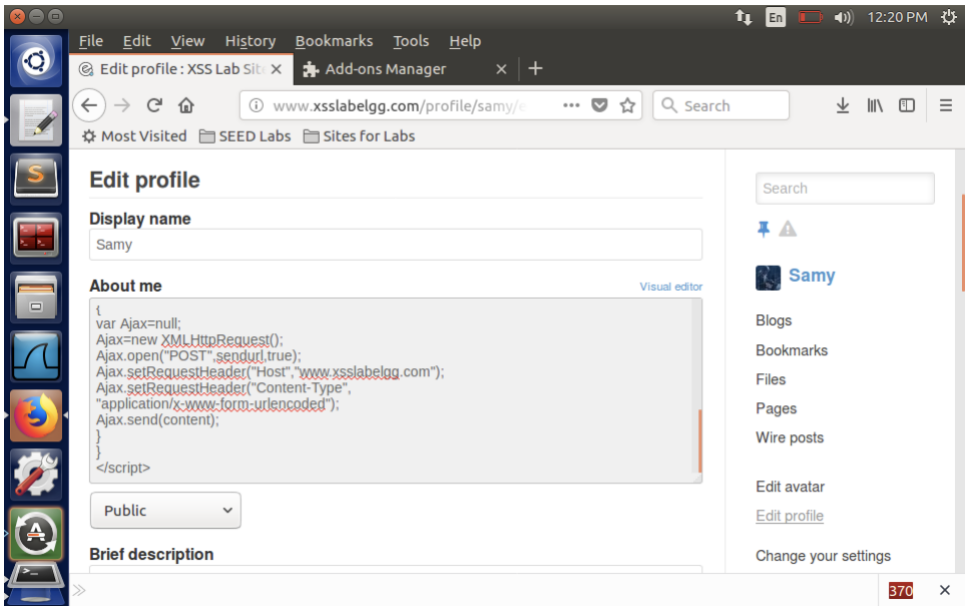

Figure 27: Placing the code in About me Field

Initially Alice's page has nothing in the About me section (Figure 28). Now when Alice visits Samy's page the XSS worm propagates to her profile. The HTTP request send the whole script through the URL and also the text "SAMY is MY HERO" which overwrites the description field as Alice visits Samy's page (Figure 29). We can see that the DOM object used in the script collects the value of the same script and pastes over the About me section of Alice's profile as seen Figure 30 and 31.

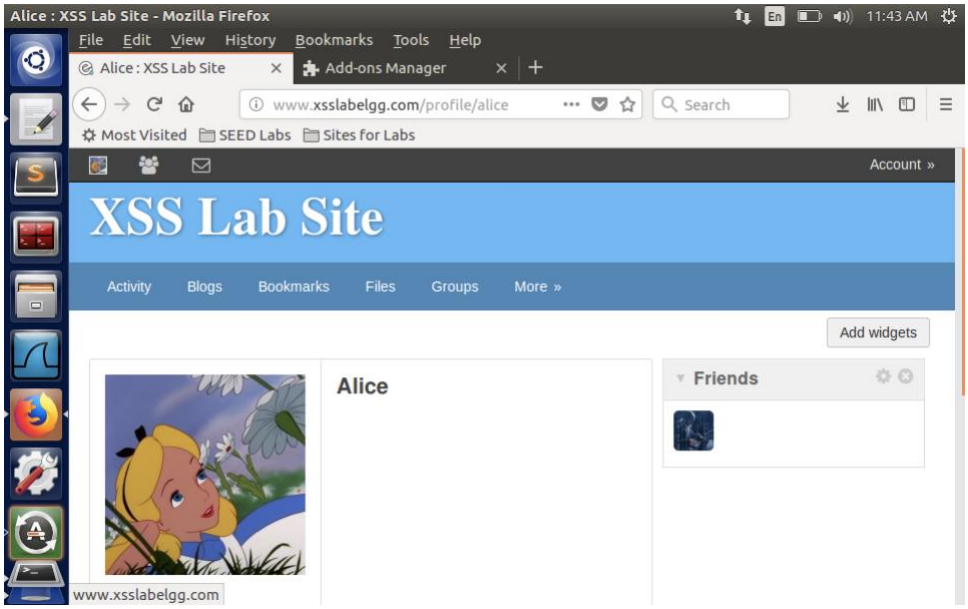

Figure 28: Alice Profile – Before Attack

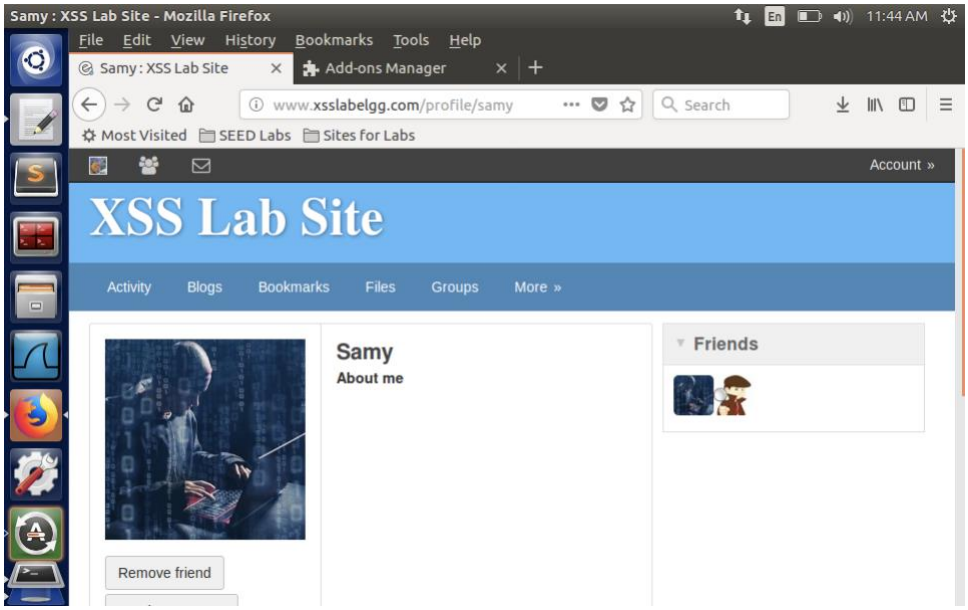

Figure 29: Visiting Samy's Profile

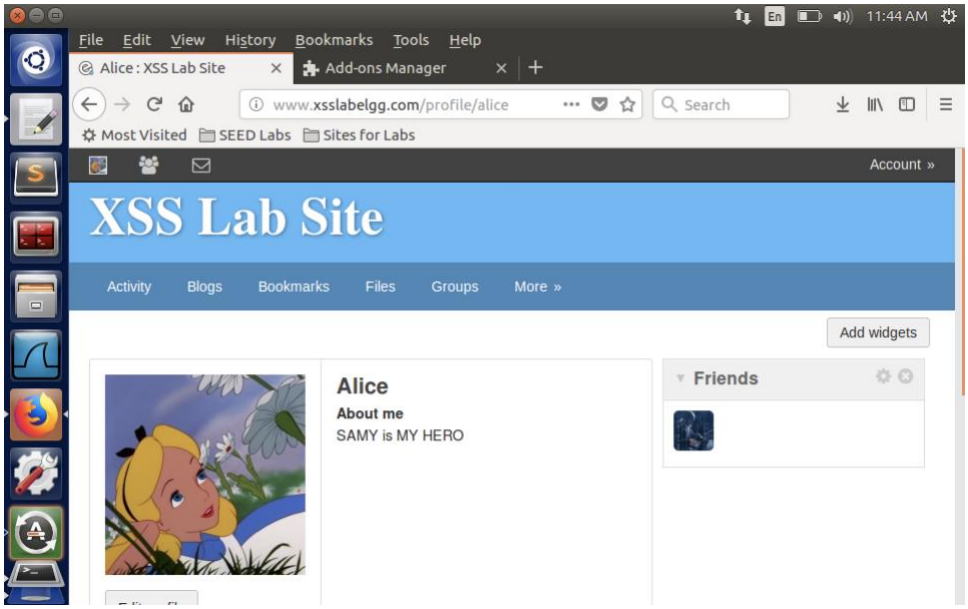

Figure 30: Alice Profile (1) – After XSS attack

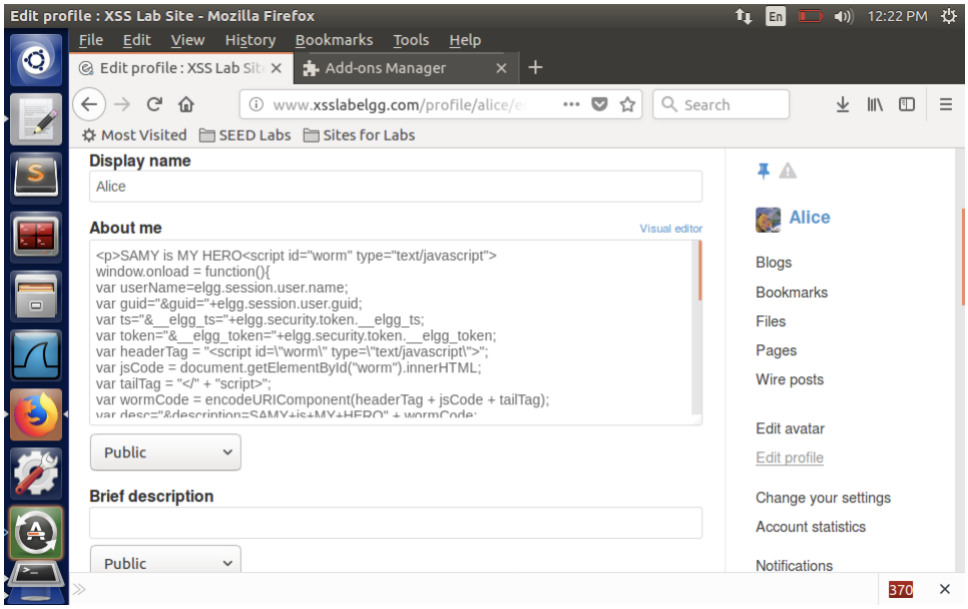

Figure 31: Alice's profile (2) – After XSS attack

Now when Boby visits Alice's profile, we can see the worm propagates in the similar fashion to Boby's account and the description is changed as "SAMY is MY HERO", as shown in Figure 34. We see the JavaScript is injected using DOM object as the previous case as shown in Figure 35 and this propagate to the next user who see Boby's profile.

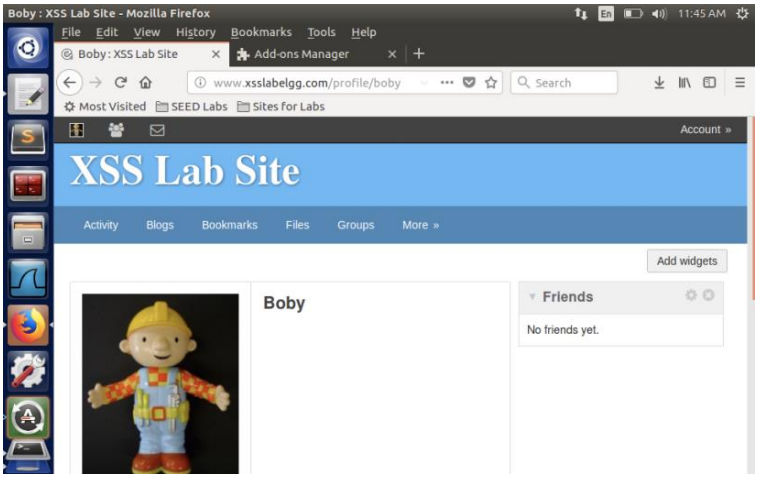

Figure 32: Boby's Profile – before XSS attack

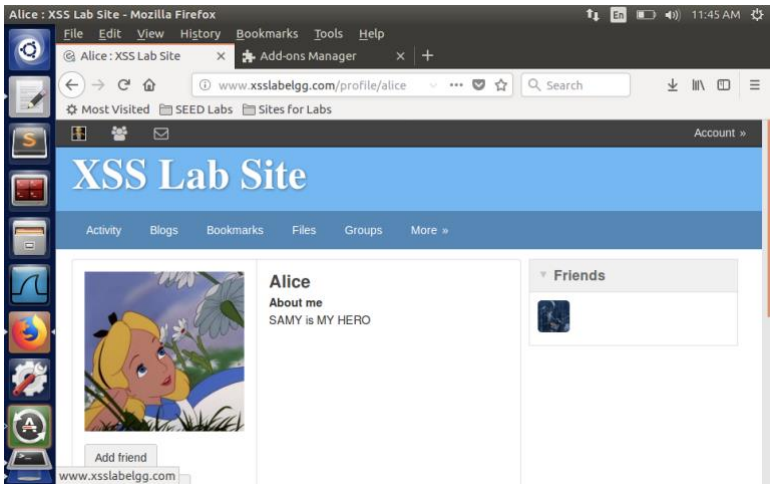

Figure 33: Boby views Alice's Profile

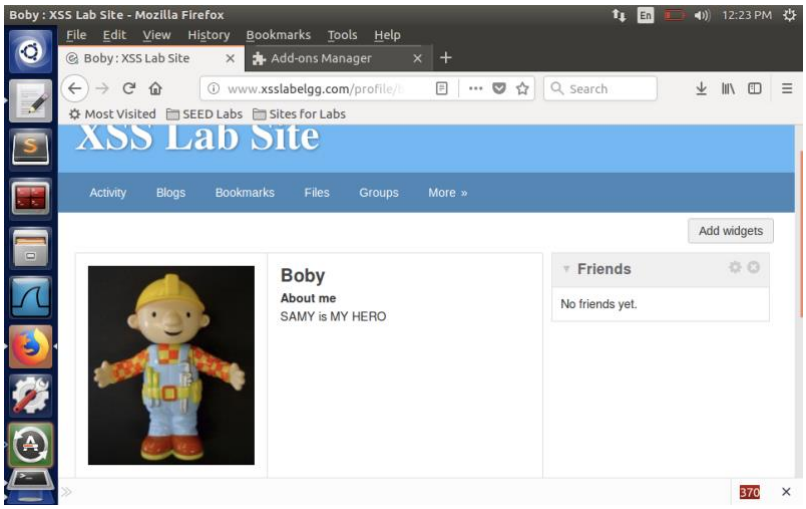

Figure 34: Boby's Profile (1) – After XSS attack

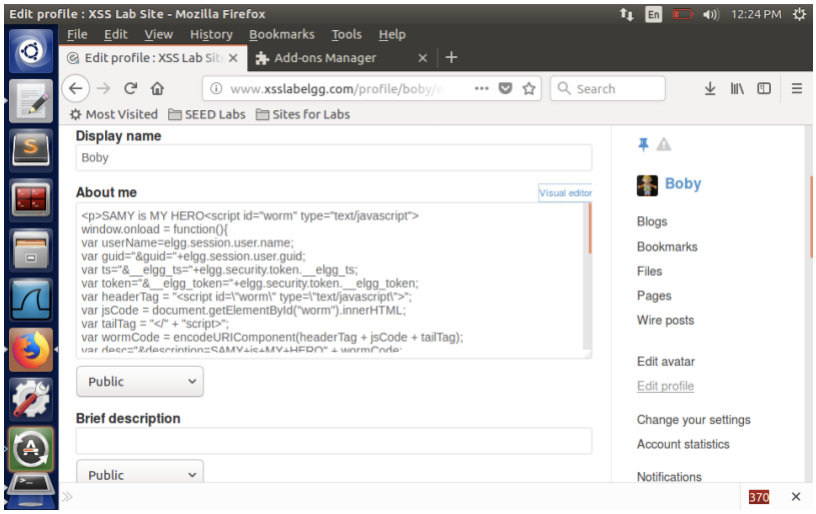

Figure 35: Boby's Profile (2) - After XSS attack

TASK 7: Countermeasures

For this task we need to apply the countermeasure to XSS which was deactivated initially. Elgg has a custim build security plugin HTMLawed. This can be found on Security and Spam plugin feature in the Administration section as shown in Figure 36. HTMLawed is a highly customizable PHP script to sanitize HTML against XSS attacks.

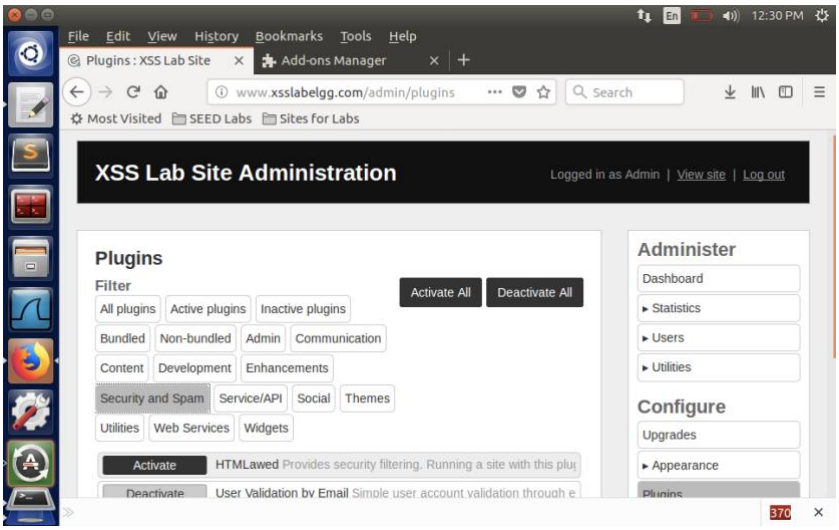

Figure 36: Activating the HTMLawed

Basically HTMLawed is a highly customizable PHP script to sanitize HTML against XSS attacks. Now going back to a victim's system, here Boby, we can see that the values are printed in the About me section includes the JavaScript code, this because the Script sanitized the embedded JavaScript and removed the script tag from the user input and hence it was treated as data. Hence the exploit fails as shown in Figure 39.

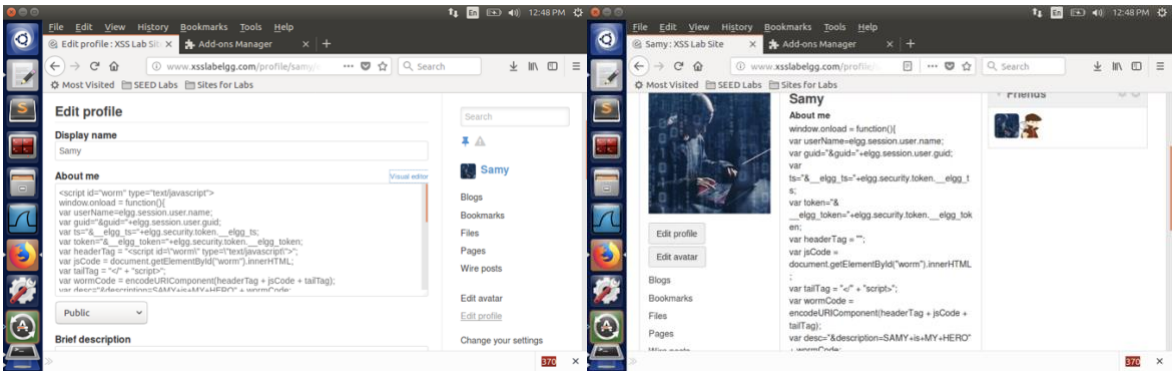

Figure 37 & 38: Checking Samy's profile (Attacker)

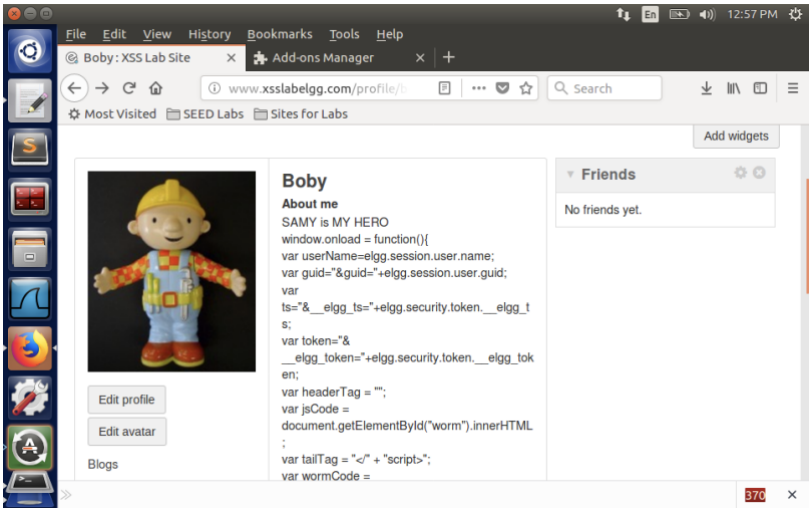

Figure 39: Checking Boby's profile (Victim)

Now we shall activate the second built-in PHP method for countering XSS attacks. We do this by uncommenting the "htmlspecialchars" function calls in the text.php (as shown in Figure 40), url.php (as shown in Figure 41), dropdown.php (as shown in Figure 42) and email.php (as shown in Figure 43) files in the "/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output/" directory (as shown in Figure 44).

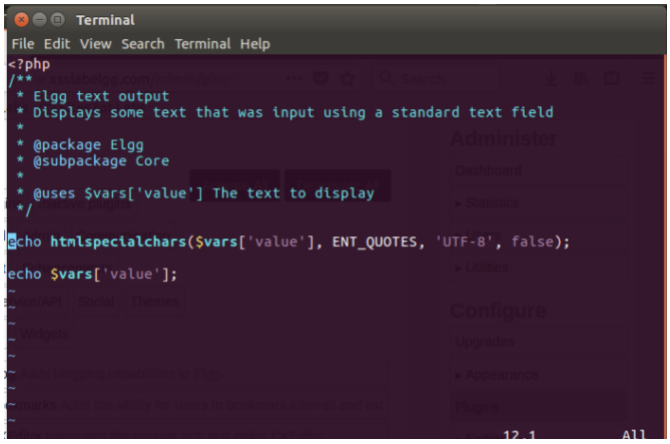

Figure 40: text.php

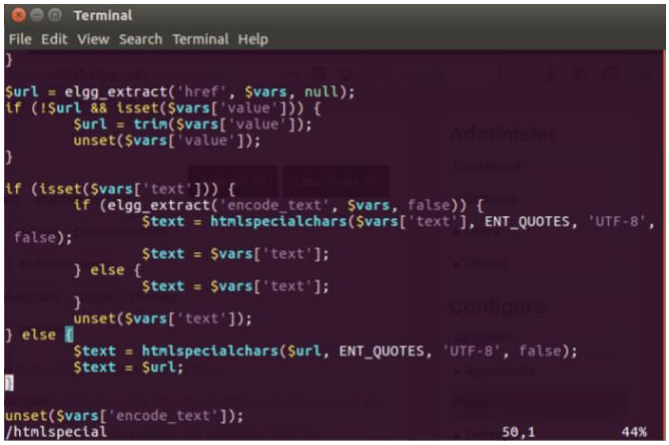

Figure 41:url.php

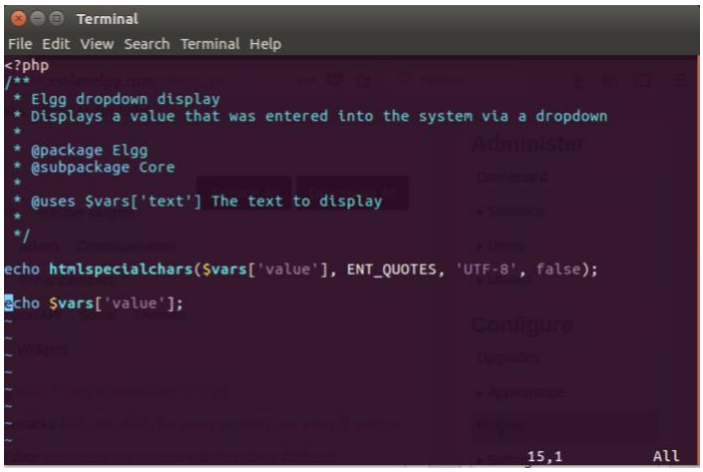

Figure 42: dropdown.php

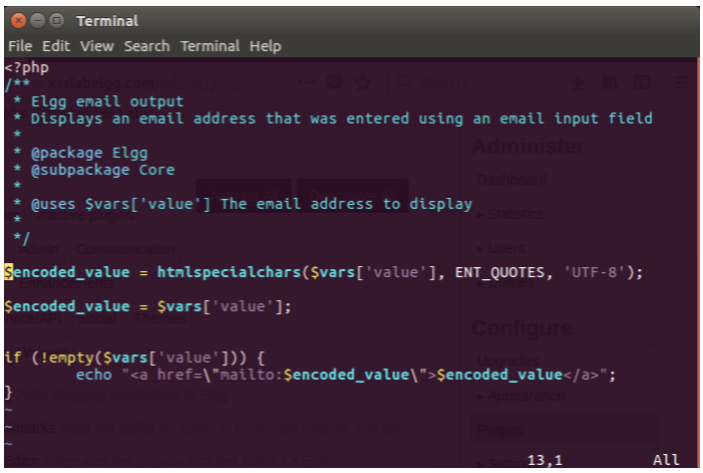

Figure 43: email.php

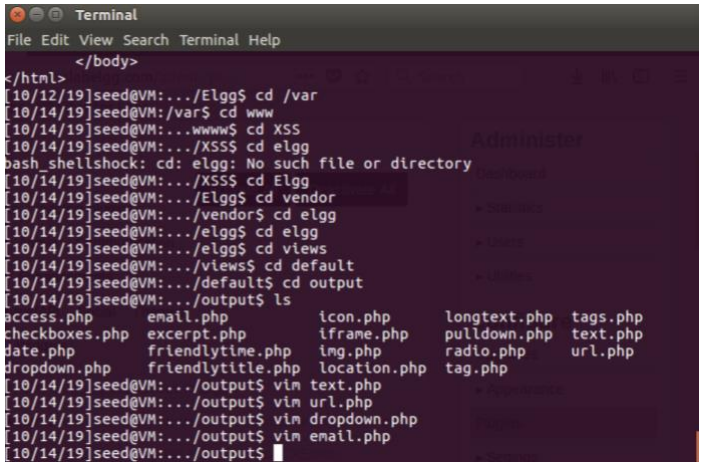

Figure 44: uncommenting the function calls

This countermeasure converts all the special characters to HTML entities. Now using the mentioned function call converts the special characters to HTML entities to preserve their meaning like "<" is represented as "&lt" and ">" is represented as "&gt". Now with the 2 countermeasures active we can see that the exploit fails as shown in the figure 45.

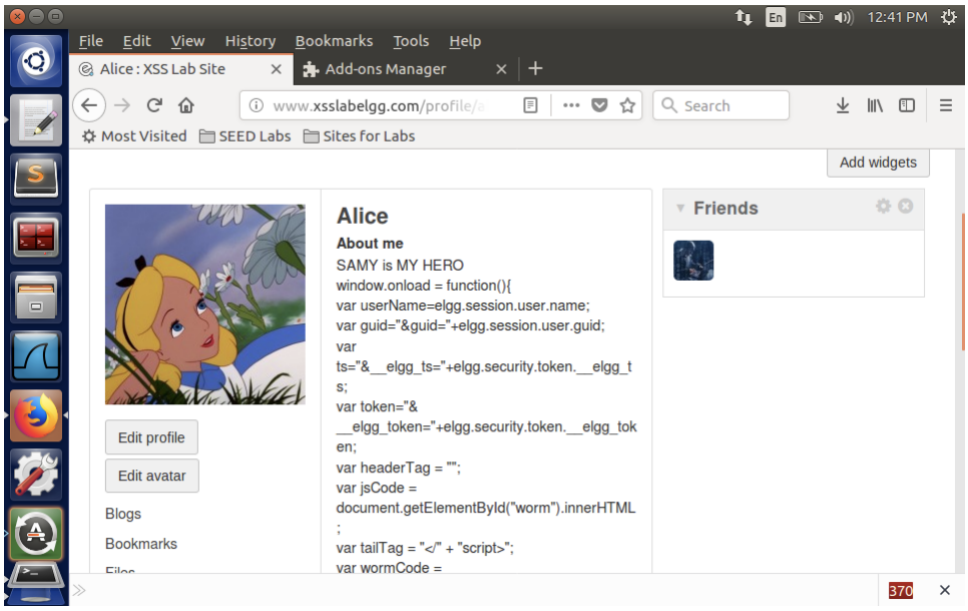

Figure 45: Viewing Alice's profile (Vicitm)

Figure 46 to 48 shows that exploit fails and prevents any propagation of the XSS worm after the activation of the countermeasures.

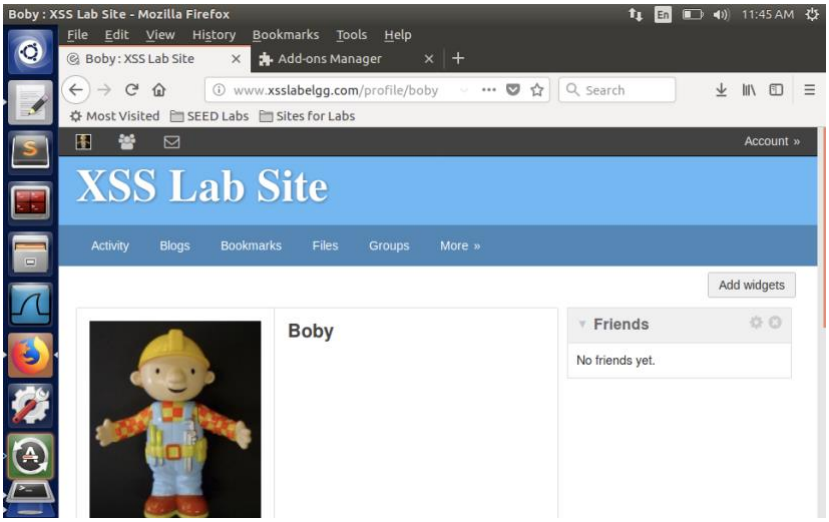

Figure 46: Before - Boby

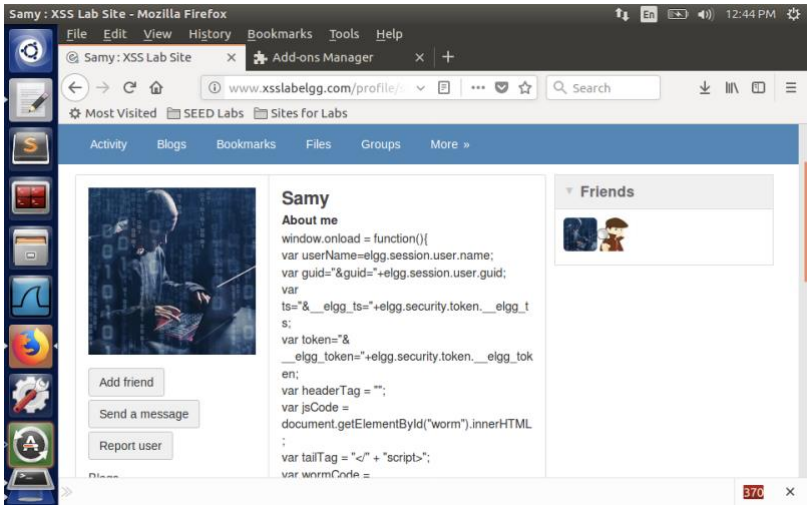

Figure 47: Viewing Samy's profile

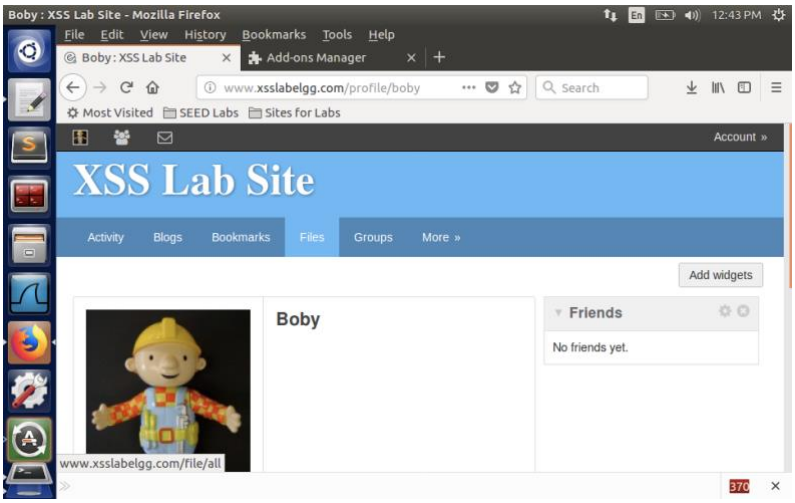

Figure 48: Countermeasure successful

PART 2: SQL INJECTION ATTACK LAB

TASK 0: Initial Setup

For this task we will be using Ubuntu version 16.04 with the preconfigured specifications. We will be using URL:www.seedlabbsqlinjection.com that is setup in the SEED lab for this exercise.

TASK 1: Get Familiar with SQL Statements

For this task we will be using MySQL as our relational database management system. Initially we login into the MySQL console using the command "*mysql -u root -pseedubuntu*" as shown in Figure 49.

```
Oct 14 11:52:08 VM systemd[1]: Starting LSB: Apache2 we
[10/14/19]seed@VM:~$
[10/14/19]seed@VM:~$ mysql -u root -pseedubuntu
mysql: [Warning] Using a password on the command line i
nterface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \
g.
Your MySQL connection id is 211
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, Oracle and/or its affiliates.
 All rights reserved.

Oracle is a registered trademark of Oracle Corporation
and/or its
affiliates. Other names may be trademarks of their resp
ective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the c
urrent input statement.
```

Figure 49: Accessing MySQL console

Once we login to the database, we can access the Users database and see the available tables as shown in Figure 50.

```
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the c
urrent input statement.

mysql> use Users;
Reading table information for completion of table and c
olumn names
You can turn off this feature to get a quicker startup
with -A

Database changed
mysql> show
    -> tables;
+----------------+
| Tables_in_Users |
+----------------+
| credential     |
+----------------+
1 row in set (0.00 sec)

mysql>
```

Figure 50: Using Users Database

Now we can see the data regarding the Alice using the select command as shown as in Figure 51.

```
mysql> select * from credential where name="Alice";
+----+-------+-------+--------+-------+----------+-----
--------+---------+-------+----------+-----------------
------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | Phon
eNumber | Address | Email | NickName | Password
                       |
+----+-------+-------+--------+-------+----------+-----
--------+---------+-------+----------+-----------------
------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |
        |         |       |          | fdbe918bdae83000
aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+-----
--------+---------+-------+----------+-----------------
------------------------+
1 row in set (0.00 sec)
```

Figure 51: Accessing Alice's data

TASK 2: SQL Injection Attack on SELECT Statement

TASK 2.1: SQL Injection Attack from webpage

For this task we will have to gain access to the web application as the administrator. We know that the username is “admin”, but we don’t know the password. We can exploit the web application using SQL injection, as shown in the Figure 52. The logic behind the exploit includes username as ‘admin’’ and then commenting the rest of the code using “#” symbol. The # will comment out the password verification in the SQL command in the database. Which gives us access to the web application with admin login as shown in Figure 53.

Figure 52: SQL injection to login as Administrator

Figure 53: Admin web page

TASK 2.2: SQL Injection Attack from command line

For this task we will have to gain access to the web application as the administrator using the command line. We can see the web page using the curl command. The curl command should be used to access the home page of the admin, hence the URL parameter should be www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27%23&Password, as shown in Figure 54. Where the unsafe_home.php is the home page of the user, and the username is set as 'admin' and ''' is encoded using URL encoding format as the %27 and the '#' is encoded as %23 in the URL. This is in similar fashion to Task 2.1, instead the data is obtained by using the curl command. The exploit is successful as shown in Figure 55,56.

```
[10/14/19]seed@VM:~$ curl www.SeedLabSQLInjection.com/u
nsafe_home.php?username=admin%27%23&Password
[2] 5086
<!--
SEED Lab: SQL Injection Education Web plateform
Author: Kailiang Ying
Email: kying@syr.edu
-->

<!--
SEED Lab: SQL Injection Education Web plateform
Enhancement Version 1
Date: 12th April 2018
Developer: Kuber Kohli

Update: Implemented the new bootsrap design. Implemente
d a new Navbar at the top with two menu options for Hom
e and edit profile, with a button to
logout. The profile details fetched will be displayed u
sing the table class of bootstrap with a dark table hea
```

Figure 54: curl command

```
<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, in
itial-scale=1, shrink-to-fit=no">

  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link href="css/style_home.css" type="text/css" rel="
stylesheet">

  <!-- Browser Tab title -->
  <title>SQLi Lab</title>
</head>
<body>
  <nav class="navbar fixed-top navbar-expand-lg navbar-
light" style="background-color: #3EA055;">
    <div class="collapse navbar-collapse" id="navbarTog
```

Figure 55: Obtaining the webpage(1)

```
e='button' id='logoffBtn' class='nav-link my-2 my-lg-0'
>Logout</button></div></nav><div class='container'><br>
<h1 class='text-center'><b> User Details </b></h1><hr><
br><table class='table table-striped table-bordered'><t
head class='thead-dark'><tr><th scope='col'>Username</t
h><th scope='col'>EId</th><th scope='col'>Salary</th><t
h scope='col'>Birthday</th><th scope='col'>SSN</th><th
scope='col'>Nickname</th><th scope='col'>Email</th><th
scope='col'>Address</th><th scope='col'>Ph. Number</th>
</tr></thead><tbody><tr><th scope='row'> Alice</th><td>
10000</td><td>20000</td><td>9/20</td><td>10211002</td><
td></td><td></td><td></td><td></td></tr><tr><th scope='
row'> Boby</th><td>20000</td><td>30000</td><td>4/20</td
><td>10213352</td><td></td><td></td><td></td><td></td><
/tr><tr><th scope='row'> Ryan</th><td>30000</td><td>500
00</td><td>4/10</td><td>98993524</td><td></td><td></td>
<td></td><td></td></tr><tr><th scope='row'> Samy</th><t
d>40000</td><td>90000</td><td>1/11</td><td>32193525</td
><td></td><td></td><td></td><td></td></tr><tr><th scope
='row'> Ted</th><td>50000</td><td>110000</td><td>11/3</
```

Figure 56: Obtaining the webpage (2)

TASK 2.3: Append a new SQL Statement

We can attempt this task by combining 2 SQL statements together using ‘;’. The logic in place is to concatenate 2 SQL statements and separate them using ‘;’. Hence we inject the value “ admin' OR 1=1 AND password='1234';delete from credential where name='Alice';# “, as shown in Figure 57. But the exploit fails as shown in Figure 58. This is because in PHP’s mysqli extension, the mysqli::query() API doesn’t allow multiple queries to run in the database server. Multiple queries can be allowed if PHP were to use the mysqli::multi_query() instead.

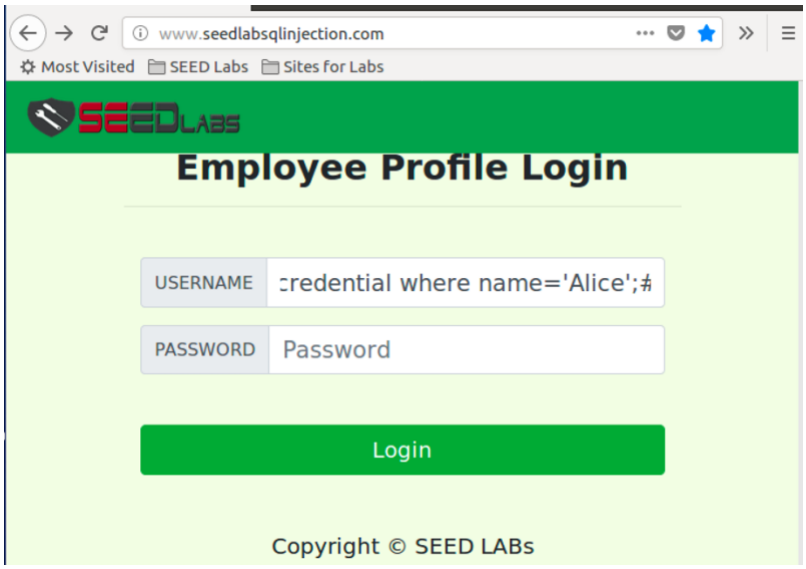

Figure 57: 2 SQL statements together

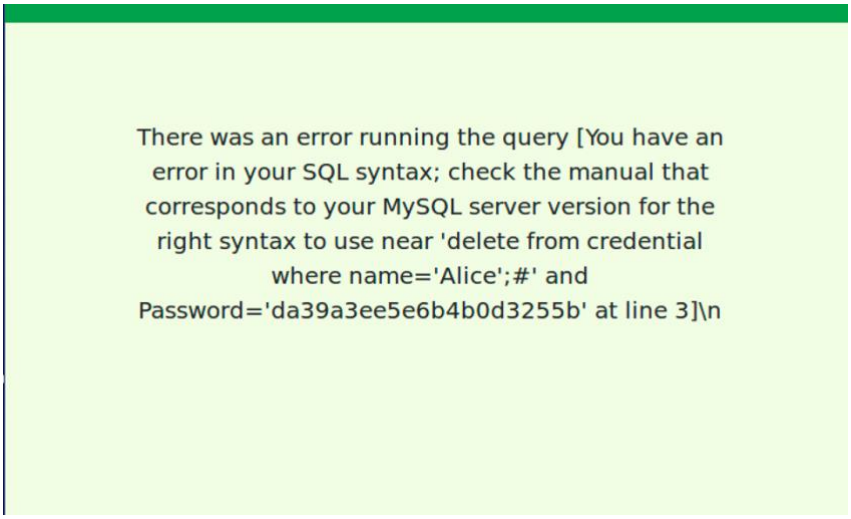

Figure 58: Exploit failing

TASK 3: SQL Injection Attack on UPDATE Statement

For this task we observer the update command used in the PHP file, and we find that there is a SQL vulnerability that can be exploited.

TASK 3.1: Modify your own salary

For this task we will need to update the salary of Alice to a higher amount. We initially observe that Alice has a salary of 20000 and also that Alice can edit her profile as shown in Figure 59.

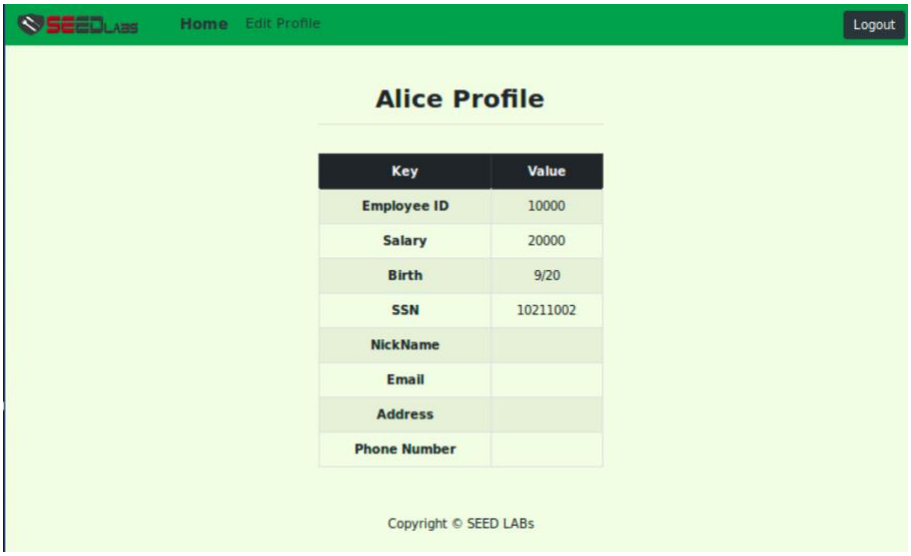

Figure 59: Alice Profile

We now inject the code “Alice', Salary=80000 where name='Alice'#” into the NickName field in the Alice’s Edit Profile as shown in the figure 60. This injected command will go to the update command in the PHP file and removes sets the NickName as Alice and the Salary as 80000 only where the name is Alice, this updates all the rows in the credential table with the name Alice with a Salary of 80000. The # sign comments the rest of the SQL command at the end.

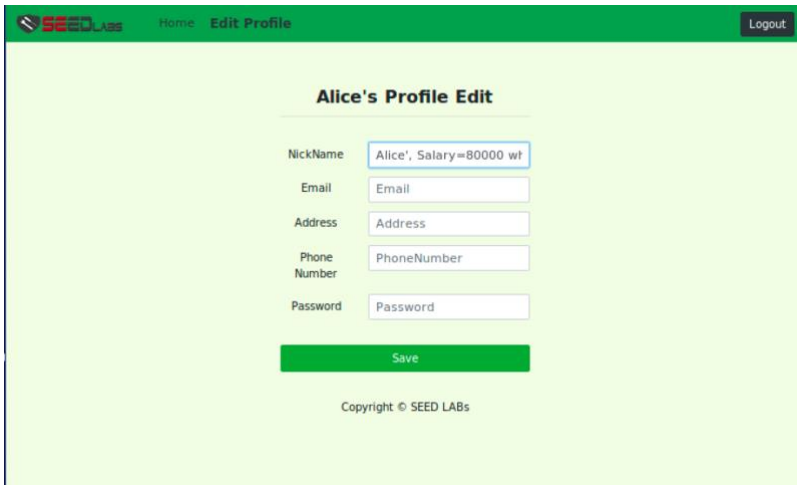

Figure 60: Injecting in Alice’s Edit Profile

Figure 61 shows that the exploit succeeded since the salary was updated to 80000.

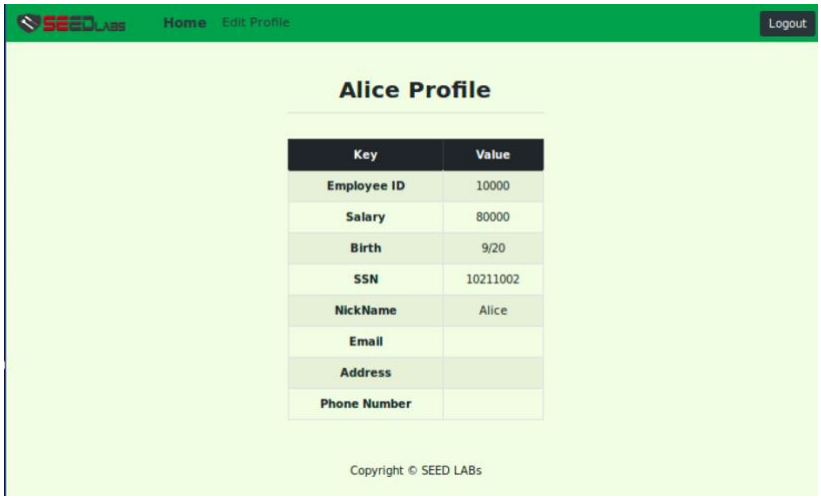

Figure 61: Updated Profile

TASK 3.2: SQL Injection Attack on UPDATE Statement

The task involves changing the Salary of Boby, using SQL injection. From our initial observation we recognize we update values in the credential table by specifying the value and the where condition. Hence we will inject the code "Alice', Salary=1 where name='Boby'#" into the Nickname field as shown in Figure 62. The nickname is set as null, and the salary is set as 1 for all the rows in the credential table where the name is "Boby". The rest of the SQL query is commented out using the #. Hence this is used to set the Salary as 1 for Boby.

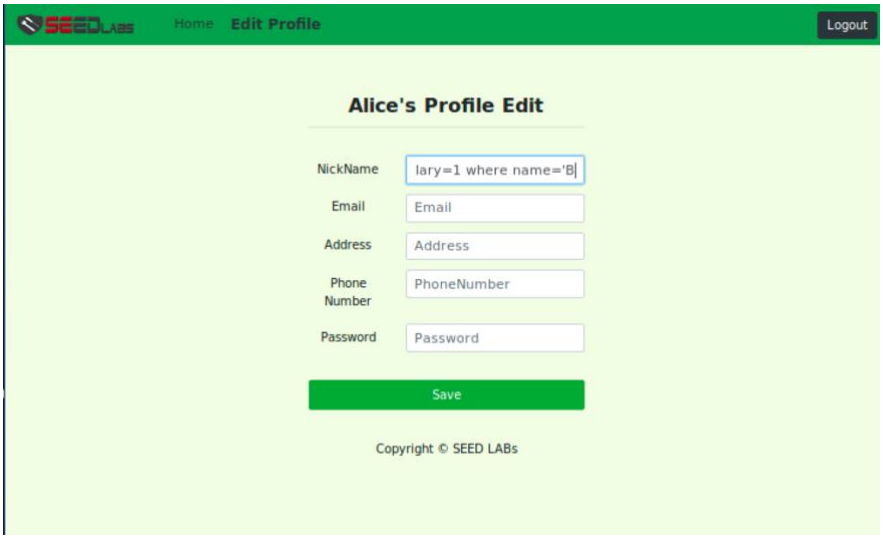

Figure 62: Alice's Edit profile

The exploit is successful as shown in Figure 63 as Boby's profile shows his Salary as 1.

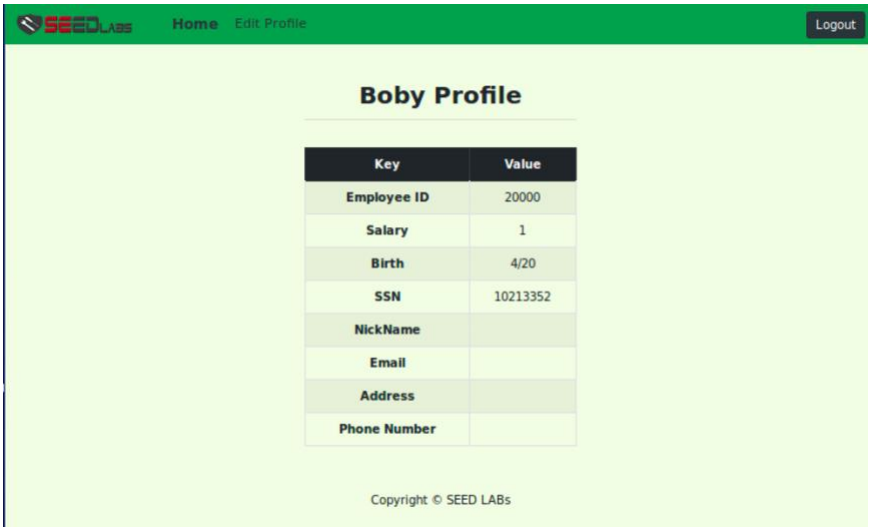

Figure 63: Exploit successful

TASK 3.3: Modify other people' password

This task involves us to change password of another user. After successfully executing this task, we have unlimited access of the victim's account leading to greater damage.

We know that passwords are not stored with their original values in the SQL database but instead recorded as the SHA1 hash value of the password string. To successfully execute the attack, instead of changing the password as we changed the salary in the above example, we will store the SHA1 value of the string we intend to store as the password.

Keeping the new password as "hello", we can obtain its SHA1 equivalent by executing:

echo -n "hello" | sha1sum | awk '{print $1}'

We correspondingly get aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d. This we can pass as the new password. Thus, we can set the password as "hello".

```
[10/14/19]seed@VM:.../SQLInjection$ echo -n "hello" | s
ha1sum | awk '{print $1}'
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
[10/14/19]seed@VM:.../SQLInjection$
```

Figure 64: Finding the Hash of hello

We execute the attack as the change salary attack in Task 3.2, but we change the input for password as the SHA1 string obtained above.

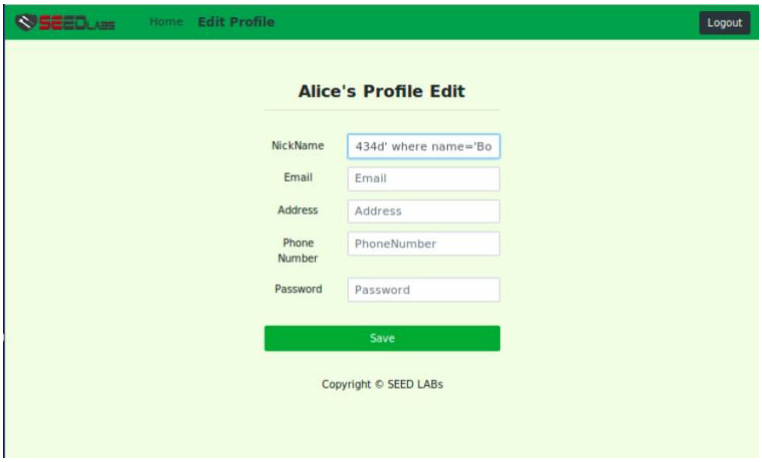

Figure 65:Injecting the hash password in the SQL command

Now, we attempt to login to Boby's profile using the credentials USERNAME = Boby and PASSWORD = hello.

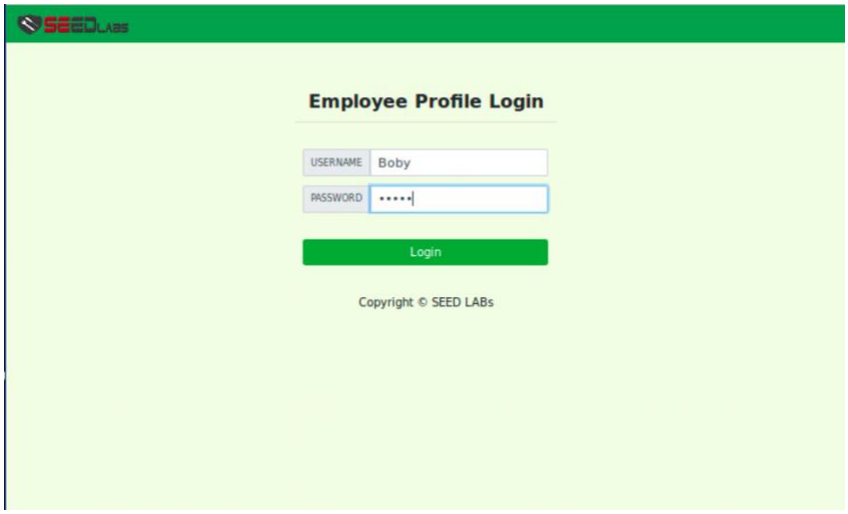

Figure 66: Using hello as the password to access Boby's account

As you can see, we are granted access to Boby's account. This is extremely dangerous as we can now mess with all the account details and activities of Boby, and at the same time lock him out of his own account such that he cannot undo the damage we have done.

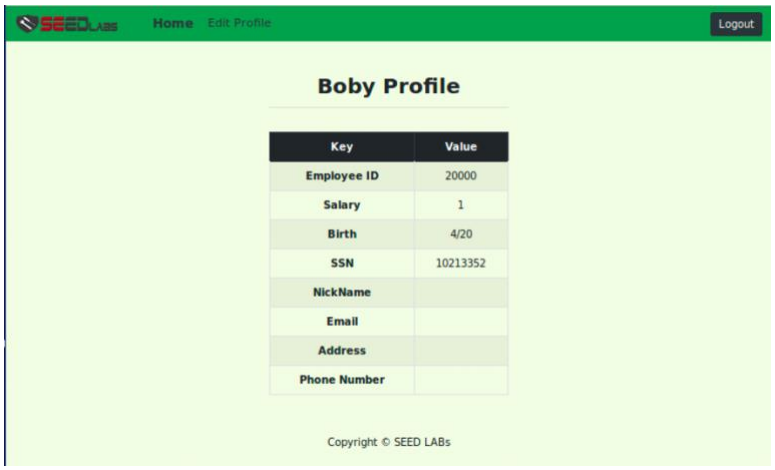

Figure 67: Exploit succeeds as the password hello works

Task 4: Countermeasure — Prepared Statement

In the above tasks, we employ an unsafe environment and execute our tasks over it. For this lab, we will switch to the safe environment using prepared statements and check if we can exploit the environment again.

The common issue with SQL statements is that they cannot differentiate between code and data. This leaves it vulnerable to attackers as they can inject SQL commands under the mask of data inputs and exploit the system. Prepared statements avoid the issue of plugging data directly into the compilation step, and hence avoid the issue of exploit through SQL statements in the data. They are fed in the compilation steps to generate a pre-compiled query which takes data input. Since the compilation does not receive and process faulty data, the problem of injection is solved.

The "safe" files on the "*/var/www/SQLInjection*" location house the SQL code updated with the prepared statements. We change all locations that point/use the unsafe files to the safe files.

Therefore, changes are:

unsafe_home --> safe_home

unsafe_edit_backend --> safe_edit_backend

Note that the unsafe_edit_frontend file does not need to be changed because it does not house the compilations. As long as it points to the safe_edit_backend file, we can be assured that the input data will pass to the prepared statements and not the compiler.

The files with changes include:

- Index.html

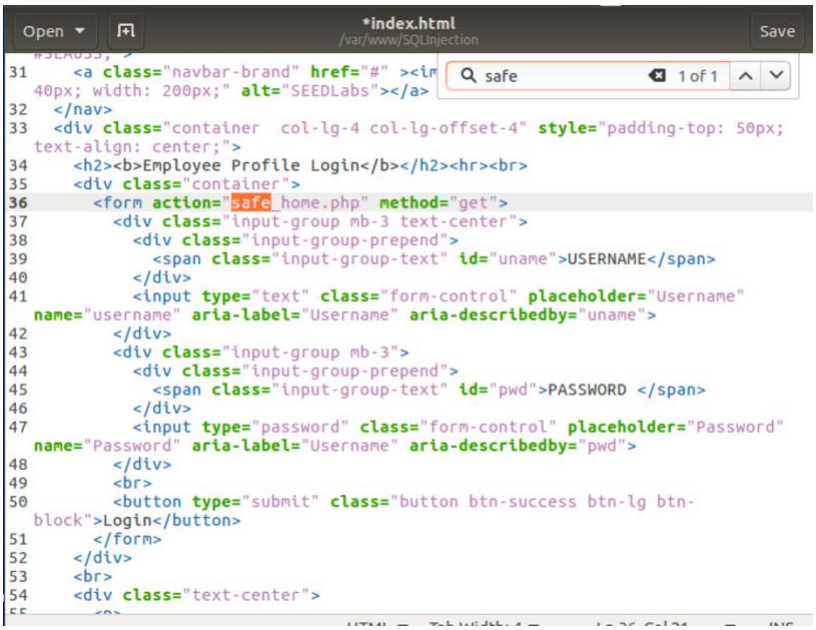

Figure 68: Index.html

- unsafe_edit_frontend.php

```
Open   *unsafe_edit_frontend.php   Save
/var/www/SQLInjection
<div class="container col-lg-4 col-lg-offset-4 text-center"
style="padding-top: 50px; text-align: center;">
    <?php
    session_start();
    $name=$_SESSION["name"];
    echo "<h2><b>$name's Profile Edit</b></h1><hr><br>";
    ?>
    <form action="safe_edit_backend.php" method="get">
      <div class="form-group row">
        <label for="NickName" class="col-sm-4 col-form-label">NickName</
label>
        <div class="col-sm-8">
          <input type="text" class="form-control" id="NickName"
name="NickName" placeholder="NickName" <?php echo "value=$nickname";?> >
        </div>
      </div>
      <div class="form-group row">
        <label for="Email" class="col-sm-4 col-form-label">Email</label>
        <div class="col-sm-8">
          <input type="text" class="form-control" id="Email" name="Email"
placeholder="Email" <?php echo "value=$email";?>>
        </div>
      </div>
      <div class="form-group row">
        <label for="Address" class="col-sm-4 col-form-label">Address</label>
        <div class="col-sm-8">
          <input type="text" class="form-control" id="Address"
name="Address" placeholder="Address" <?php echo "value=$address";?>>
        </div>
      </div>
```

Figure 69: unsafe_edit_frontend.php

- safe_edit_backend.php

```
Open   *safe_edit_backend.php   Save
/var/www/SQLInjection
  $conn = getDB();
  // Don't do this, this is not safe against SQL injection attack
  $sql="";
  if($input_pwd!=''){
    // In case password field is not empty.
    $hashed_pwd = sha1($input_pwd);
    //Update the password stored in the session.
    $_SESSION['pwd']=$hashed_pwd;
    $sql = $conn->prepare("UPDATE credential SET
nickname= ?,email= ?,address= ?,Password= ?,PhoneNumber= ? where ID=$id;");
    $sql->bind_param("sssss",$input_nickname,$input_email,$input_address,
$hashed_pwd,$input_phonenumber);
    $sql->execute();
    $sql->close();
  }else{
    // if passowrd field is empty.
    $sql = $conn->prepare("UPDATE credential SET
nickname=?,email=?,address=?,PhoneNumber=? where ID=$id;");
    $sql->bind_param("ssss",$input_nickname,$input_email,$input_address,
$input_phonenumber);
    $sql->execute();
    $sql->close();
  }
  $conn->close();
  header("Location: safe_home.php");
  exit();
  ?>

</body>
</html>
```

Figure 70: safe_edit_backend.php

Now, we restart the server to make note of our changes. Without the reload, the changes would not be reflected in the behavior of the application. Thus, injections might still be possible.

Using the command "*sudo service apache2 reload*", we reload the apache server for the changes.

```
[10/14/19]seed@VM:.../SQLInjection$ sudo service apache
2 reload
[10/14/19]seed@VM:.../SQLInjection$
```

Figure 71: reloading apache2 server

Now we attempt to replicate the attacks executed before. The simplest would be to access the admin without entering a password. As before, we attempt this through injection by typing the username as "*admin' #*".

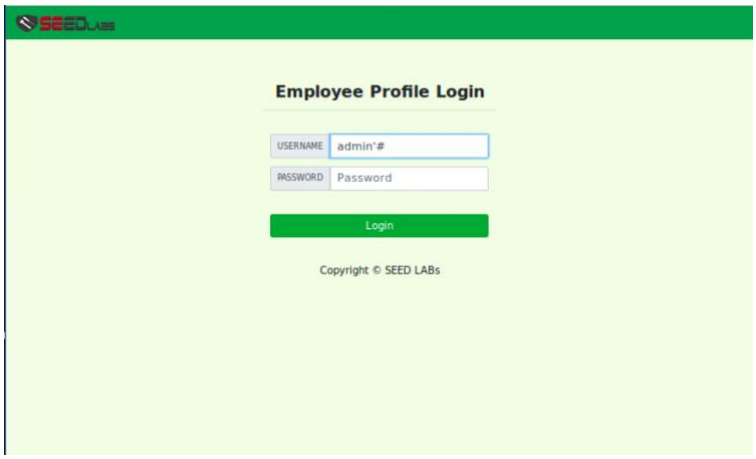

Figure 72: repeating the exploit

We note that the attack does not work and returns an error saying that the username we entered does not exist. Thus, we can conclude the command we entered in the username field is processed as a normal string (input data) and not as the SQL code. This is because we have introduced the prepared statements in the new code, which disallow the user input to directly go to the compiler.

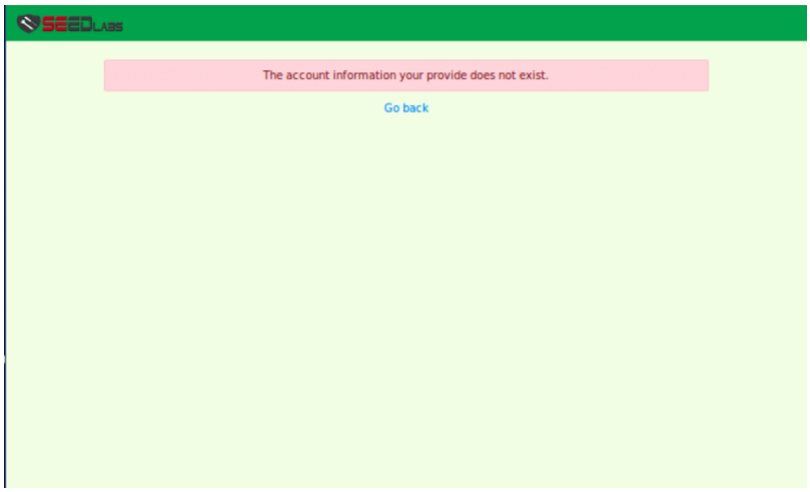

Figure 73: Exploit fails