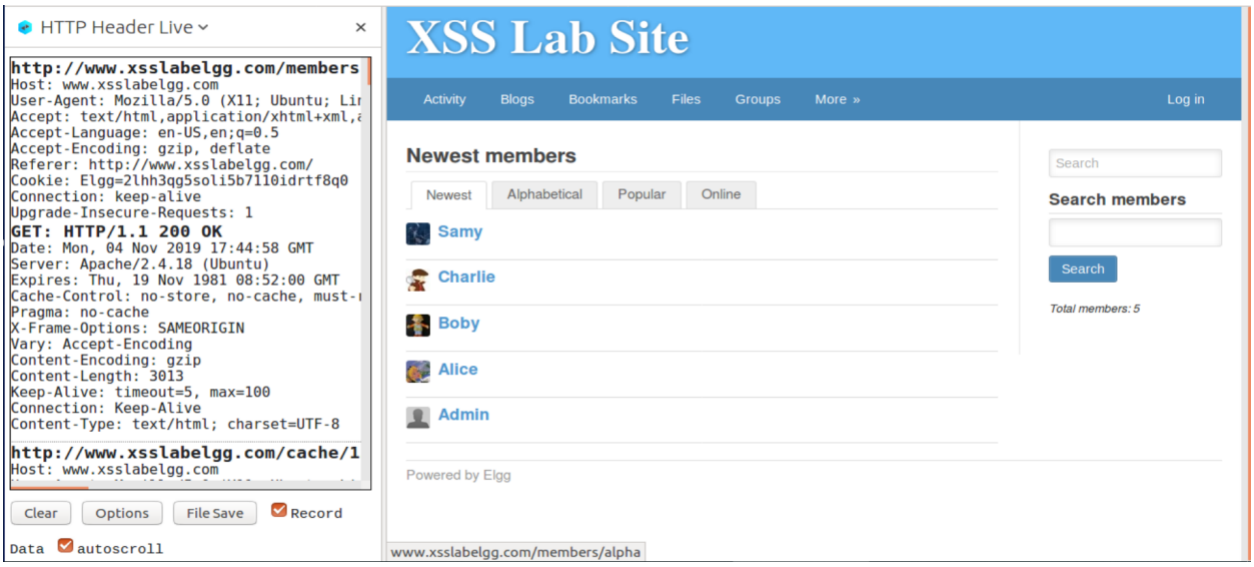
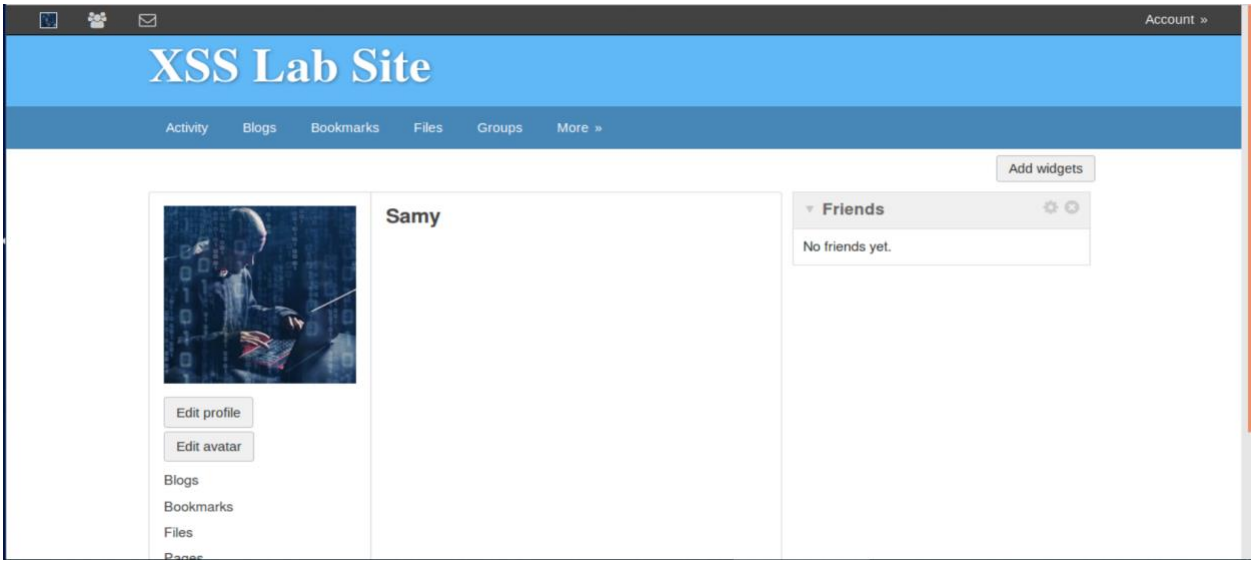
LAB 10: CROSS-SITE SCRIPT (XSS) ATTACK LAB

```
[11/04/19]seed@VM:~$ cd /var/www/XSS/Elgg/
[11/04/19]seed@VM:.../Elgg$ sudo service apache2 start
[11/04/19]seed@VM:.../Elgg$
```

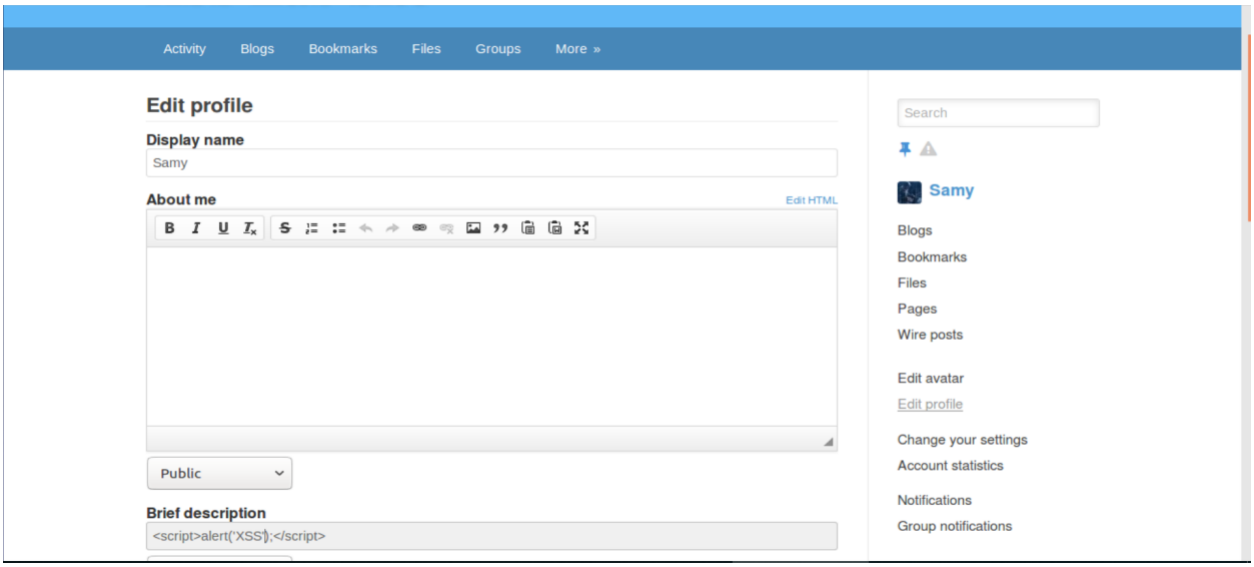


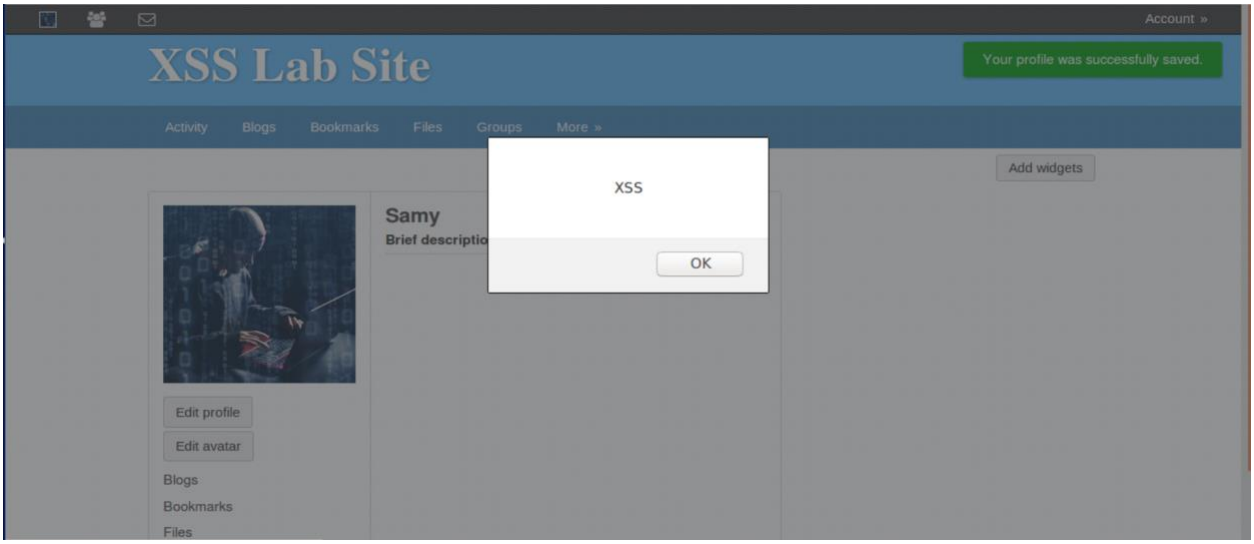
Task 1: Posting a Malicious Message to Display an Alert Window

The below screenshot shows Samy's profile without the attack code in the brief description.

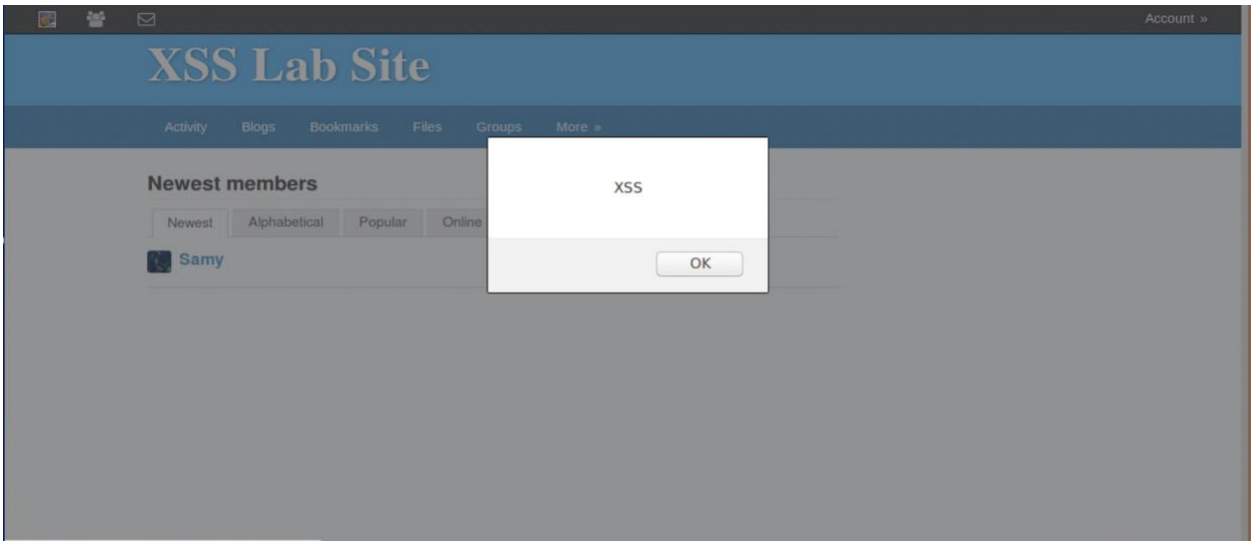


Then Samy adds the malicious code in his brief description and saves his profile.

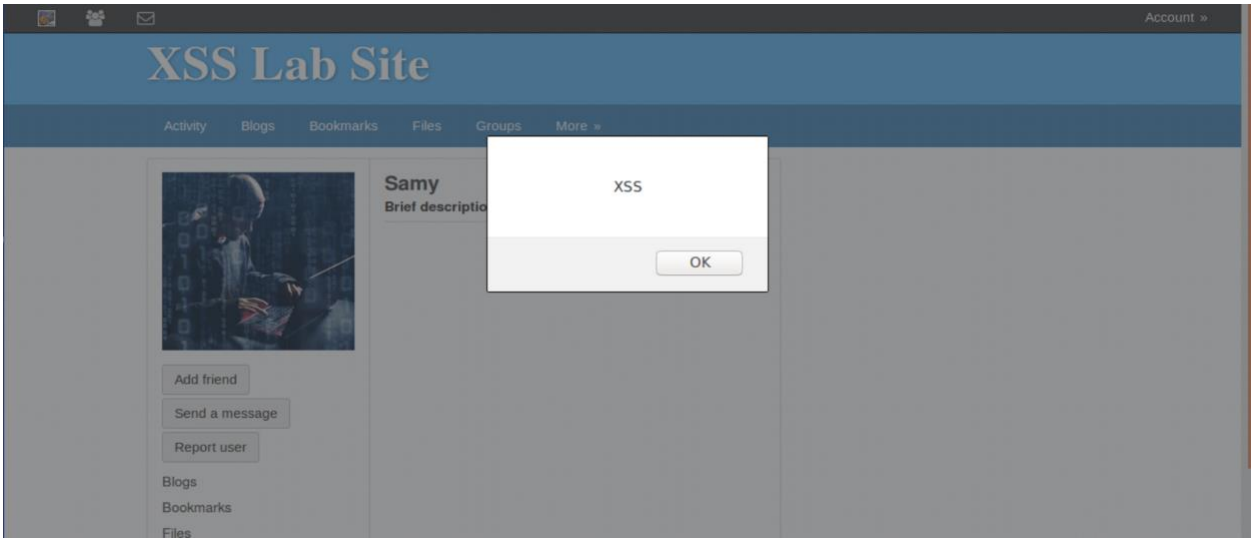




As soon as Samy saves the profile, an alert window pops up. This happens because at that time the script is running.



Now, Alice logs into her account and goes to the members' page and the alert command is triggered. This is because the malicious code is in the brief description and the brief description is visible in the members' page along with the member name.

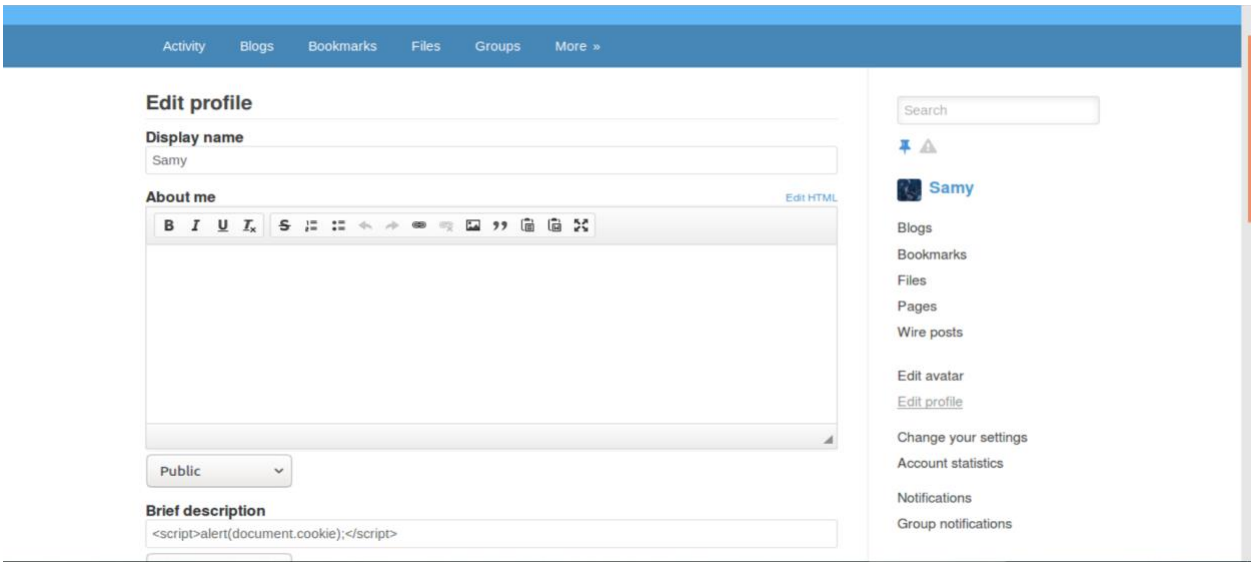


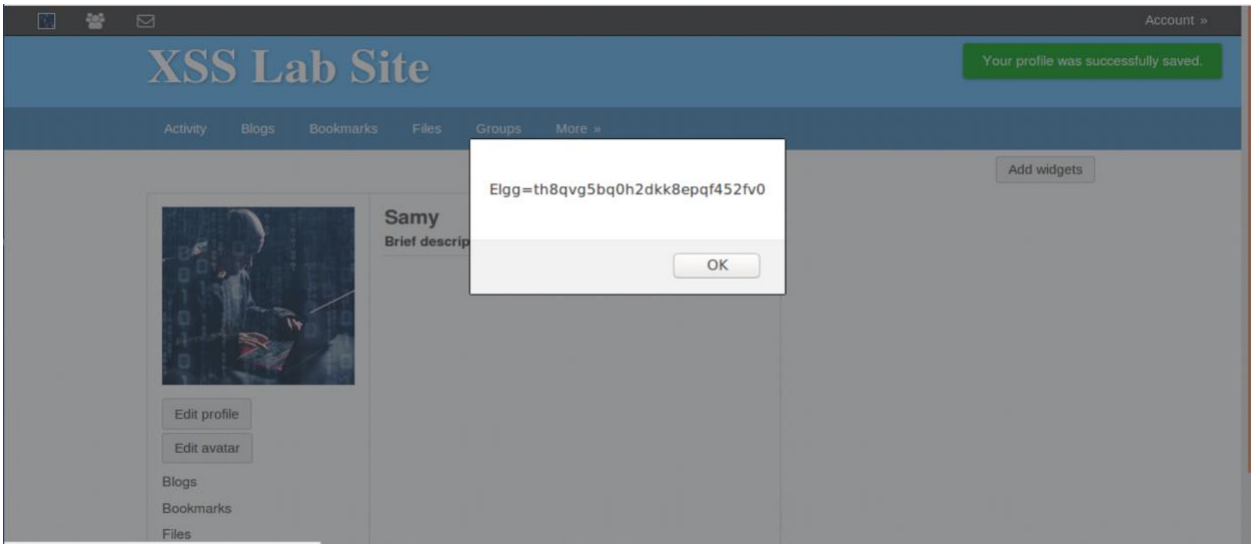
Again when Alice clicks on Samy and opens his profile, the alert command is triggered again and the window pops up.

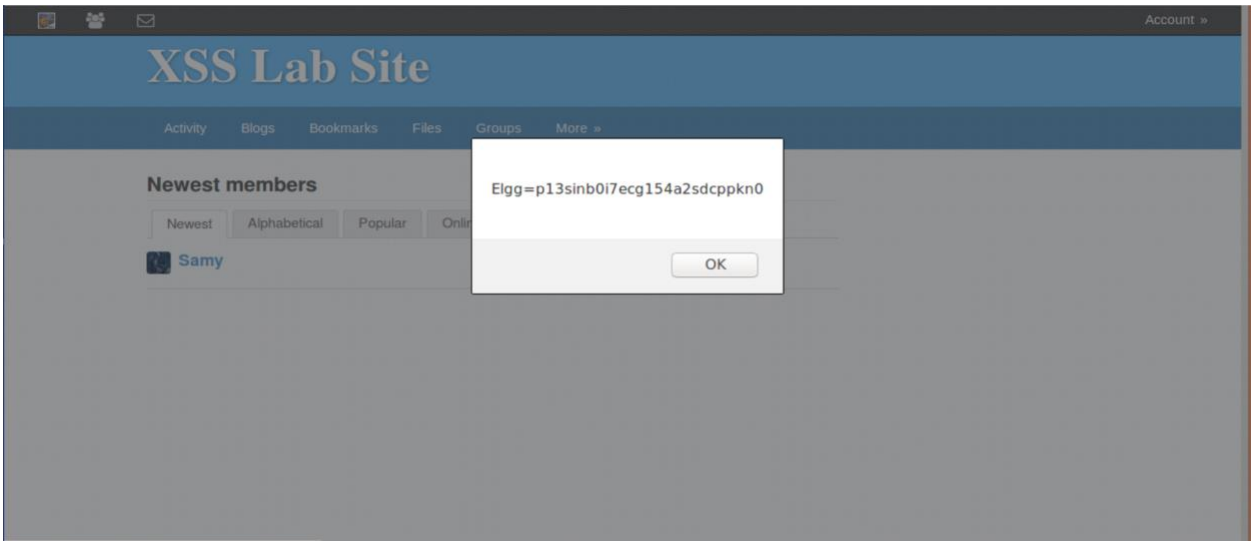
We write the malicious code in the brief description so that when the victim opens the attacker's profile, the javascript code is executed and the alert window is displayed.

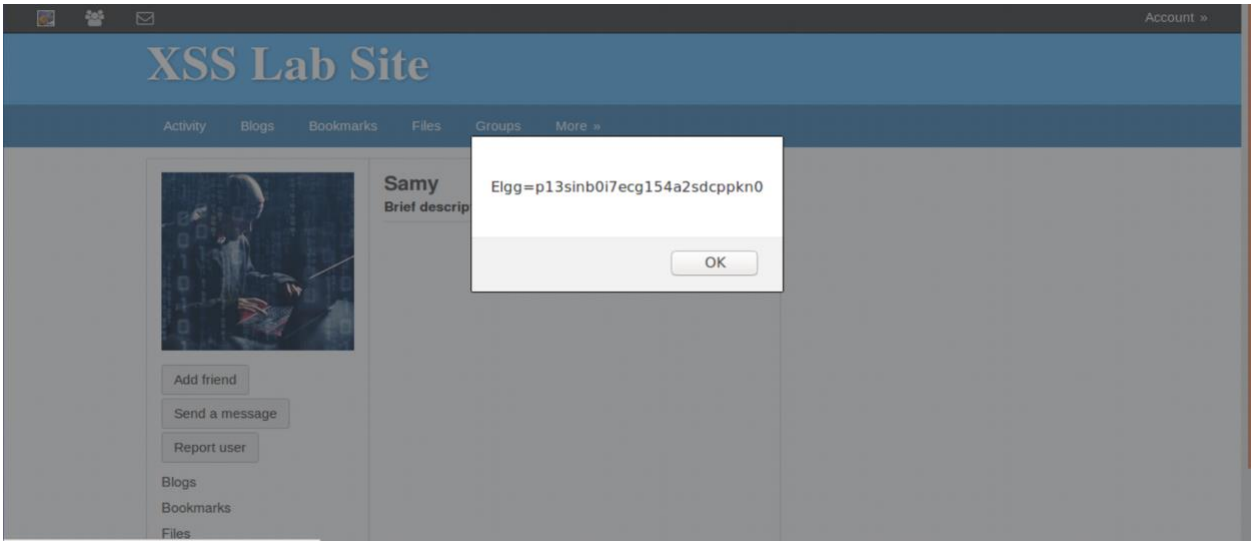
Task 2: Posting a Malicious Message to Display Cookies

Samy adds the malicious code in his brief description to display the cookies and saves his profile.



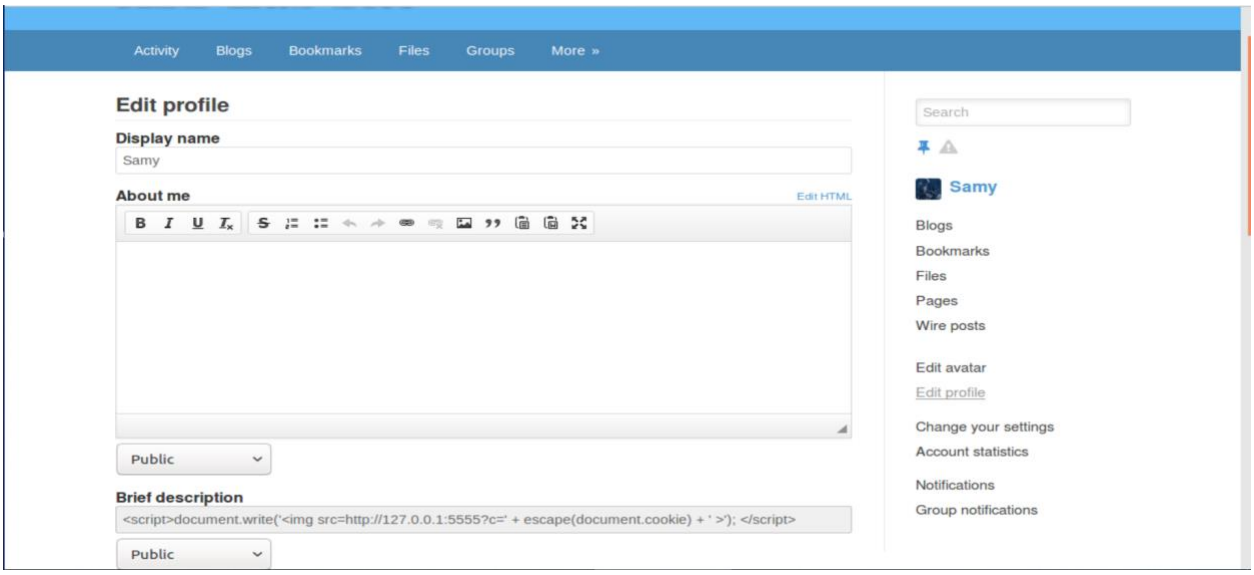






In the above screenshot we notice that, when the victim visits the page of the attacker, the victim's cookie is displayed. The victim that is Alice here sees her cookies as an alert when she visits the members' page and Samy's profile.

Task 3: Stealing Cookies from the Victim's Machine



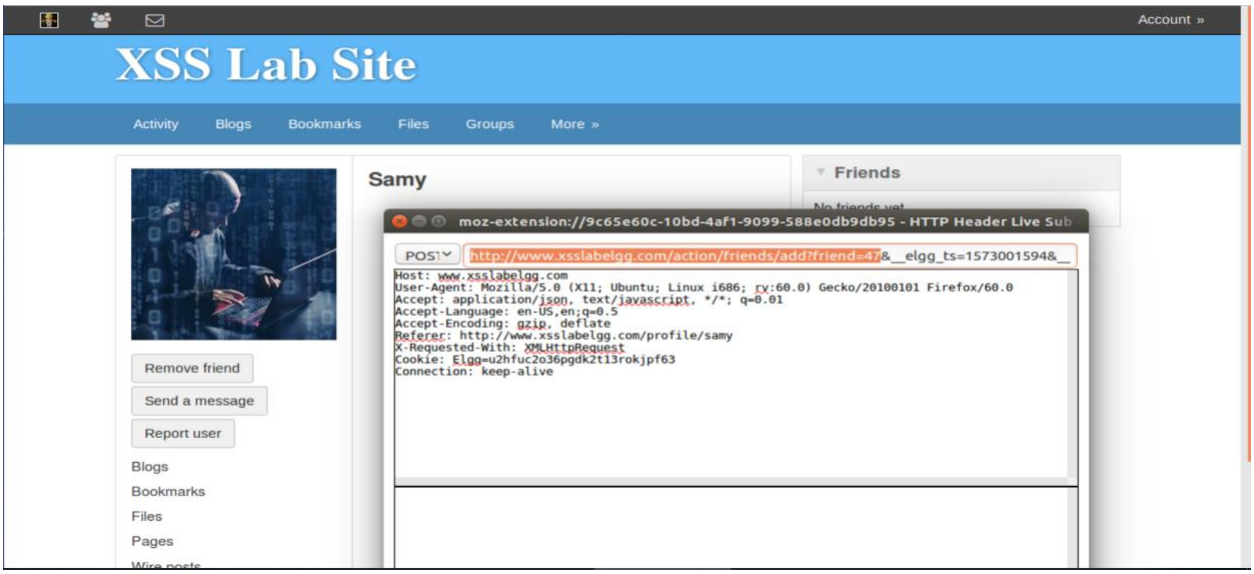
In this task, we modify the malicious code in the brief description such that the cookie information is returned to the attacker who is listening on port 5555 using the netcat command. The victim does not know that the cookie information is sent to the attacker. The escape API is used so that the cookie is encoded and sent back to the attacker. Then the attacker specifies the IP address of the machine he is listening to the connection on. In this case the address is sent back again in a loop. Hence, when the victim opens the profile page of the attacker, the javascript is executed and the cookie information is sent back due to which a GET request is sent back to the attacker along with the cookie information since the code is trying to load the image from the URL in the img tag.

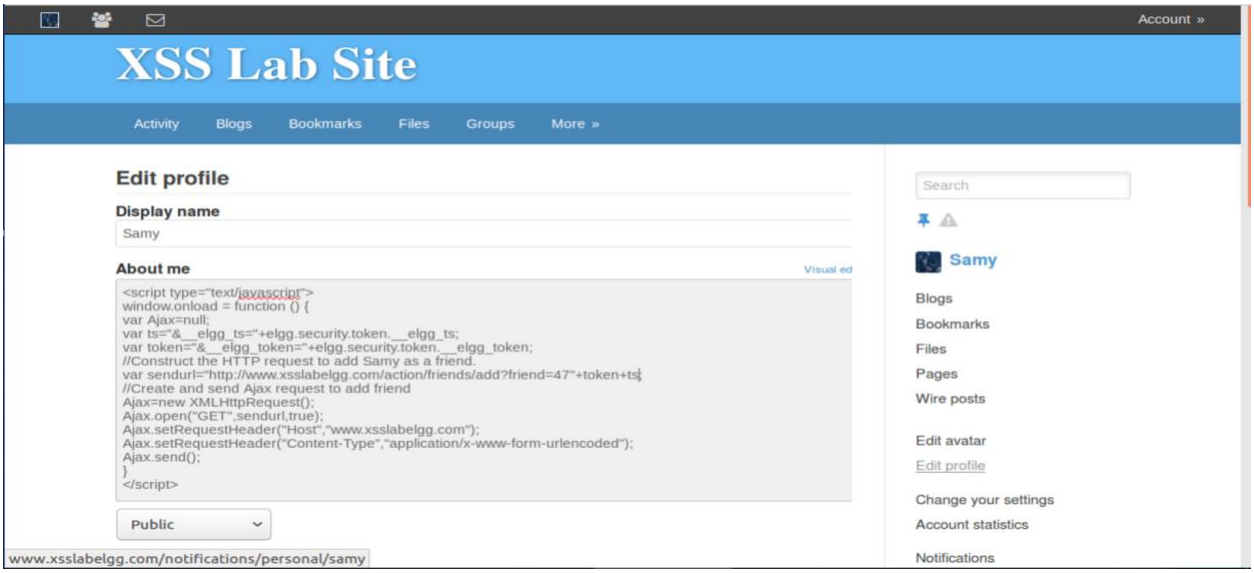
```
[11/05/19]seed@VM:~$ nc -l 5555 -v
Listening on [0.0.0.0] (family 0, port 5555)
Connection from [127.0.0.1] port 5555 [tcp/*] accepted (family 2, sport 51992)
GET /?c=Elgg%3Dda2tr88nt1dpc75h9nrvit3fo3 HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefo
x/60.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy
Connection: keep-alive
```

The screenshot above shows the attacker has started netcat and is listening to connections on port 5555.

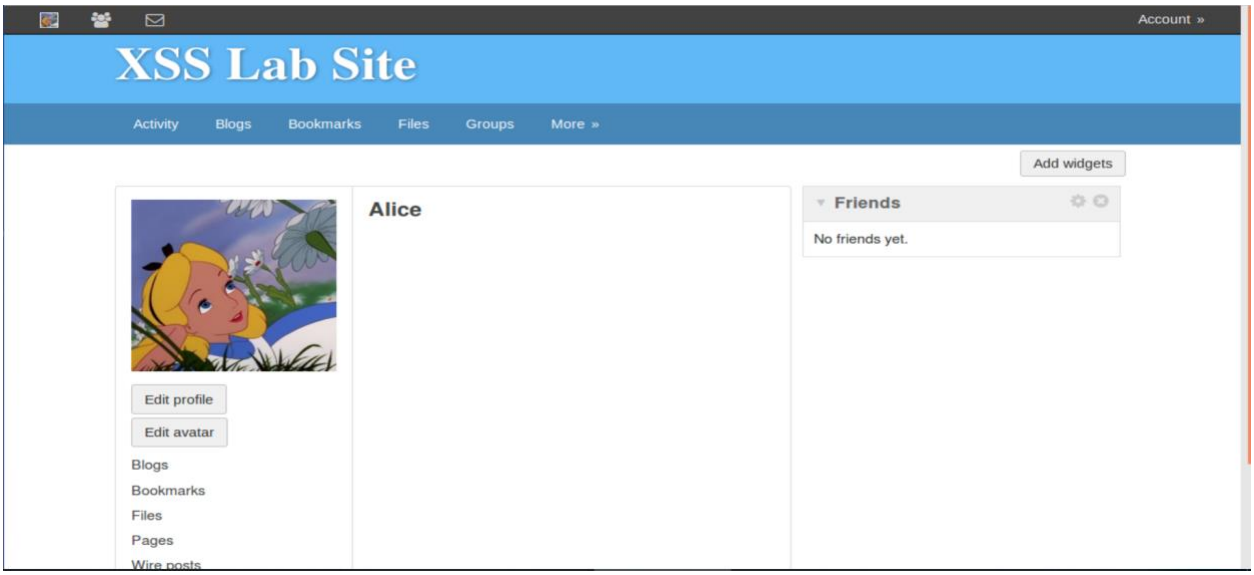
Task 4: Becoming the Victim's Friend

For his task, from Boby's accout we add Samy to investigate and get the cookies, tokens, ts and Samy's guid as shown from the HTTP live header below. Then we edit the malicious code to be used in the plaintext.





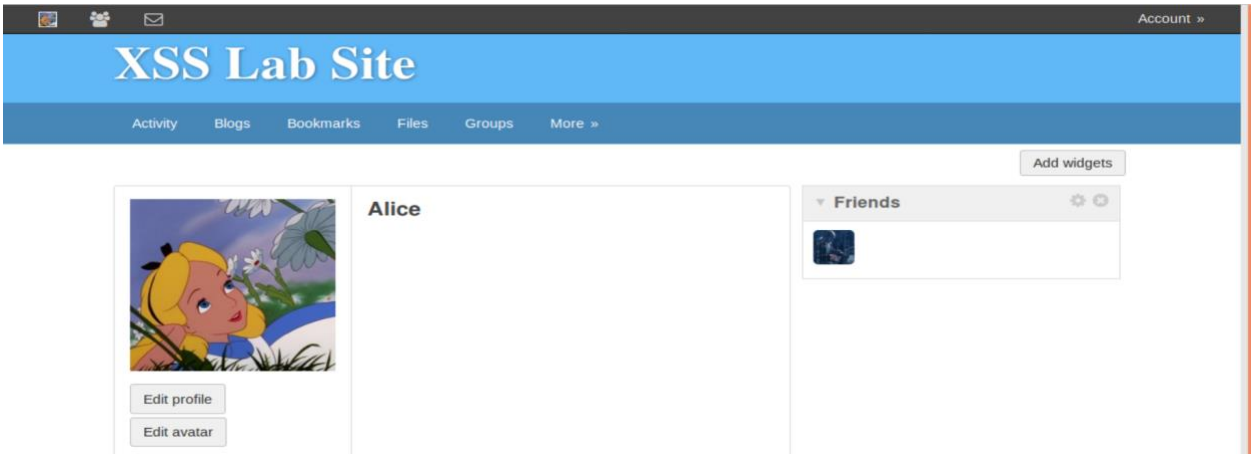
BEFORE ATTACK:



In this screenshot we see that Alice currently has no friends.

Then Alice visits members' page and visits Samy's profile.

AFTER ATTACK:



In this screenshot we notice that Samy is now Alice's friend. Hence, we can consider our attack to be successful.

Question 1: Explain the purpose of Lines ① and ②, why are they are needed?

These two lines are used and stored in a variable to make accessing easier. Hence, which makes the attack also easy, instead of searching the value they are directly loaded from the variable. Line 2 is needed because when the attacker requests for the token to the browser and when the victim clicks on the link or views the victims token is sent, Line 1 is there to keep a check on the current time such that when the

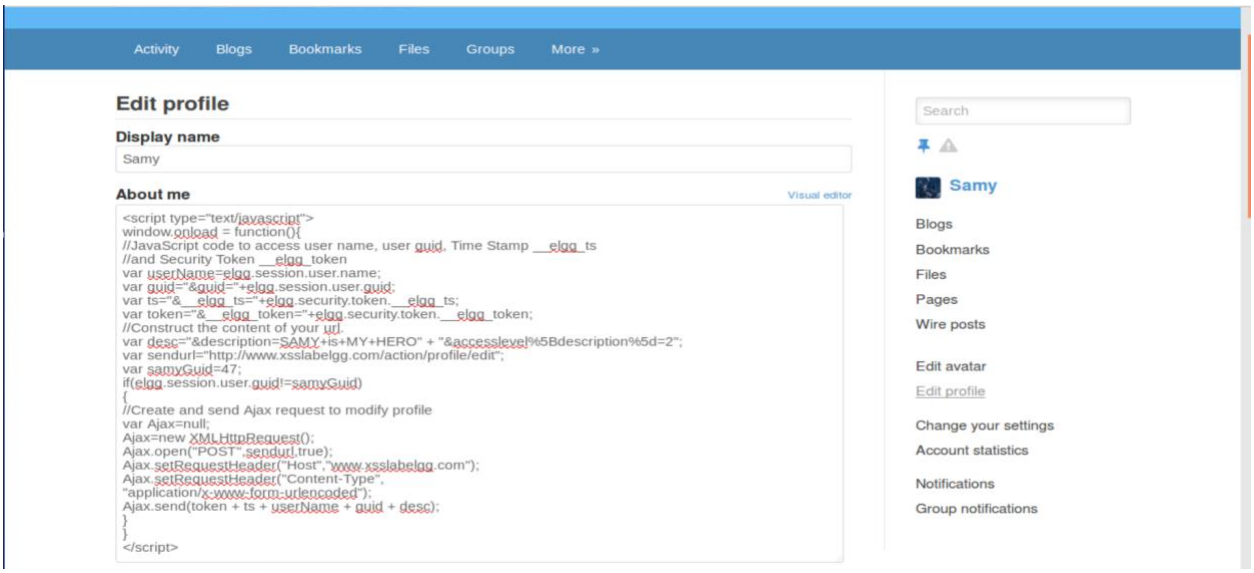
browser checks and if latest it is proceeded. If the token was of previous time-stamp then browser will discard. Hence to keep a check line 1 is used.

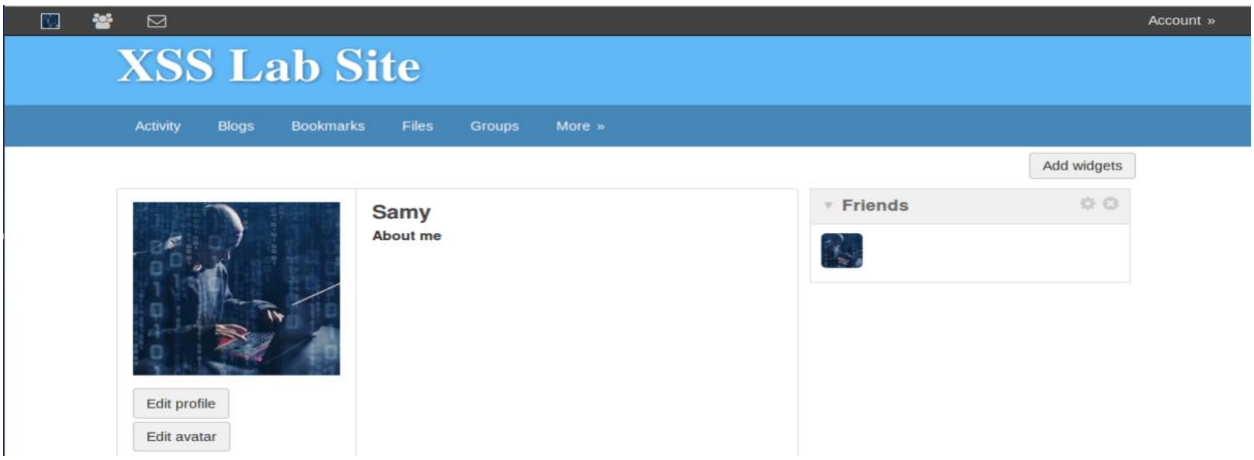
Question 2: If the Elgg application only provide the Editor mode for the "About Me" field, i.e., you cannot switch to the Text mode, can you still launch a successful attack?

Yes, though the attack only works on the text mode and not on editor mode because in editor mode the editors may add additional formatting data to the text due to which there can be problems in the javascript code and attack may not be successful. But the attack can possibly work by using an extension to remove the formatting data from HTTP requests.

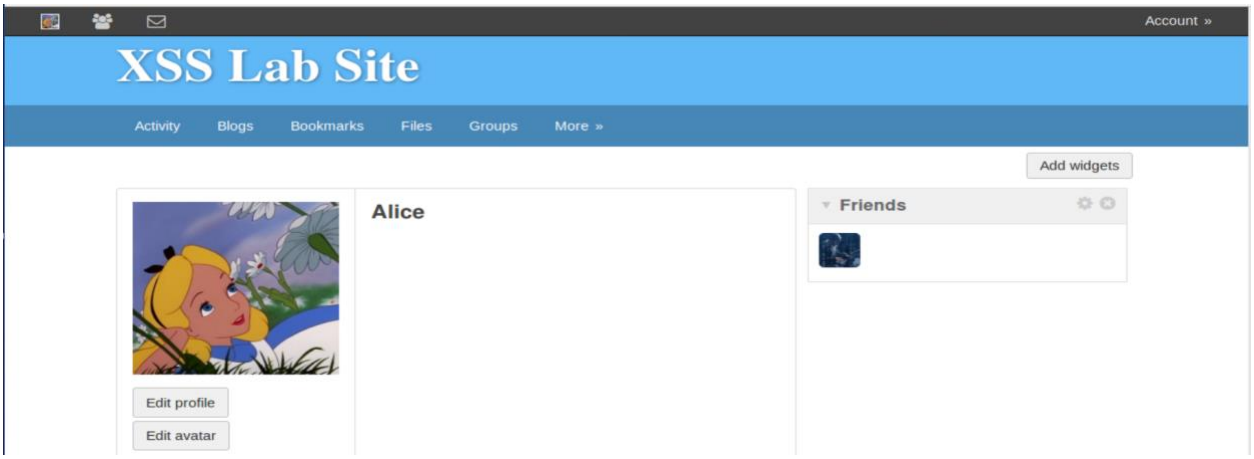
Task 5: Modifying the Victim's Profile

In this task Samy logs into his account and goes to his profile, clicks on edit html to enter the visual editor/plaintext mode. Then he injects the malicious code i.e; the javascript code into the about me field. The parameters passed in this malicious code is the token, ts, text and a condition on the guid. Apart from his which ever user visits his profile their profile will be modified.

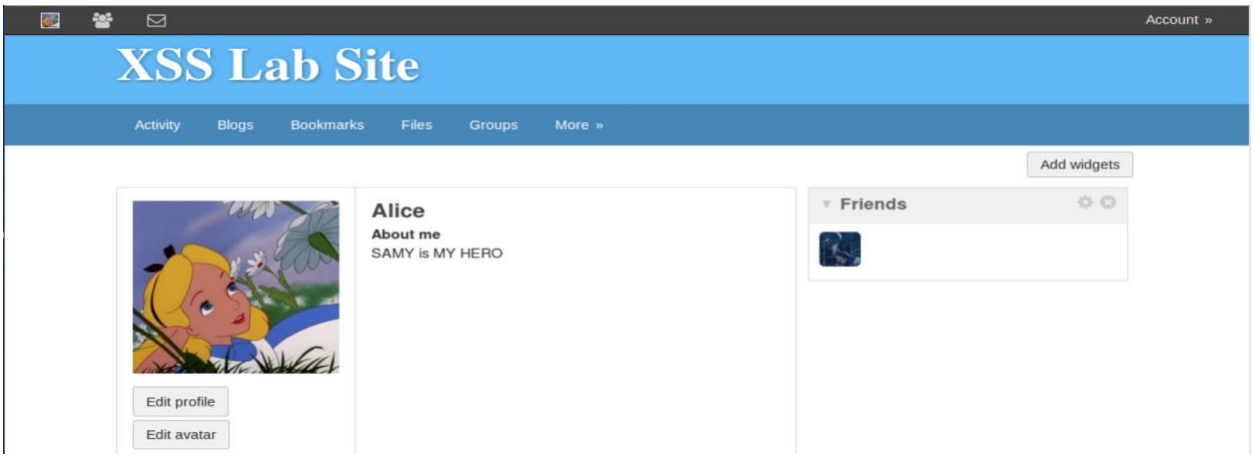




We see that there is nothing on Samy's profile under about me after he edited his profile and it was saved. Even when Alice or any other user visits Samy's profile they will not be able to see this.



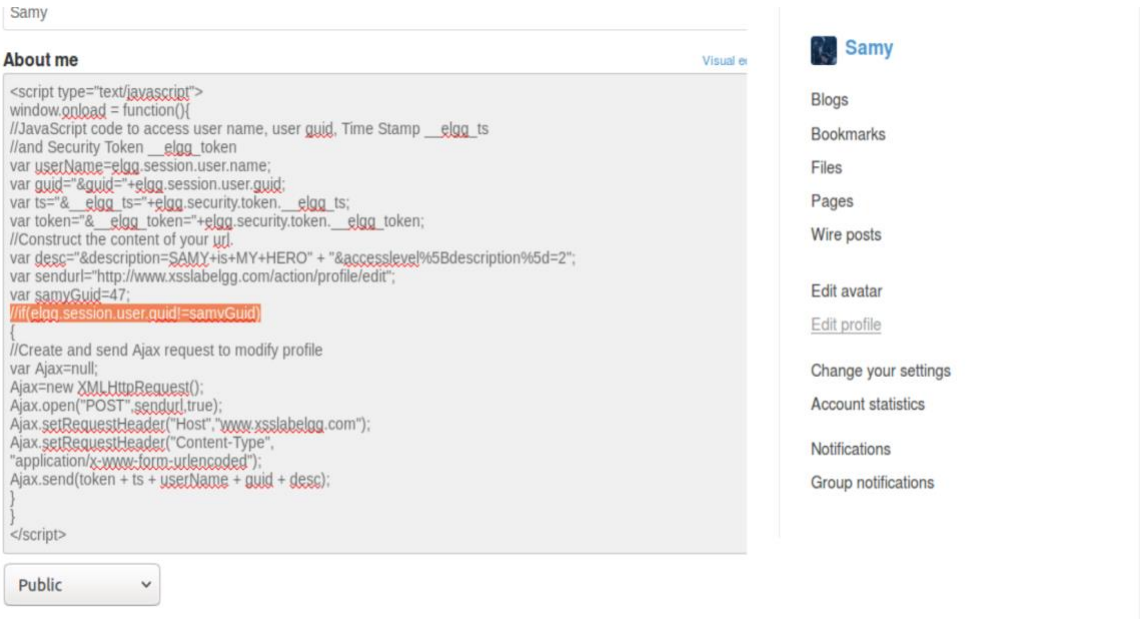
We see that Alice does not have any brief description currently

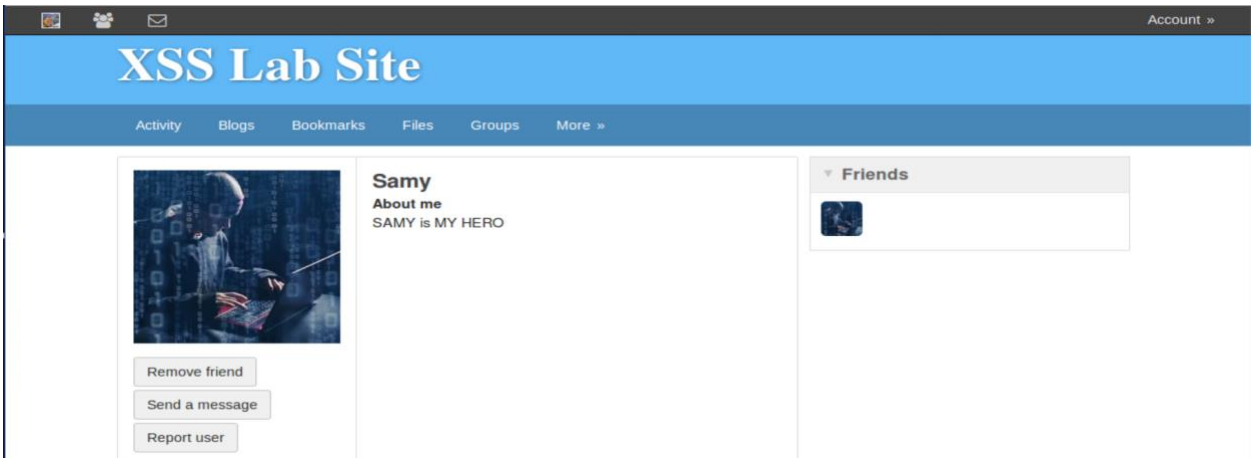


She then visits Samy's profile and returns back to her profile and notices that her profile has been modified and now there is a brief description stating SAMY is MY HERO. Hence the attack is successful.

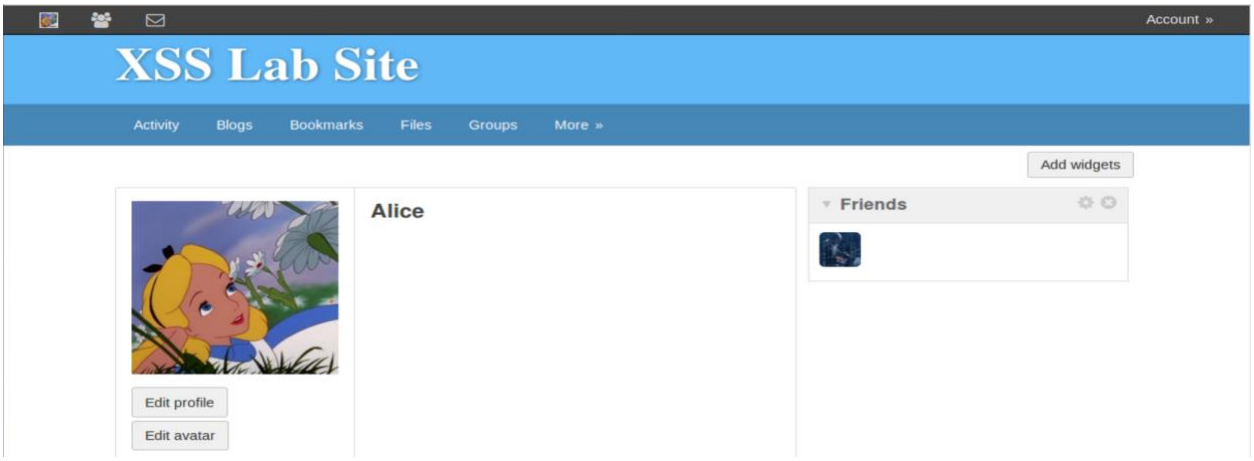
Question 3: Why do we need Line ①? Remove this line and repeat your attack. Report and explain your observation.

Now we uncomment the if condition which we explain above in the same attack as shown below in the screenshot the uncommented line and we do the same task again to see what happens.





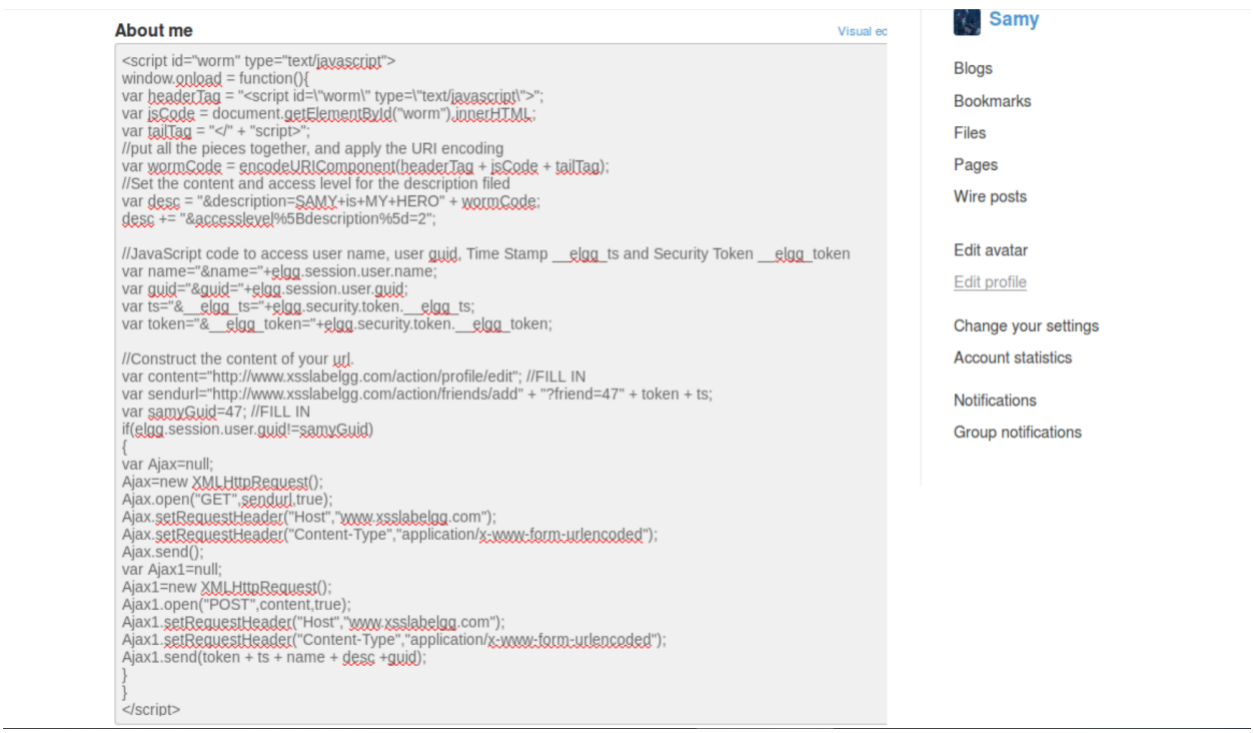
Alice visits Samy's profile and she notices that SAMY is MY HERO is under about me on his profile. This means that the attack is not successful on others but that text is only posted on his profile.



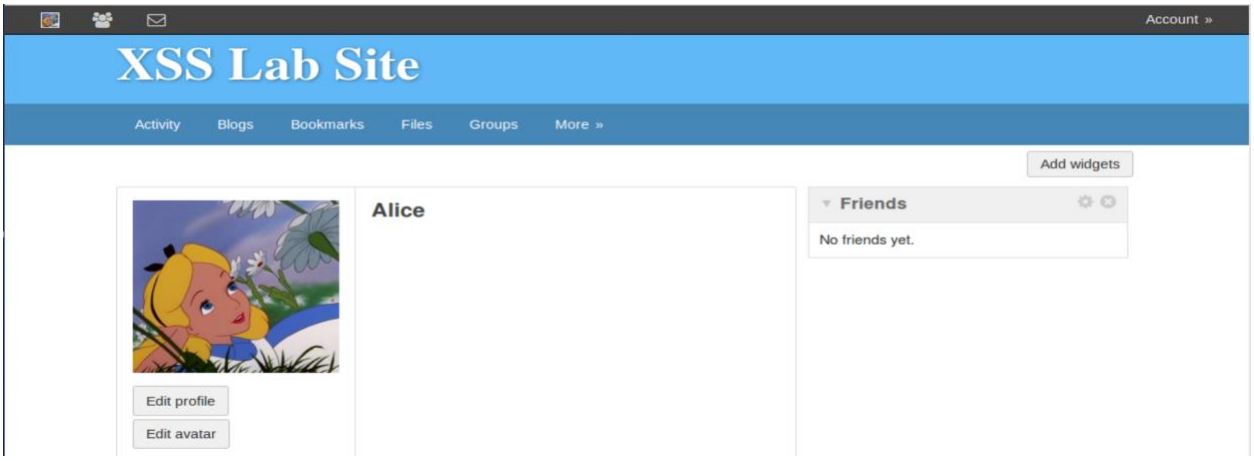
No such change is seen on Alice's profile.

Task 6: Writing a Self-Propagating XSS Worm

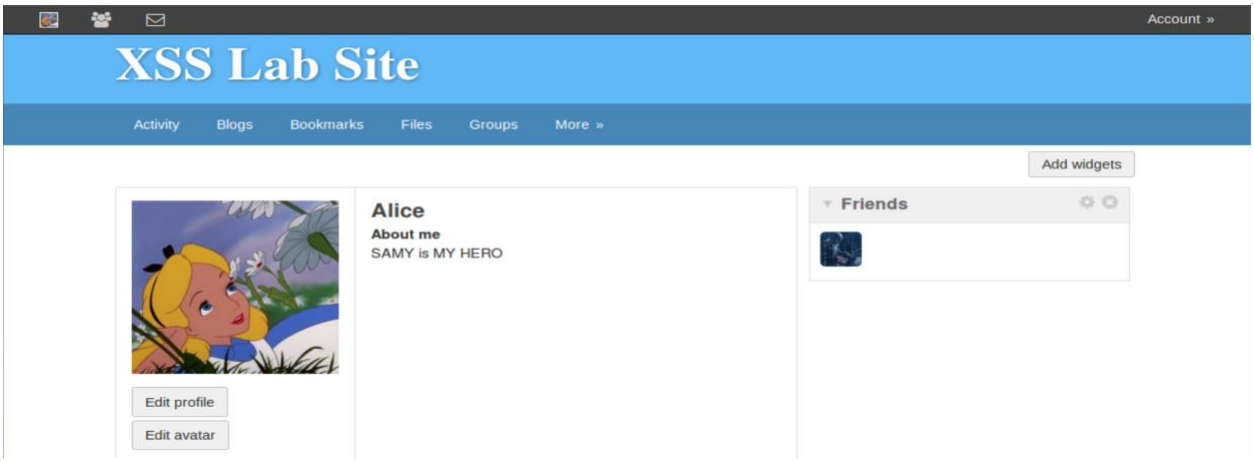
We modigy Samy's code from Task 5 and use the DOM approach.



Self-propagating malicious code

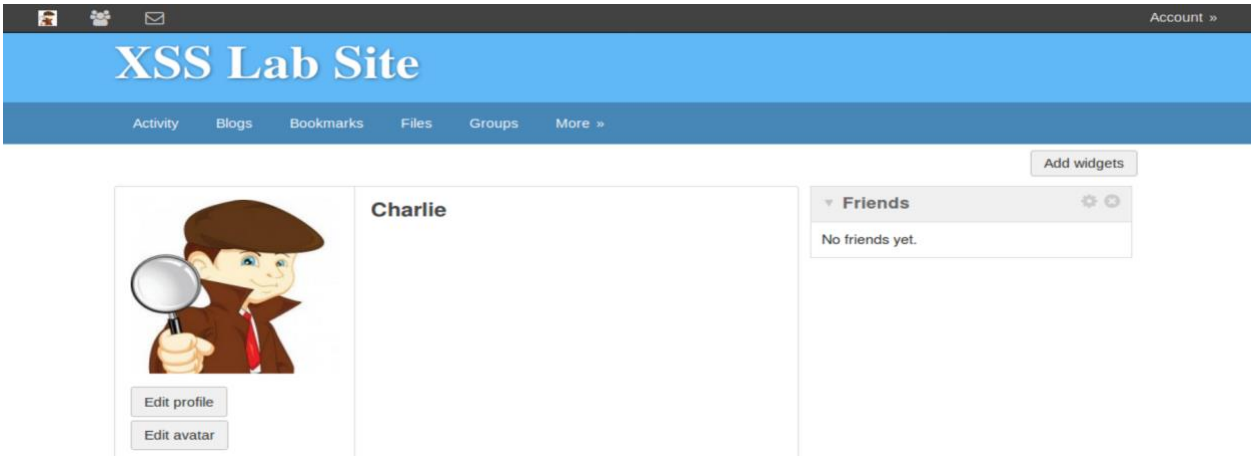


This is the observation taken before the attack. Alice logs onto her account and we notice that she currently has nothing on her profile.



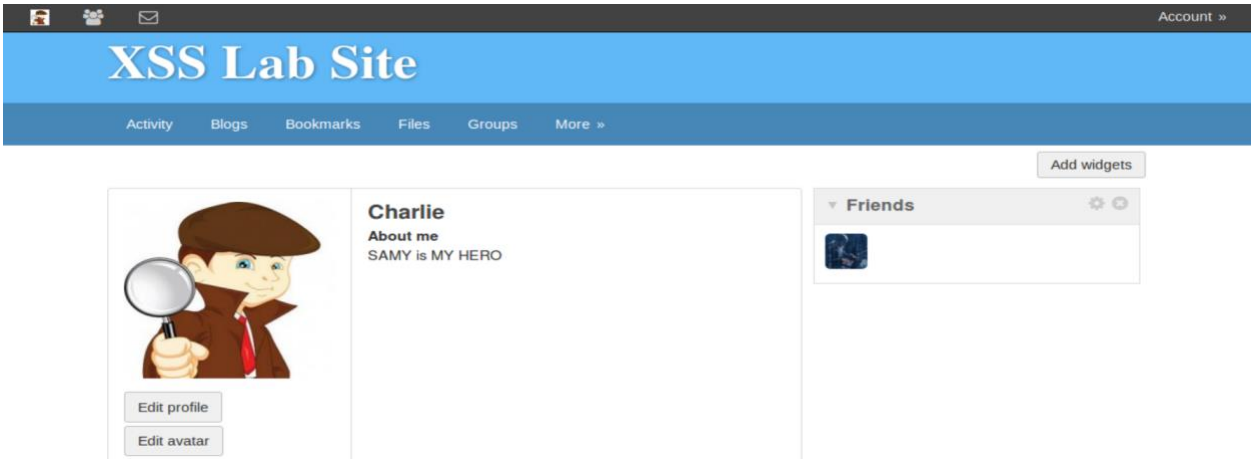
Then Alice visits Samy's profile. The above screenshot is after the attack.

Next, we see what happens when Charlie visits Alice's profile.



Charlie's profile before the attack

After Charlie visits Alice's profile.



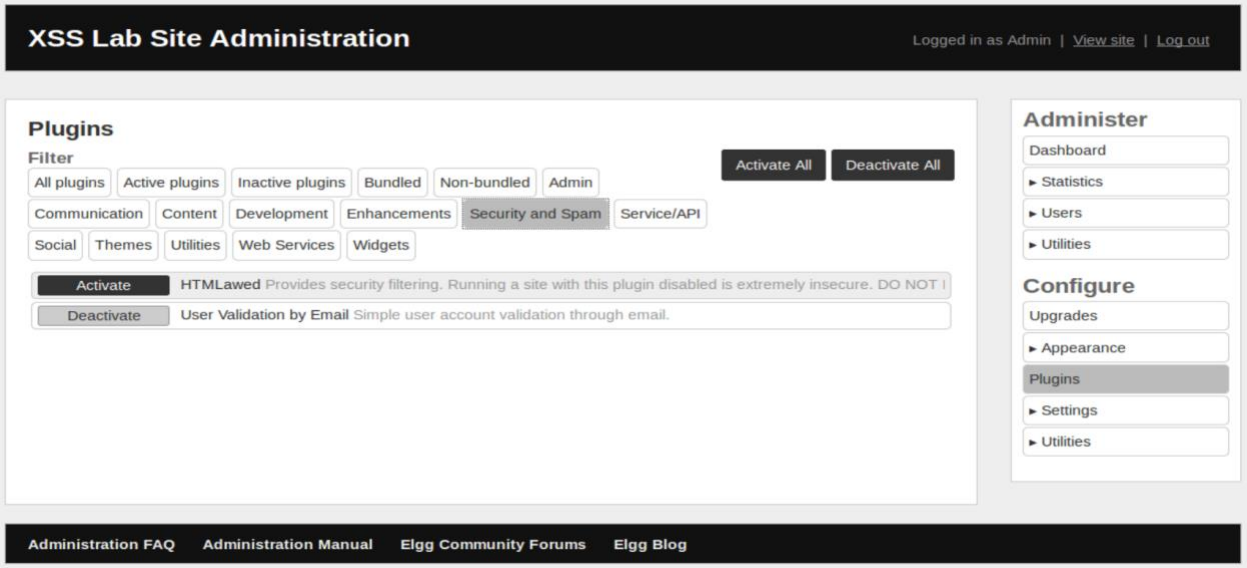
After the attack

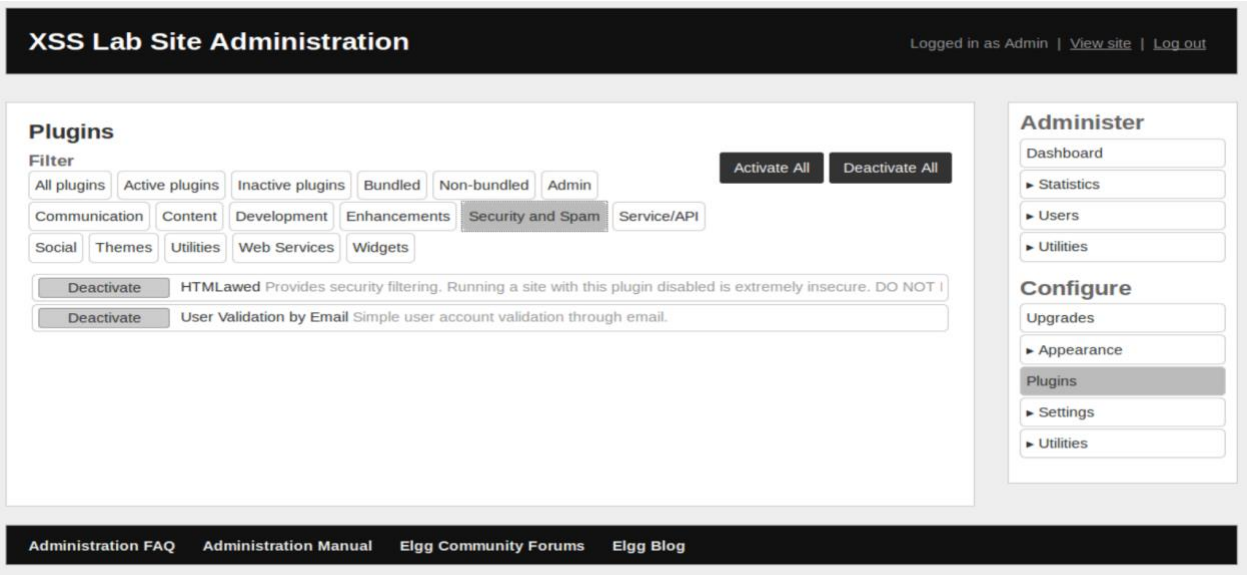
We perform the same steps as in the previous case, but here this is self-propagating malicious code. So, when a user visits the infected victim's profile, he also gets infected. In the above example, Samy is the attacker, he places a worm in his profile. Alice visits his profile and gets affected. When Charlie visits Alice's profile he also gets affected.

Task 7: Countermeasures

HTMLawed

We log onto Admin's account and enable the plugin. When we post the JavaScript directly on Samy's profile in the 'About Me' we notice that the script tags are removed from the JavaScript.





Once this countermeasure is turned on, all the script tags are disabled and all the malicious code is shown in the profile. Hence, the attack doesn't work since there are no script tags. Nothing is executed.

HTML Special Characters

```
[11/05/19]seed@VM:~$ cd /var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output/
[11/05/19]seed@VM:.../output$ gedit text.php
[11/05/19]seed@VM:.../output$ cat text.php
<?php
/**
 * Elgg text output
 * Displays some text that was input using a standard text field
 *
 * @package Elgg
 * @subpackage Core
 *
 * @uses $vars['value'] The text to display
 */

echo htmlspecialchars($vars['value'], ENT_QUOTES, 'UTF-8', true);

echo $vars['value'];
[11/05/19]seed@VM:.../output$
```

```
[11/05/19]seed@VM:.../output$ gedit url.php
[11/05/19]seed@VM:.../output$
```

```
if (isset($vars['text'])) {
        if (elgg_extract('encode_text', $vars, false)) {
                $text = htmlspecialchars($vars['text'], ENT_QUOTES, 'UTF-8', true);
                $text = $vars['text'];
        } else {
                $text = $vars['text'];
        }
        unset($vars['text']);
} else {
        $text = htmlspecialchars($url, ENT_QUOTES, 'UTF-8', true);
        $text = $url;
}
```

```
[11/05/19]seed@VM:.../output$ gedit dropdown.php
[11/05/19]seed@VM:.../output$ cat dropdown.php
<?php
/**
 * Elgg dropdown display
 * Displays a value that was entered into the system via a dropdown
 *
 * @package Elgg
 * @subpackage Core
 *
 * @uses $vars['text'] The text to display
 *
 */

echo htmlspecialchars($vars['value'], ENT_QUOTES, 'UTF-8', true);

echo $vars['value'];
[11/05/19]seed@VM:.../output$
```

```
[11/05/19]seed@VM:.../output$ gedit email.php
[11/05/19]seed@VM:.../output$ cat email.php
<?php
/**
 * Elgg email output
 * Displays an email address that was entered using an email input field
 *
 * @package Elgg
 * @subpackage Core
 *
 * @uses $vars['value'] The email address to display
 *
 */

$encoded_value = htmlspecialchars($vars['value'], ENT_QUOTES, 'UTF-8');

$encoded_value = $vars['value'];


if (!empty($vars['value'])) {
        echo "<a href=\"mailto:$encoded_value\">$encoded_value</a>";
}
[11/05/19]seed@VM:.../output$
```

We uncomment the htmlspecialchars function in each of the files: text.php, url.php, dropdown.php, and email.php as shown in the above screenshot.

We notice that on enabling the countermeasure, the special characters are encoded in the JavaScript that is not directly posted on Samy's profile but it is can be seen when we click on view page source as shown below.

```
amy</span><h2 class="p-name fn">Samy</h2><p class='profile-aboutme-title'><b>About me</b></p><div class='profile-aboutme-contents'><div class="elgg-output mtn">
"worm").innerHTML;<br />var tailTag = "&lt;/" + "script&gt;";<br />//put all the pieces together, and apply the URI encoding<br />var wormCode = encodeURIComponent(headerT

g_token<br />var name="&amp;name="+elgg.session.user.name;<br />var guid="&amp;guid="+elgg.session.user.guid;<br />var ts="&amp;__elgg_ts="+elgg.security.token.__elgg_ts;<

/edit"; //FILL IN<br />var sendurl="http://www.xsslabelgg.com/action/friends/add" + "?friend=47" + token + ts;<br />var samyGuid=47; //FILL IN<br />if(elgg.session.user.gu
```

After uncommenting in each of the above files, the attack is not successful since html encoding encodes the special characters like <,> which are used as tags. Therefore, none of the tags can be used to attack. Hence, the attack is not successful.