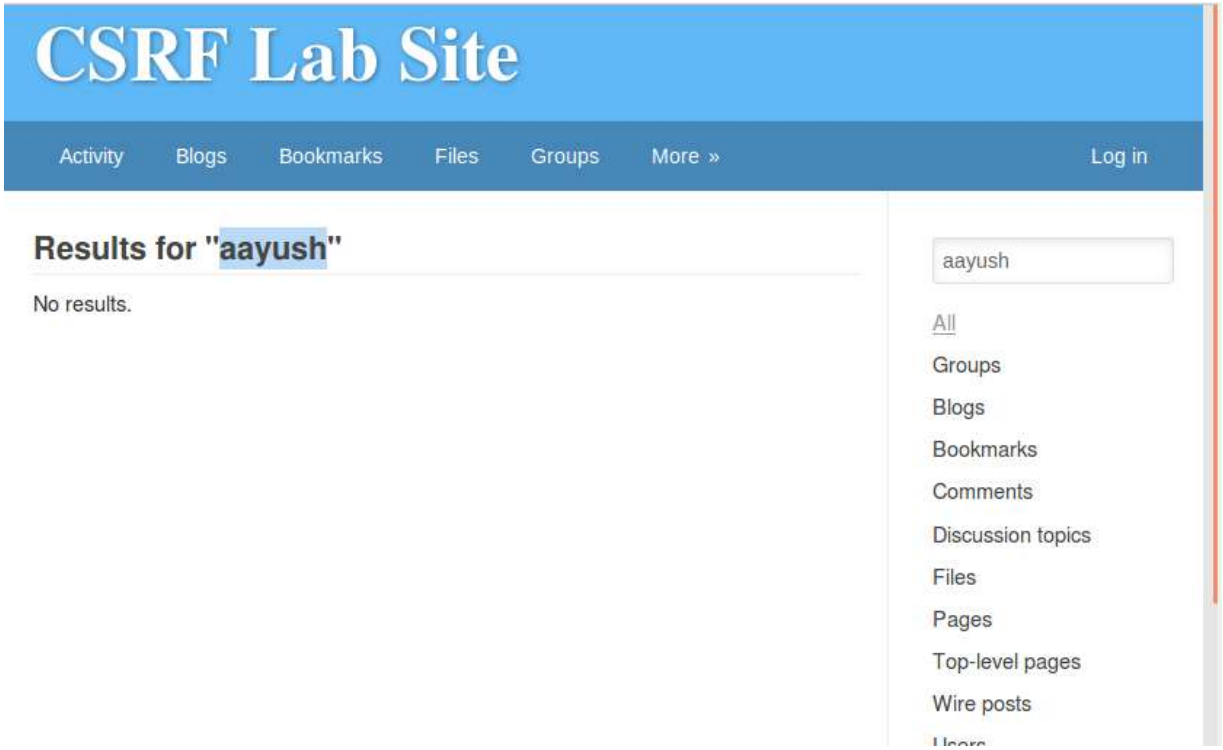
PES UNIVERSITY
INTERNET SECURITY
LAB 7 - CSRF ATTACK

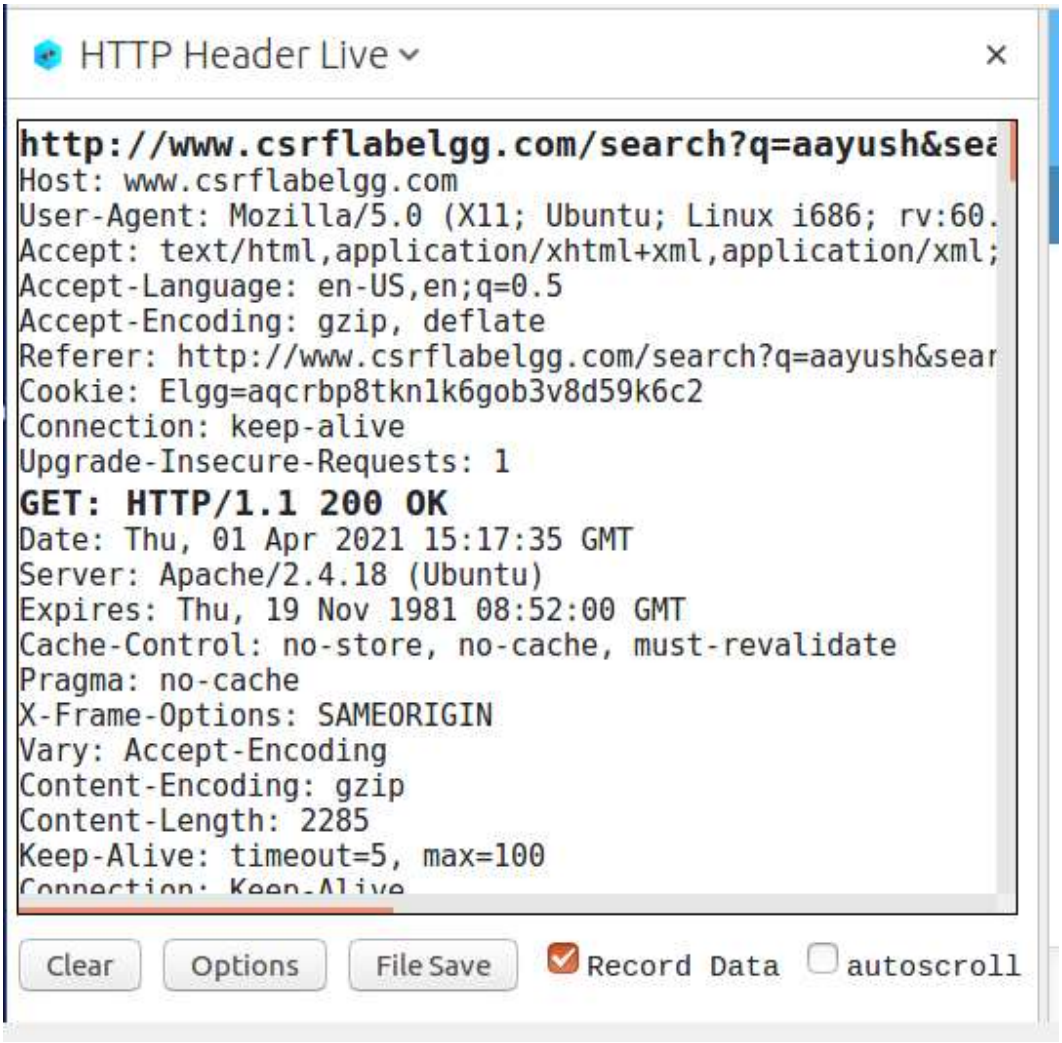
Aayush Kapoor PES2201800211

TASK 1:

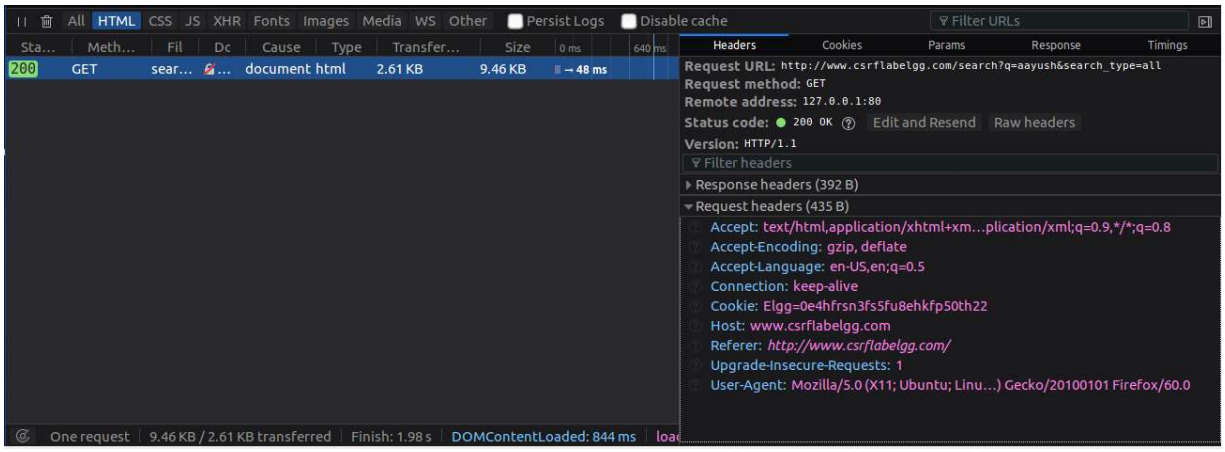
HTTP GET

Here I search for my name in the search bar of the website, and using the developer tools, I see the request generated. The following shows the web activity:





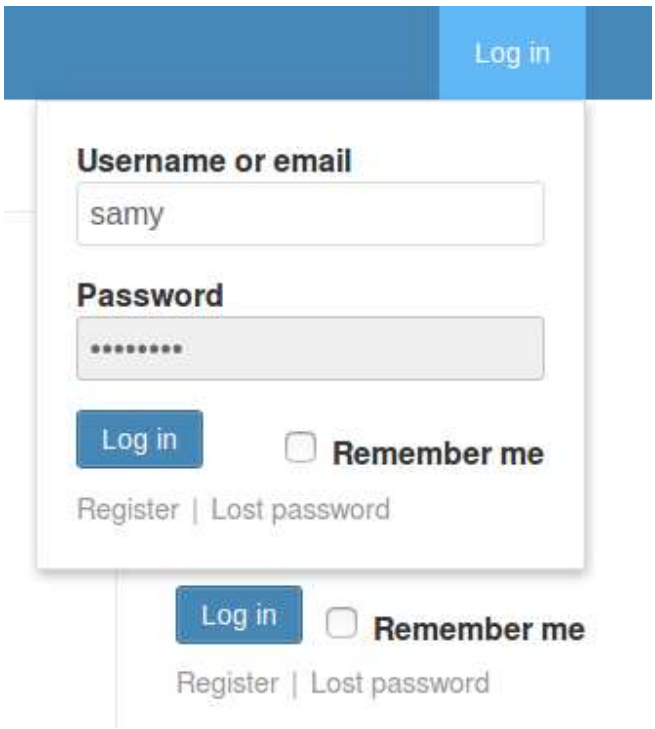
The HTTP request seen in the developer tools is as follows:

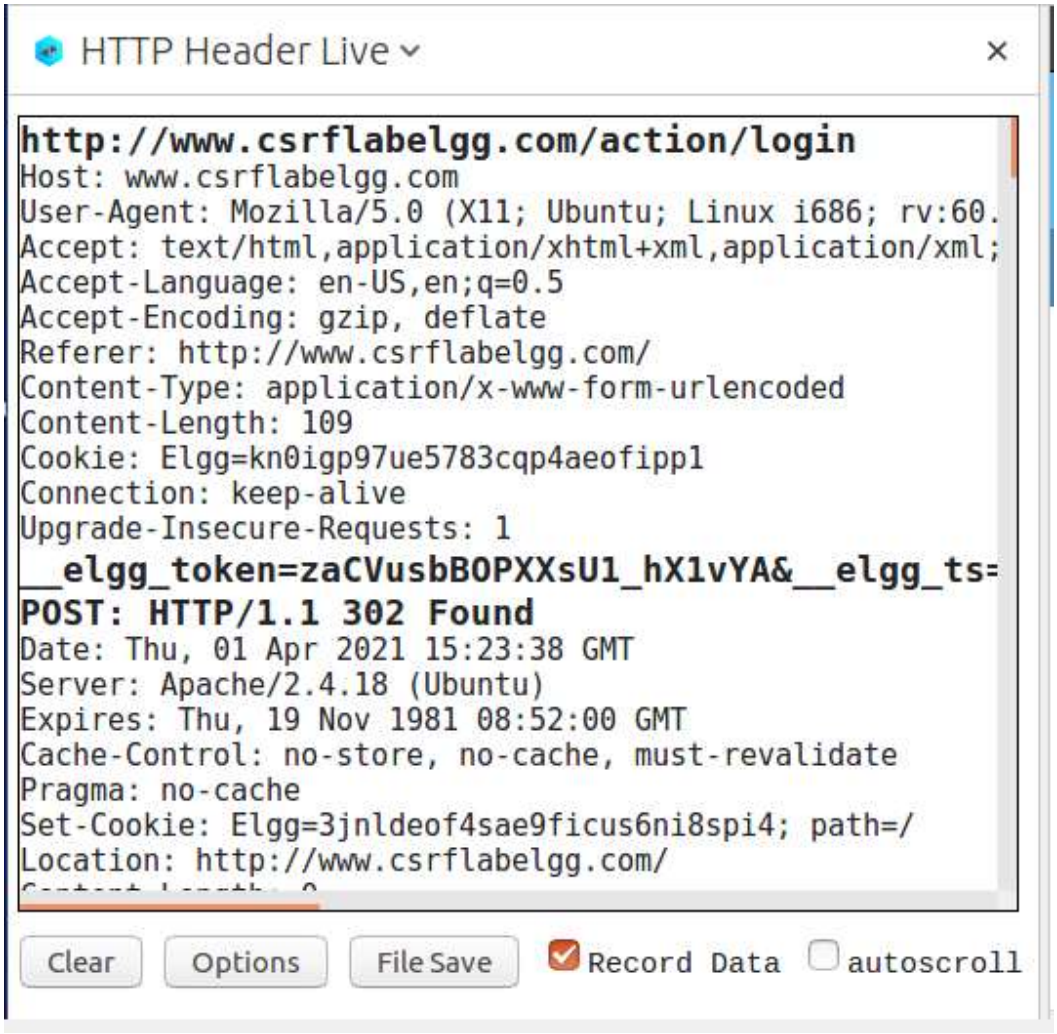


We see that the HTTP request has the method as GET. We also notice that there are Params sent in this request, from the Params tab. The parameters have q's value set to megha, the string I entered, and search type has all since All is selected as the area to search. Along with this, the Accept fields are also sent to indicate the server the type of data accepted by the browser. We also see that the browser is sending the cookie in the request along with its own information in the User-Agent field.

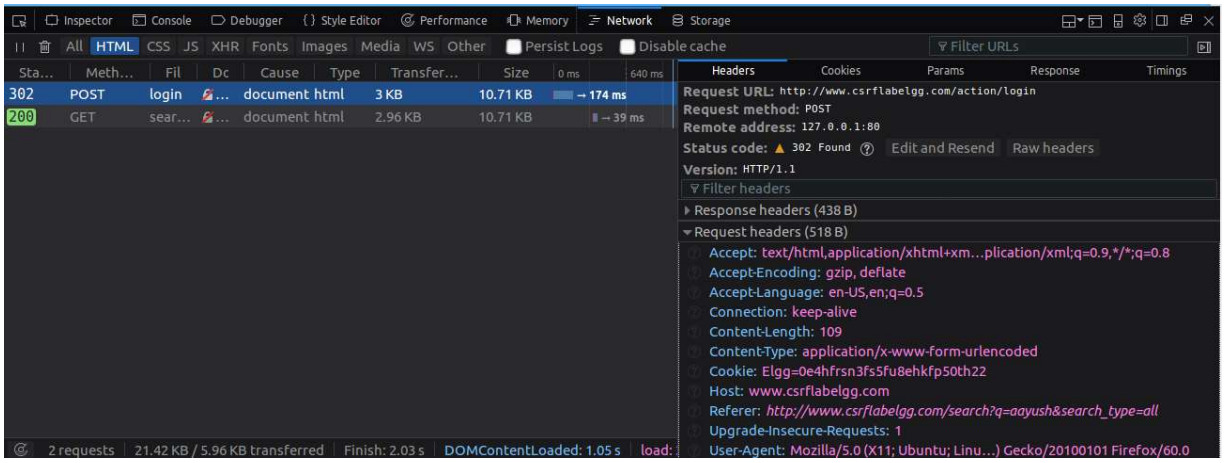
HTTP POST

We try to log in using one of the credentials given in the manual, since the login is eventually a form. The following shows this activity:

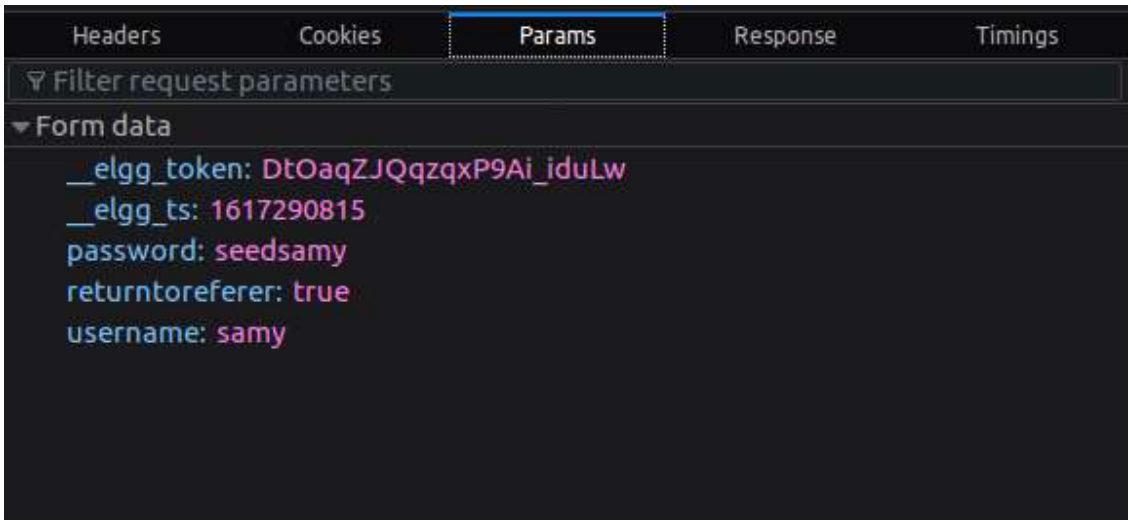




On looking at the HTTP request in the web developer tool, we see:



Here, as expected 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. In order to explore that, we see the content of the Params Tab:

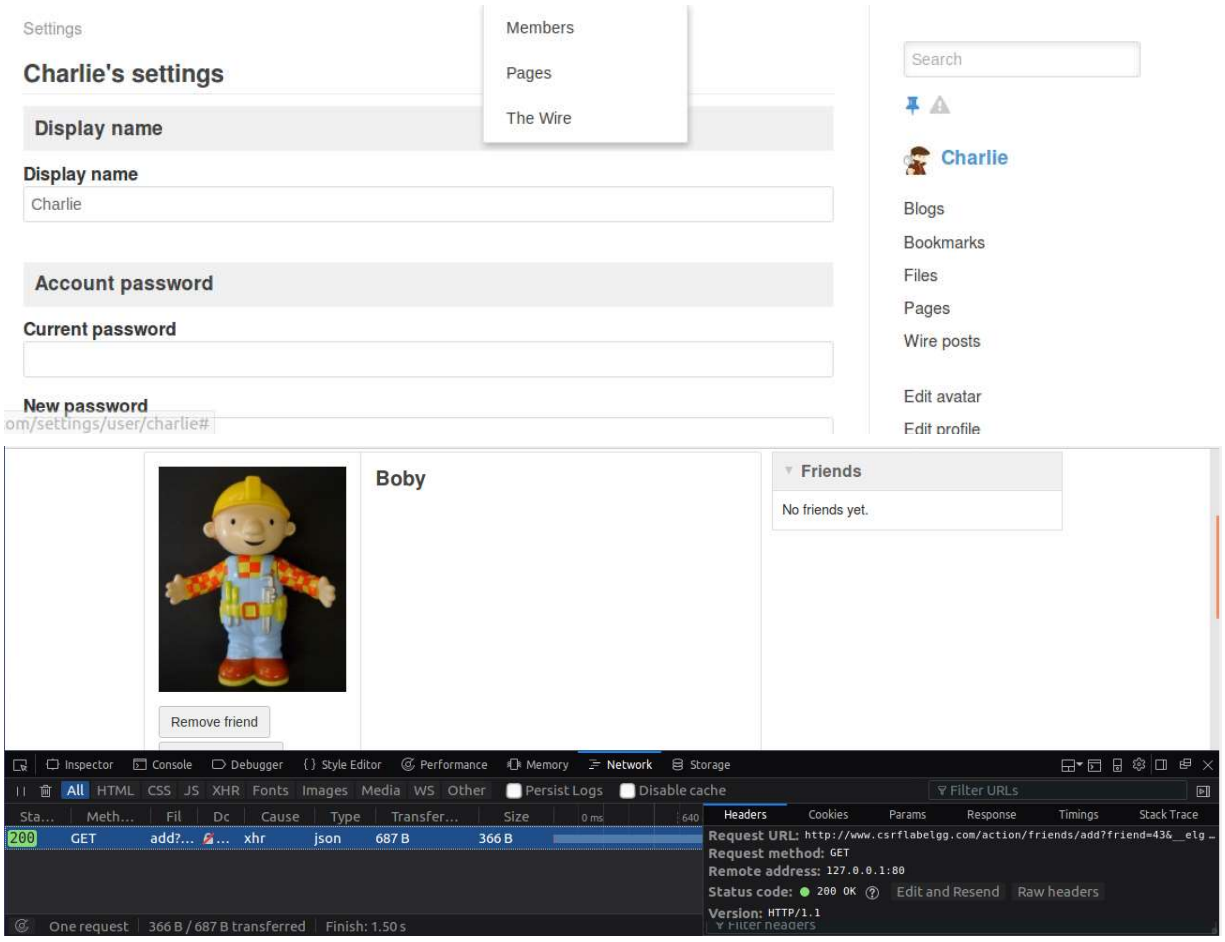


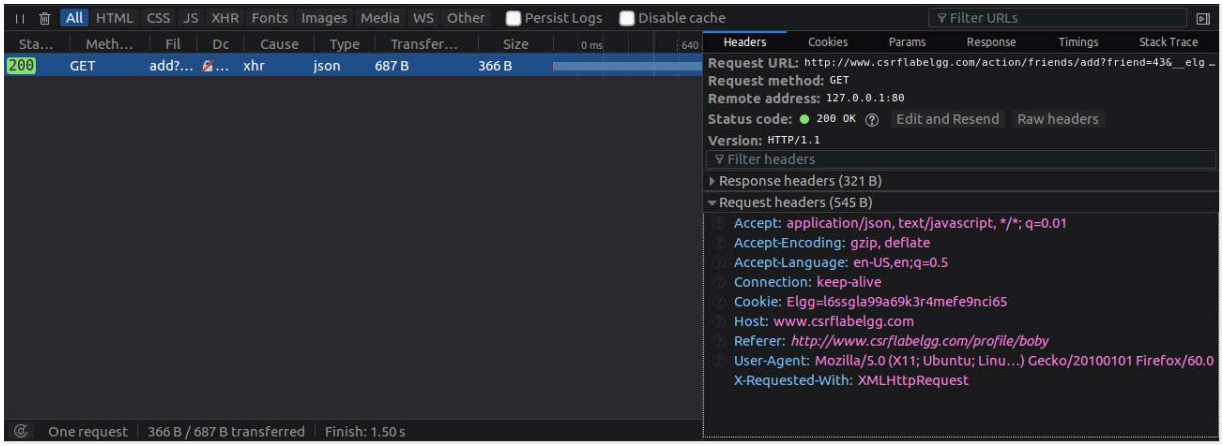
Here, we see that some parameters are sent in the HTTP request. Username and Password are the fields that I entered. Return to referrer is set to True indicating that the result will be returned to the referer set in the HTTP request. The first two parameters are the token and the timestamp, which are the countermeasures to the CSRF attack and will be explained later in the lab.

There is also a Referer field present that indicates the source website of the request. This field can let the server know whether the request is cross-site or same-site and hence can be used as a countermeasure to CSRF attack; however not all the browsers incorporate this and also this might be invading an individual's privacy.

TASK 2:

In order to create a request that will add boby as a friend in Alice's account, we (Boby) need to find the way 'add friend' request works. So, we assume that we have created a fake account named Charlie and we first log in into Charlie's account so that we can add Boby as Charlie's friend and see the request parameters that are used to add a friend. After logging into Charlie's account, we search for Boby and click on the add friend button. While doing this, we look for the HTTP request in the web developer tools and see:





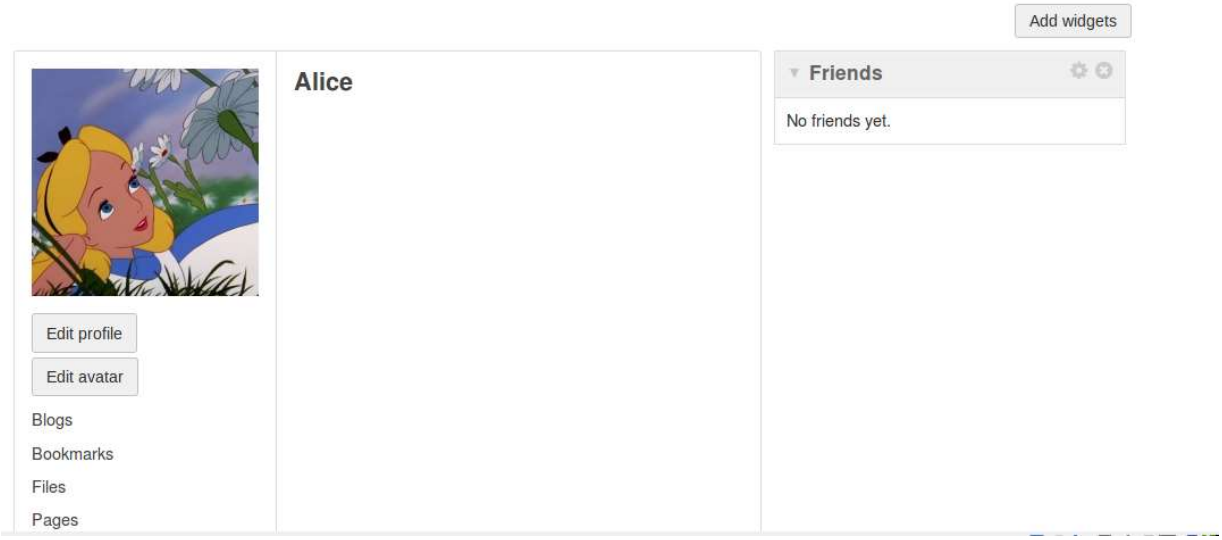
Since it's a GET request, the URL has the parameters and we see that friend has a value of 43. It also has some other parameters in the request, that can be clearly seen in the Params tab:

So, we know that since Charlie tried to add Boby as a friend, a request was made with the friend value as 43, which must be of Boby.

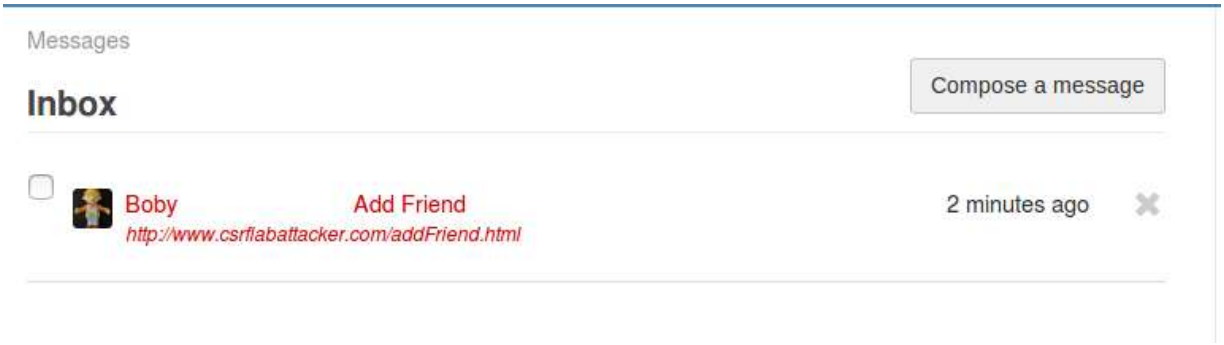
In order to generate a GET request we use the img tag of HTML pages, which sends a GET request as soon as the web page loads in order to display the image. Here, we specify the image width and height as 1 so that it's very small and not visible to Alice. This helps in hiding the intention of the web page – here, that of adding a friend. elgg_ts and elgg_token are countermeasures but have been disabled for the lab. So, we won't need to include those two fields. The attacker website's code is as following:

```
[04/01/21]seed@PES2201800211_AAYUSH-A:.../Attacker$ pwd
/var/www/CSRF/Attacker
[04/01/21]seed@PES2201800211_AAYUSH-A:.../Attacker$ sudo nano addFriend.html
[04/01/21]seed@PES2201800211_AAYUSH-A:.../Attacker$ cat addFriend.html
<html>
<body>
        <h1>Hi Alice. This is Bob, I have been added your friend via CSRF attack....</h1>
        <img src="http://www.csrflabelgg.com/action/friends/add?friend=43" alt="image", width="1" height="1">
</body>
</html>

[04/01/21]seed@PES2201800211_AAYUSH-A:.../Attacker$
```

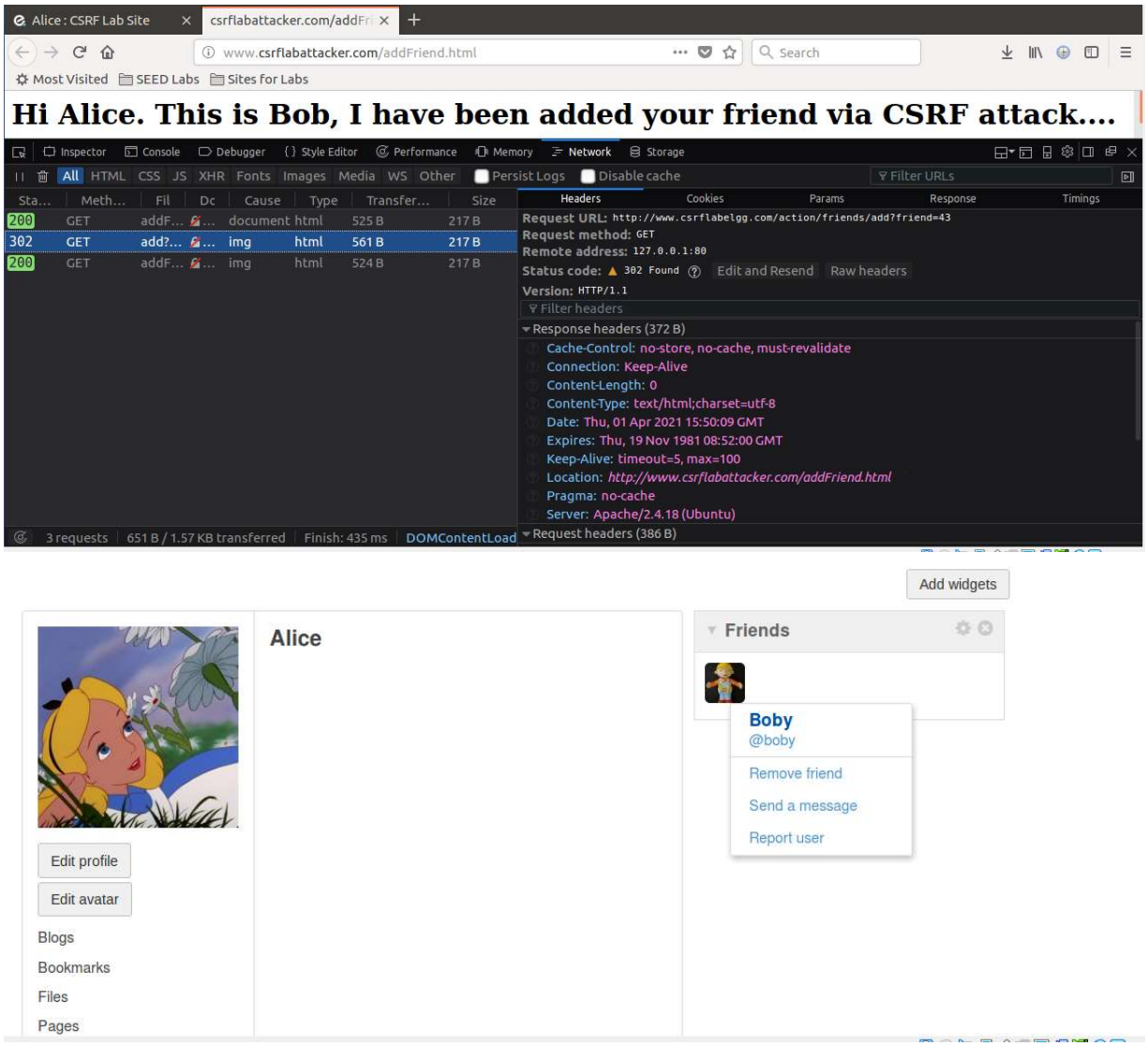
Now, in order to see if the attack was successful, we log into Alice's account because one of the prerequisites of CSRF attack to be successful is that the victim must have a valid session with the targeted website. On logging into Alice's account, we see that she does not have any friends currently:



Message from Boby with malicious link:



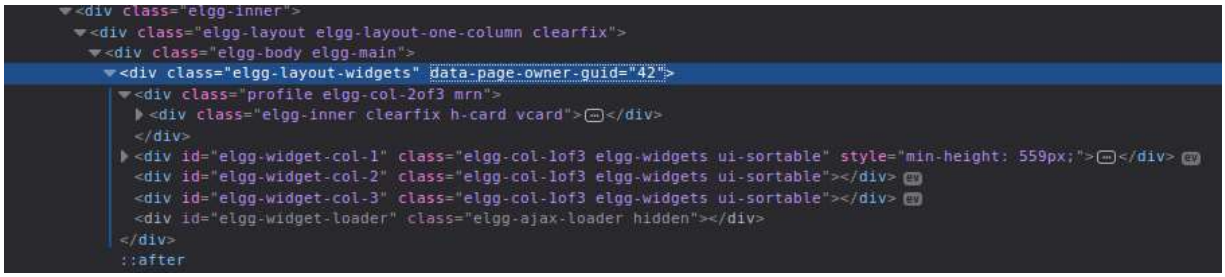
The following proves that Boby has been added as a friend:



Hence, we were successful in adding Boby as Alice's friend without Alice's intention. The following shows the content of the HTTP request when the malicious website loads and we see that the URL that we specified is sent in an HTTP GET request as soon as the link is clicked, and friend with GUID 43 is added to the current session, that of Alice.

TASK 3:

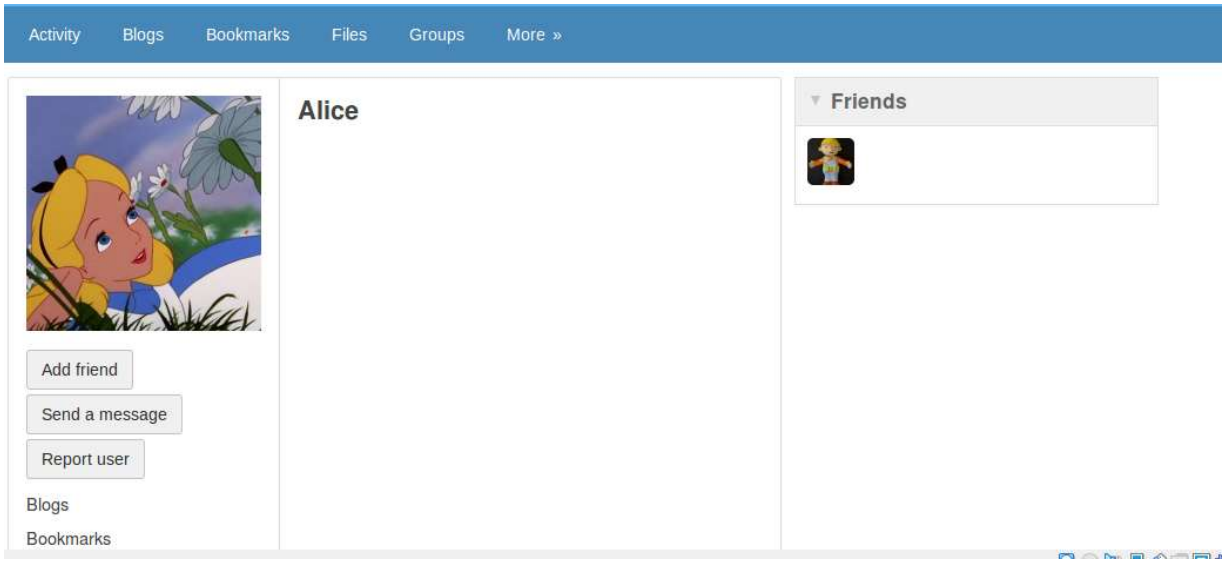
In order to find the GUID of Alice, all we need to do is search for her profile from anyone's account on the website and then use the inspect element feature to look for the web page owner's guid – which will indicate Alice's GUID. Here we get her guid as 42.



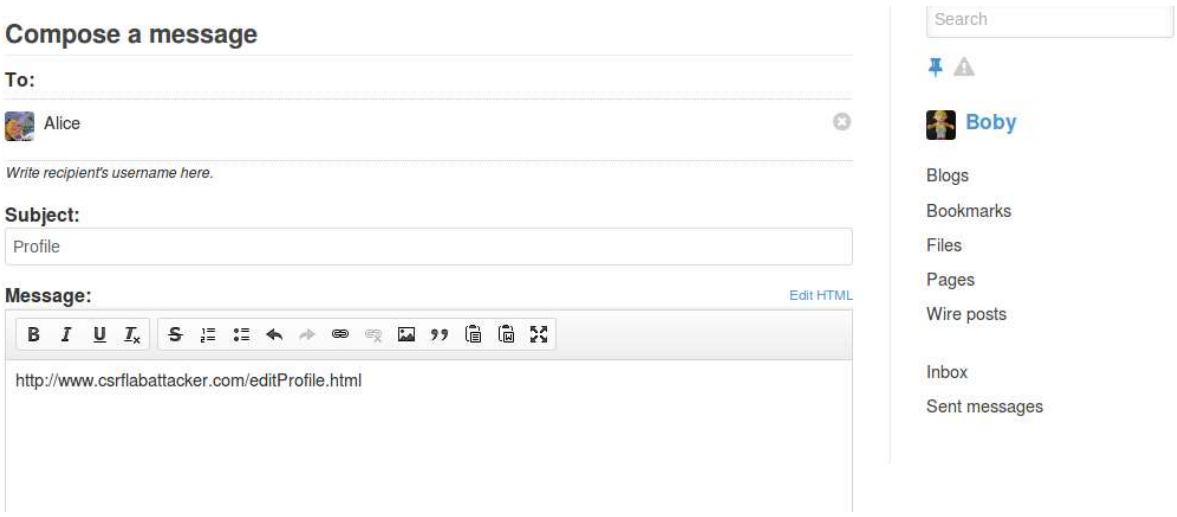
Here, we see that Alice's guid is 42. Using this value, we construct a web page named editProfile.html in the var/www/CSRF/Attacker folder, which is associated with the malicious web page. We set the name and guid to that of Alice's and Description is what we want to store i.e. 'Boby is my Hero'. We also set the access level of the description to 2, so that we can see the changes. The URL for the POST request is the one of the targeted site with edit profile open. The program of the website is given in the next screenshot:

```
<html>
<body>
        <h1>This page forges an HTTP POST request.</h1>
        <script type="text/javascript">

        function forge_post()
        {
                var fields;
                fields += "<input type='hidden' name='name' value='Alice'>";
                fields += "<input type='hidden' name='description' value='Profile Edited'>";
                fields += "<input type='hidden' name='accesslevel[description]' value='2'>";
                fields += "<input type='hidden' name='description' value='Boby is my Hero'>";
                fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
                fields += "<input type='hidden' name='location' value='INDIA'>";
                fields += "<input type='hidden' name='accesslevel[location]' value='2'>";
                fields += "<input type='hidden' name='guid' value='42'>";

                var p = document.createElement("form");

                p.action = "http://www.csrflabelgg.com/action/profile/edit";
                p.innerHTML = fields;
                p.method = "post";

                document.body.appendChild(p);
                p.submit();

        }

        window.onload = function() { forge_post(); }
        </script>
</body>
</html>
```

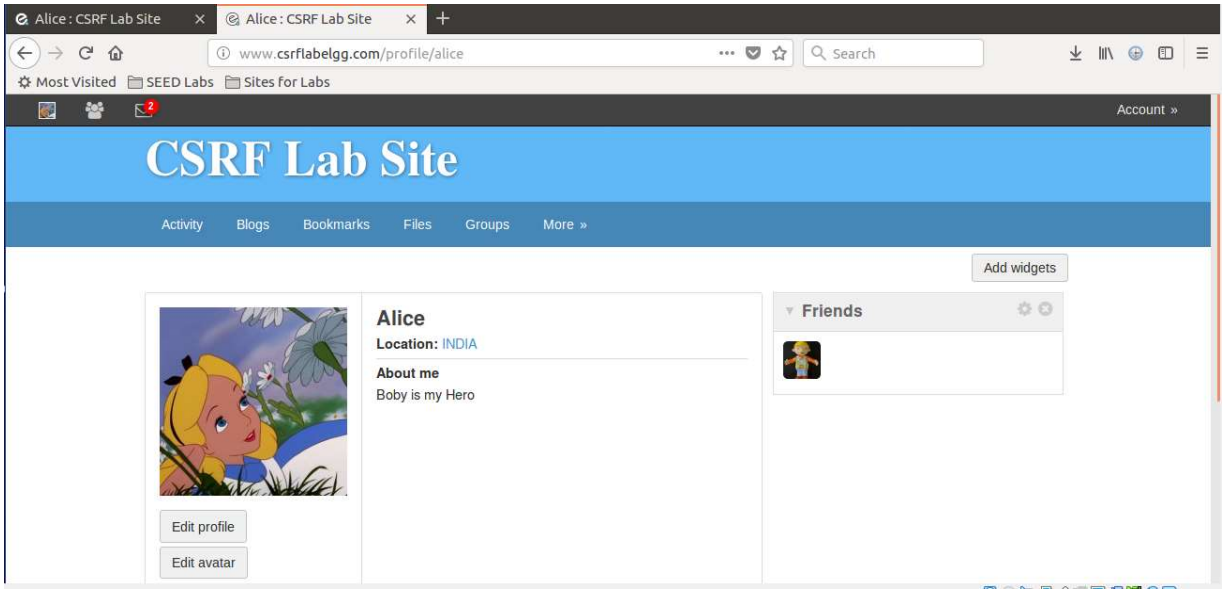
Next, in order to demonstrate a successful attack, we login into Alice's account. The following shows her current profile data:



Boby sending mail to Alice with malicious link to edit profile:

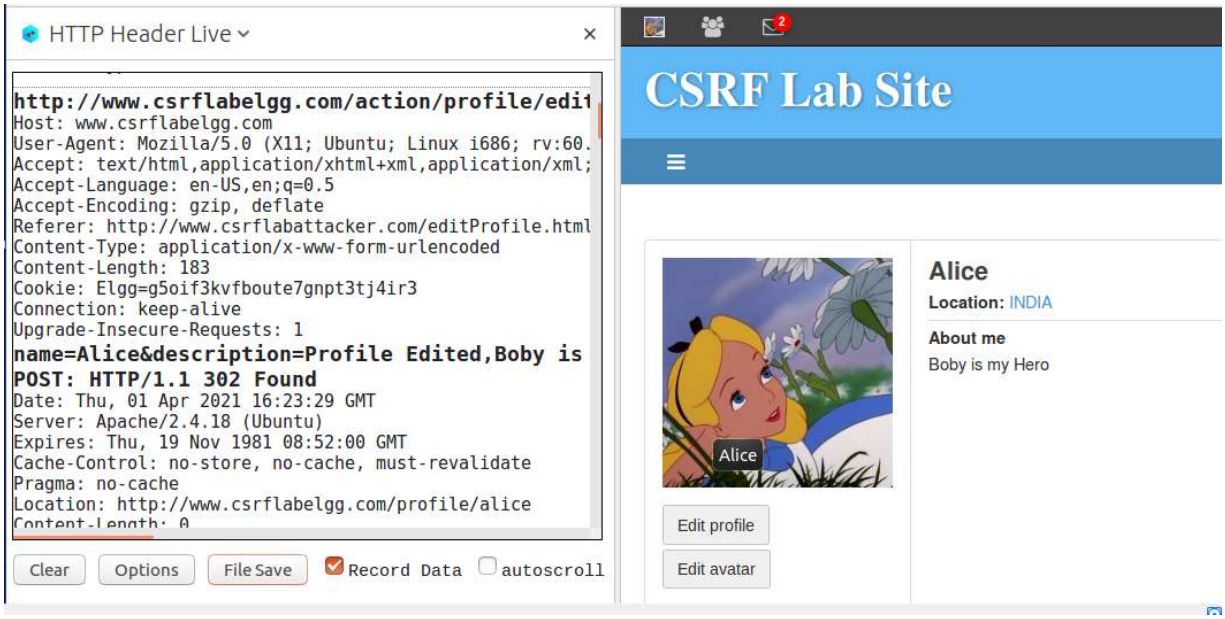


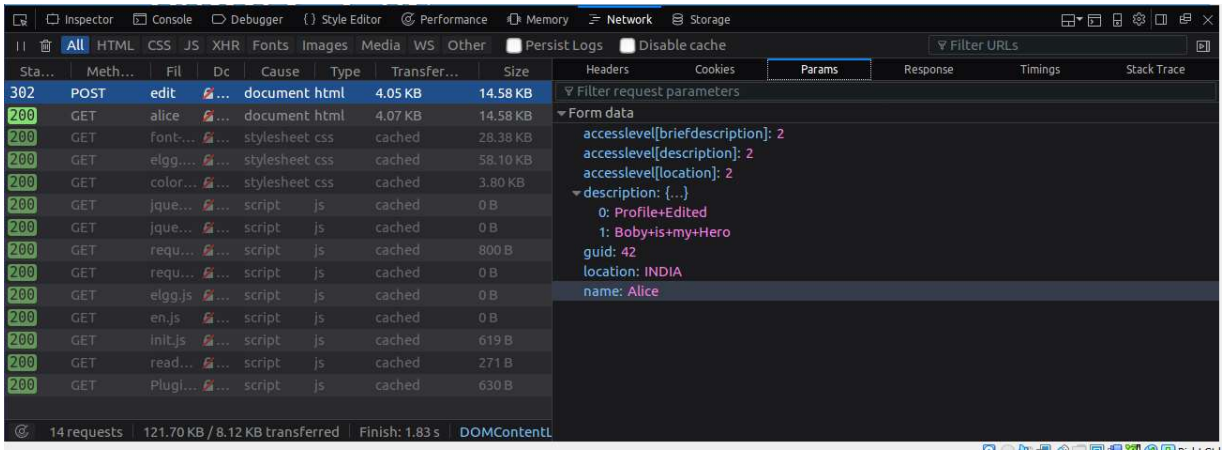
Then, we assume that Alice clicks on the link to the malicious website sent to her in a message and we see the following web page:



This indicates that we were able to successfully edit Alice's profile.

The following HTTP Request is generated as a result of clicking on the malicious website.





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

-> This problem can be solved in the way we found Alice's GUID, by searching Alice on the platform and then doing an inspect element to see the web page owner's GUID. This does not require to know Alice's credentials. If the website did not contain any GUID in its source code, hence we wouldn't be able to use the first approach, we can try to just entire Alice's name as username and some random password and look at the HTTP Request or Response. If any of those had Alice's GUID, we could use that as well.

Question 2: If Boby 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.

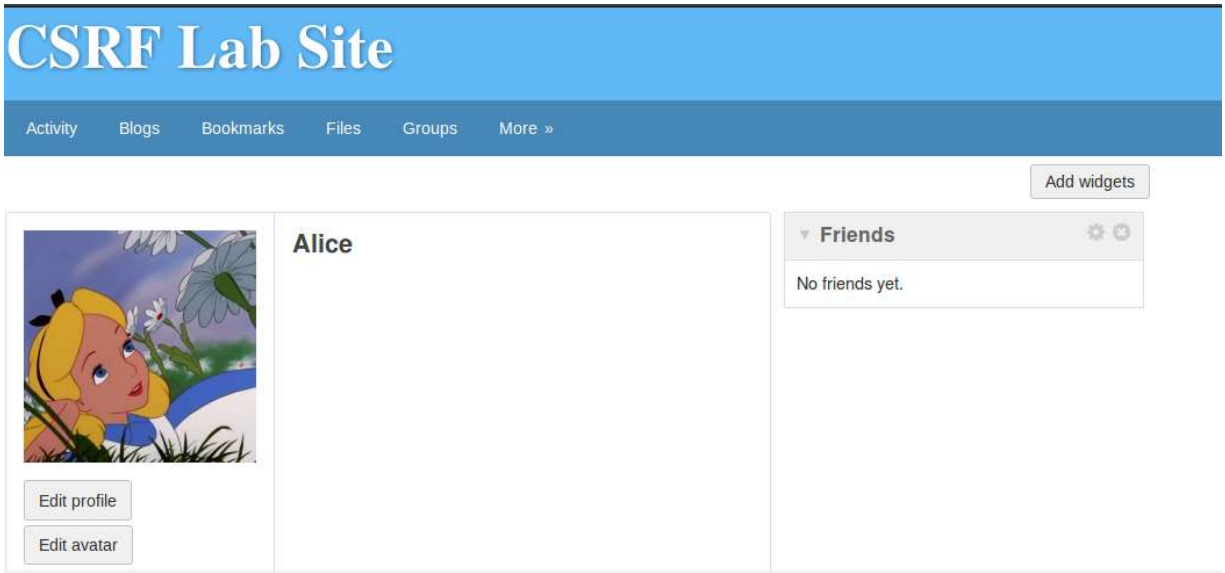
-> In this case, Boby would not be able to launch the CSRF attack, because his malicious web page is different than that of the targeted website. In that case, we do not have access to the targeted website's source code and hence cannot derive the GUID as before. Also, the GUID is sent only to the targeted website's server and not to any other website, hence we would not get the GUID from the HTTP request from elgg to the attacker's website.

TASK 4:

We now enable the CSRF countermeasure by commenting out the return True statement. Due to this statement, the function always returned true, even when the token did not match. So, by commenting it out, we are performing the check on token and timestamp and only if they are the same, we return true. If the tokens are not present or invalid, the action is denied, and the user is redirected. This can be seen in the following:

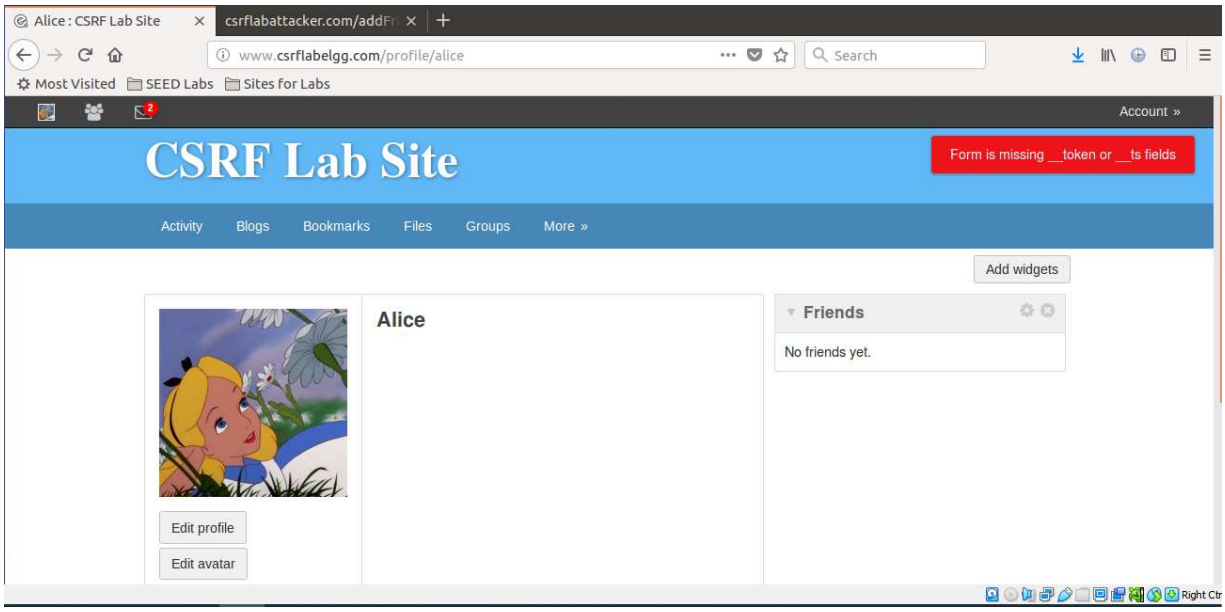
```
/**
 * @see action_gatekeeper
 * @access private
 */
public function gatekeeper($action) {
        //return true;

        if ($action === 'login') {
                if ($this->validateActionToken(false)) {
                        return true;
                }

                $token = get_input('__elgg_token');
                $ts = (int)get_input('__elgg_ts');
                if ($token && $this->validateTokenTimestamp($ts)) {
                        // The tokens are present and the time looks valid: this is probably a mismatch due to the
                        // login form being on a different domain.
                        register_error(_elgg_services()->translator->translate('actiongatekeeper:crosssitelogin'));

                        forward('login', 'csrf');
                }
```

I removed the results of the previously done attacks, to have Alice's account as follows:

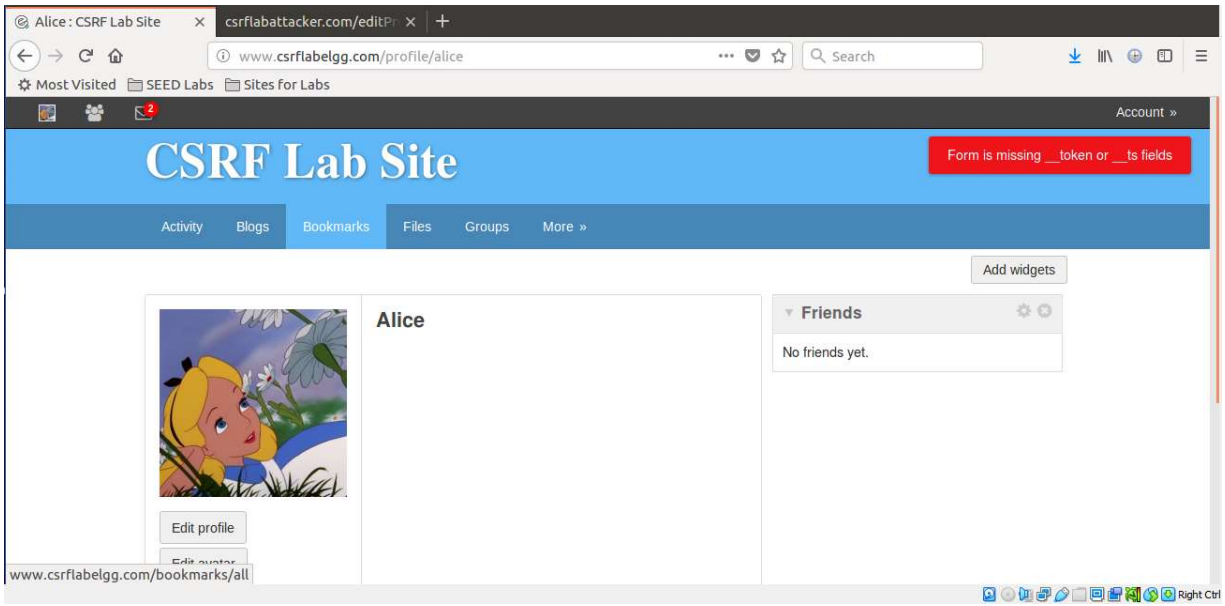


Elgg security token is a hash value (md5 message digest) of the site secret value (retrieved from database), timestamp, user sessionID and random generated session string.

We then perform the same attacks. On doing GET request attack:



On doing the POST request attack:



We see that we see an error during both the times and our attack is unsuccessful. We get the error that the token and timestamp fields are missing and hence the action was not performed successfully. On looking at the HTTP headers for both GET and POST requests, we see no token and ts fields being sent. This is because we are constructing the HTTP request and have not specified any parameters for timestamp and secret token.

We can see these secret tokens when we are logged in into Alice's account, but any other user on the platform will not have Alice's credentials and hence won't be able to find these values. Also, anyone cannot guess these values because even though it's easy to find the timestamp value, we need two values to pass the test – timestamp and the secret token, and the secret token is a hash value of the site secret value – that is retrieved from the database, timestamp, user sessionID and random generated session string. Even though we would know the timestamp and user sessionID from the previous practices, it is impossible to get the site's secret value which is stored in its own secret database and a string that is generated randomly. Hence, the attacker cannot guess nor find out – that requires having valid credentials – the secret tokens, and hence the attack will not be successful anymore.