Tanmay Fadnavis

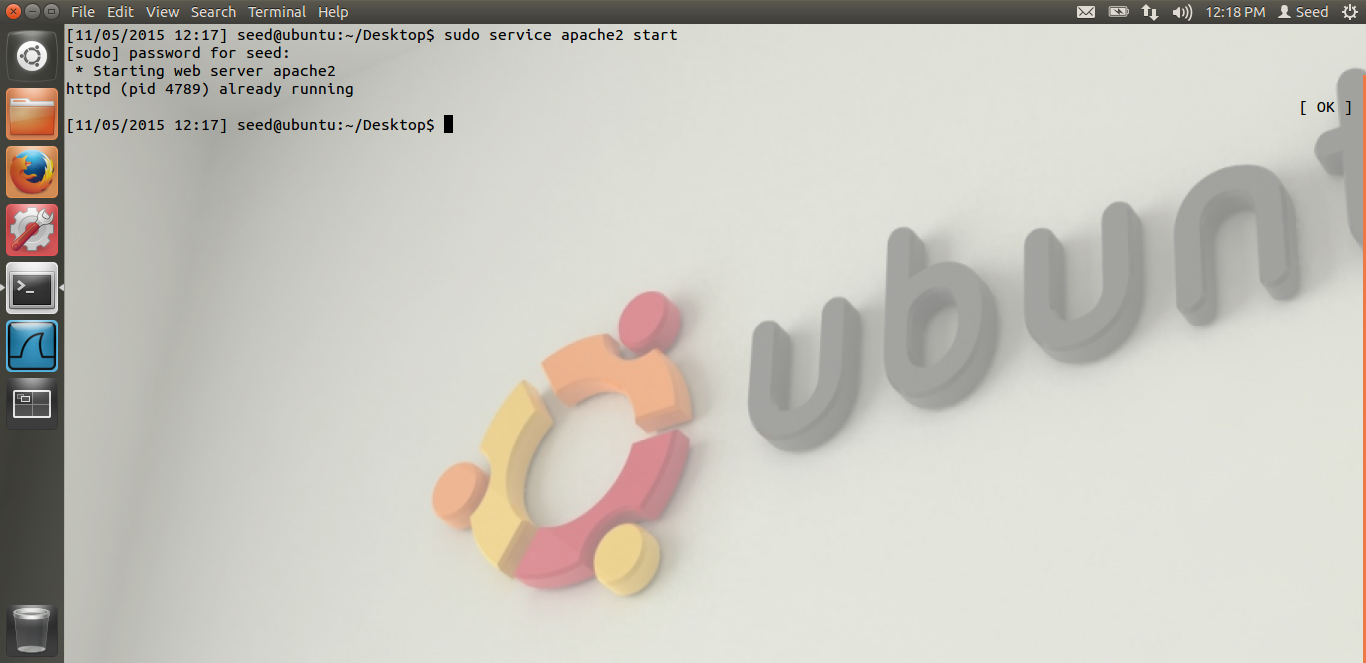
SUID: 971141760

Computer Security

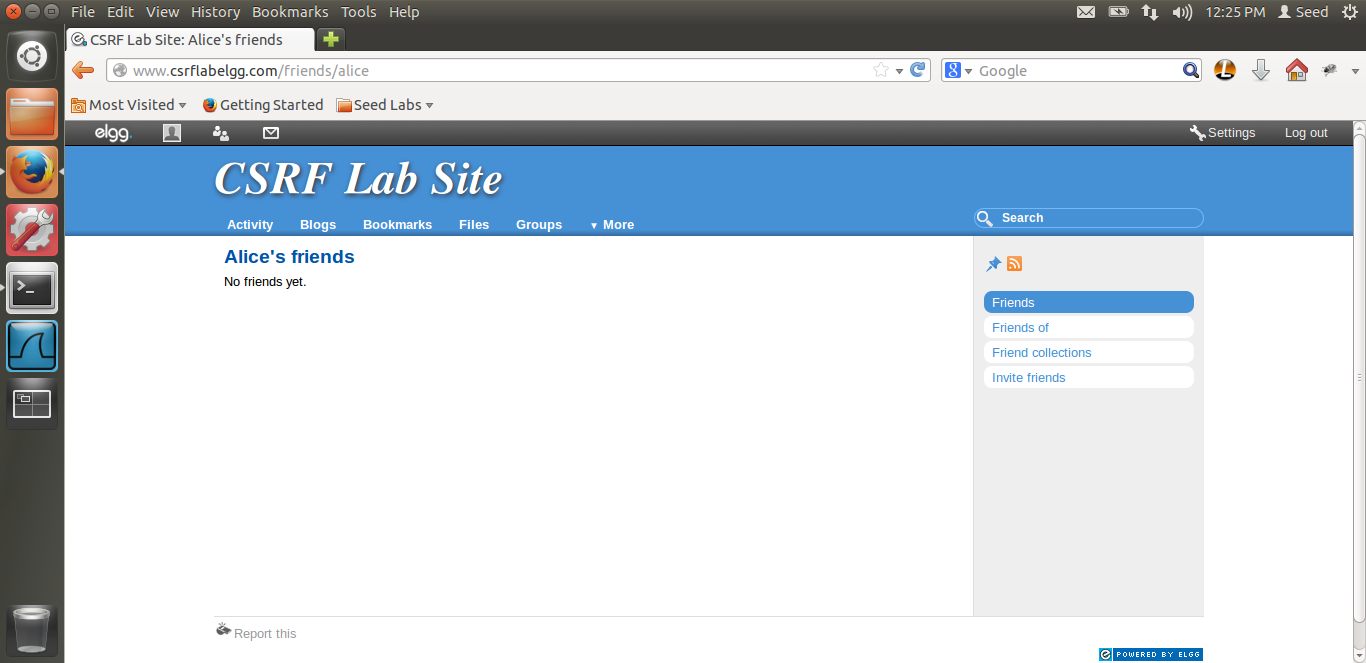
Lab-9

# TASK 1: CSRF ATTACK USING GET REQUEST

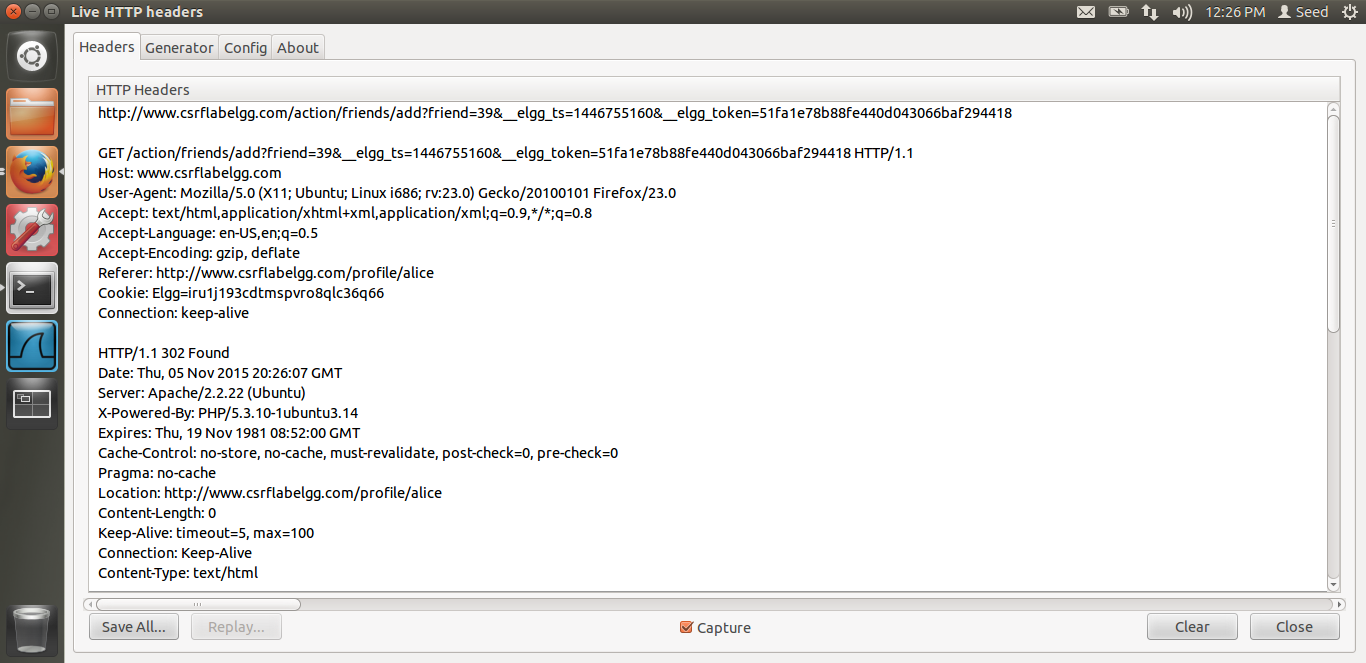
The screen shots for task 1 are below.



Here, in the above screen shot, we can see that I have started the apache server.



This is the screen shot of Alice’s friends page before the CSRF attack. We can see that Alice has no friends as of now. Now, Boby will try the CSRF attack on alice and lets see whats the output.

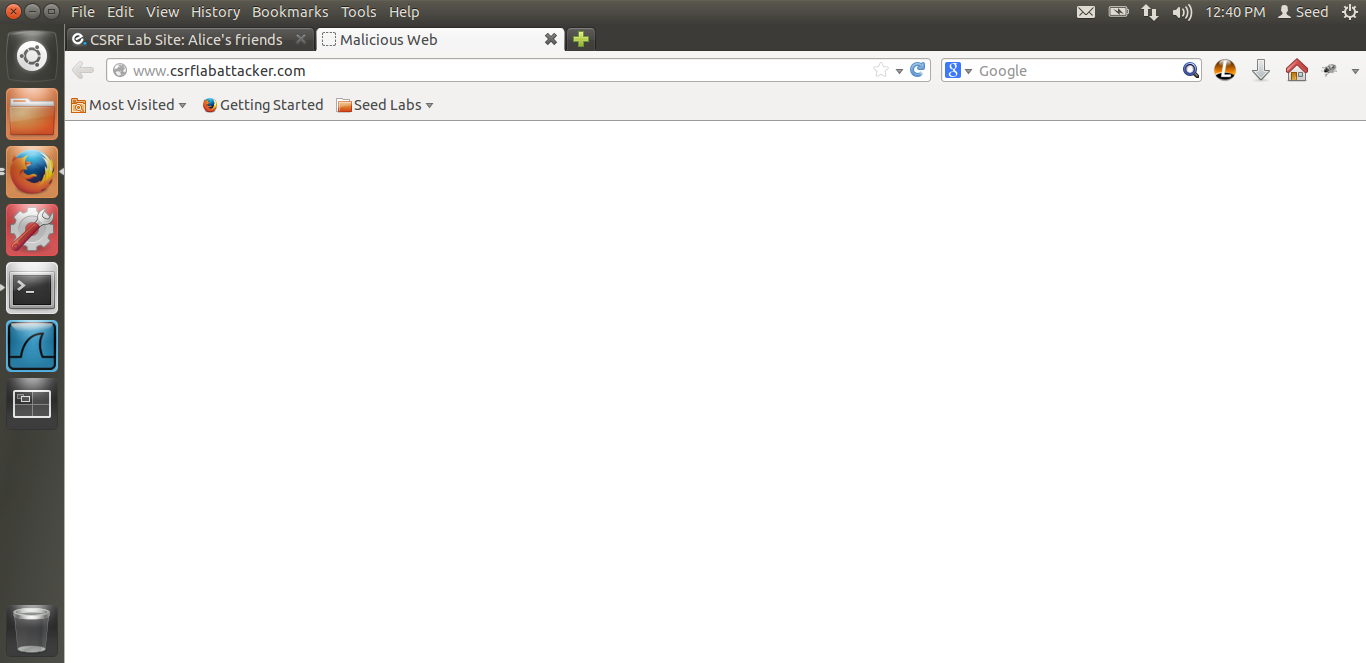


From the live HTTP header, we can see the URL which is being hit, when we add a friend, on the csrflabelgg site. We just need Boby’s GUID, which is 40. Below we can see the exact URL to be hit. As this is a GET request, we just need to put the img src i.e. the image tag’s source as the url above. So, once Alice visits our malicious webpage of the csrf attacker, the src in the image tag will be hit. As the source is the url to add alice as a friend, our attack is launched. Also, one important thing is, Alice needs to be logged in for this attack is successful.

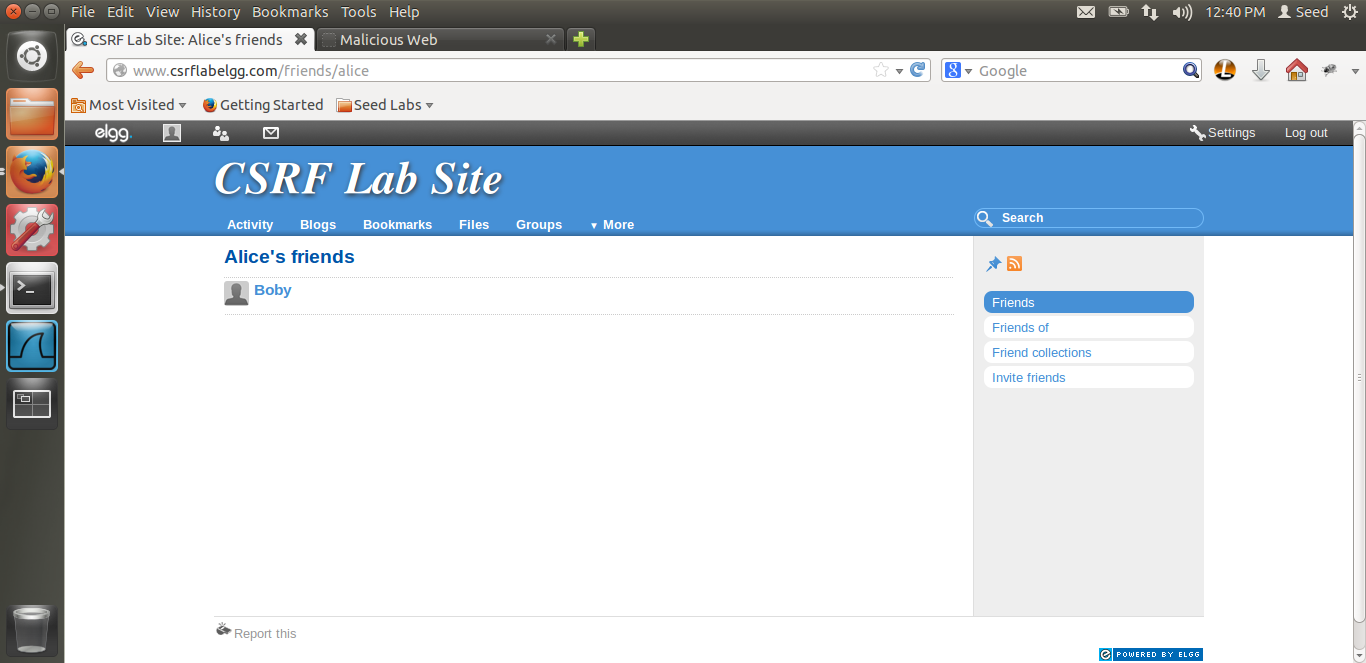
Reason: Reason is, we are hitting the csrflabelgg.com url. Thus, Cross site scripting works because, when we hit a particular web-site the browser sends all the cookies related to that website along with the request. Hence, Alice needs to be logged in, to get her information from the cookies. Thus, CSRF attack works. Now let us look at the code and the output.



This is the code above. This is the index.html file, this webpage is loaded when Alice visits the malicious ww.csrfattacker.com In this page, you can see the img tag and the src is the code to add Boby as a friend. Thus, as soon as Alice visits our malicious web page, this URL is hit and Boby is added as a friend. Let us see whether the attack is successful.

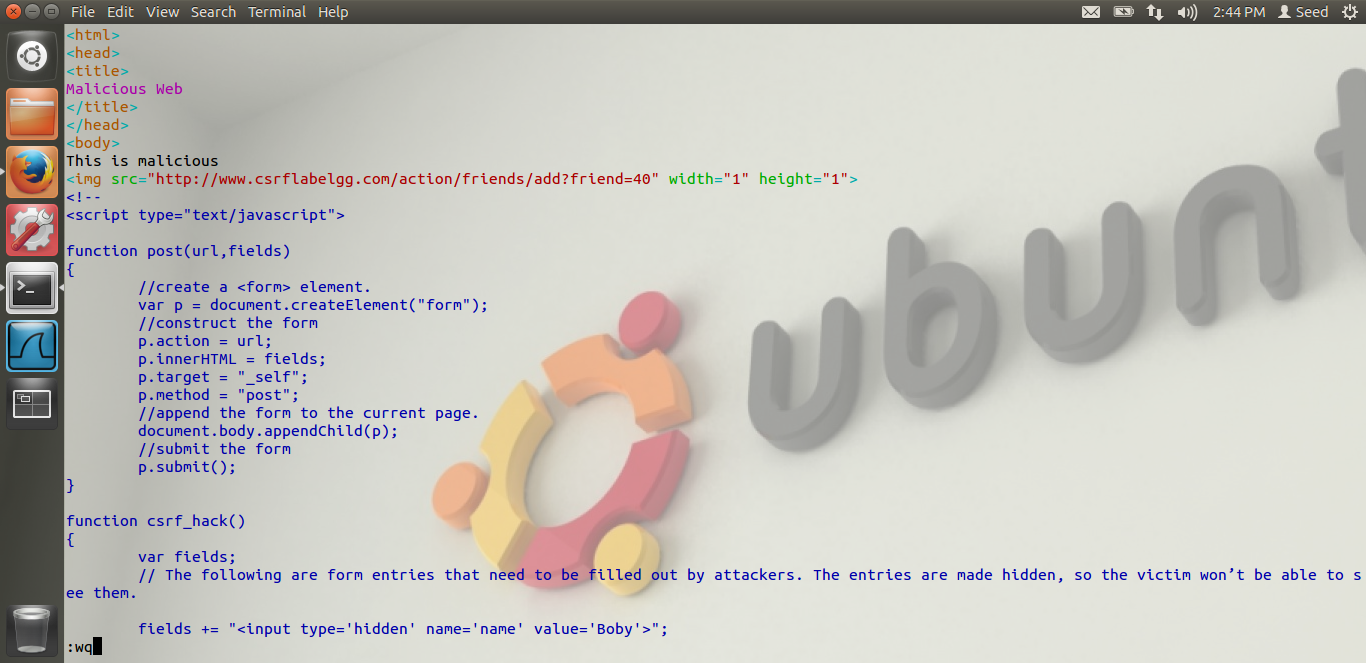


Thus, alice has visited our malicious web-site.



Now, you can see that Boby has been added as a friend as soon as we re-fresh the web-page. Thus our CSRF attack is successful. Reason, as I explained above, the src in the img tag is the url to add Boby as a friend. This img tag will generate a GET request for this. Also, as alice has an active session with elgg, all the cookies related to elgg will be sent by the browser and thus the CSRF attack will be launched.

But, as I was performing the countermeasure, I realized that, as the countermeasure was off, we don’t even need the elgg\_ts and the elgg\_token in the GET url. Hence I tried the attack besides that and the attack works. Below is the snapshot of the code.

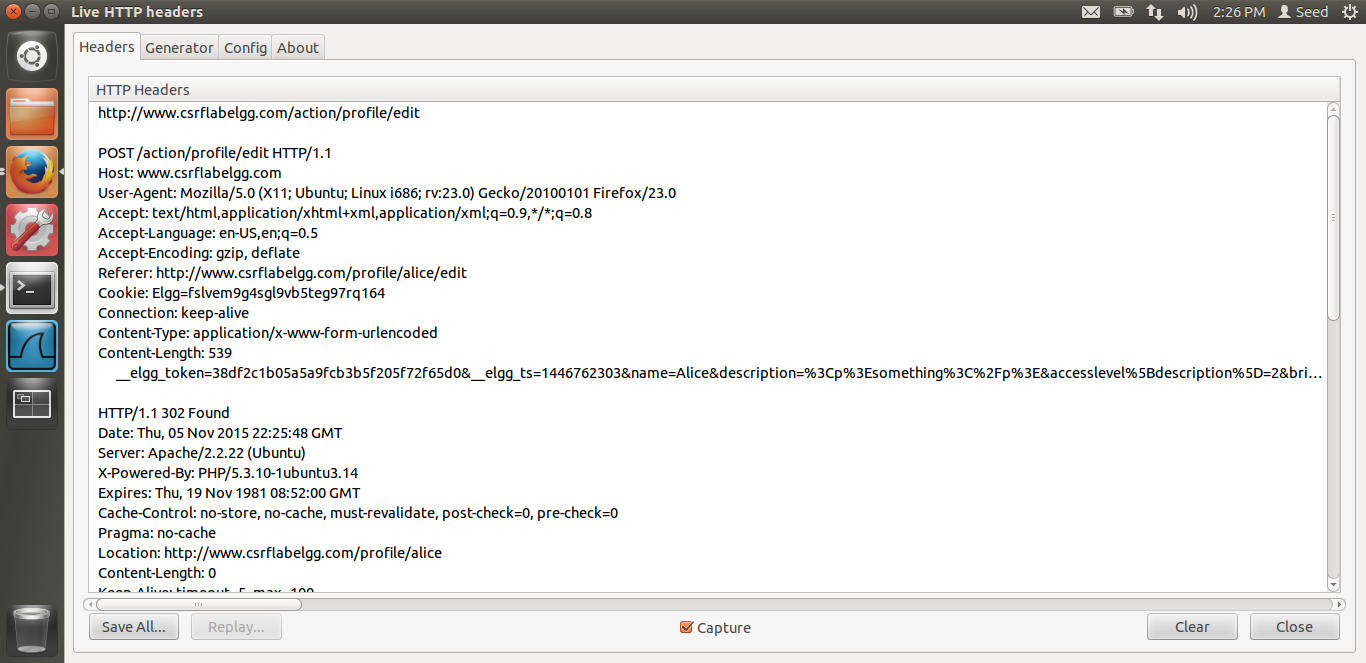


As you can see, the link only has the url to add the friend, but it does not contain the elgg\_ts and the elgg\_token value as the counter measure is off. Thus, my task-1 is completed.

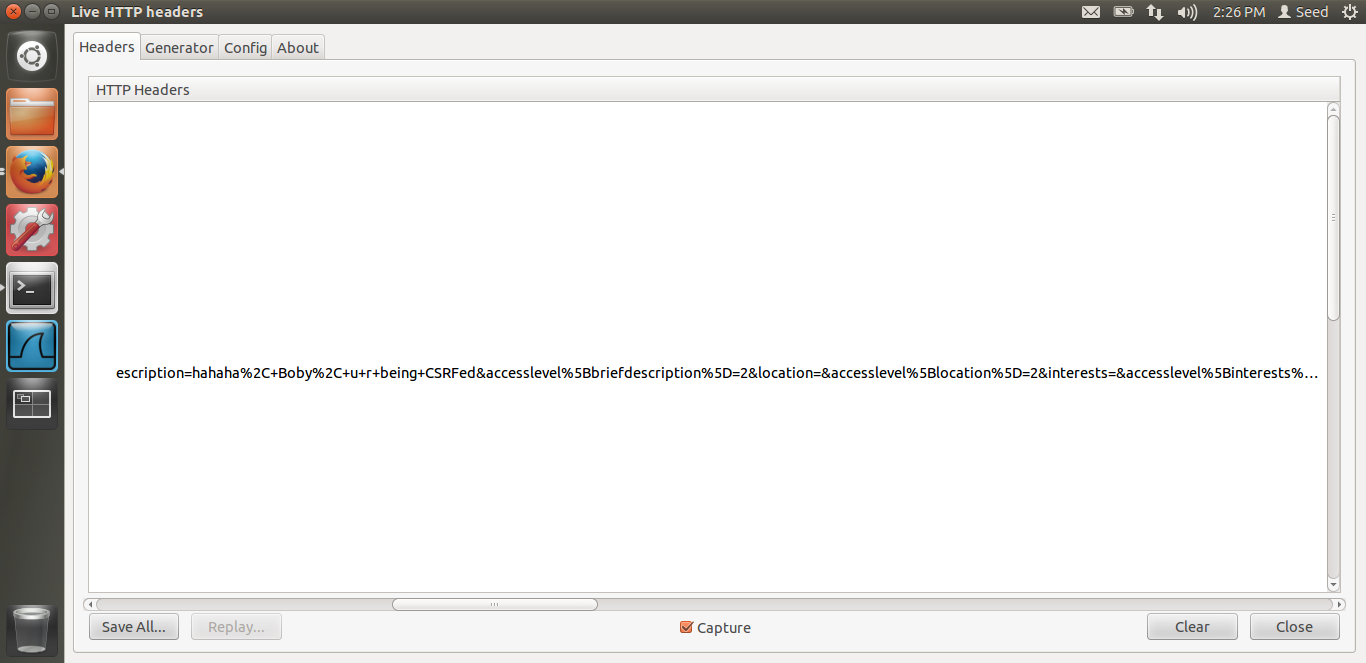
# TASK 2: CSRF ATTACK USING POST REQUEST

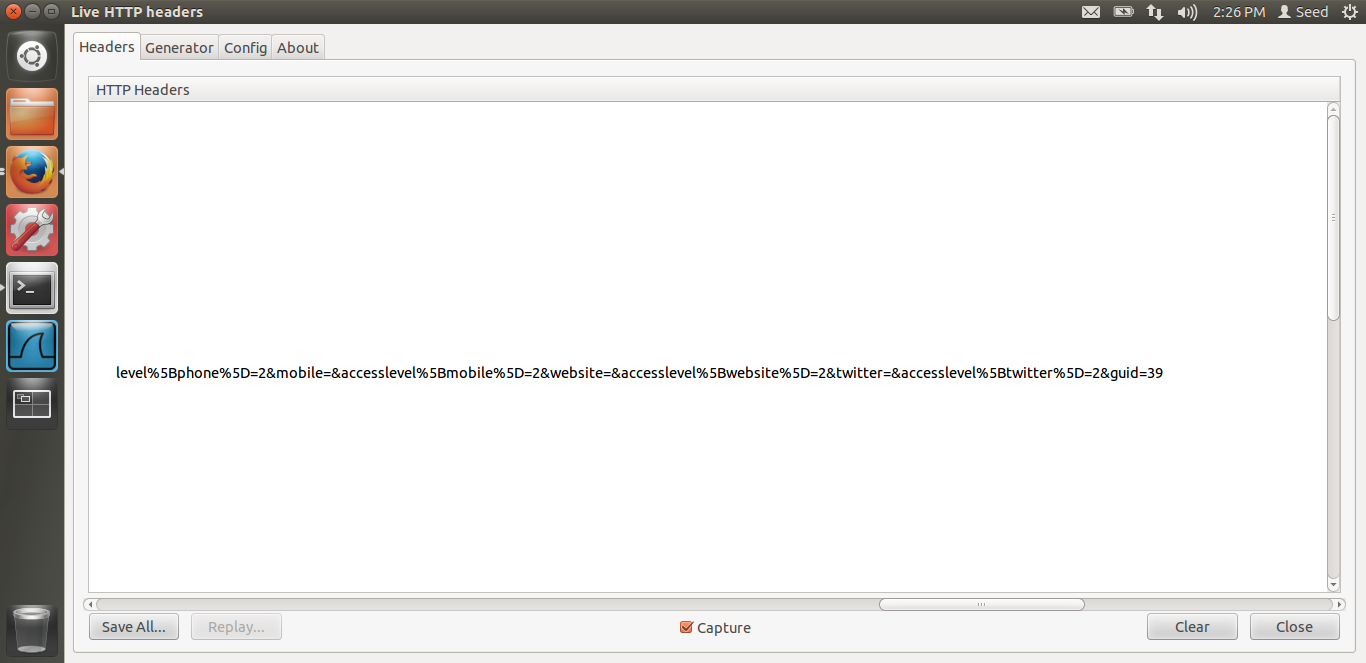
The screen shots for Task-2 are below.

In this task, Alice is the attacker now and Boby is the victim.

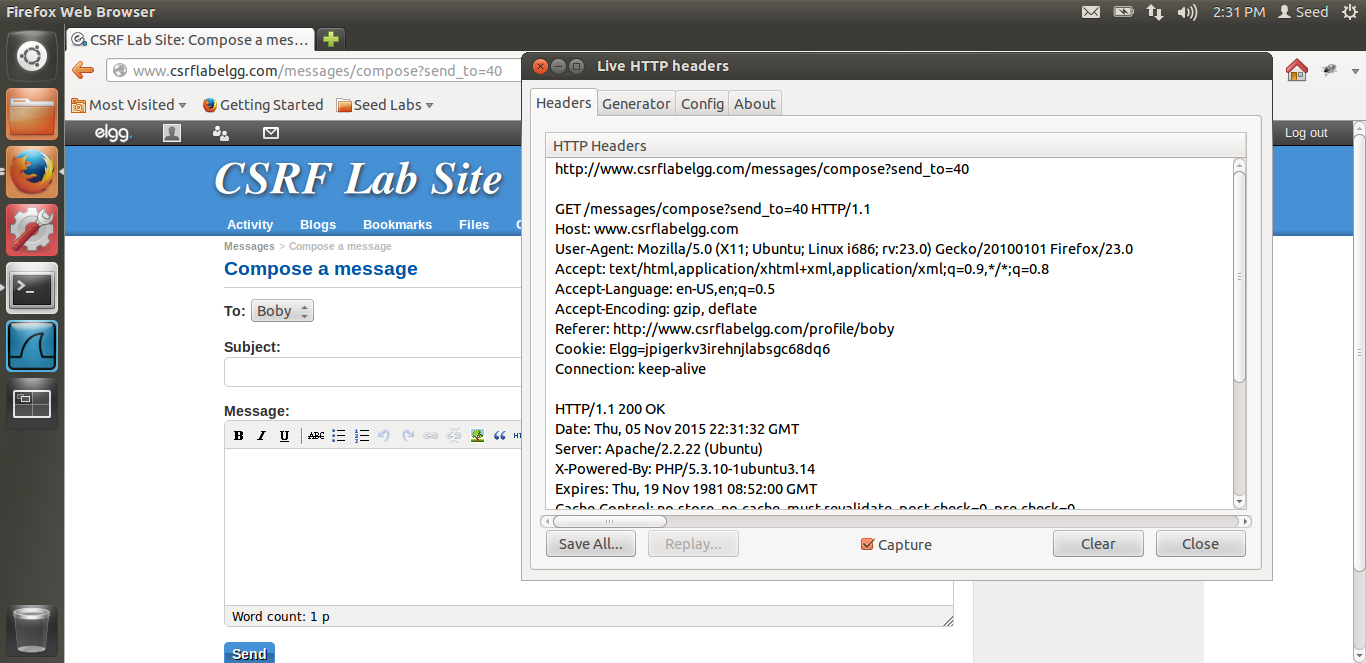


I just edited Alice’s profile to see the Live Http header and the url being hit. As you can see from the screen shot, the url is /profile/edit and rest of the information is in the content part, i.e. brief description, description, etc.

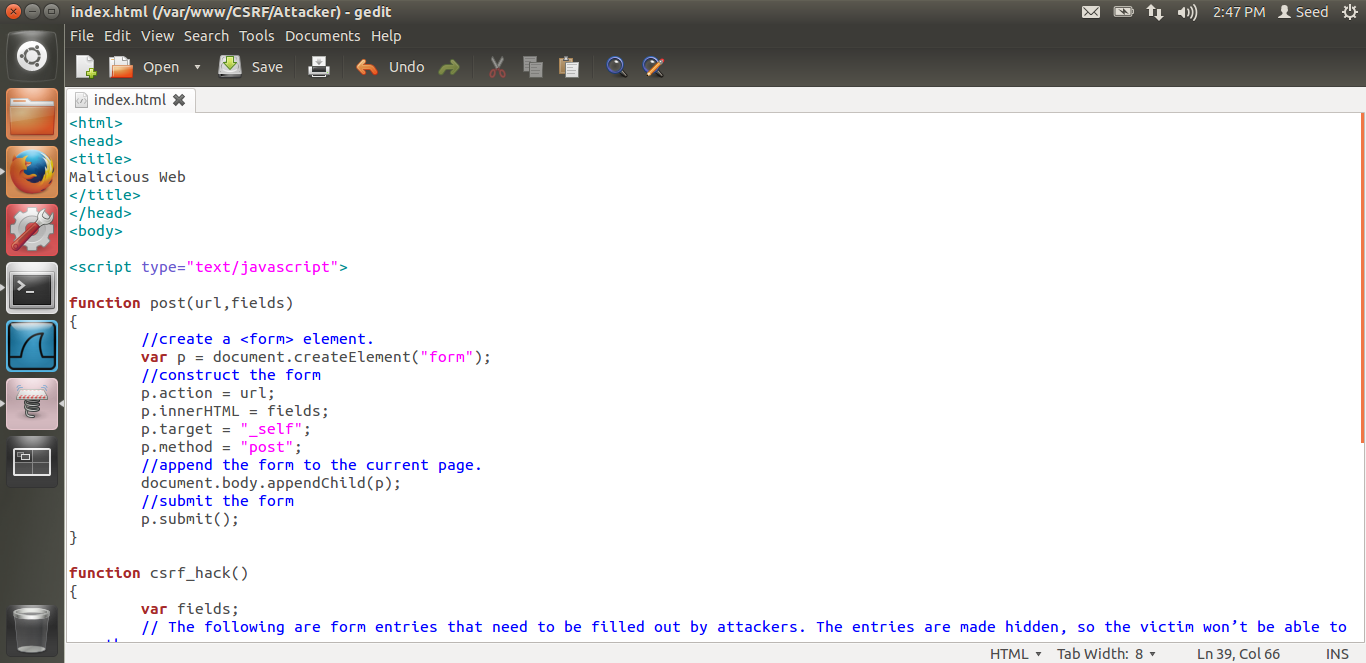




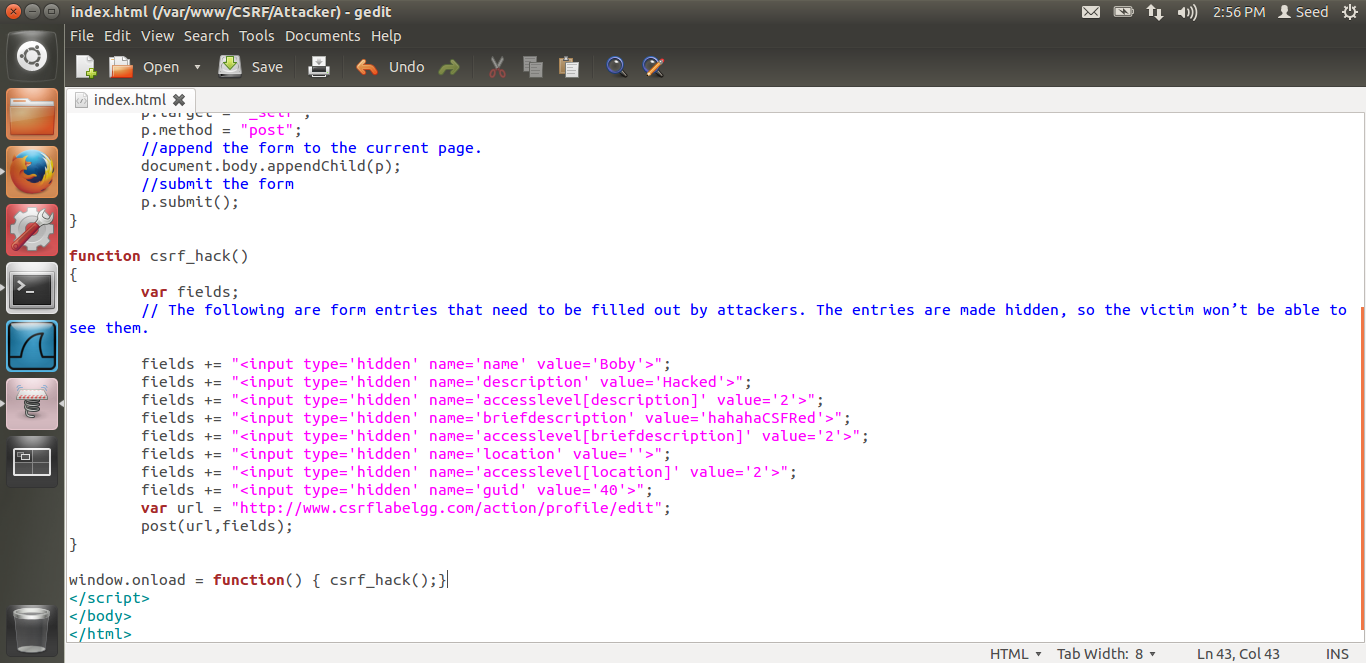
These are above just the screen shots for the same request. In the end, as you can see, we need the victim’s GUID which is being sent along with the content.



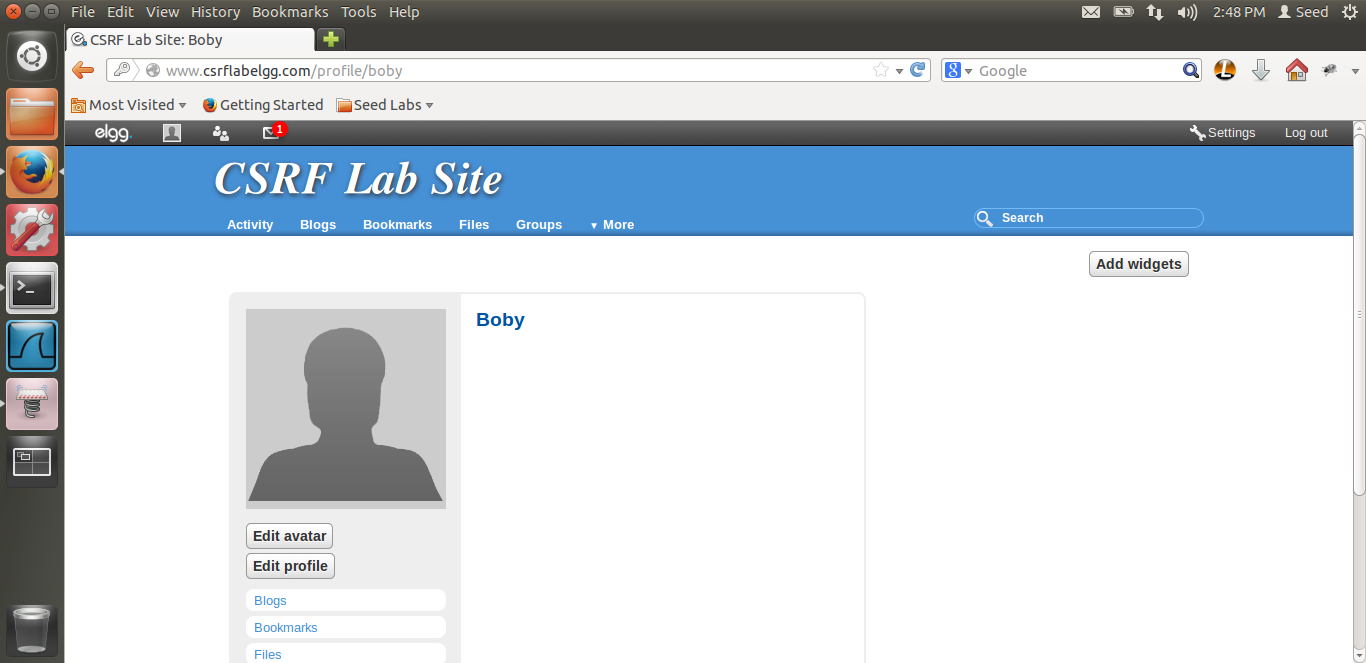
Now, we need to attack Boby. Hence, we need Boby’s GUID in the request. What I did was, as Boby is already my friend, i.e. alice’s friend, alice just wants to modify Boby’s profile, I tried to send message to Boby from alice. I saw in the LiveHttpHeader. As you can see from the screen shot, in the URL the send\_to=40, i.e. this must be Boby’s GUID. Thus, alice has got Boby’s GUID. Now we just need to write the javascript for the POST request. The sample code is given by Dr. Du. I am just adding in few details. Let us see the code.



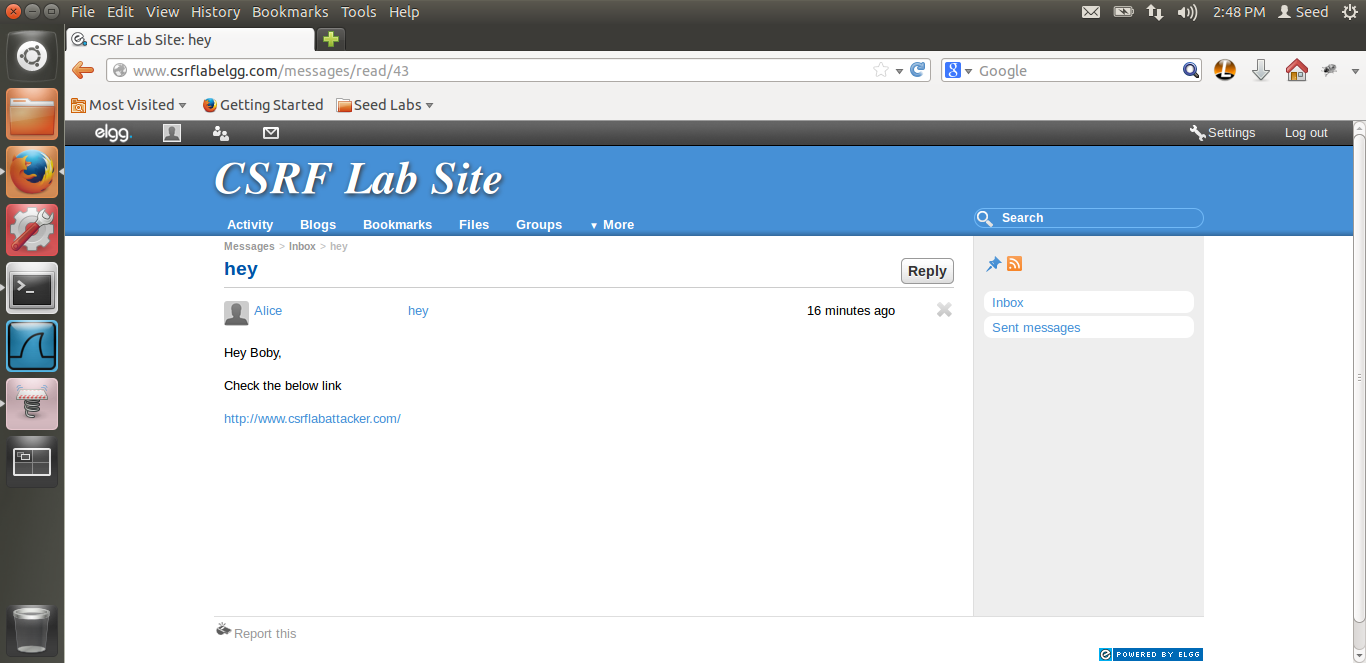
This is the code given by Dr. Du. This code, I am again modifying in the index.html, the webpage which is loaded when Boby will visit the malicious website. As you can see there is a post function which takes the url and the fields. It dynamically generates the form, The form is hidden from Boby. This form has all the information for the attack. Now, using the form , we are generating the POST request. This form is hidden. Also we are simulating the submit button. Whenever the submit button is clicked in the form, a POST request is sent to the url, which is mentioned in the form’s url field. Thus, I am generating the POST request and doing the CSRF POST attack. Let us see the rest of the code.



In the above screen shot, we can see that the form is hidden. In the name field, the name is Boby, i.e. Boby is the victim. In the description, I am just adding Hacked. This is the description field which will be edited/modified in Boby’s profile, with Hacked value. In the brief description, I am adding some data. In the guid, you can see I have put Boby’s GUID i.e. 40. And finally in the URL, I have put the /profile/edit URL. This I got from the liveHTTPHeader. Thus, also, this function is called when the window is loaded, window.onload, i.e. when Boby visits our malicious webpage. Thus the code is ready. Let us see whether the attack is successful.



Boby has logged in. You can see Boby’s profile before the attack. As you can see, there is nothing in his profile.



Boby checks his message. He has received a message from alice which has a link to the malicious website which alice has developed. Now Boby will visit the website. Let us see what happens.



Boom! As soon as Boby visited the website and then visited his profile, his profile has been edited. Thus, our CSRF POST attack is successful. You can see the edited profile in the screen shot above.

Thus, the reason I mentioned above. Due to the hidden form, the POST request was sent to the csrflabelgg server along with the cookies which has Boby’s aunthentication and other information and our POST CSRF attack was successful.

Questions in the lab.

Question 1: Alice needs Boby’s GUID. As described earlier, the same method can be used to get Boby’s GUID. I sent a message to Boby and checked the LiveHttpHeader. From the header I was able to get Boby’s GUID, from the url as described above. Thus, in a similar way. Alice can get Boby’s GUID by sending him a message and checking the request which is being sent. As the request involves Boby, definitely Boby’s GUID can be found out from there.

Question 2: The answer is No. Alice won’t be able to launch attack on anybody who visits her webpage. The reason being, for this attack, alice needs the victim’s GUID beforehand. Thus, even in the attack we made, Boby’s GUID was know and we need to put that GUID in the javascript code. As alice cannot know the GUID of all the people who are visiting her website, alice won’t be able to launch the attack on everyone. Thus the answer is No.

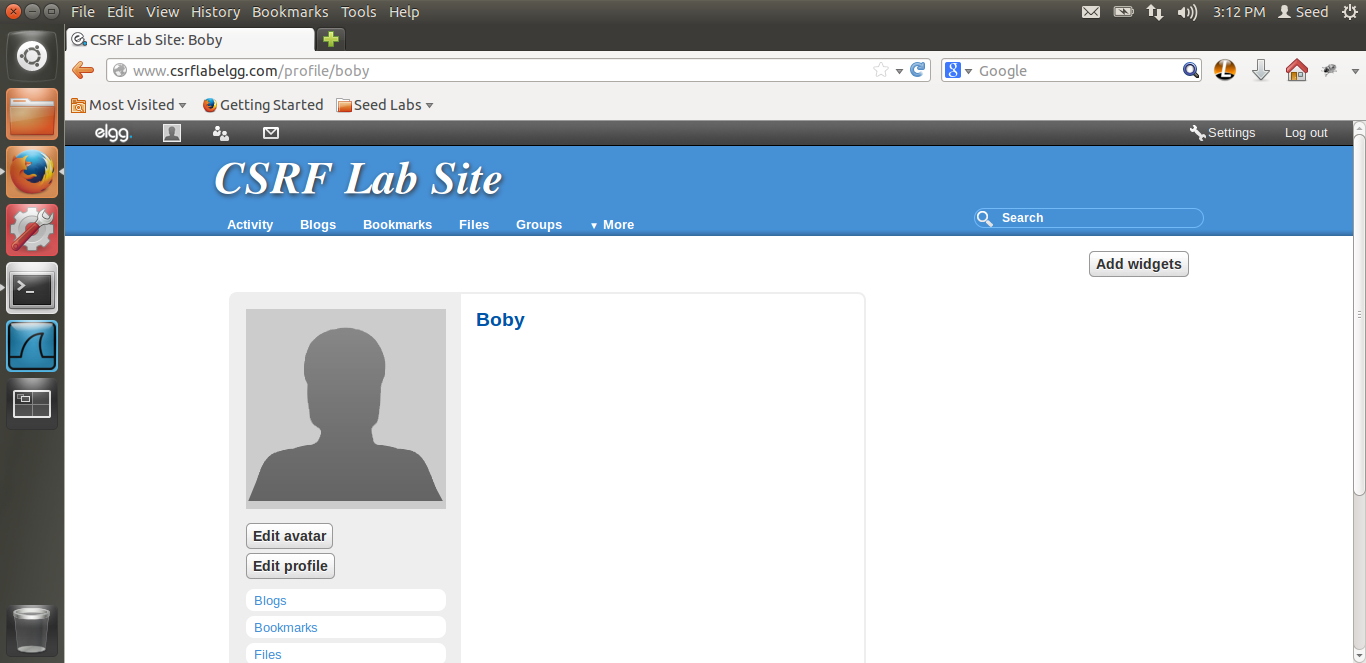
# TASK 3: IMPLEMENTING A COUNTER MEASURE FOR ELGG

The screen shots for Task-3 are below.

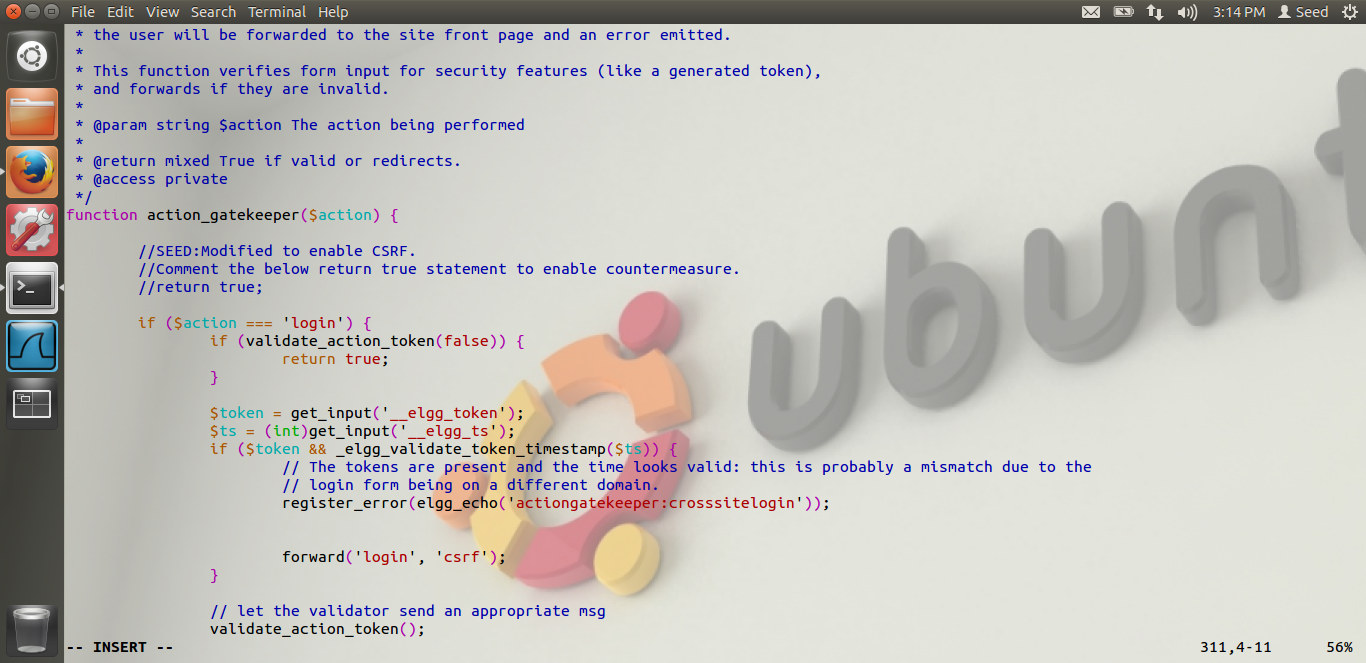
This is the task for countermeasures.

The Elgg application uses a secret token approach.

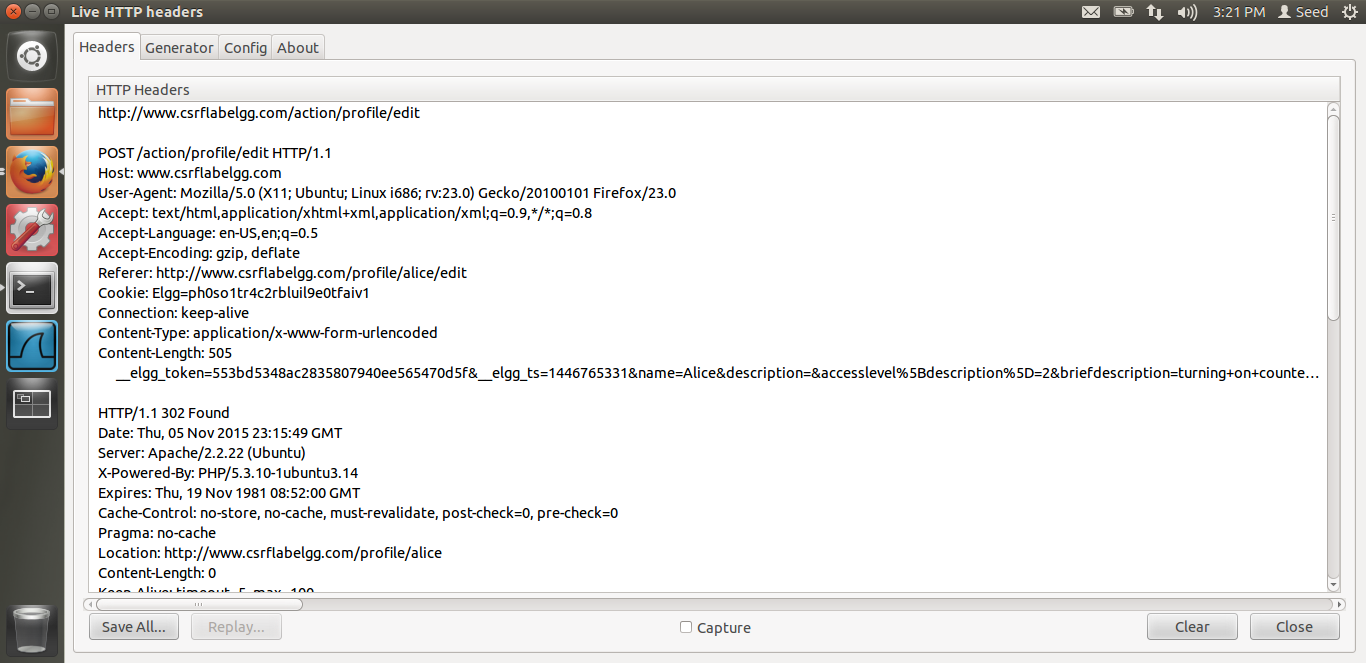
As described in the lab, elgg embeds 2 parameters elgg\_ts and elgg\_token in the request as the counter measure. These 2 parameters are added in the URL in the GET request and the HTTP message in the POST request. Thus, for every request, these 2 values are being checked as described in the lab. This is the countermeasure for the CSRF attack. Reason being, this is dynamically added in the request. As the request from the malicious website won’t have these tokens, automatically, the elgg server would detect that this is a Cross site request and thus, this would prevent the CSRF attack. Let us see what happens when I enable the counter measure.



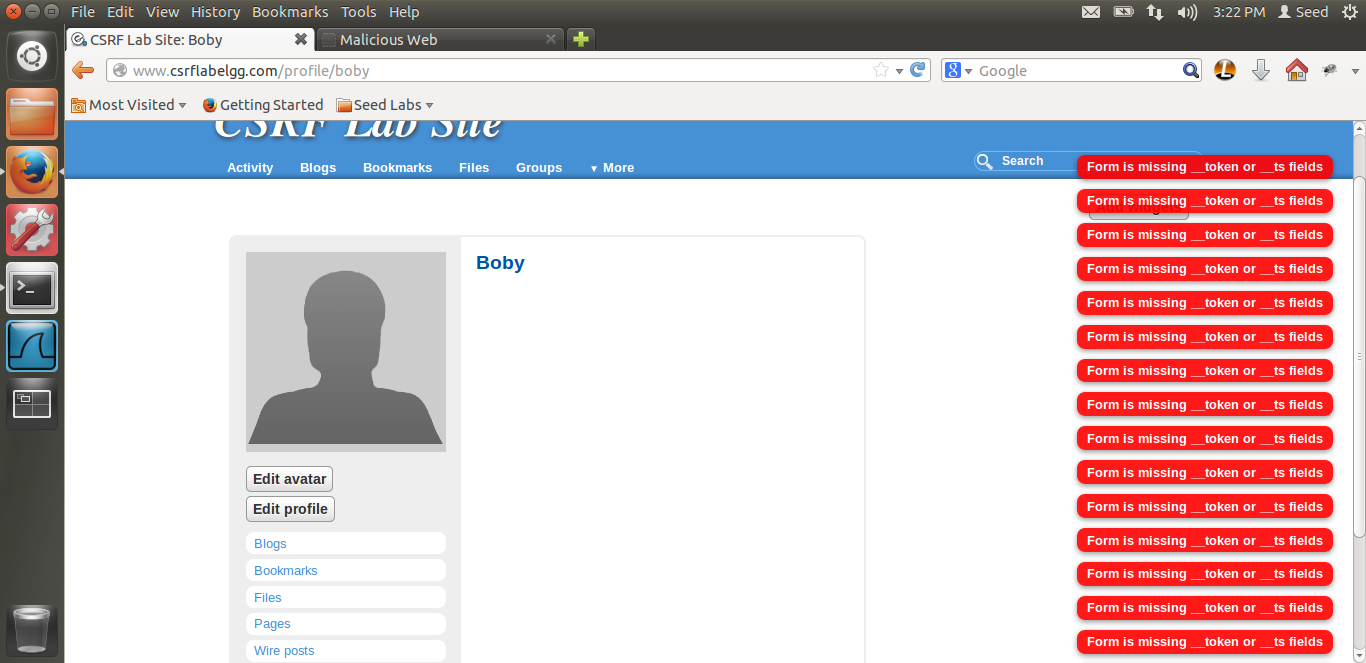
I cleaned Boby’s profile before the attack. This is the profile now.



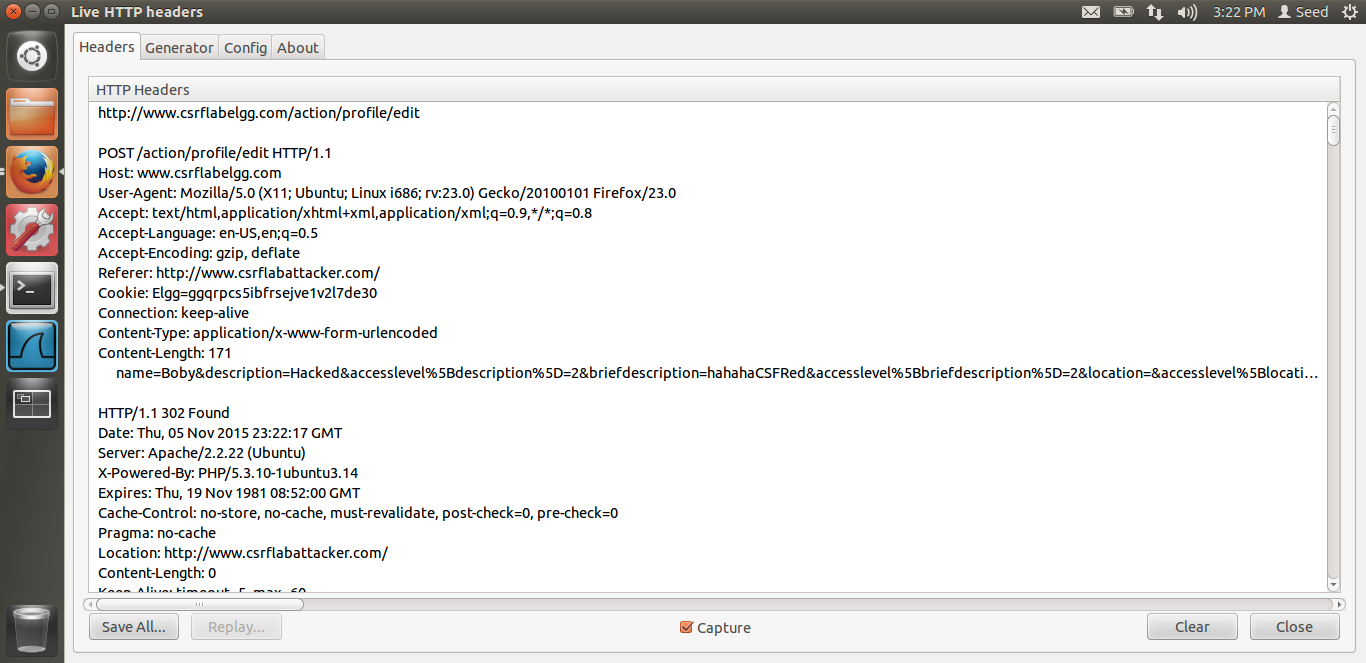
Thus, in the above screen shot, you can see that I have commented the return true line. Thus, I am enabling the counter measure.



In the above HTTP live header, we can see that, this is request from the actuall elgg webpage. In this, we can see that \_\_elgg\_token and the \_elgg\_ts value in the content.



After Boby visits the malicious web-page, you can see that the profile is not edited. Thus the CSRF attack of alice failed. Also you can see the errors, Form is missing \_token or \_ts fields. Thus, this shows that due to the missing elgg\_ts and the elgg\_token fileds, the elgg web server was successful able to defend against the cross site attack and thus the counter measure worked. Thus, the attack was not successful. Now let us see the request which the malicious website sends.



In the above screen shot, we can see the request which the malicious website sends. We can clearly see in the content, that the elgg\_ts and the elgg\_token value is missing. These values were present in the screen shot before, i.e. when the actuall website sends the request. Thus, I have proved the difference in the request sent by the actuall elgg page and the cross site page. We can see that its missing the elgg\_ts and the elgg\_token values. The corresponding error can seen on the screen.

The attacker, alice in our case cannot get these values, the secret token. The reason is that these values are in the webpage and not in the cookie. These values are generated dynamically as explained in the lab. As, the cross site attack is a different DOM structure altogether, it can only send the information present in the cookie of that web-page, as the cookies are attached by the browser. But the attacker’s webpage has no access to the original victim’s dom. The secret token values are in the DOM, hence the attacker cannot send these values and hence the CSRF attack is unsuccessful. Thus, we have completed task-3 and Lab-9.