Spring 2020

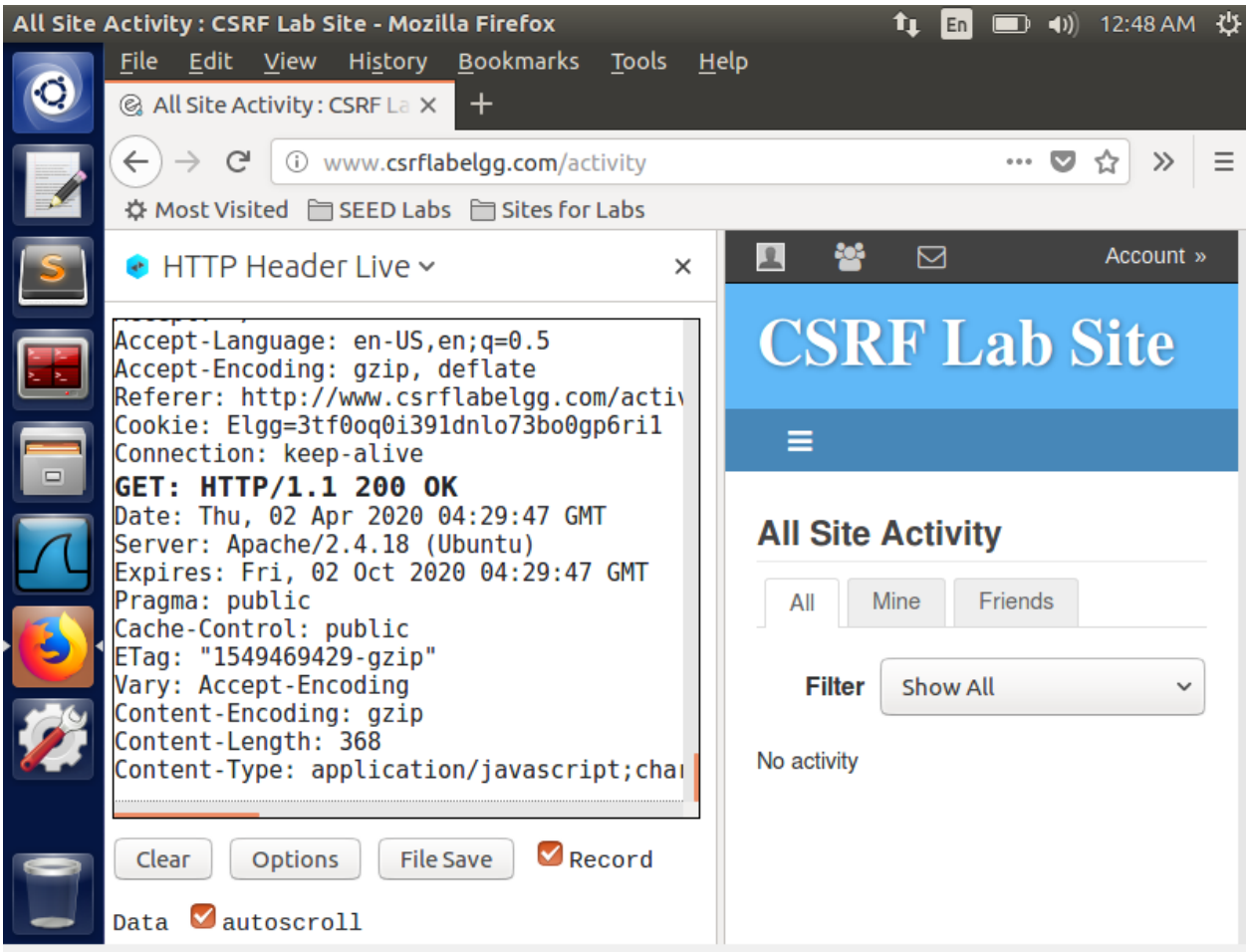
CSE-5382-001 – Secure Programming

Homework Assignment 7 – Cross-Site Request Forgery (CSRF) Attack

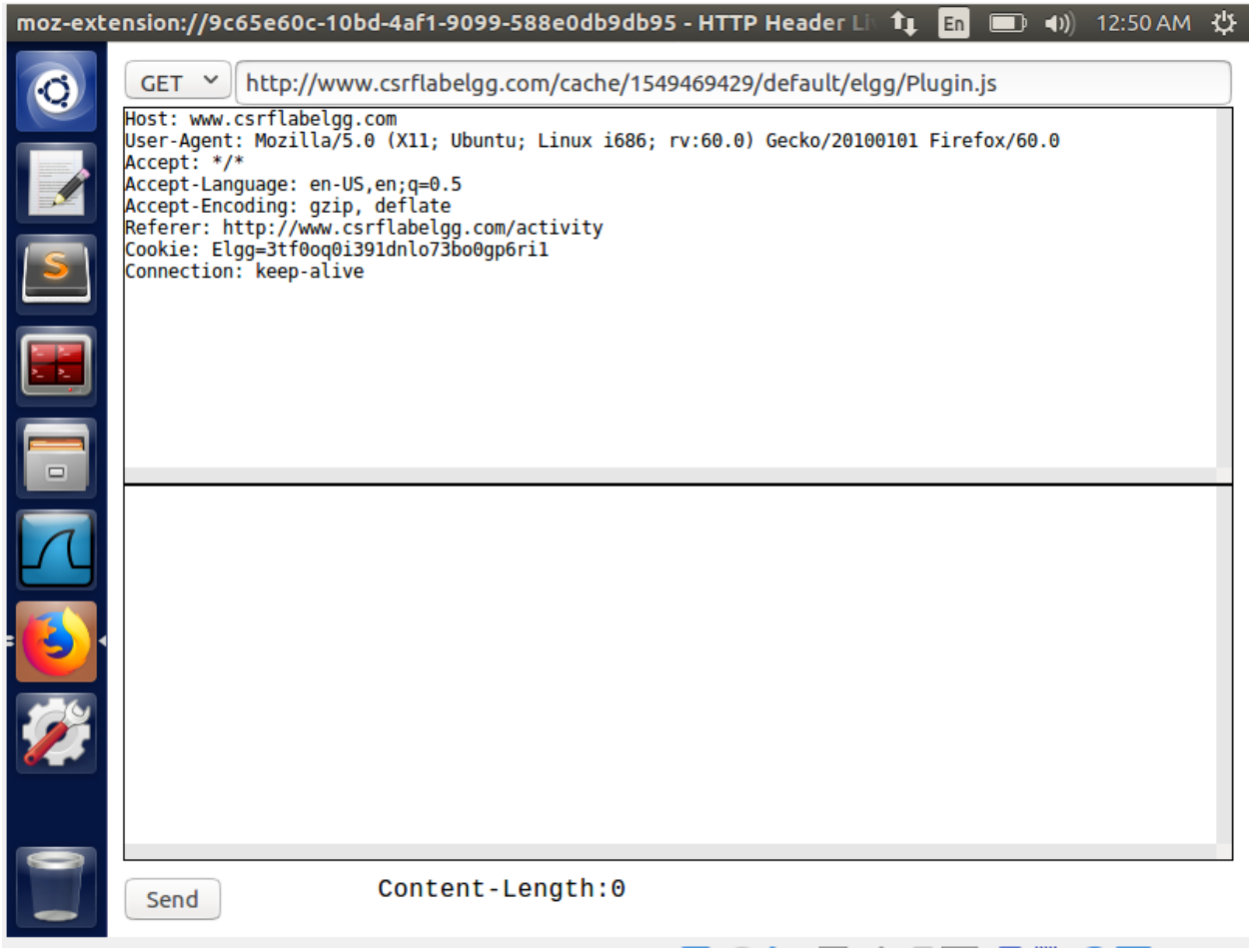
Name	UTA ID
Goutami Padmanabhan	1001669338

3.1 Task 1: Observing HTTP Request.

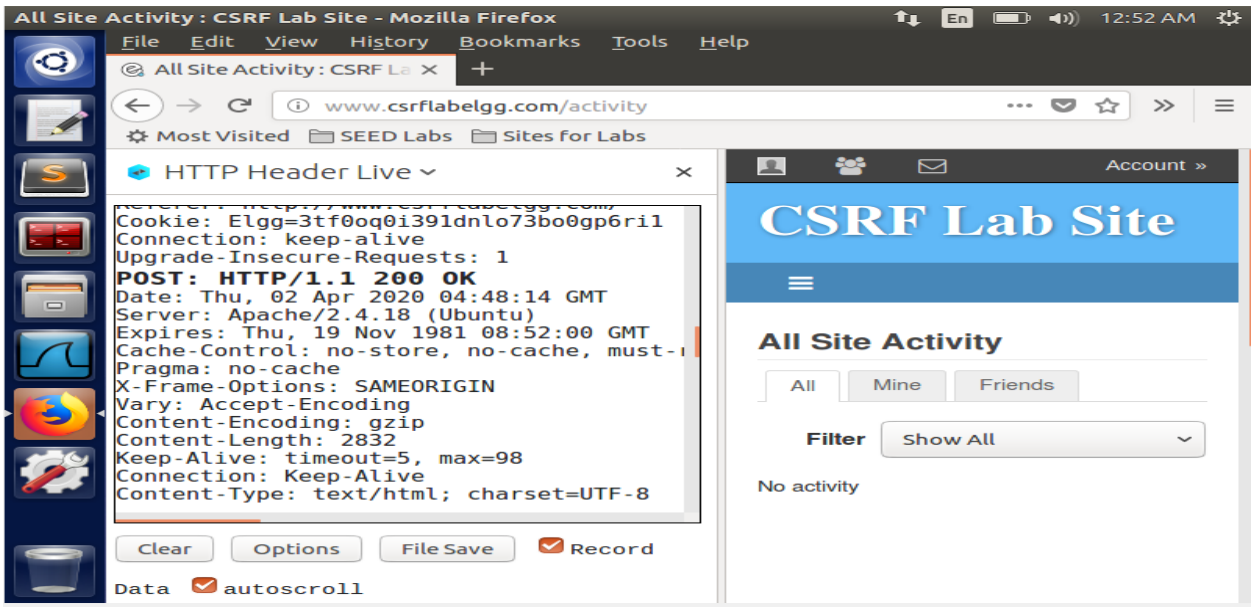
The purpose of this task is to understand and capture the GET and POST HTTP request by using the Firefox add on "HTTP Header Live". In order to understand how to capture these, we go to the social networking website www.csrflabelgg.com.



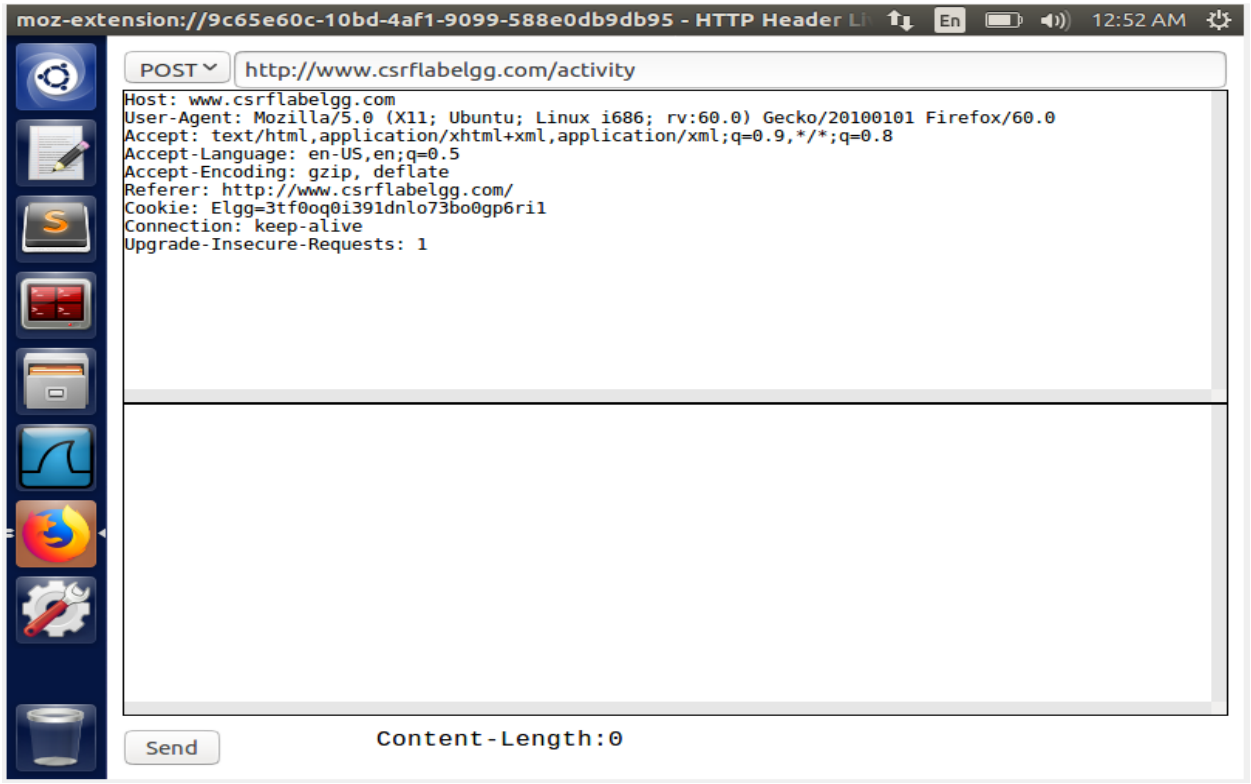
Once we go into the social networking website, we open the add on from the sidebars and check what kind of requests we get. When we click something on the website, the add on shows GET requests.



When we click on the GET request, we get the URL for the request and shows details like Host, Referer, Cookie, Connection, etc., shown like below.



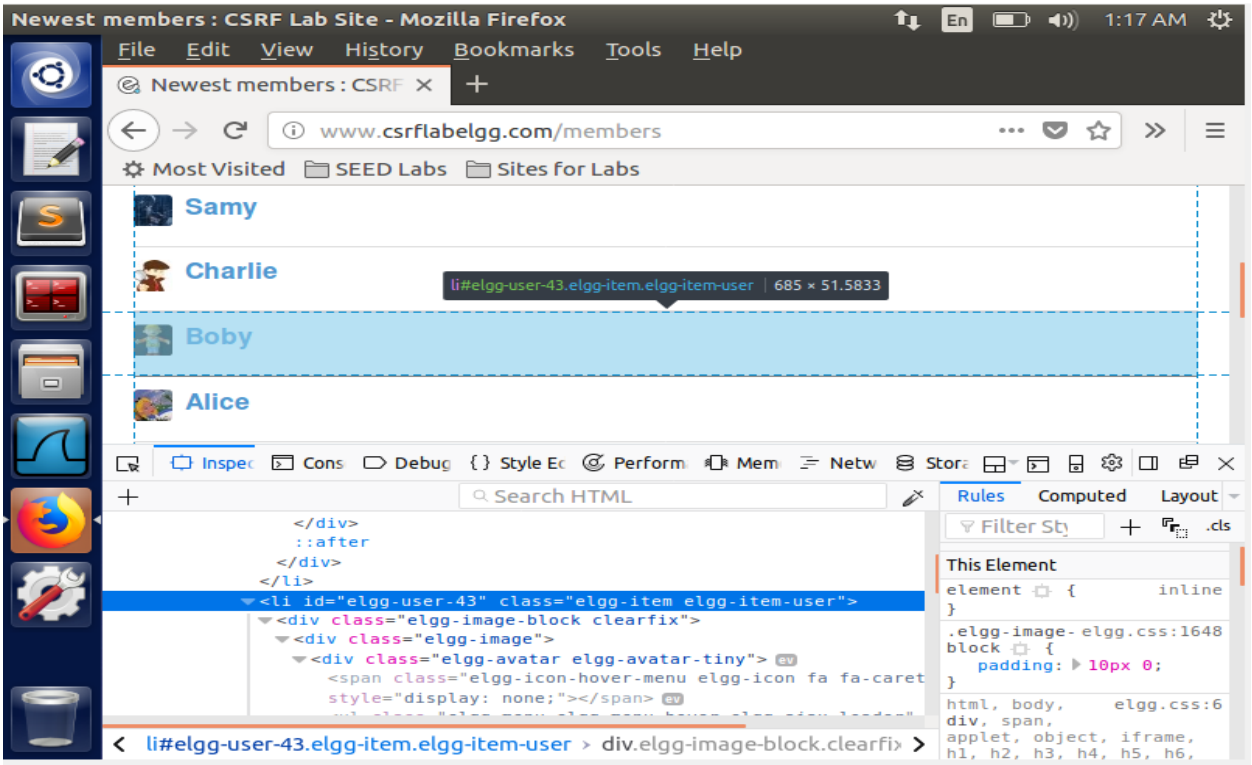
We login to the admin account on the website and check the add on. Since we have posted a request, we will find the POST request. When we click on the POST request in the add on, we get a page that shows the URL for the POST request along with other details.



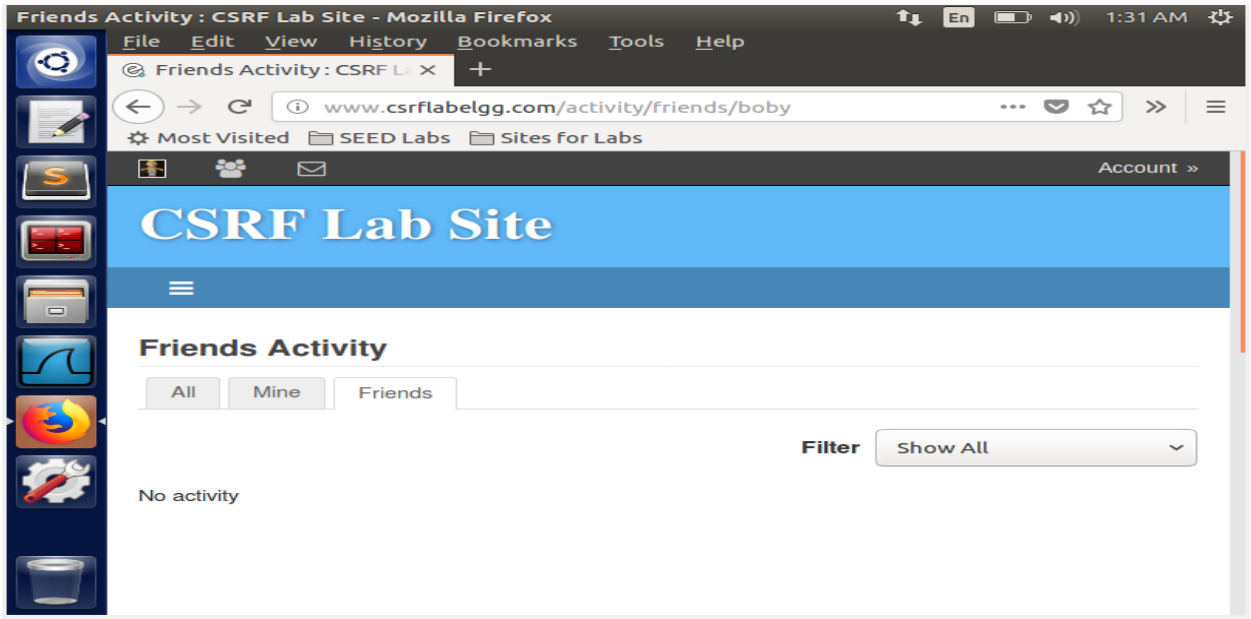
Conclusion: When we use the "HTTP Header Live" add on, we can view the background GET and POST HTTP requests in a website whenever we make changes or navigate through the website. Each time there is a POST or GET request, we see that we get the URL of the request and other details like Host, Referer, Cookie, Connection, etc.

3.2 Task 2: CSRF Attack using GET Request

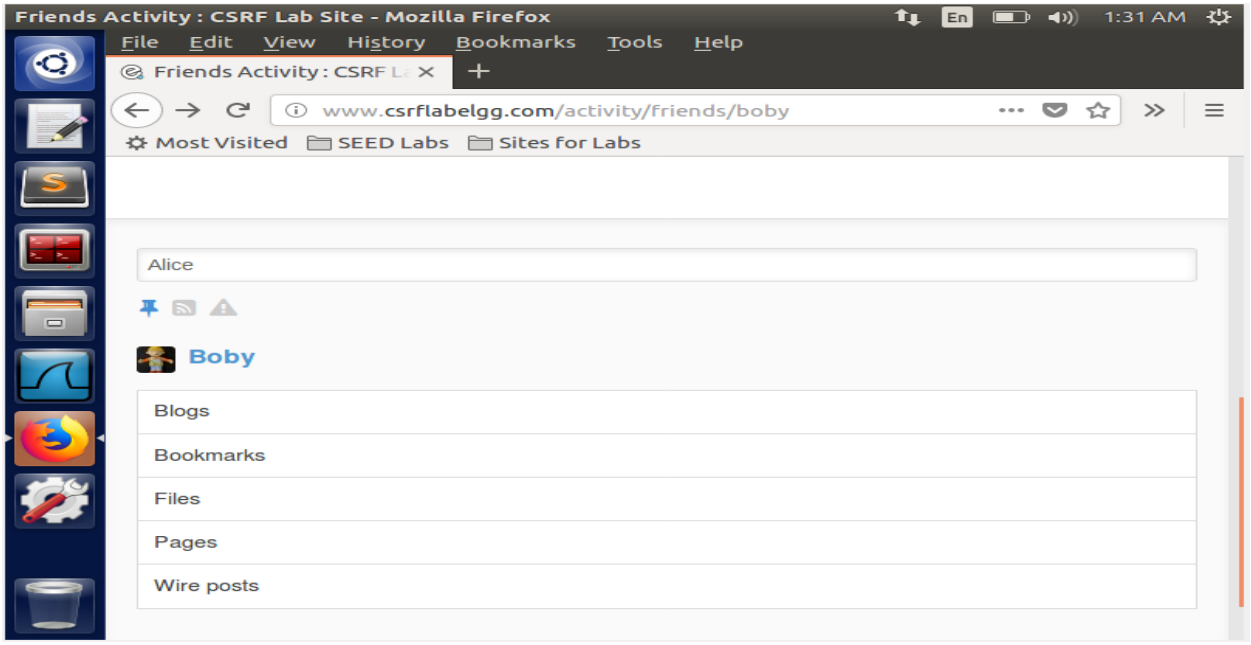
The purpose of this task is to perform the CSRF attack using the GET request without using JavaScript code. We try to add friend to the victim user. Here the attacker is Boby and victim is Alice. Boby tries to add himself to Alice's friend list.



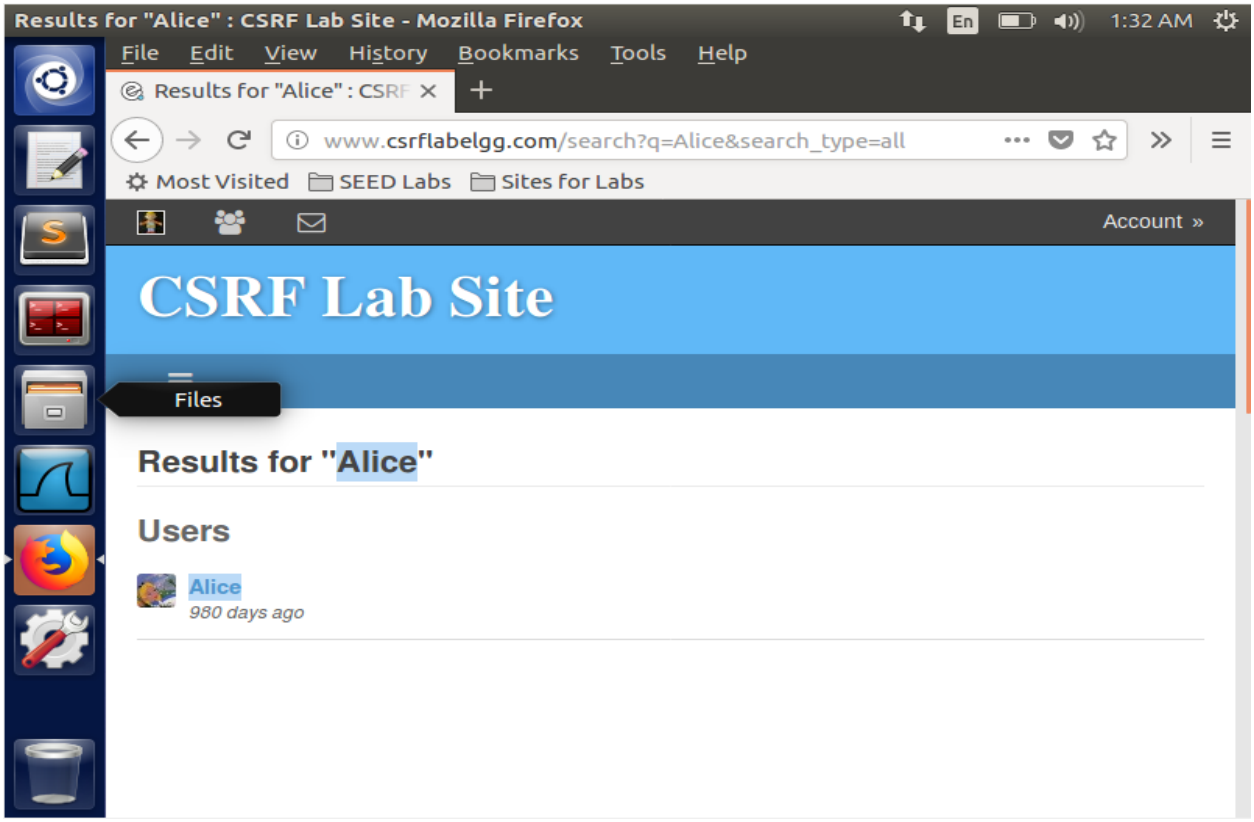
In order to add Boby to Alice's friend list, we need to know Boby's user id. We go to the Elgg website's members list and use the inspect element on Boby to find the user. Here the user is 43 as shown.



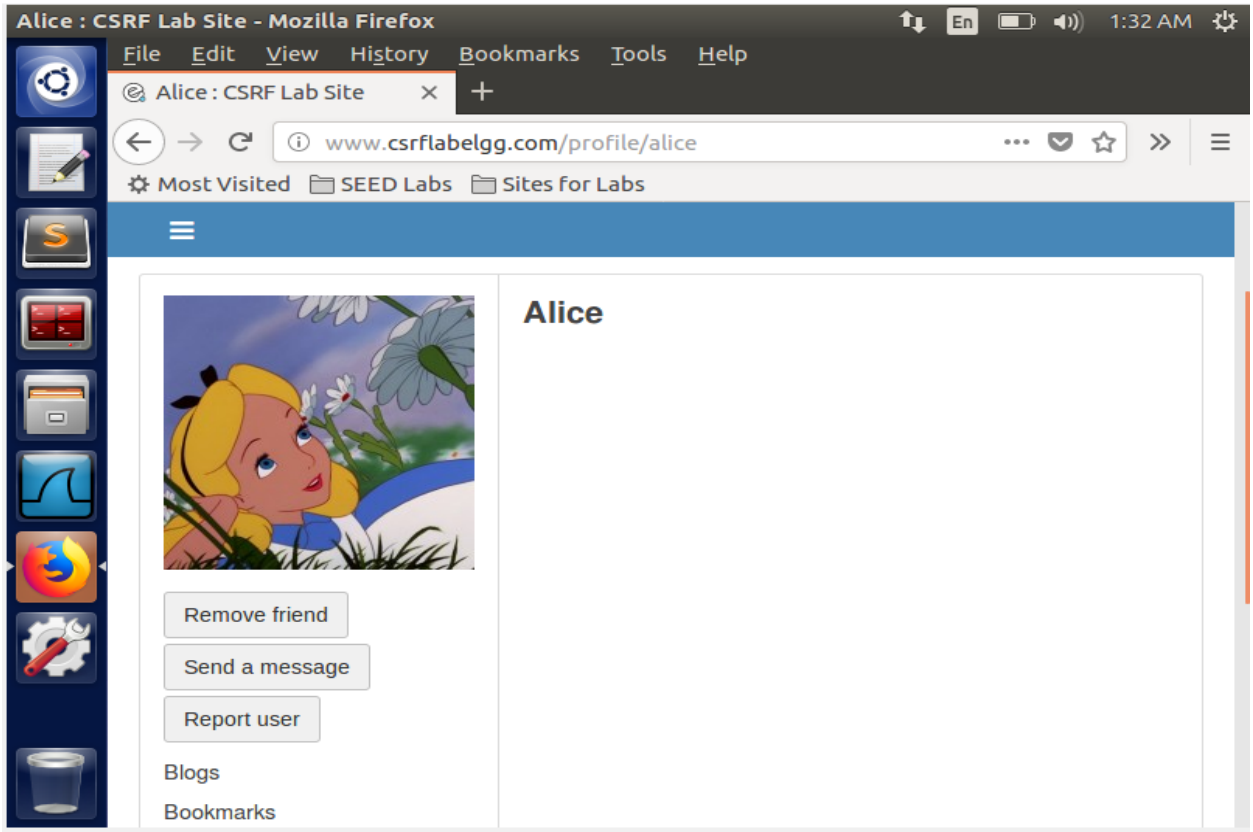
We then go and login to Boby's account using the username passwords given already. Initially, we see that Boby does not have any friends in his Friend list. Then we go and search for Alice in the search bar found at the end of the website's page.



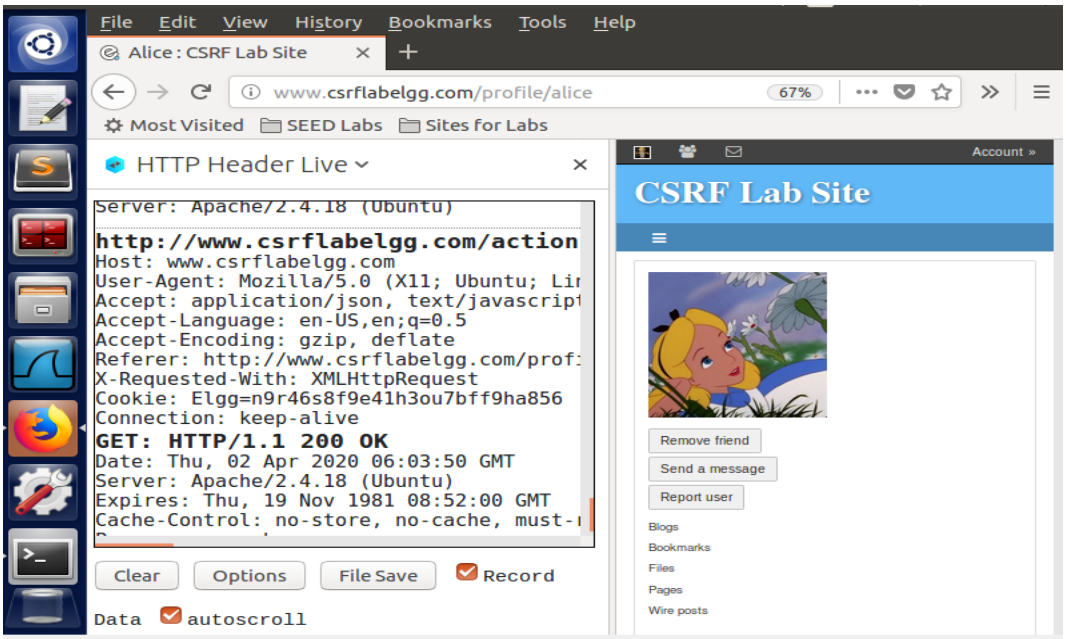
We retrieve Alice from the search as shown in the screenshot.



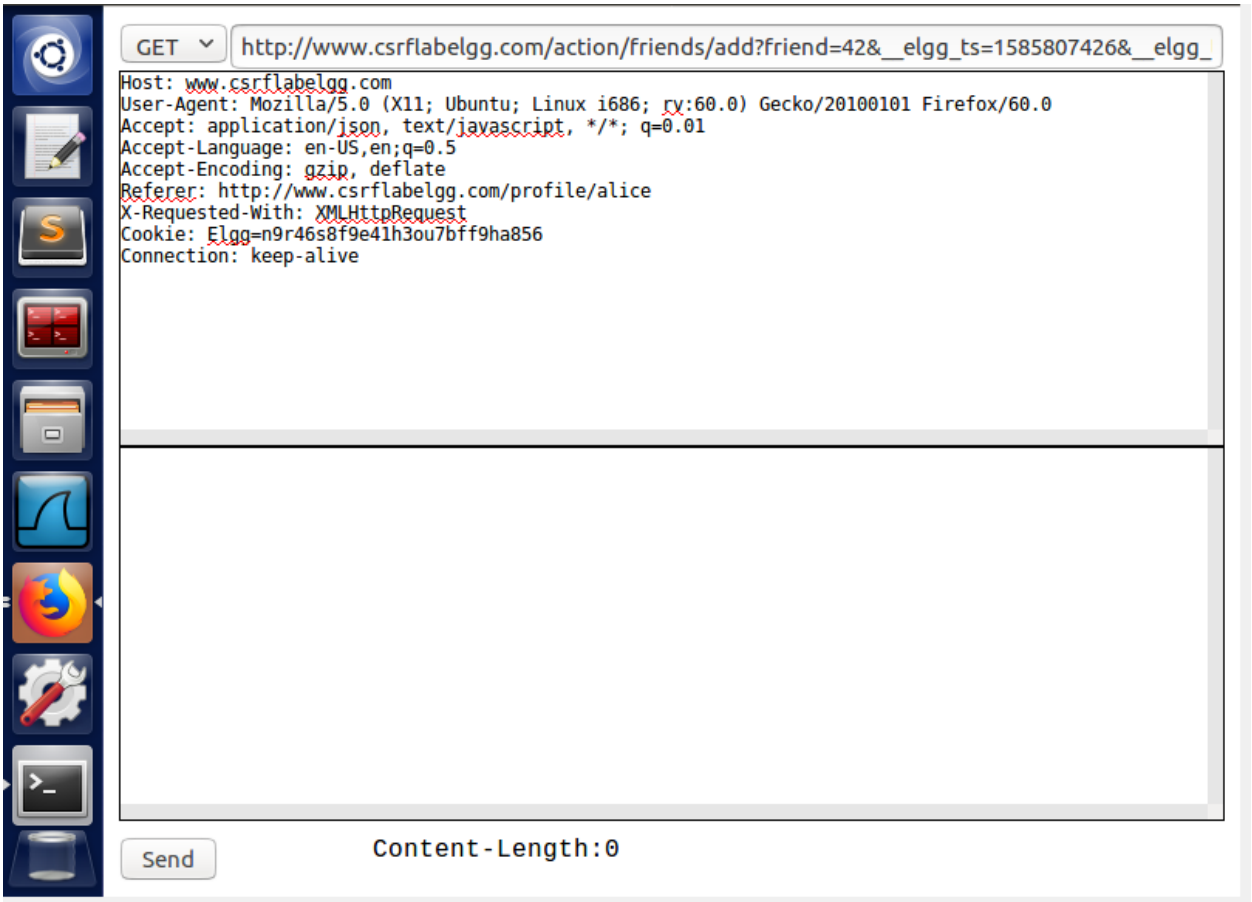
We go to Alice's profile from Boby's account and click 'Add Friend'. We send a friend request to Alice.



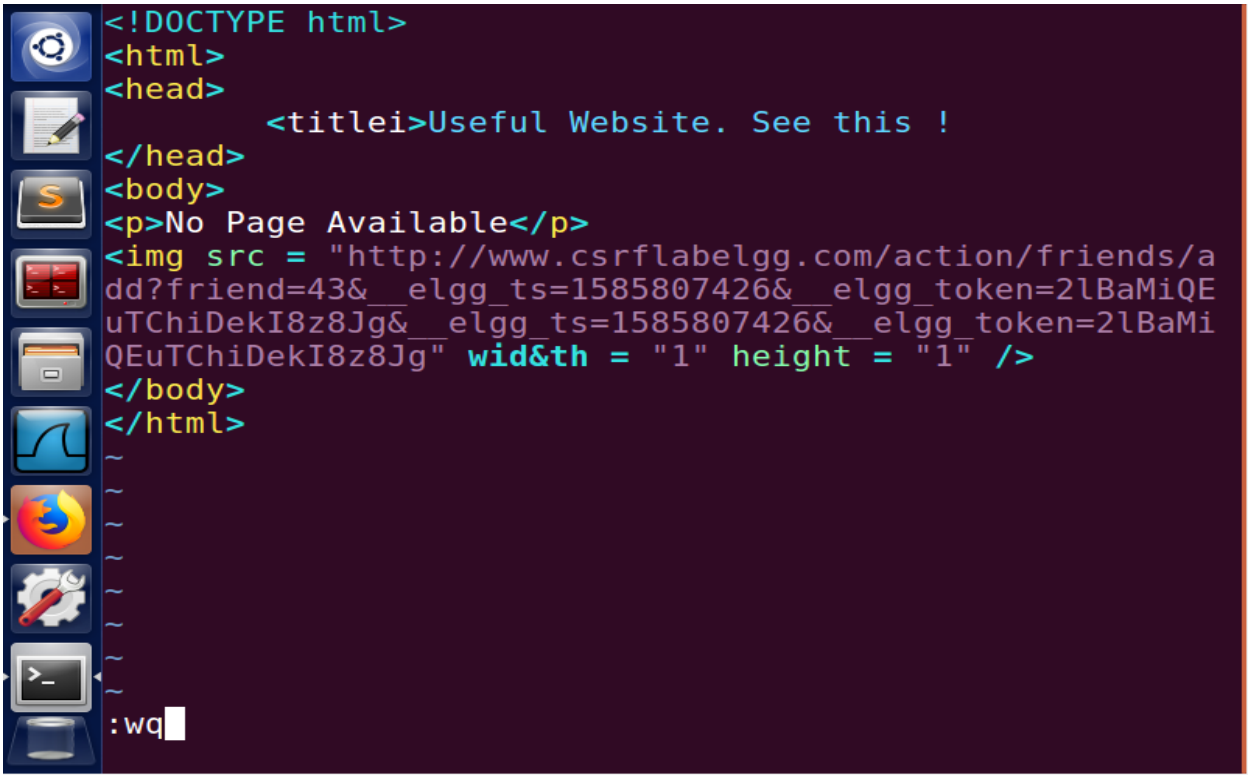
When this is done, we simultaneously check the "HTTP Header Live" add on to find the GET request for XMLHttpRequest.



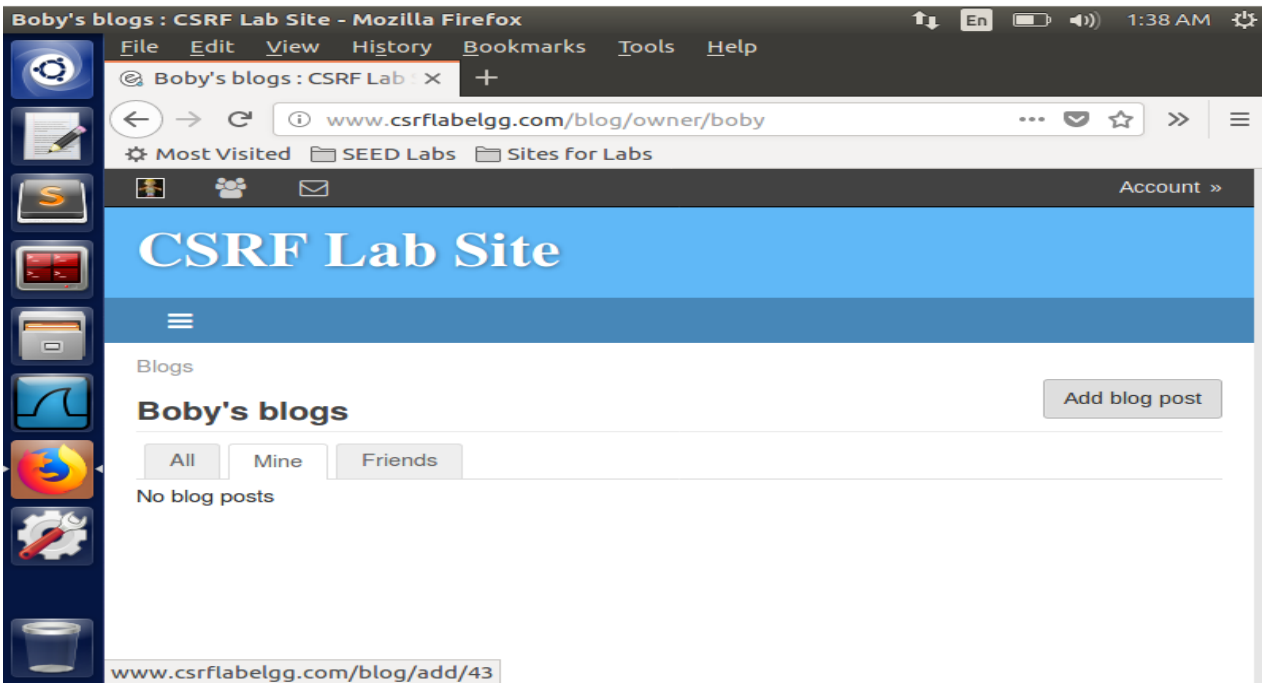
As shown in the screenshot, we copy the URL from the GET request and the URL has elgg_ts and elgg_token in it which we use to append in our URL string while creating a malicious website.. We also see the Cookie in the details



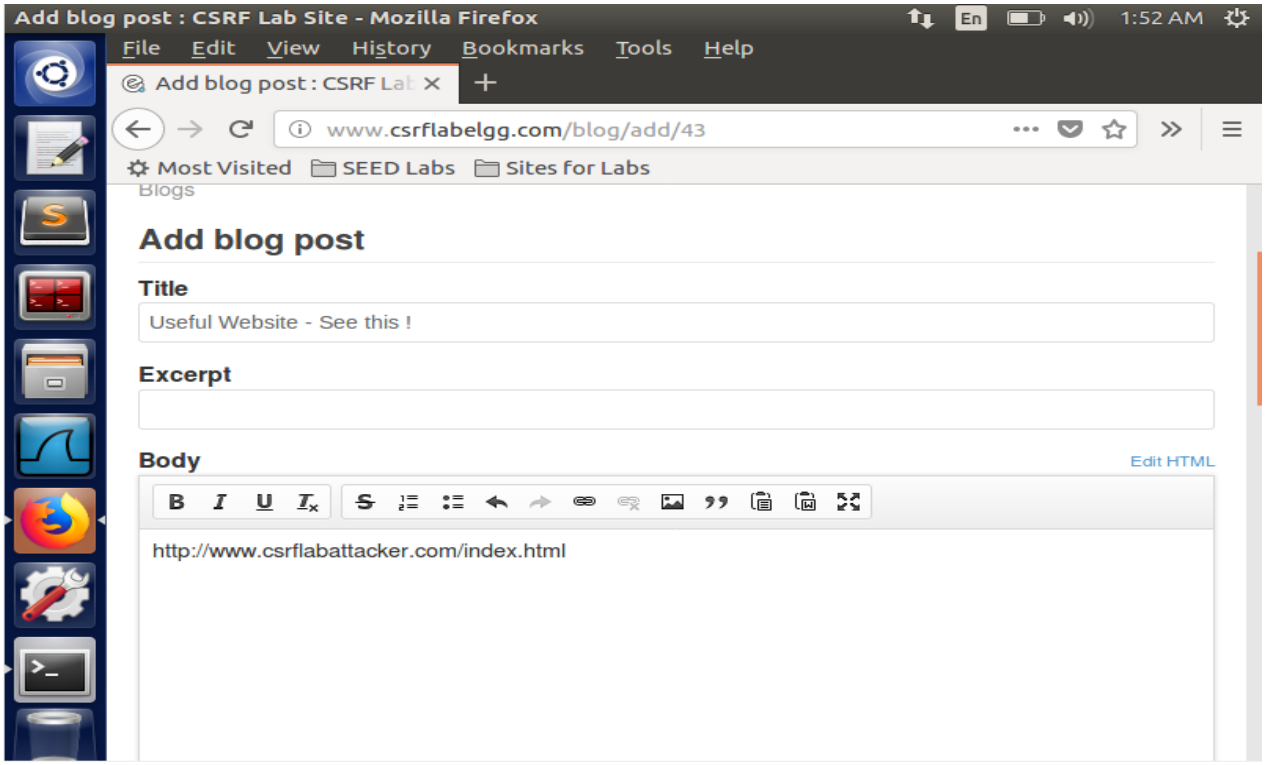
We use this URL from the GET request to create the malicious site in the attacker folder. For this we use the img tag in HTML and give the web URL we have got. Here instead of user 42 (Alice's user id), we give Boby's user id 43.

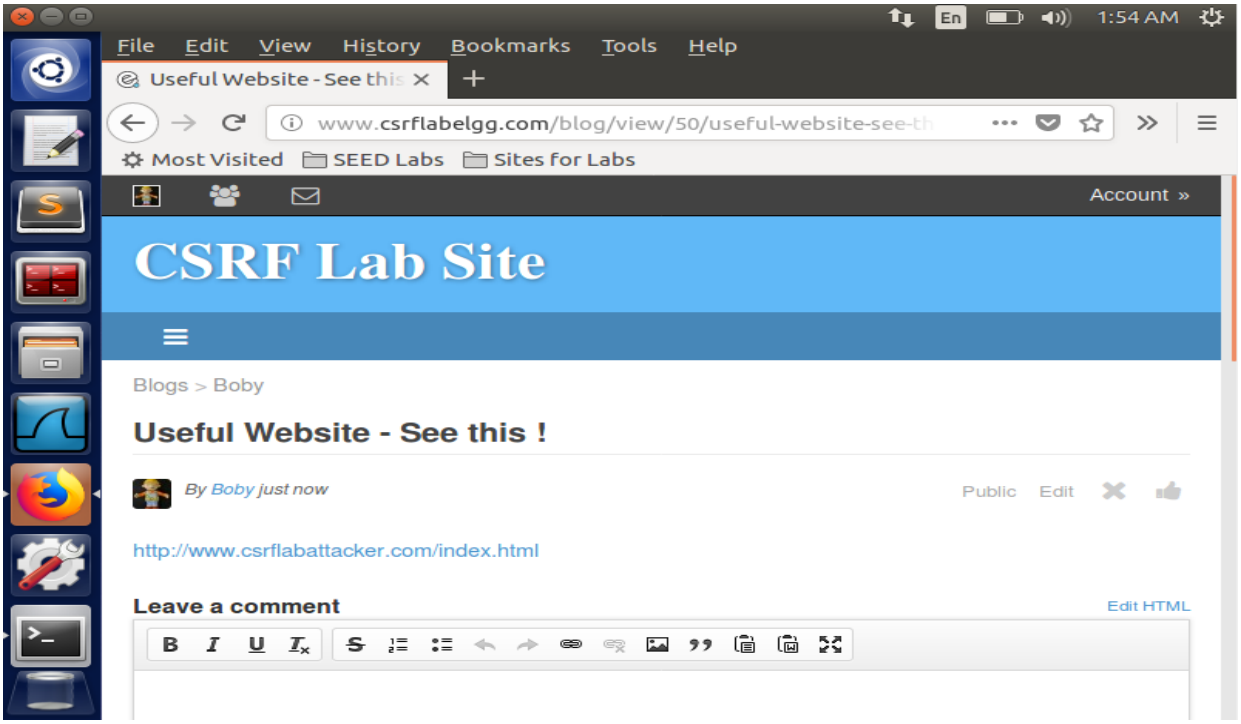


This index.html is created in the folder /var/www/CSRF/Attacker. Once the malicious website is created, we restart the Apache service to make sure the malicious website works.

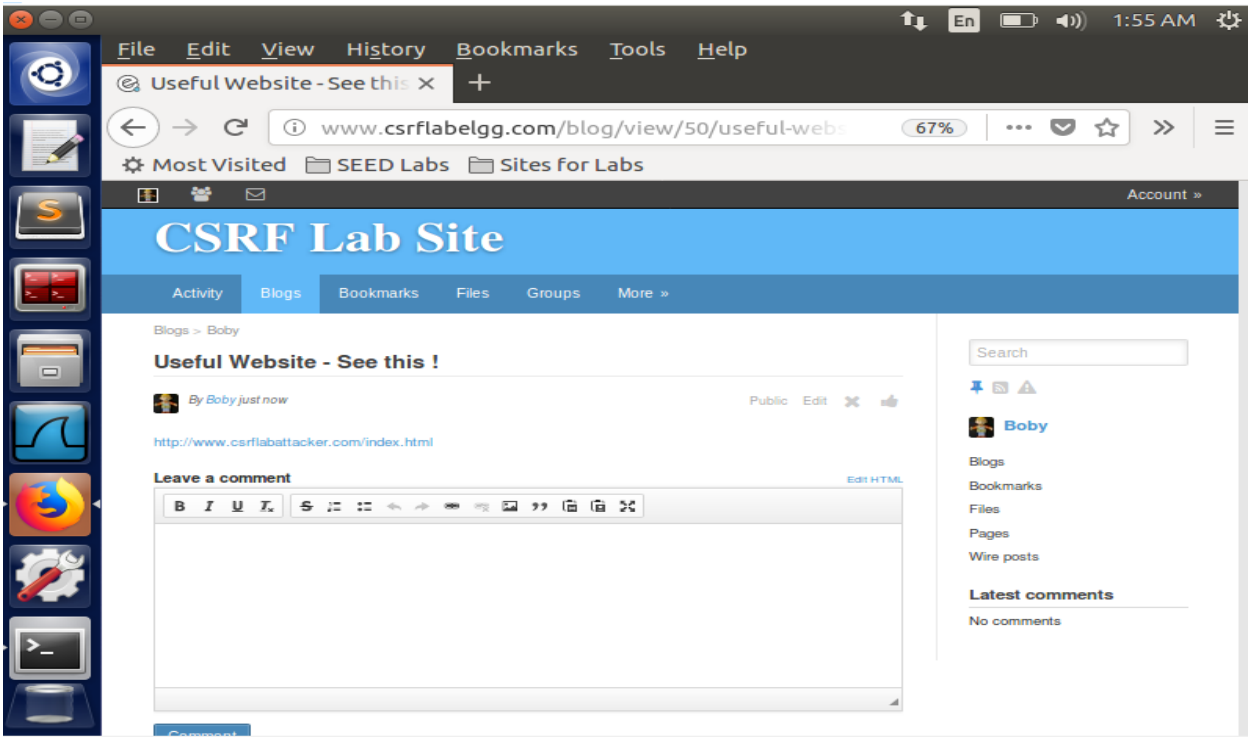


After creating the malicious website in the Attacker folder, we login as Boby and create a blog post. The blog post is created to mislead Alice. The blog post contains the malicious URL that we have created.

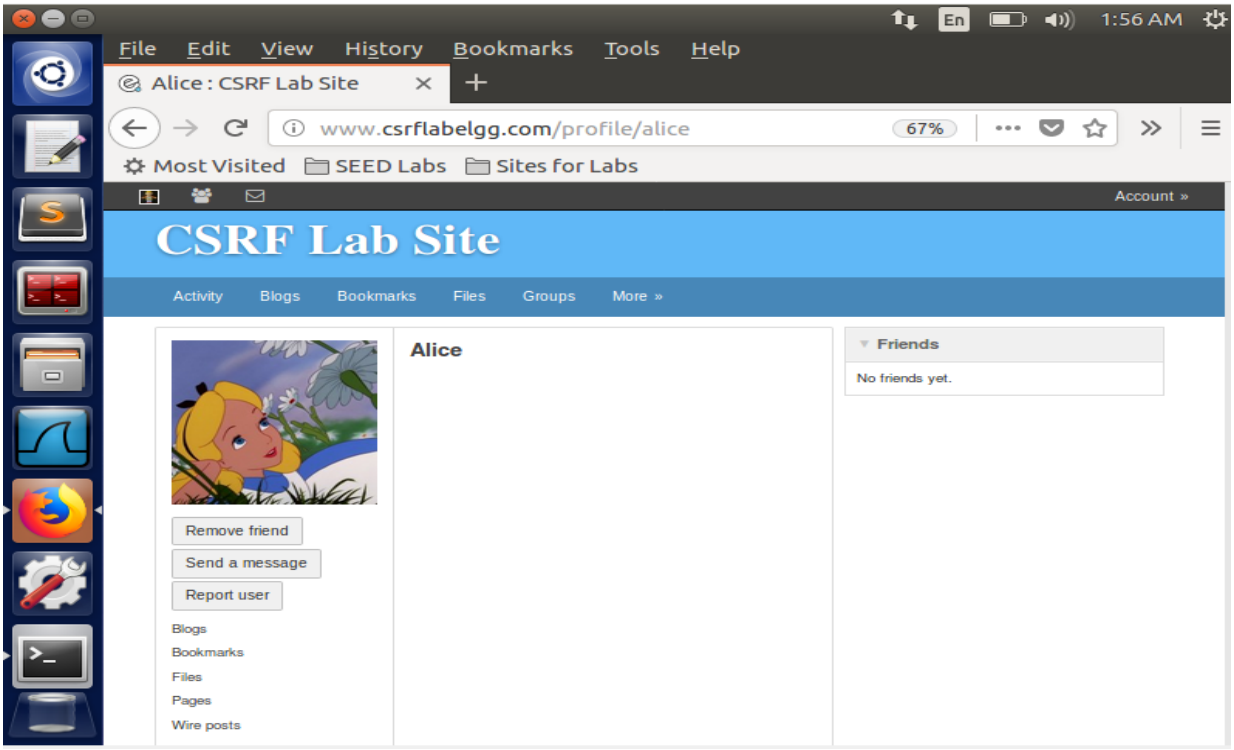




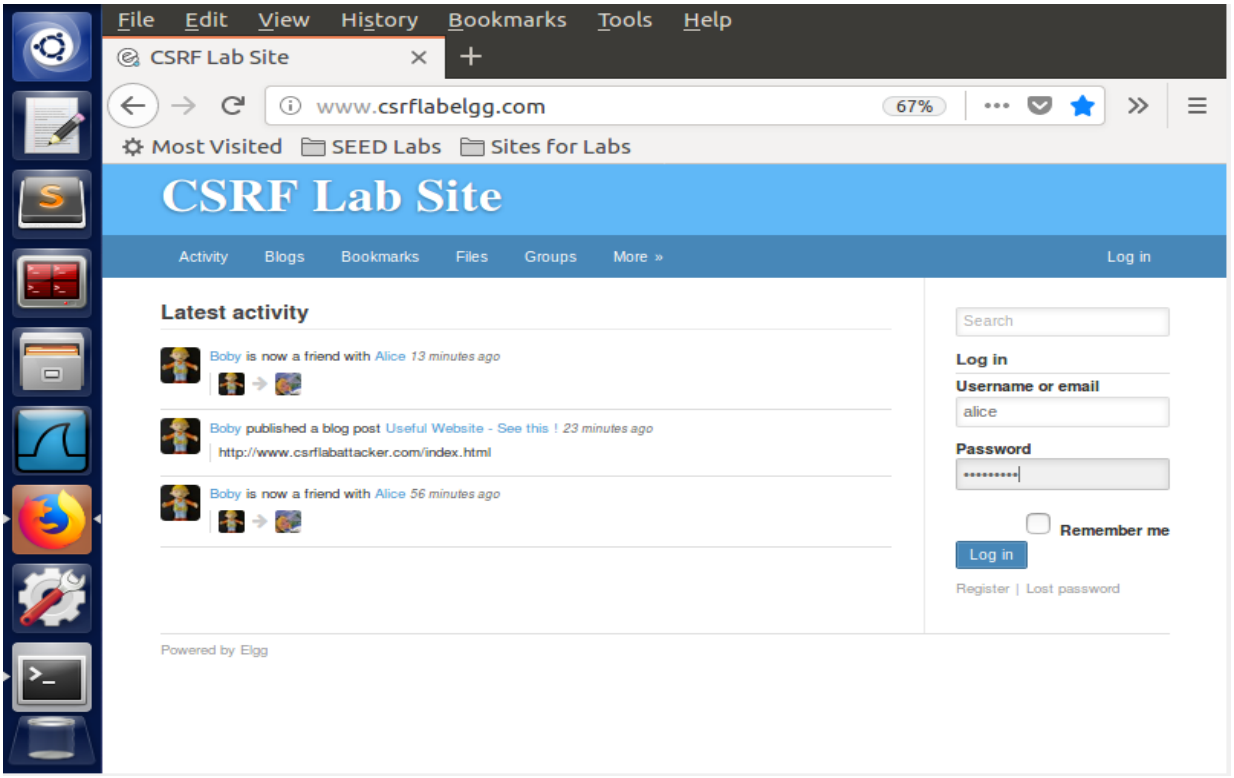
Once the blog post ids created we go and see the blogs of Boby and we make sure we find the post that was created just now.

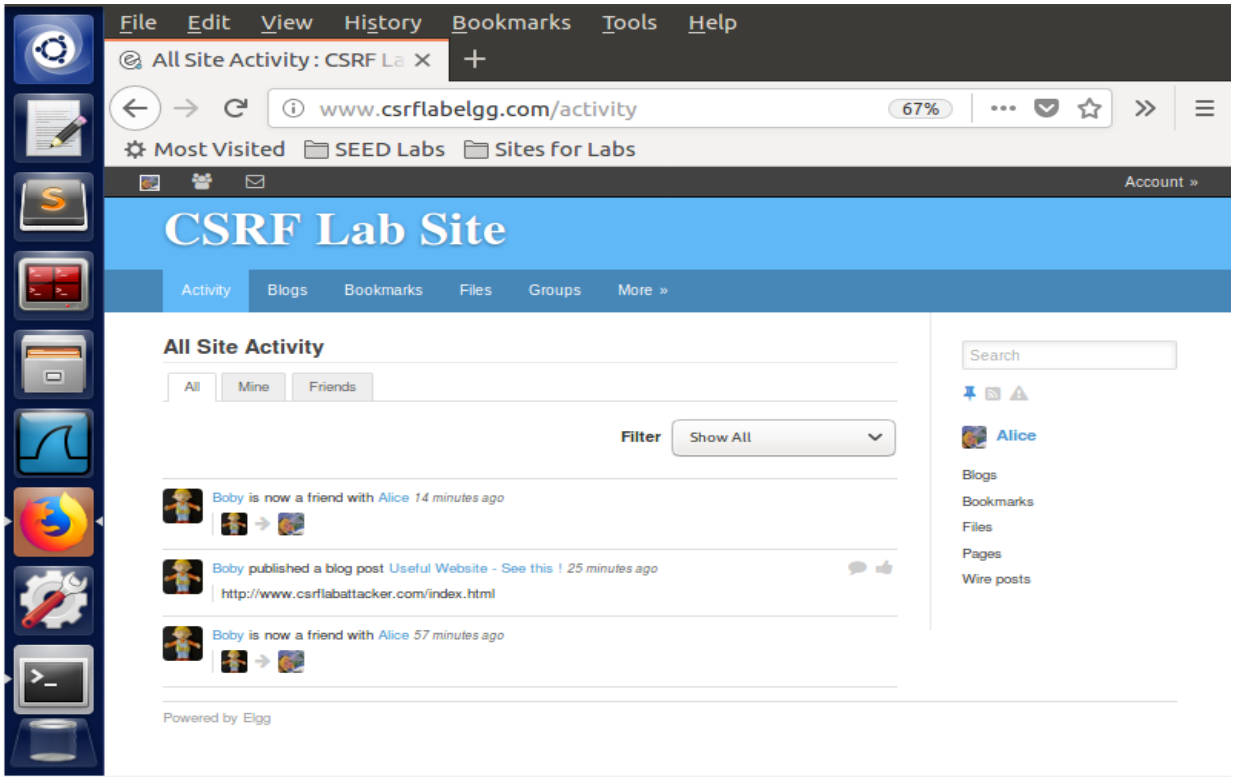


As Boby, we check Alice's profile to see no friend in her friend list initially.

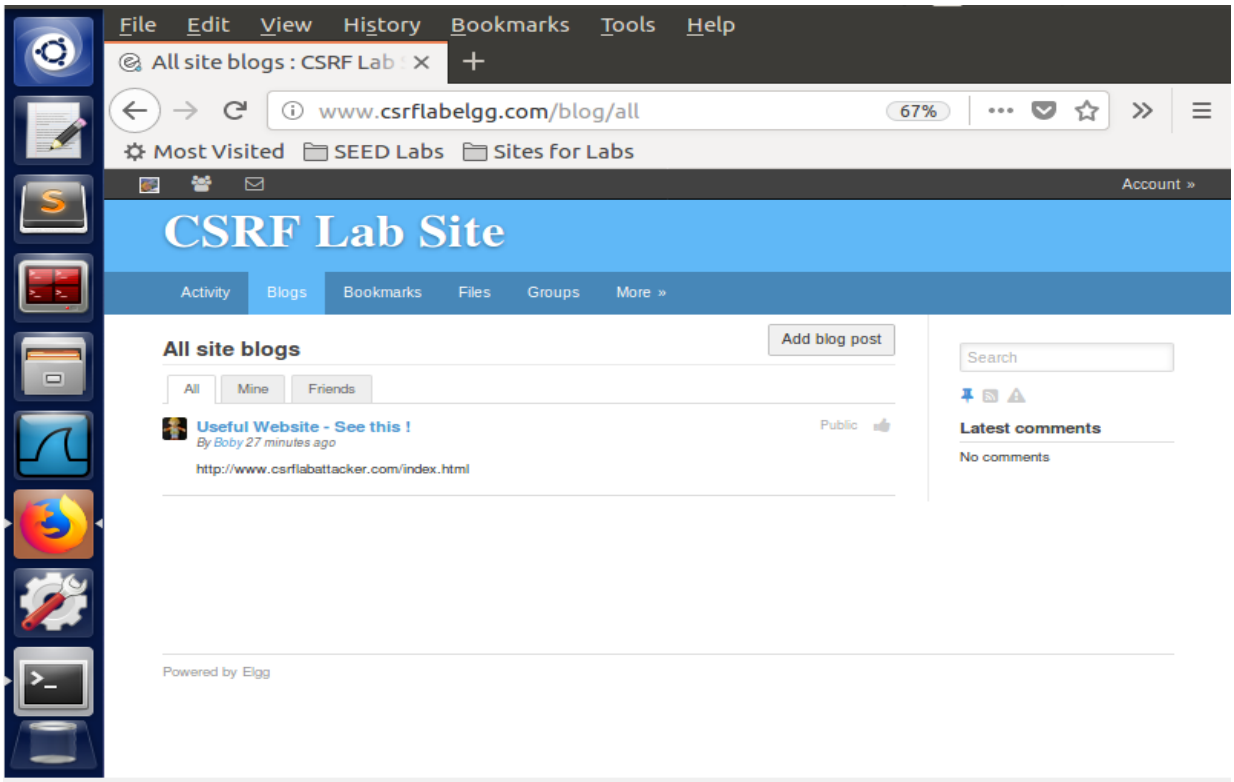


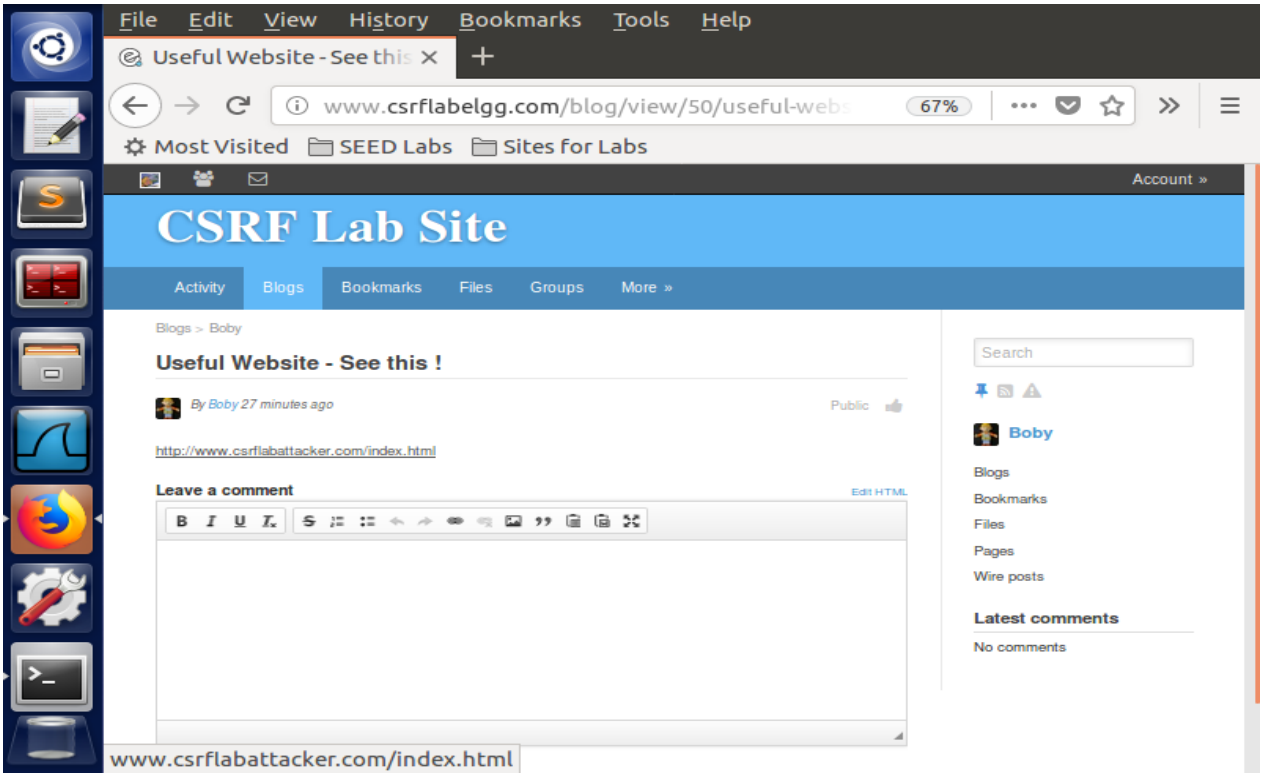
We then login to Elgg website as Alice and check her profile.



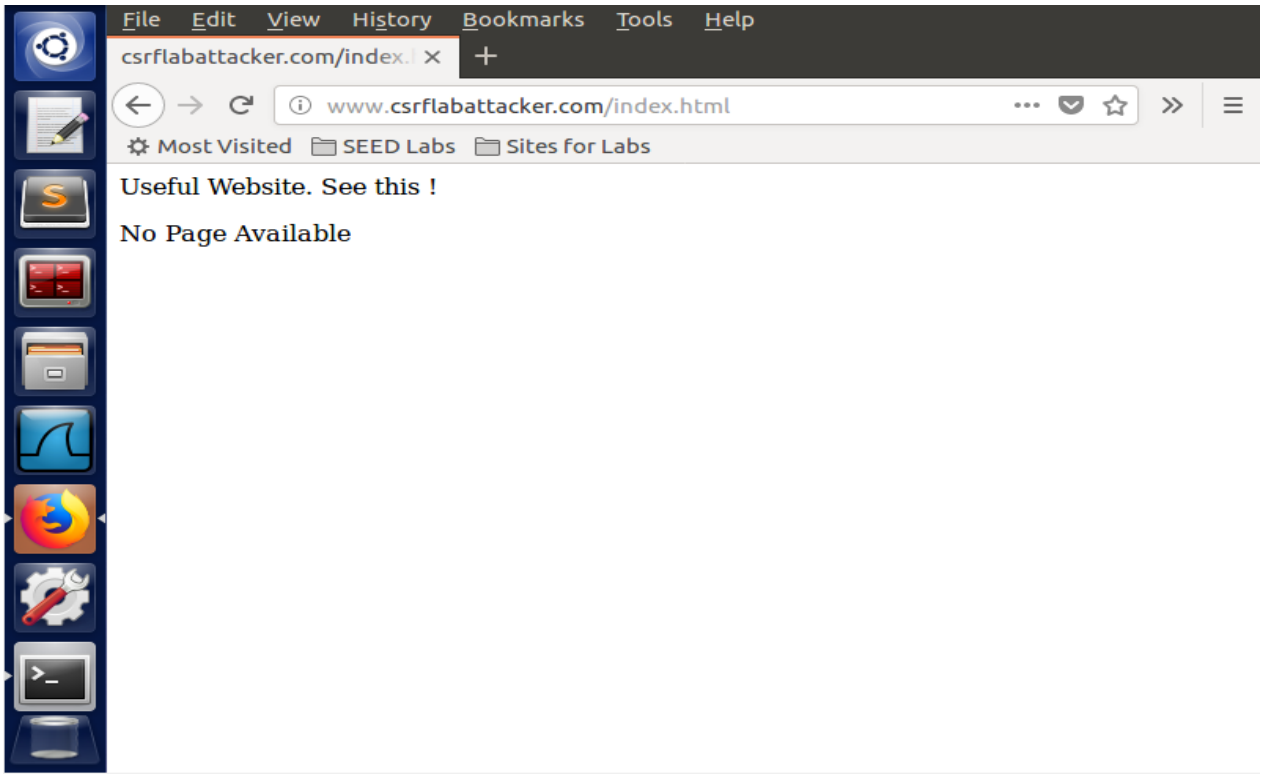


As Alice, when we go and see the blogs and we find a misleading blog created by Boby earlier.

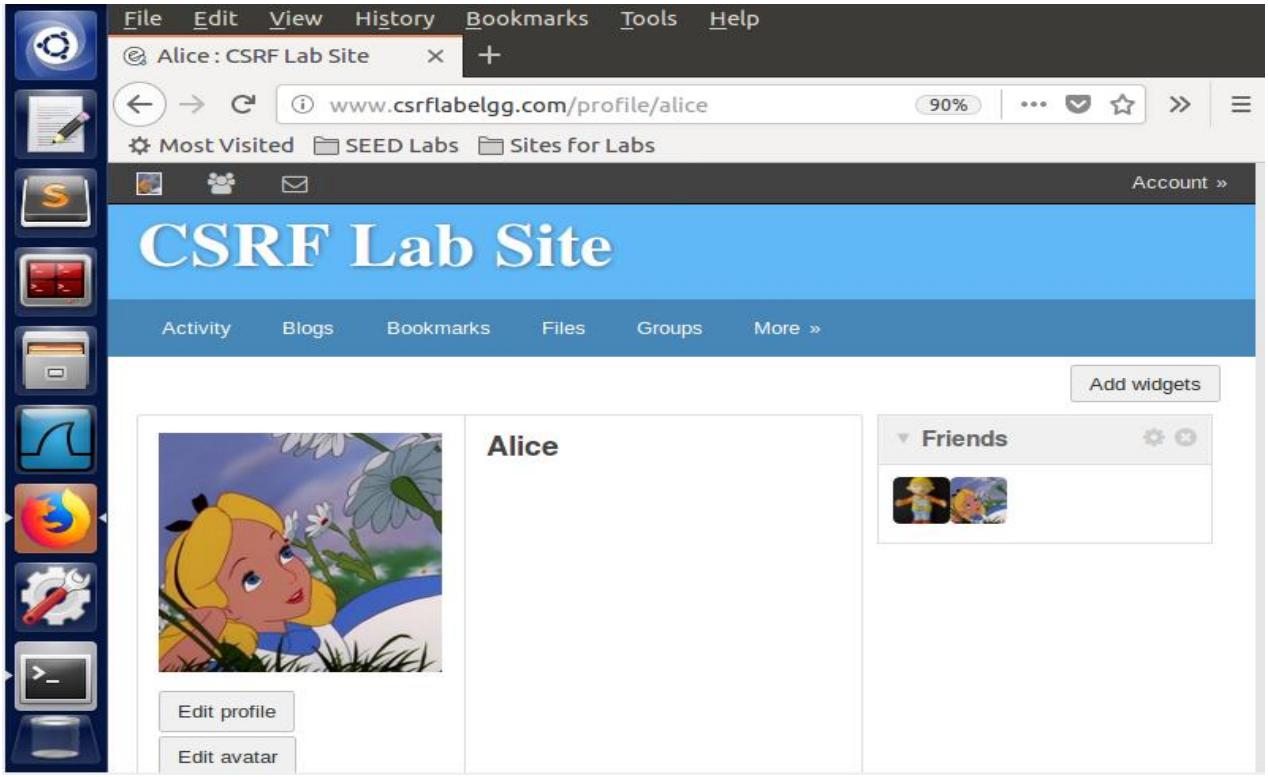




When Alice clicks the malicious website, it displays “No Page Available”. Here Alice clicks the website without knowing it is malicious and unknowingly adds Boby as her friend.



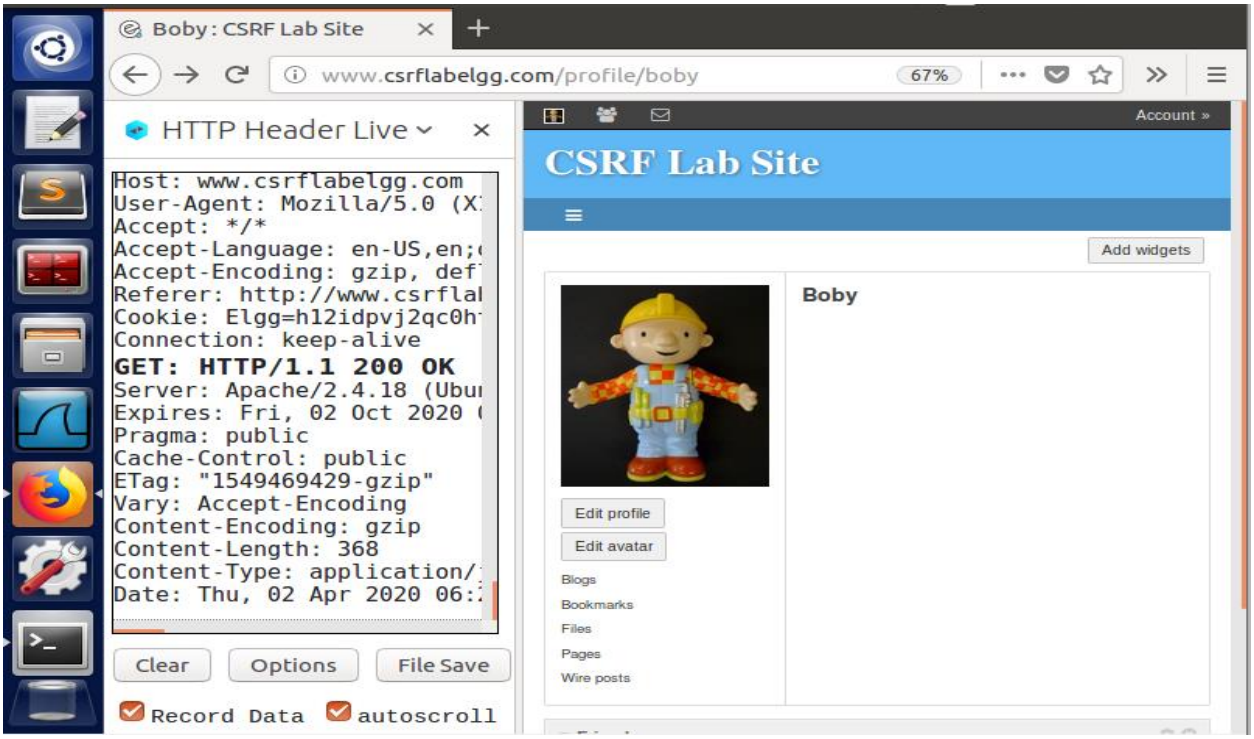
Now when we go and see Alice's friend list, we see that Boby is already there in her Friend List.



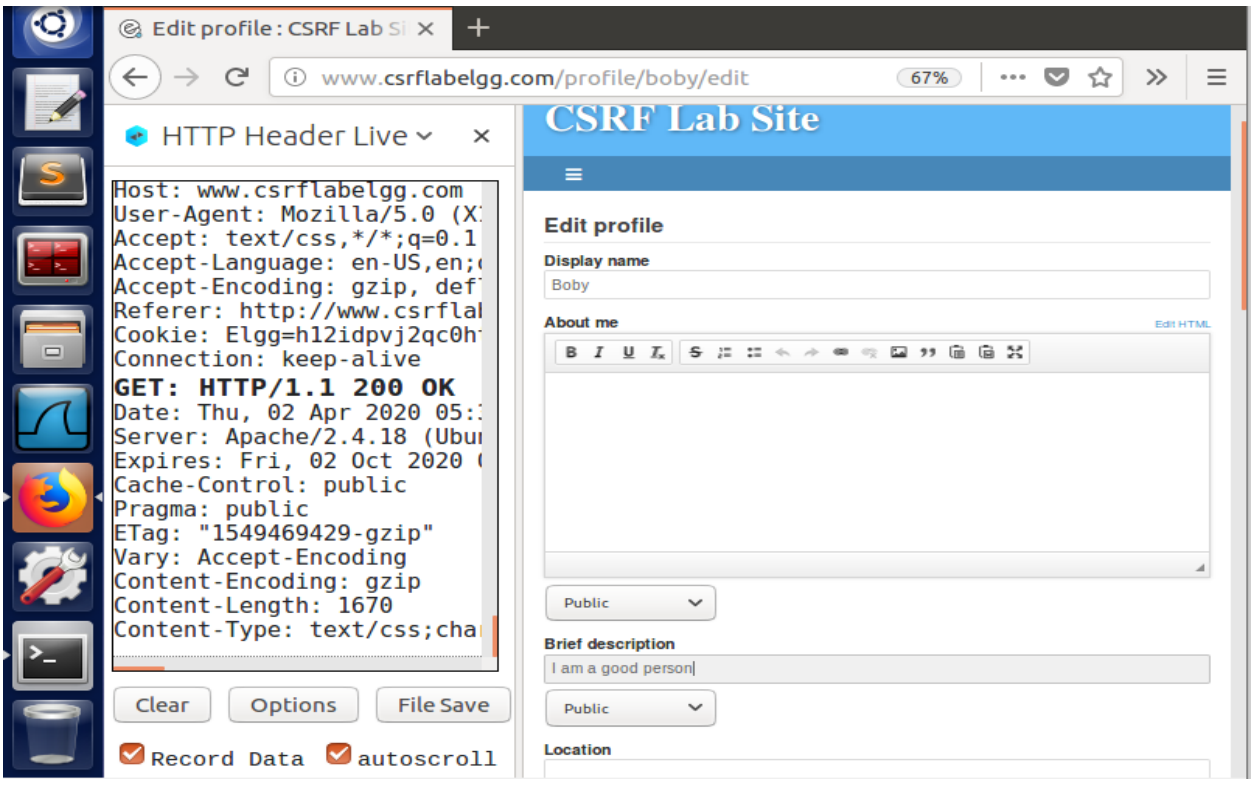
Conclusion: Boby just finds his own user number or id so that he can add it to friend list of Alice. This user can be found from the members page by using the Inspect element. Boby then finds the GET request URL and simply creates a malicious website which can be publicly posted. The malicious URL is created by adding the user, elgg_ts and elgg_token values to the GET request URL string and placing it in the src attribute of img tag. When Alice clicks on the URL she gets attacked unknowingly and the Boby gets added to Alice's friend list. Alice will not have any clue whether he/she is attacked. This is cross-site request forgery by using GET request URL. Here the trusted website is www.csrflabelgg.com and the malicious URL injected through trusted website using GET request is http://www.csrflabattacker.com/index.html.

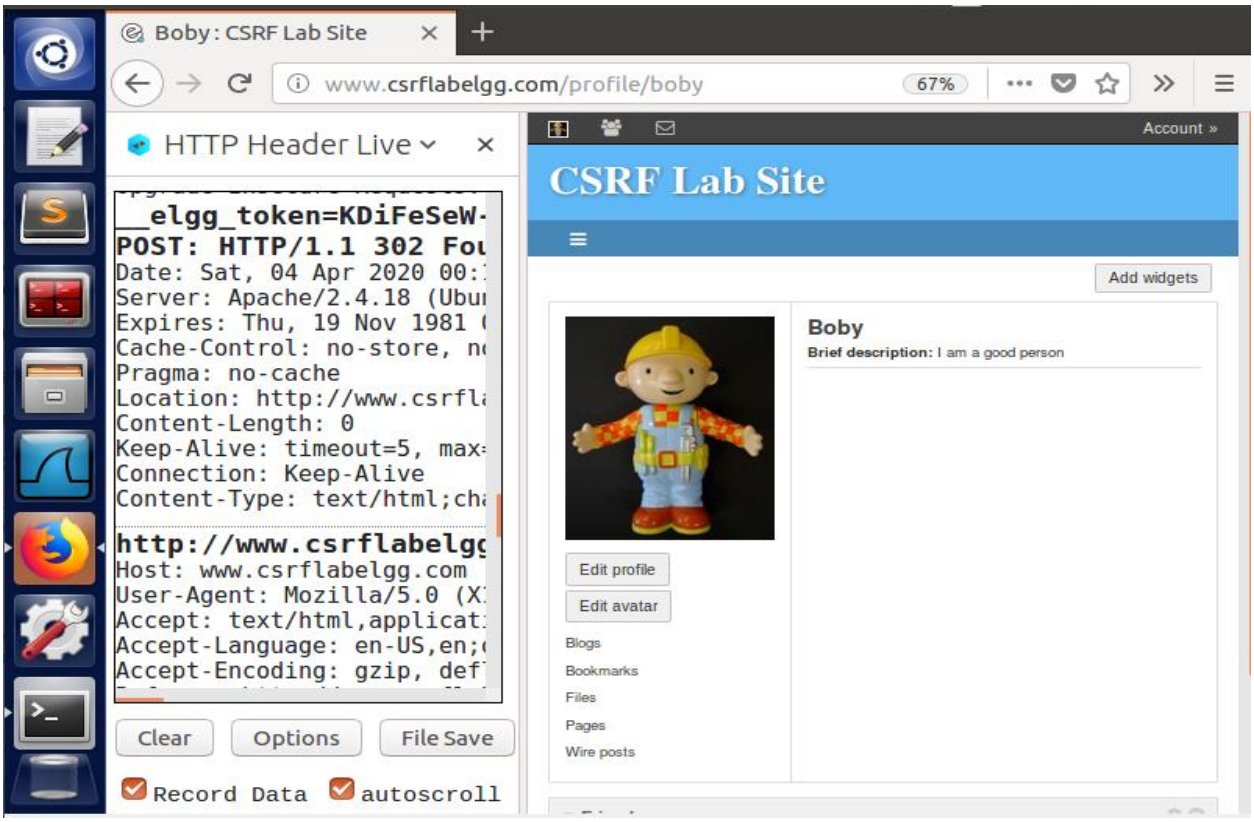
3.3 Task 3: CSRF Attack using POST Request

The purpose of this task is to make Alice to say "Boby is my Hero" in her profile without her knowledge using CSRF attack by Boby with POST request. In order to attack Alice, we must first get the POST request URL from HTML message body. We perform a sample POST request in Boby's account for this purpose.

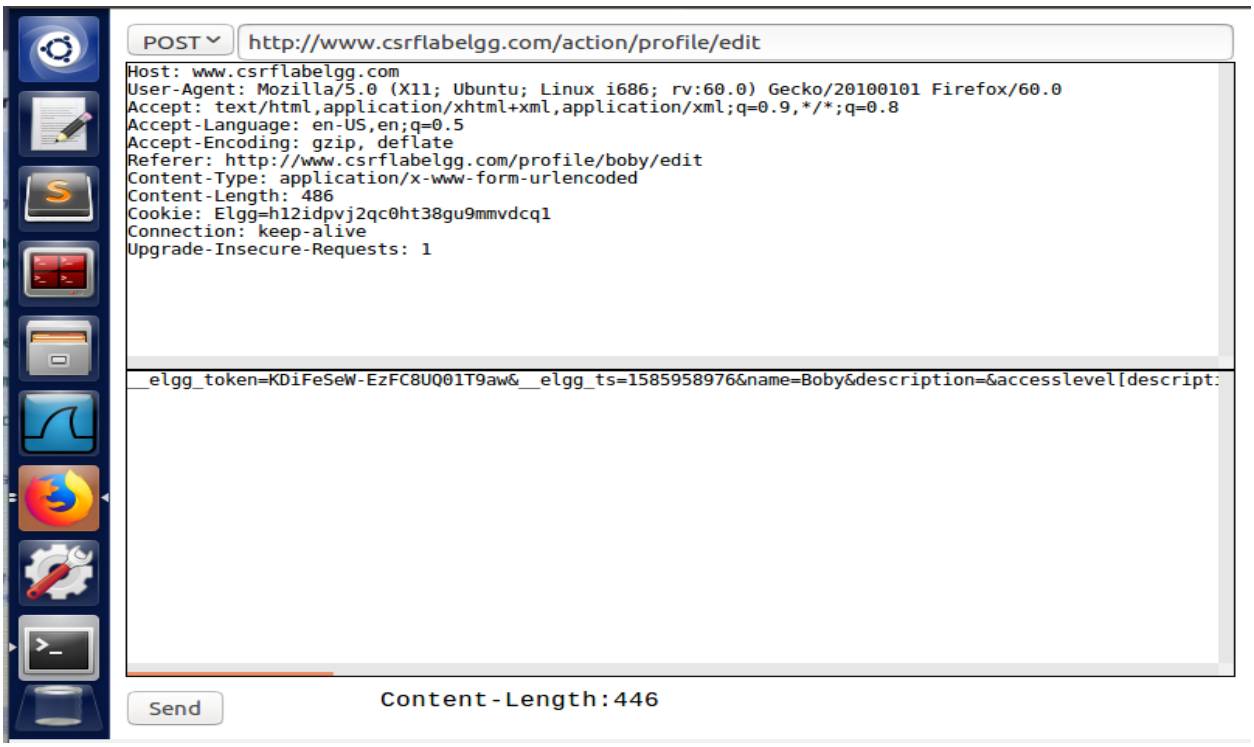


We have logged in as Boby and noted the HTTP Header live add on for all kinds of GET and POST requests. Now we try to edit the description in Boby's profile and save it. While performing the 'SAVE' operation, there will be a POST request whose URL and message body we note down.





When we click the POST request we have, by saving the profile description, we can see HTML message body that has values for elgg_token, elgg_ts and the values accesslevel, briefdescription and other such patterns. Our aim to change Alice's description in her profile. So we note down briefdescription values.

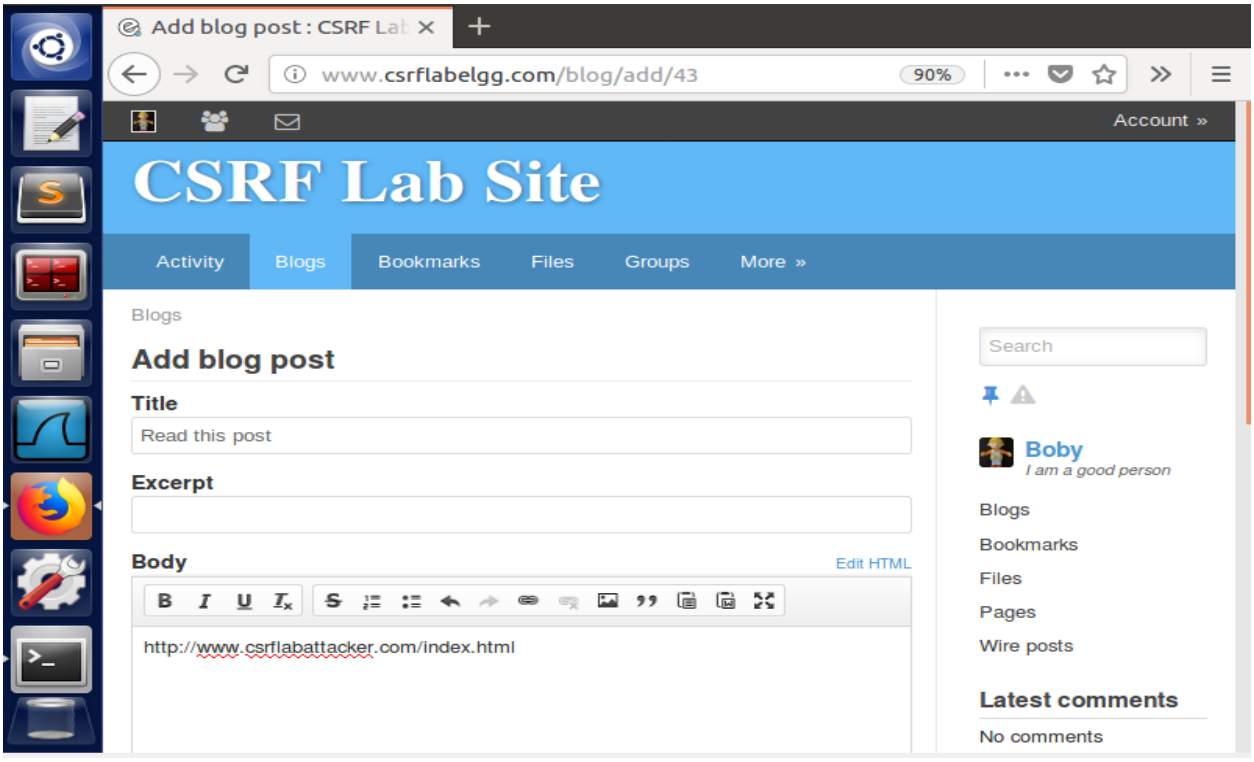


index.html

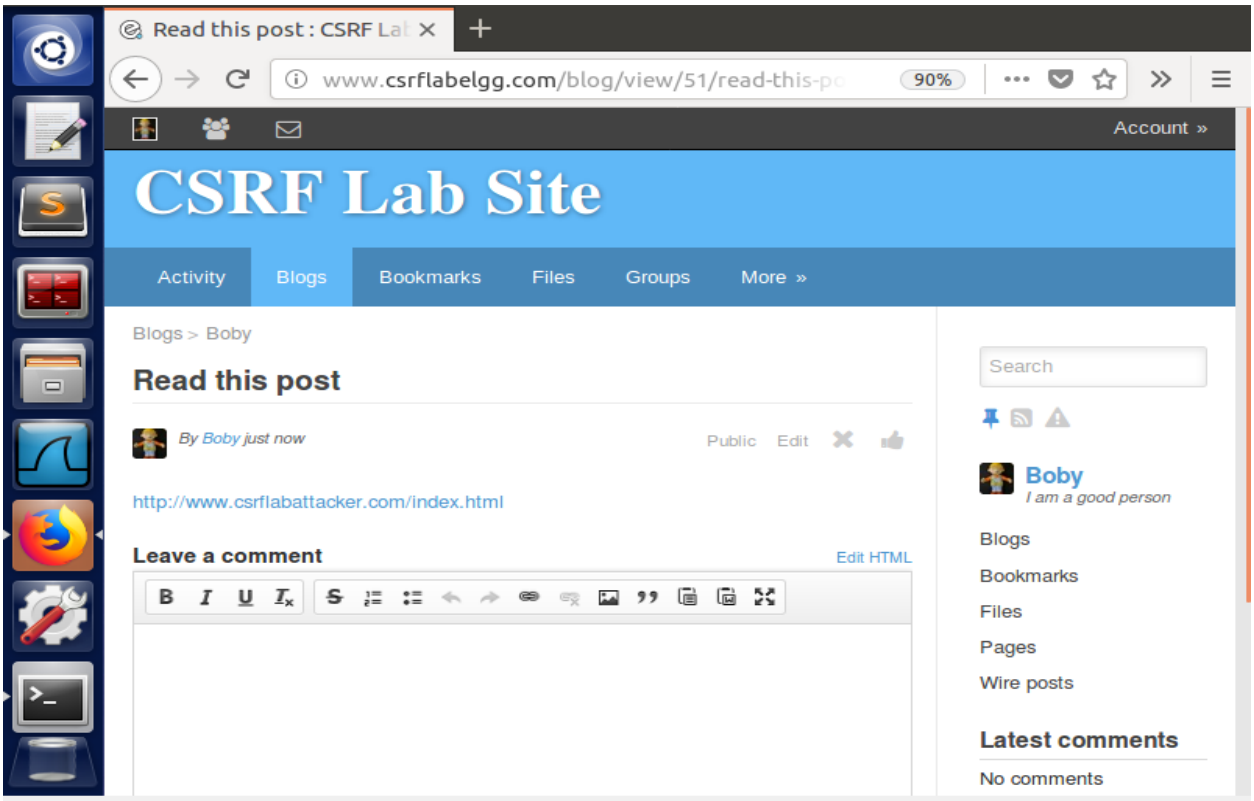
```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">
function forge_post()
{
var fields;

// The following are form entries need to be filled out by attackers.
// The entries are made hidden, so the victim won't be able to see them.

fields += "<input type='hidden' name='name' value='Alice'>";
fields += "<input type='hidden' name='briefdescription' value='Boby is my Hero'>";
fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
fields += "<input type='hidden' name='guid' value='42'>";

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

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

// Append the form to the current page.
document.body.appendChild(p);

// Submit the form
p.submit();
}
// Invoke forge_post() after the page is loaded.
window.onload = function() { forge_post();}
</script>
</body>
</html>
```

As shown below, we navigate to the folder /var/www/CSRF/Attacker and create a index.html page. In this HTML page we use JavaScript fields to write all the patterns and values we just noted down. All values are made hidden so that the victim does not come to know that he/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. A complete restart is done to the Apache 2 service.

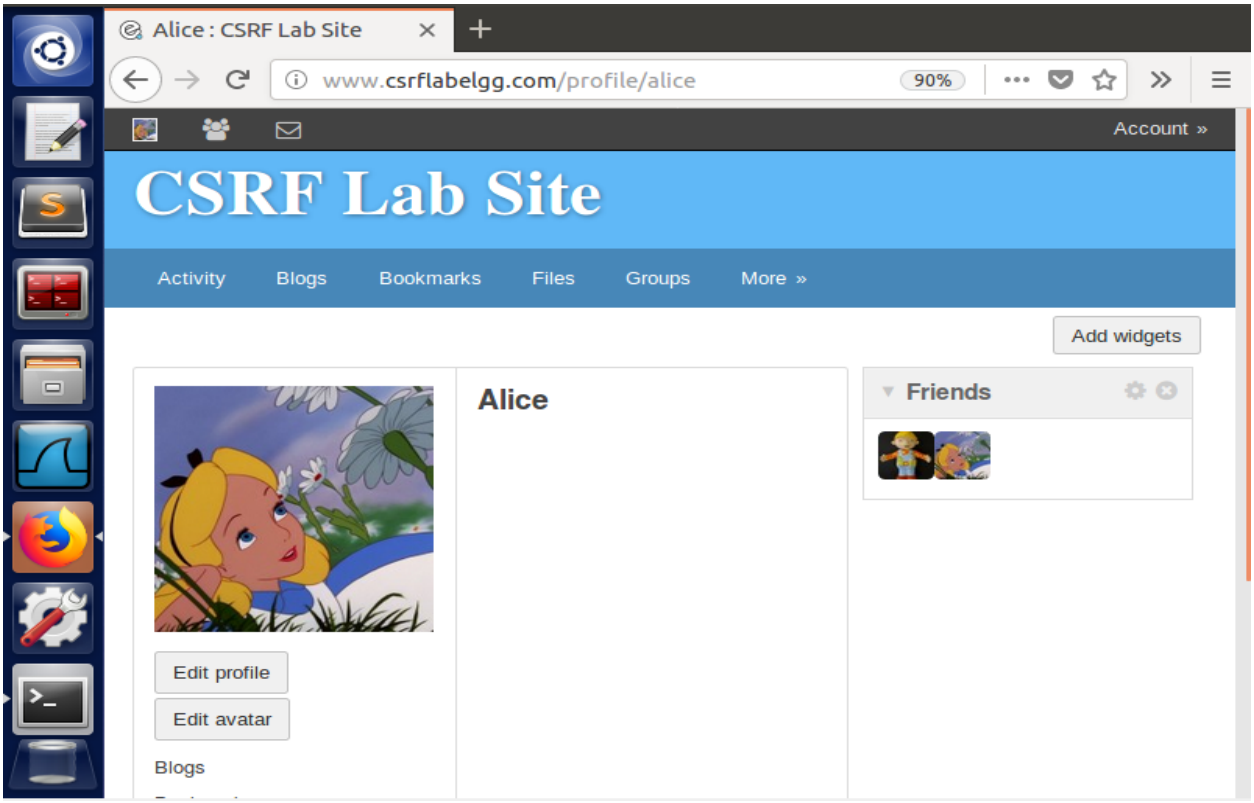
We have logged in as Boby. Boby now creates a blog post which has the attacker URL http//www.csrflabattcker.com/index.html.

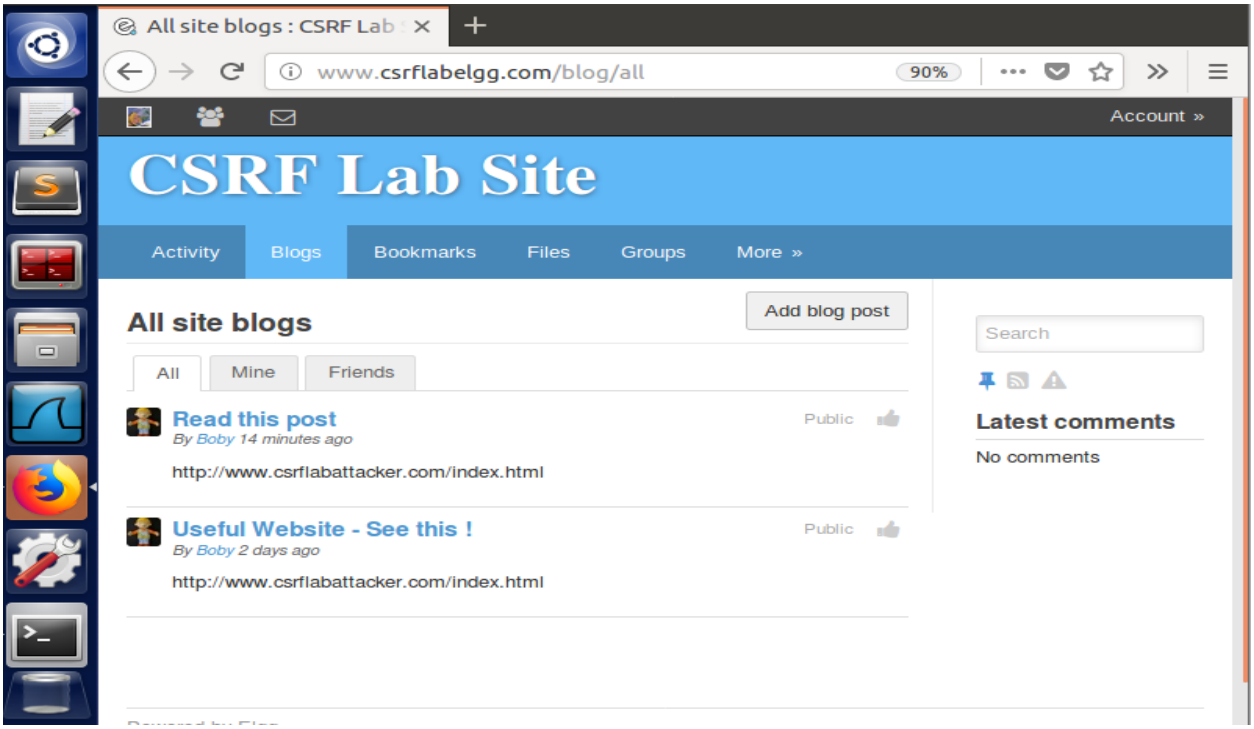


He then saves this blog post to be available to Alice expecting that Alice would click on this post and her description gets changed.

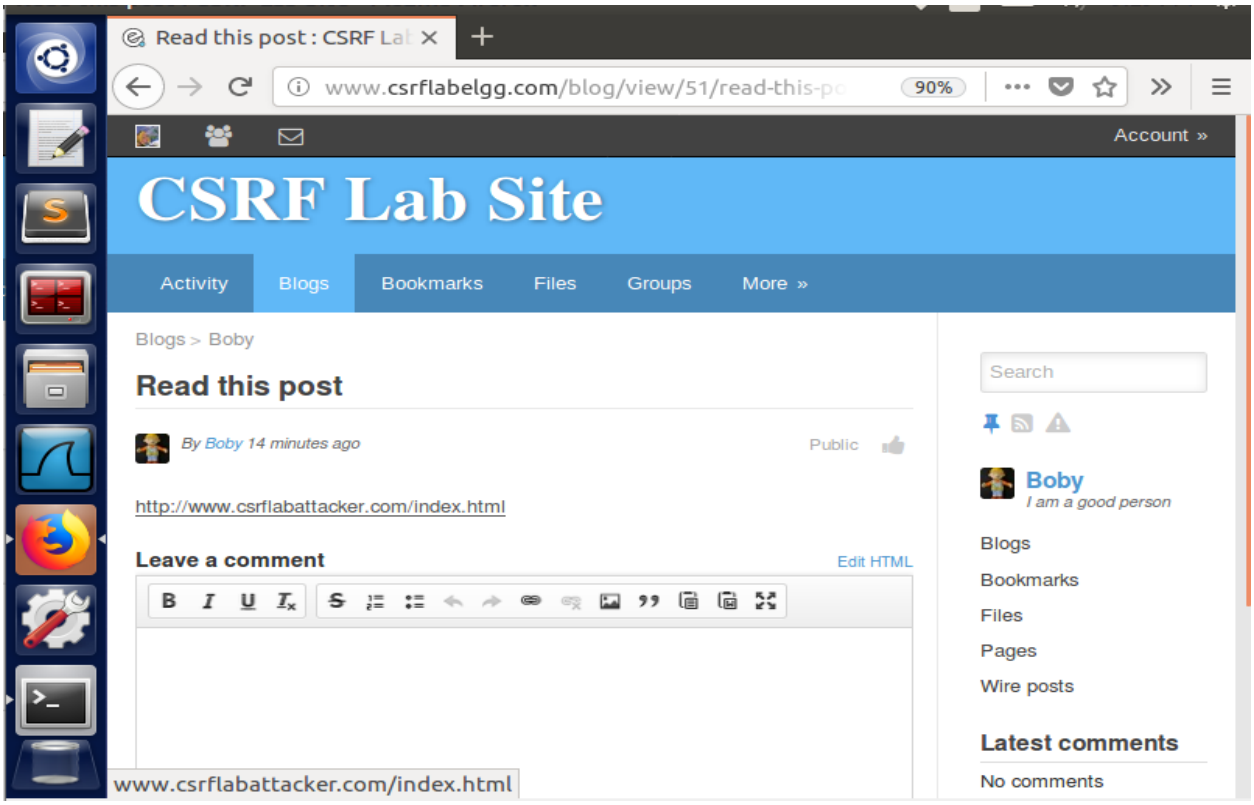


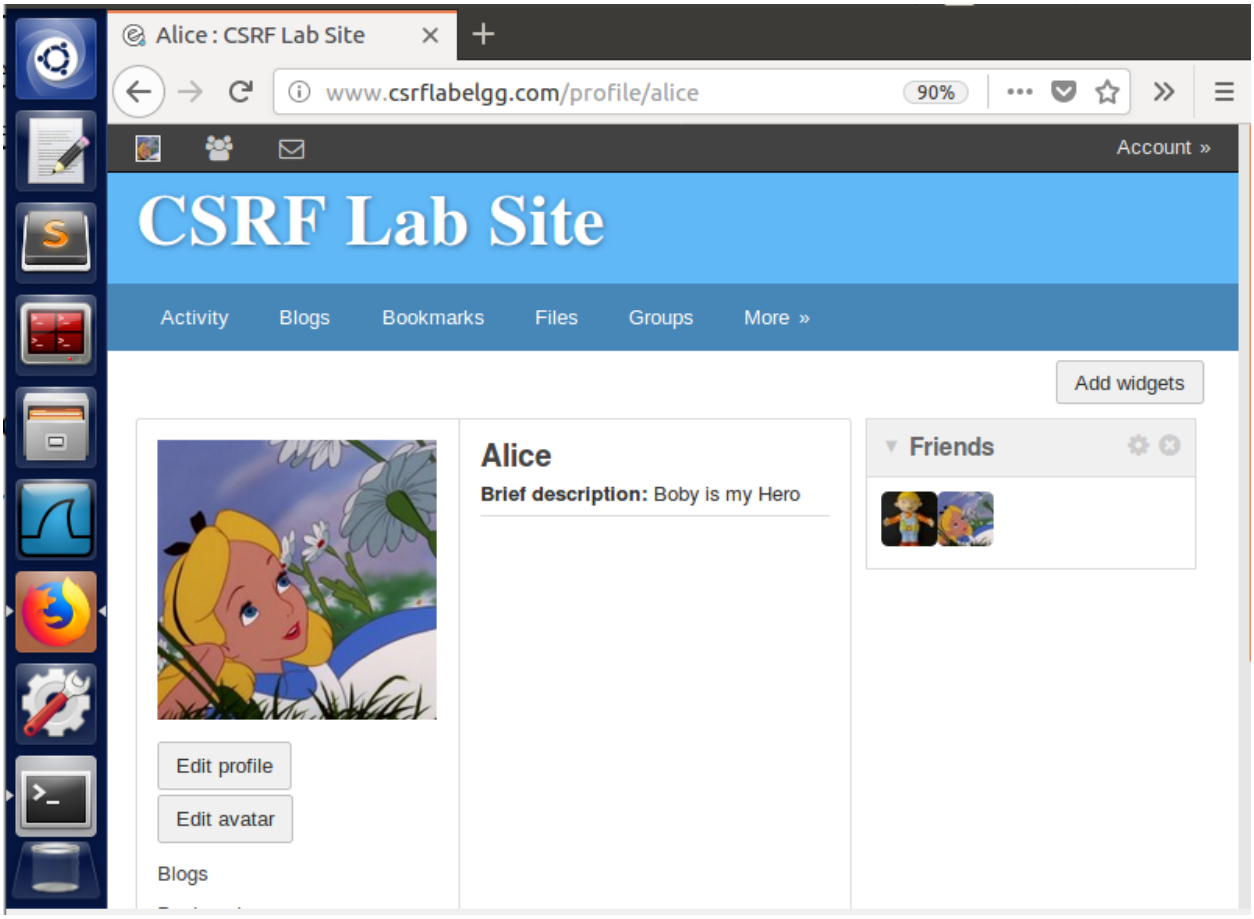
Initially, before Alice clicking the malicious URL, Alice profile has no description as shown in the screenshot.





We have logged in as Alice now. Alice goes and checks the new blogs in the Elgg website. She finds a new blog post by Boby and curious goes and checks the post. She goes and clicks the malicious URL in the post.

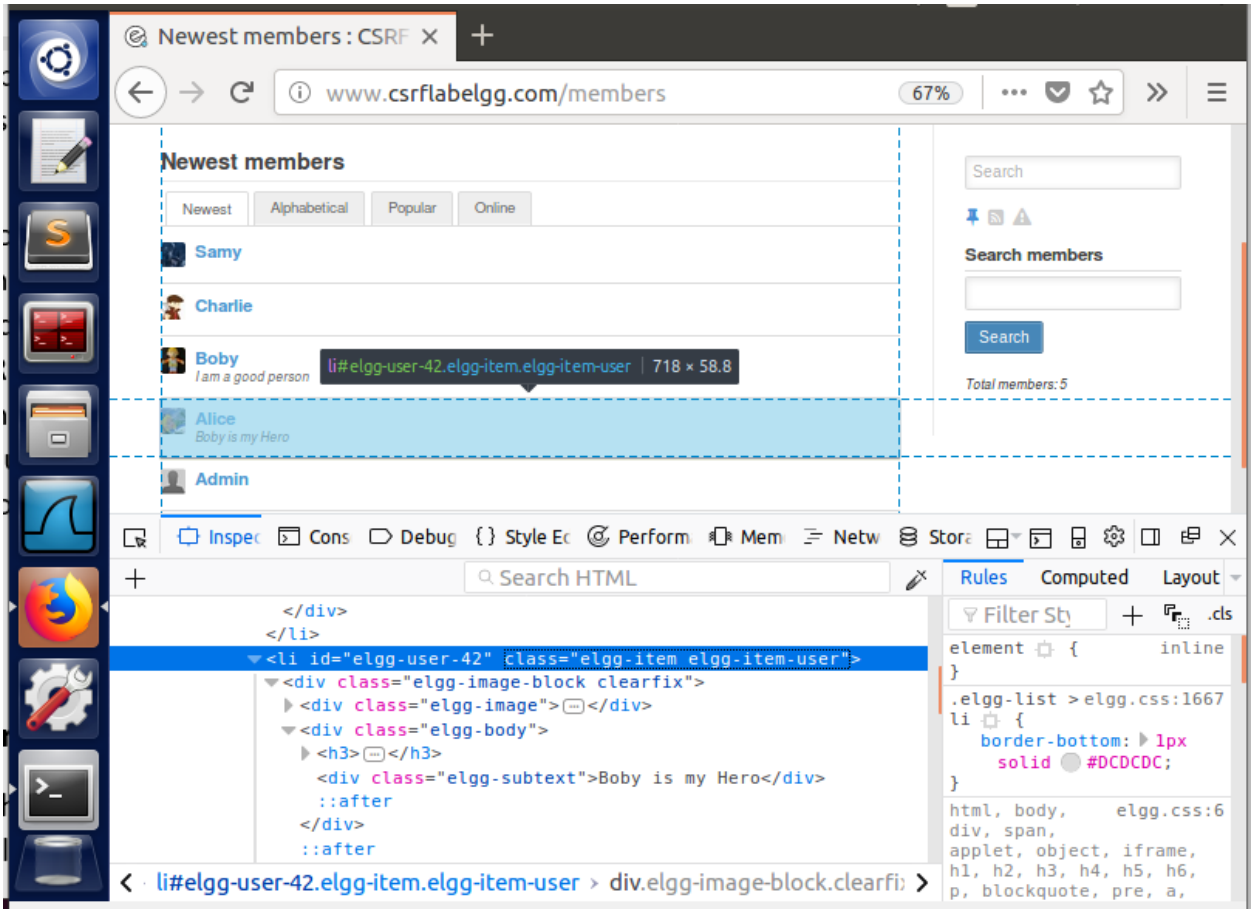




Once Alice clicks the URL it redirects her to her profile page where she sees that her description has been changed to 'Boby is my Hero' without her knowledge. Thus, Boby's attack was successful.

Conclusion: Here the attack is done using the POST request. Boby wanted to change Alice's description saying 'Boby is my hero'. So, he did the dsame thing with his profile in order to know the exact URL and get the POST request. He edited his own description in his profile and noted down the POST request using Http Header Live add on. He knew that the SAVE button would give a POST request to the Elgg website. He noted down the accesslevel, briefdescription, elgg_ts, elgg_token, etc. in order to create a JavaScript code. He creates a malicious URL, http://www.csrflabattacker.com/index.html and puts the JavaScript code in it which sends the POST request to Elgg website posing as Alice. The Elgg website will have the form submission as if Alice has requested. Now he creates a blog post with the malicious URL. When Alice comes and sees the post, she clicks on the malicious URL. This URL injects the malicious code in the trusted Elgg website. It appears to the Elgg website as if Alice has changed the description and sent a form submission to the trusted website. But when Alice goes and investigates her profile, she sees a change in her description without her changing it. This is due to the cross-site forgery request sent by Boby using the POST request. The forged HTTP request needs Alice's user id to work properly.

Answer to Question 1: As I explained earlier in Task 2, we go to the members page http://www.csrflabelgg.com/members and use the 'Inspect' element in the FireFox browser to get Alice's user id.



Answer to Question 2: No. Boby cannot launch the attack to anybody who visits his malicious web page. As explained earlier in this task, Boby has used the user id of Alice to perform this attack only on Alice. So he has to know the user id of each and every person he want to attack. That is impossible. He doe does not know who will visit his malicious URL beforehand. So, he will not be able to add the user id of everyone. However, if he knows the user id of the victim he wants to attack (in this case, Alice with user id 42) he can change the JavaScript code to that user Id. Whenever that particular user id has an active session with the Elgg website and clicks the malicious URL, only that person gets attacked. The attack happens only when the user id of the victim and the user id present in the malicious URL match. Otherwise, the attack cannot happen.

3.4 Task 4: Implementing a countermeasure for Elgg

The purpose of this task is to turn on the countermeasure and perform the CSRF attack on Alice again by Boby. This time Boby tries to change Alice's profile description to 'I love Boby'.

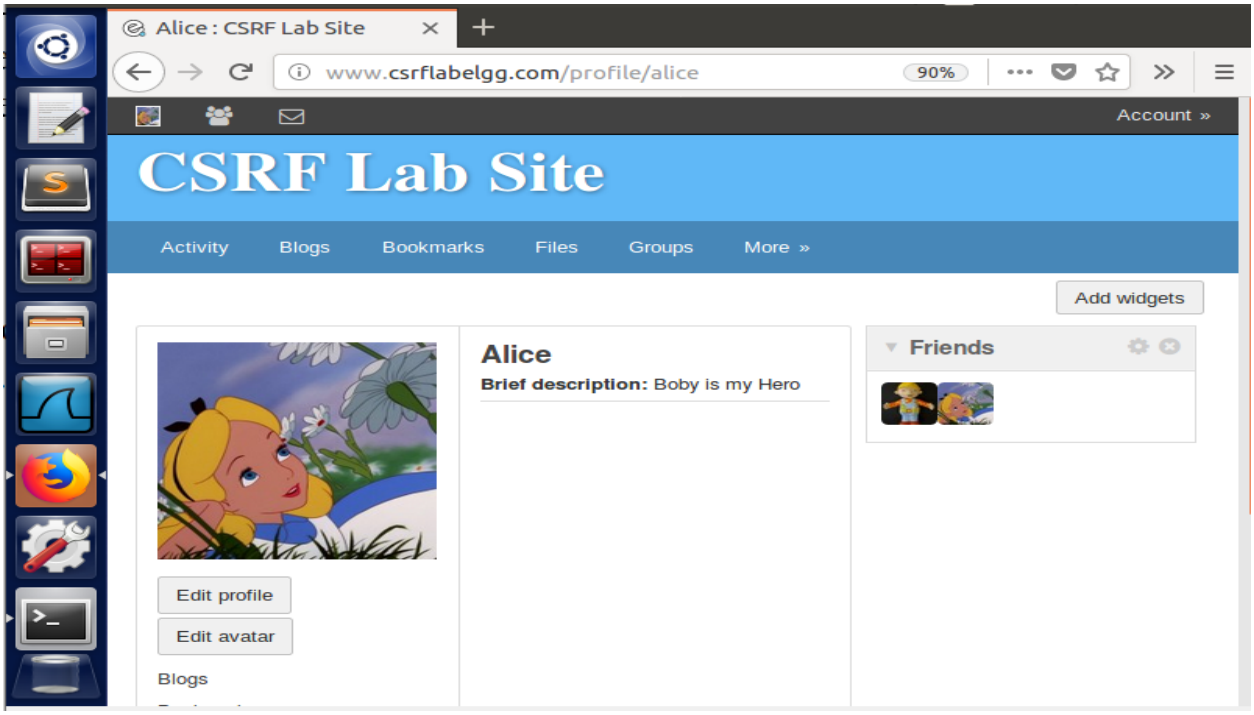
We go to the specified folder in the task /var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg and find the PHP file ActionsService.php. Then we edit this file and go to the function gatekeeper (). In this we comment out the return true statement as shown in the below screenshot. By doing this we are turning on the countermeasure of the Elgg website.

We then go to the attacker folder to modify the index.html which is our malicious code and do a complete Apache 2 service restart to make it active.

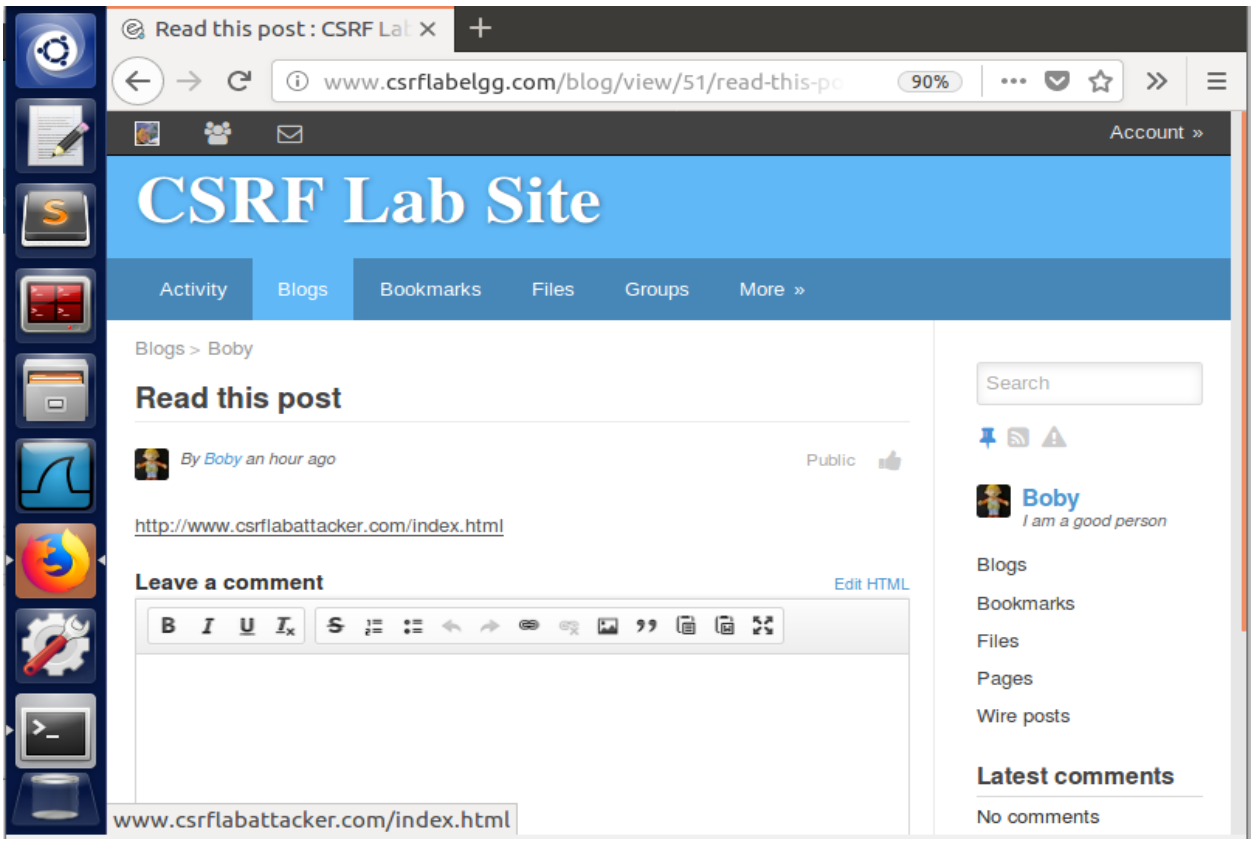
index.html

```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">
function forge_post()
{
var fields;

// The following are form entries need to be filled out by attackers.
// The entries are made hidden, so the victim won't be able to see them.

fields += "<input type='hidden' name='_elgg_ts' value='1585958976'>";
fields += "<input type='hidden' name='_elgg_token' value='KDiFeSeW-EzFC8UQ01T9aw'>";
fields += "<input type='hidden' name='name' value='Alice'>";
fields += "<input type='hidden' name='briefdescription' value='I love Boby'>";
fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
fields += "<input type='hidden' name='guid' value='42'>";

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

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

// Append the form to the current page.
document.body.appendChild(p);

// Submit the form
p.submit();
}
// Invoke forge_post() after the page is loaded.
window.onload = function() { forge_post();}
</script>
</body>
</html>
```

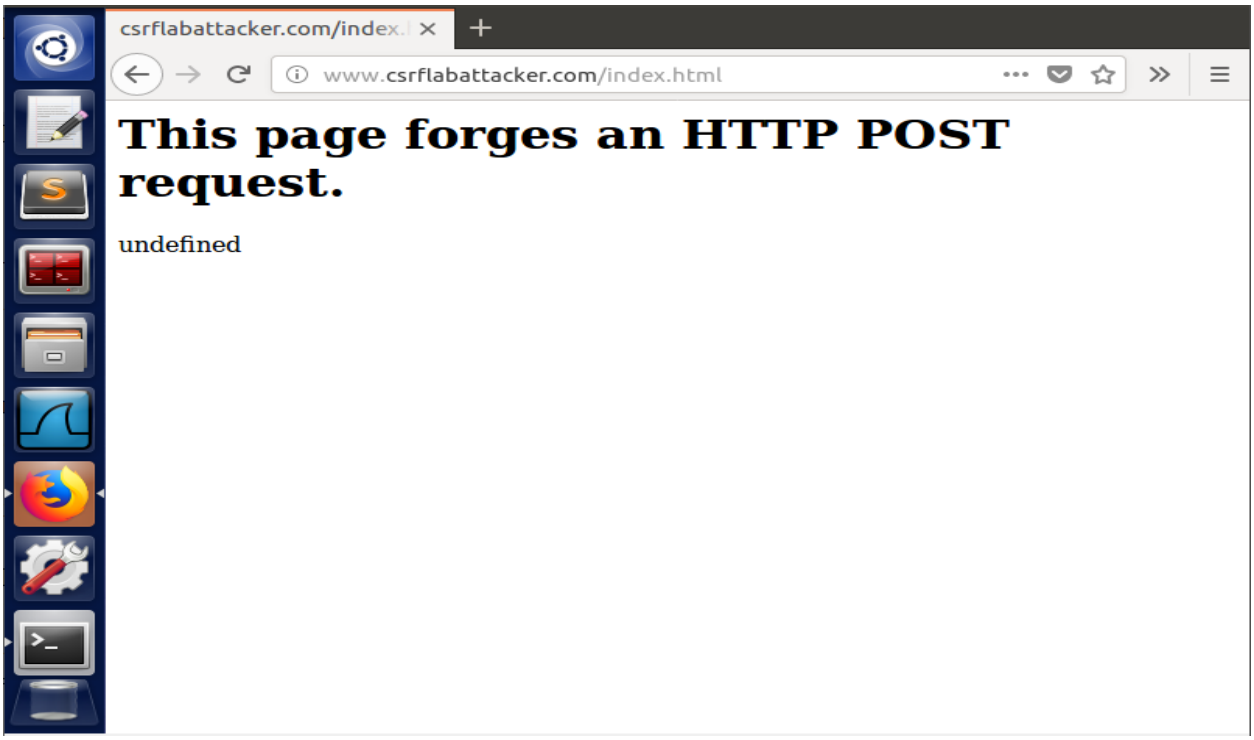
The changes made in our malicious code is that we have added two secret tokens in our JavaScript code. One is the elgg_ts i.e. the timestamp and the other is the unique elgg_token that we got from Boby's POST request from Task 3. We make these fields also hidden so that Alice and the trusted Elgg website does not know how and when the attack happened.



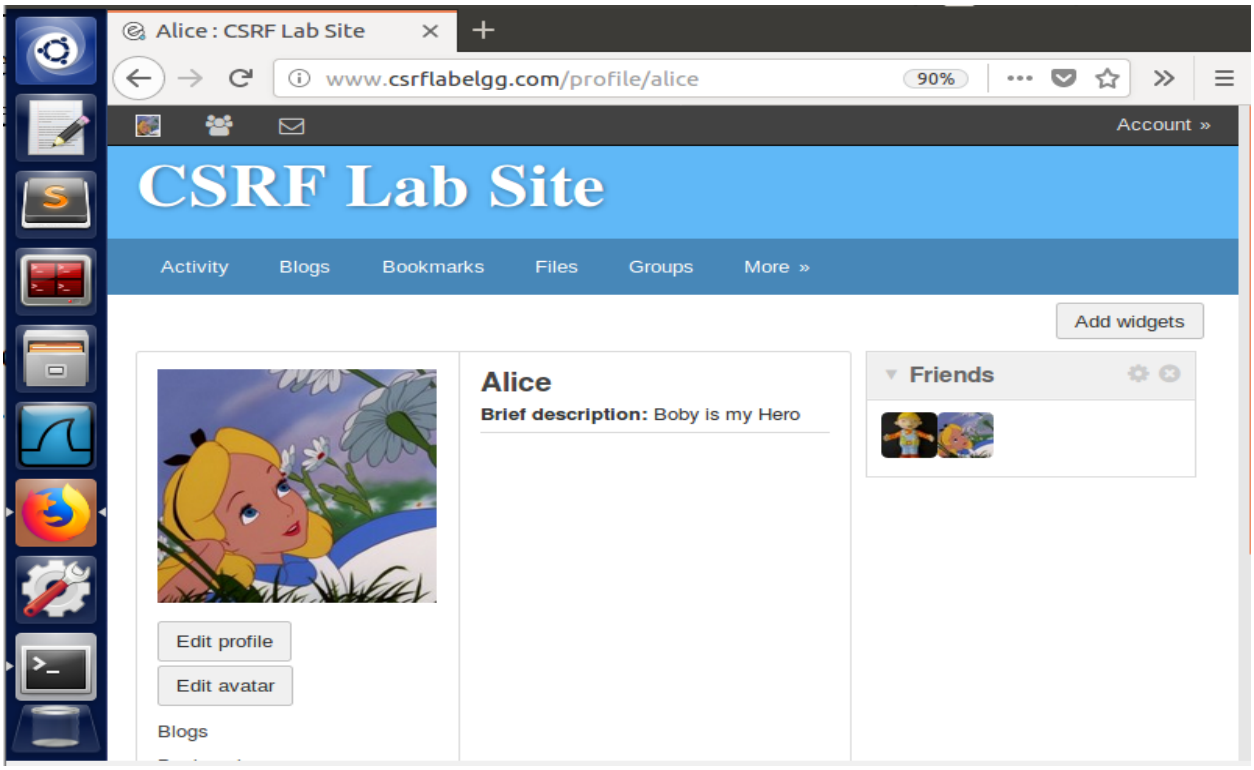
We login to Alice's account now. Initially, Alice's sees that her description is not changed and it's the same as before. She goes and checks the Blogs in her website and goes to the same blog created by Boby as before. She clicks that malicious URL again this time.



This time, she sees a page warning her about the attack. She gets a display saying ‘This page forges an HTTP POST request undefined’.



Alice is warned by this message and goes and checks her profile again to see if she is being attacked. She sees that her profile description has not changed at all and she is not attacked.



Conclusion: After turning on the countermeasure by commenting the 'return true;' statement in the gatekeeper () function, Alice goes and clicks the same malicious URL and see that her profile description is still not changed. This time Boby included the two secret values that is unique to each and every user. He assumed that the values are not unique and tried the attack. But since Alice had a diffrenet timestamp (elgg_ts) and different and unique secret token (elgg_token), she was not attacked by Boby.

The secret tokens in the HTTP request captured using Firefox's HTTP inspection tool HTTP Header Live are elgg_ts and elgg_token.

Earlier, when the countermeasure was off, in the ActionsService.php file, we saw that the gatekeeper () function allowed the POST request even when the validateActionToken was false. When the validateActionToken was false, it was changed to true. Now, we turn on the countermeasure by commenting this 'return true' statement. This clearly sends the information to the trusted website and the victim about the malicious attack.

Each time a POST request form submission is made to the trusted Elgg website, a unique elgg_token and elgg_ts timestamp is created. This the reason why the attacker Boby cannot get this information from the HTTP Header Live add on of FireFox. Whenever there is a POST request, it checks whether the two secret values are valid for the current active session. Since, these values are not unique, it sends a warning message to the Elgg website and the victim user. It is identified that it is a cross-site request and a forgery and does not allow it. This prevents Boby from finding out the secret tokens from the web page.

References:

https://github.com/aasthayadav/CompSecAttackLabs/blob/master/8.%20CSRF%20Attack/Lab%208%20CSRF%20Attack.pdf