**ASSIGNMENT – 7**

Name: **Sudharsan Srinivasan**

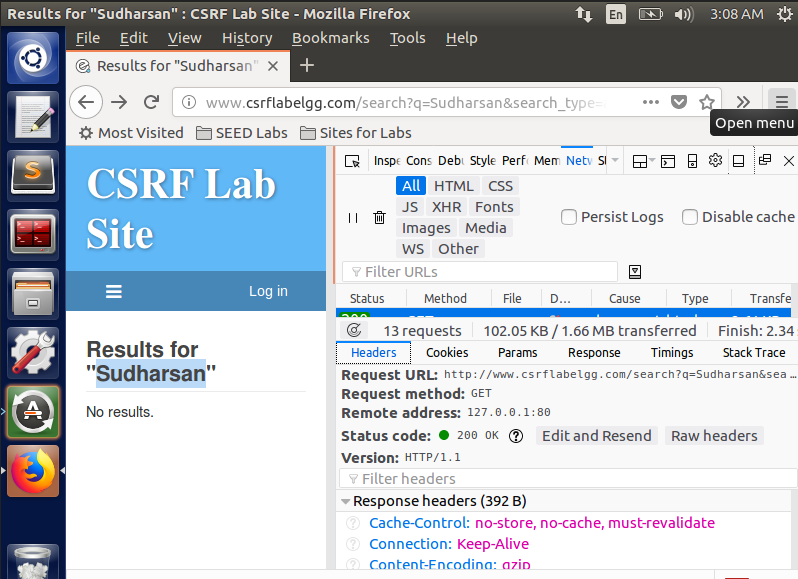
UTA ID: **1001755919**

**Environment:**

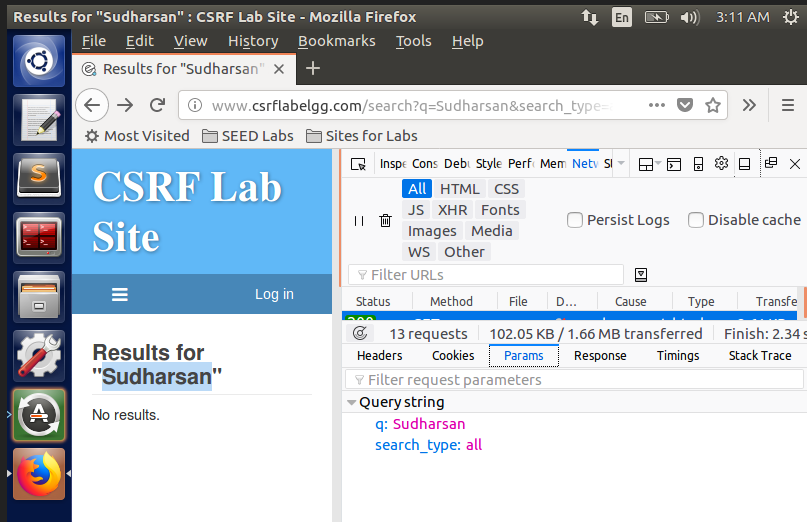
* The tasks are carried out in a web browser with **HTTP Headers Live** add-on installed to track the **GET** and **POST** requests. **Inspect Page Element** option is also used to track the status of the requests, parameters and headers involved etc.
* The websites used for the tasks are [**www.csrflabelgg.com**](http://www.csrflabelgg.com)to login to user accounts etc and [www.csrflabattacker.com](http://www.csrflabattacker.com) to launch attacks from, onto the user accounts.

**Task 1 – HTTP Request Observe:**

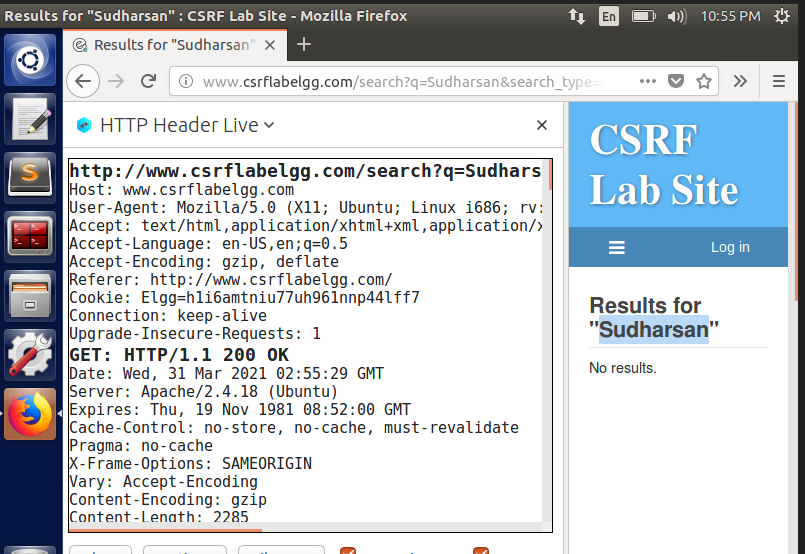
* This task requires to observe the HTTP request and responses corresponding to actions performed on the site [www.csrflabelgg.com](http://www.csrflabelgg.com)
* A typical action involves searching for name in the Search bar. In the below image, the username "**Sudharsan**"is searched in the search bar and results are displayed for the same.



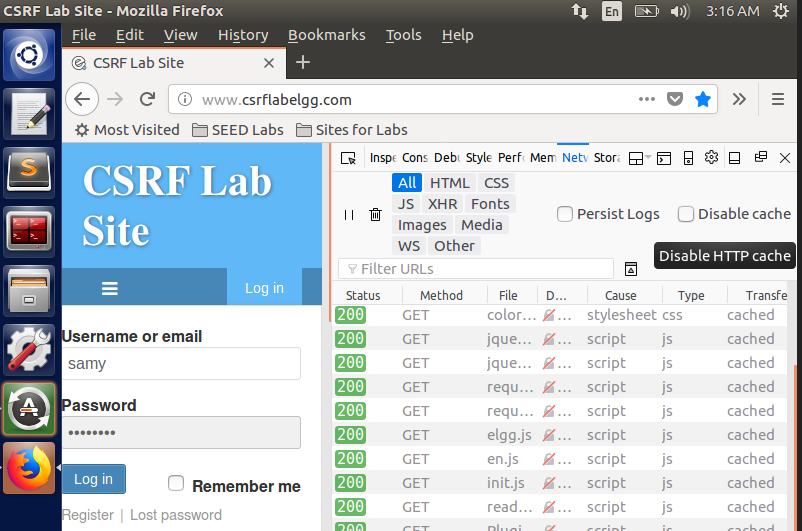
* The developer tool option provides the corresponding HTTP headers and parameters that are run for the operation performed, in this case, searching for a name.



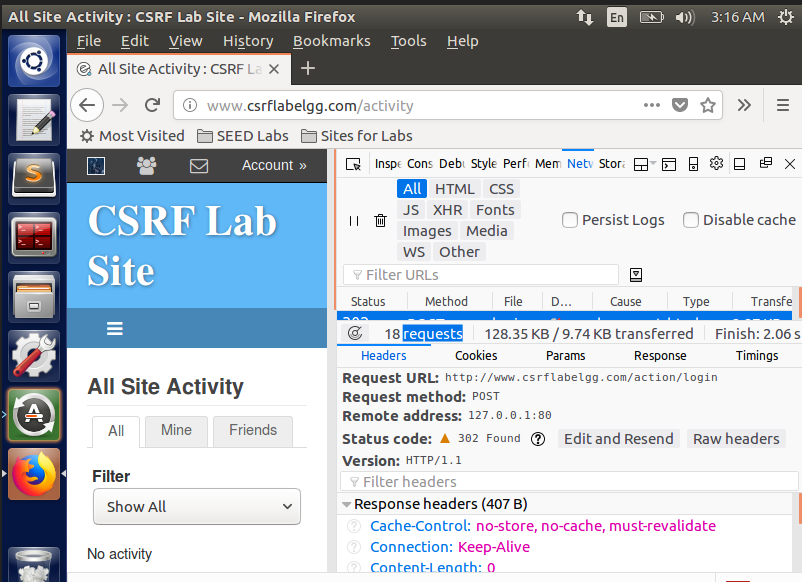
* The corresponding **HTTP Header Live** add-on is also shown below:



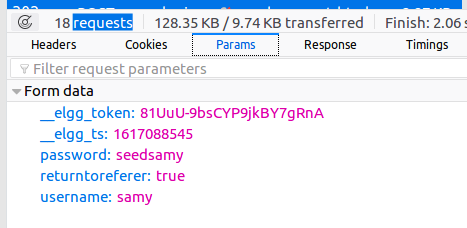
* The **params** tab above, shows the query string that was given in the search bar. It is also important to note that the method used for this request is **GET**. The browser also sets cookies corresponding to every activity performed.
* Next, using the login form, we try to login using one of the existing user credentials provided, let’s take the username **sammy** for this task. The credentials are provided in the **login** form.



* Now, this request has to be processed using a **POST** method, since POST method uses the information provided in the login form and validates the credentials, finally returning the status of the request.



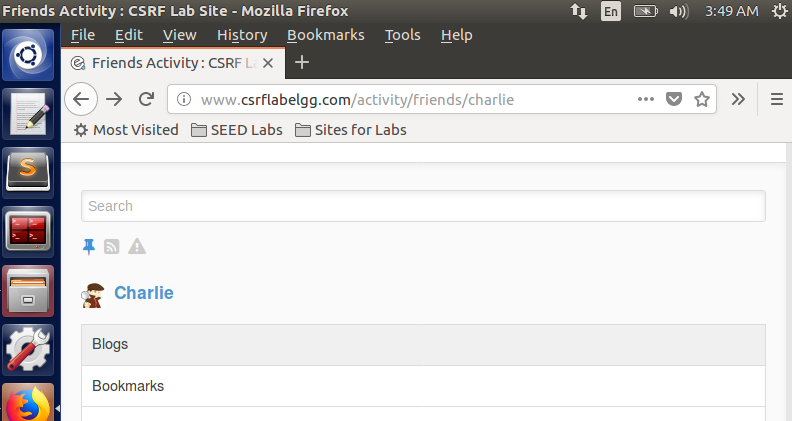
* The account has been logged in using the method **POST**. The Headers tab also contains the URL requested, the corresponding address and a Status code **302** indicating that the account has been located.



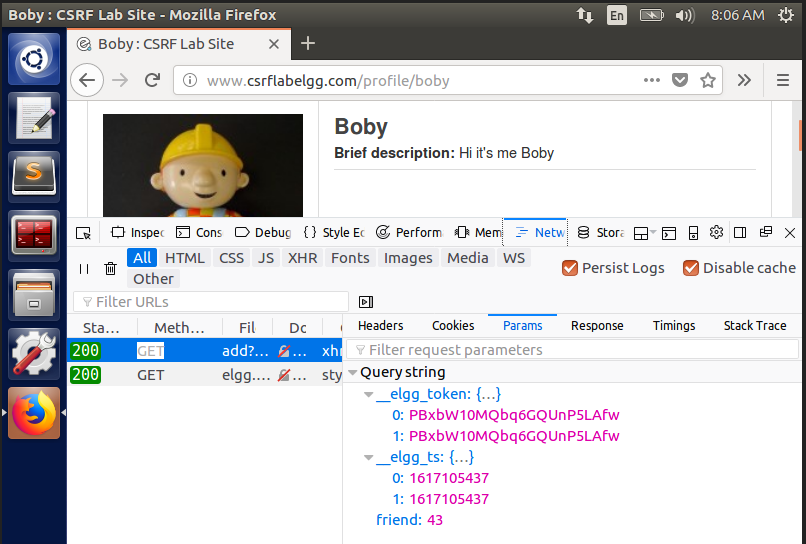
* Similarly, the **Params** tab shows the parameters that have been used in the login form, namely the username and password. The first two parameters shown are the **elgg token and timestamp**, which serves as the countermeasure to prevent **CSRF attacks.** The **returntoreferer** is used to identify whether the request is a **cross-site or same-site.**

**Task 2- Using GET Request to launch CSRF Attack:**

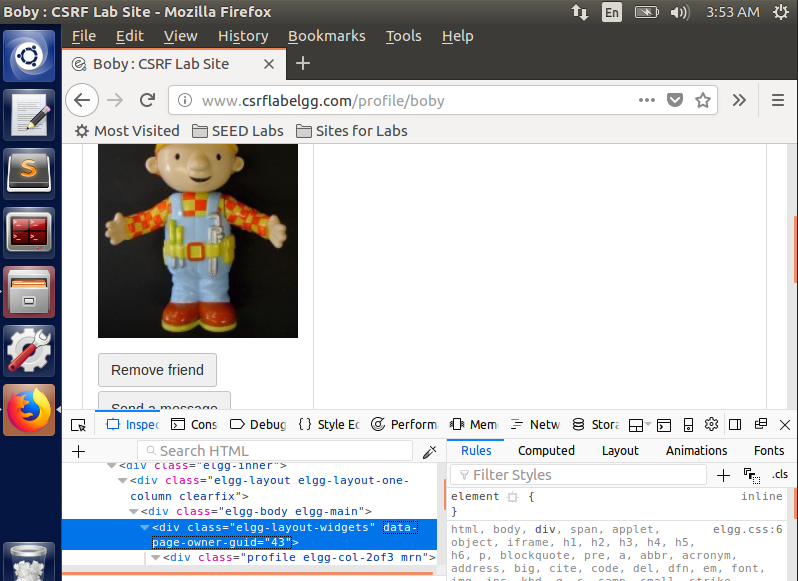
* The goal of this task is to use one account to launch an attack on another account and get added to their friend list without their knowledge. In this case, we have 2 accounts: **Alice and Boby.** The goal is to add Boby to Alice’s friend list without Alice’s knowledge.
* Here, in this case, Boby is the attacker and Alice is the victim.
* Before being able to add Boby as a friend to Alice’s account, it is important to know Boby’s GUID. This can be found by adding Boby as a friend in another account, say Charlie’s account.



* In Charlie’s account, search for Boby and click on **Add Friend** button. Notice on the Networks tab in the Developer tools, there is a **GET** request generated corresponding to the action. Click on that request and head to **Params** tab to see the value **friend:43**. This is the id of Boby.



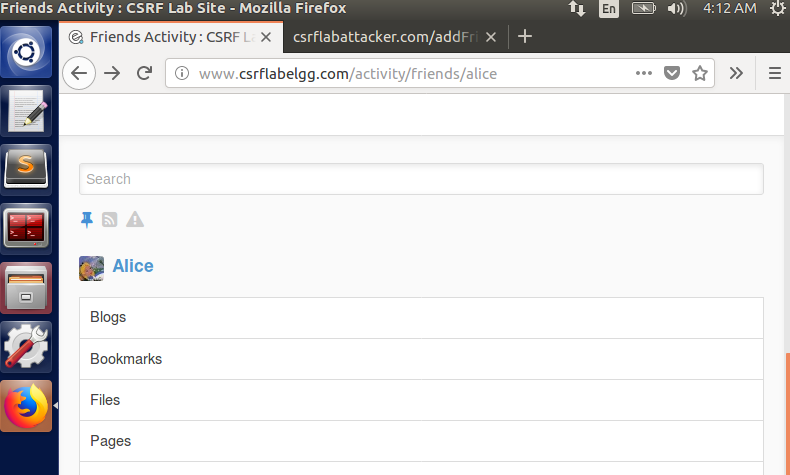
* The GUID can also be checked in the source code of the entire site, by going into **Inspect Element** and selecting the **Inspector** tab.



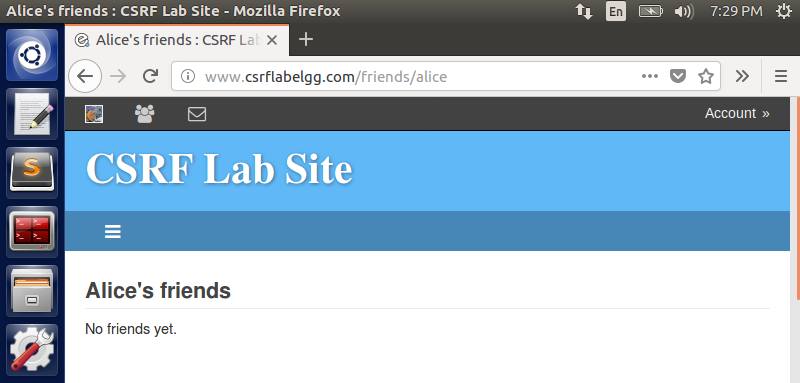
* Now that we know the GUID of Boby (43), we construct the attacker side code with a website link to send to Alice. The attacker code **addFriend.html** is stored in the path **var/www/CSRPF/Attacker**. The html file contents is shown below:



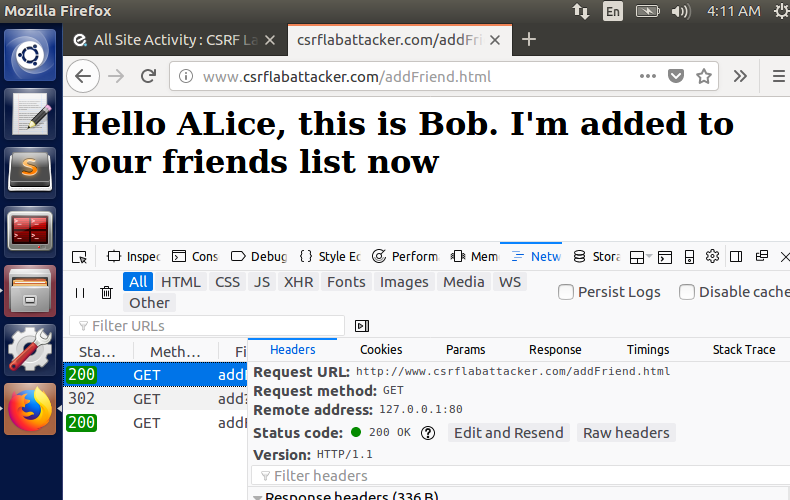
* Here the contents in the **h1** tag are what will be displayed in the page when Alice navigates to the malicious site. To generate GET request, we use **img src** command in HTML, which generates GET request when web page is loaded.
* After all the necessary setup is done, to check for the successful launch of CSRF attack, we log into Alice’s account as below:



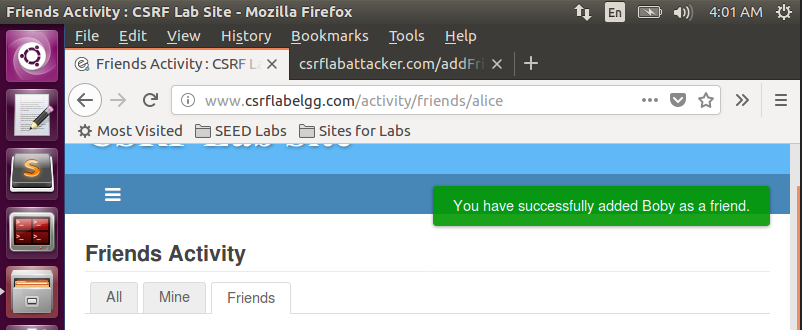
* Initially before navigating to the malicious website link, it can be seen from below that Alice has no friends yet in her list.

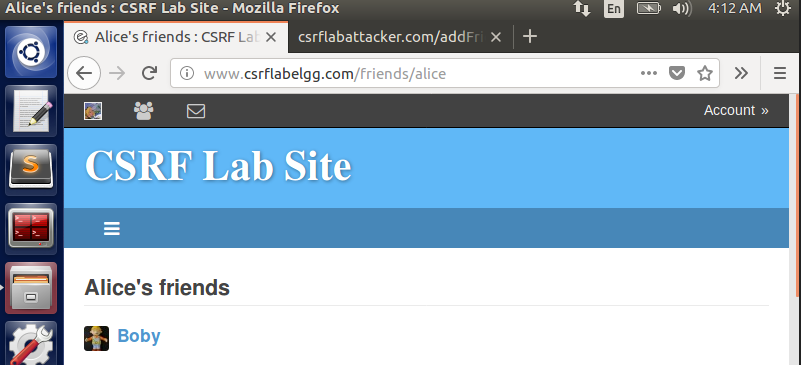


* Now, Alice navigates to the malicious website link provided to her. She goes to [www.csrflabattacker.com/addFriend.html](http://www.csrflabattacker.com/addFriend.html) and she gets the following message displayed.



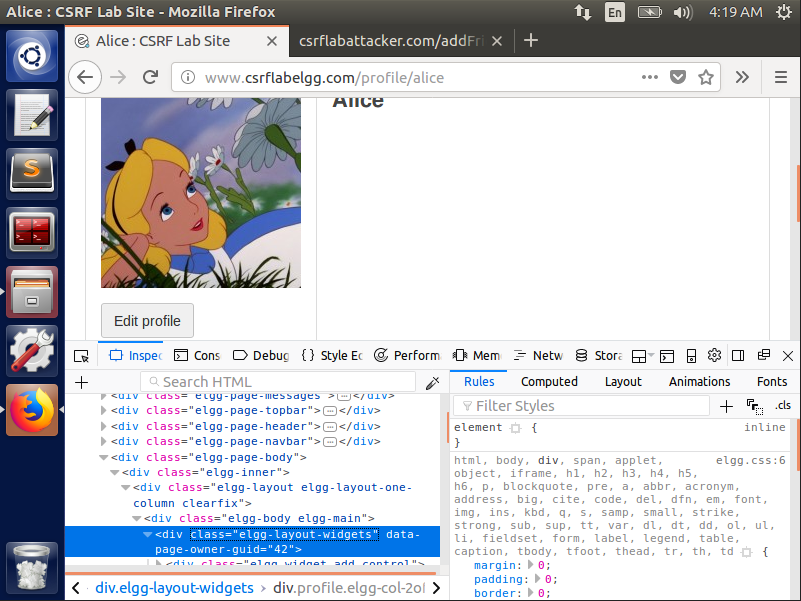
* Now, as a result of Alice visiting that site, a CSRF attack has been launched on her account without her knowledge and Boby gets added to her friends list, which can be seen from below, proving that the attack turned out to be successful.



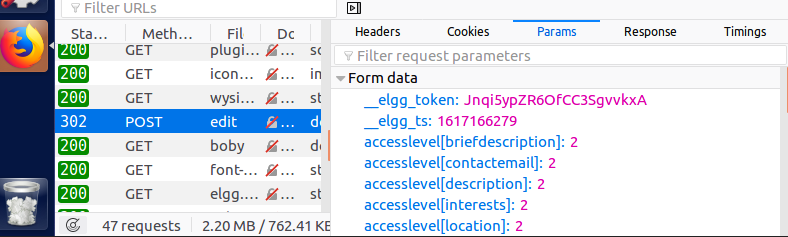


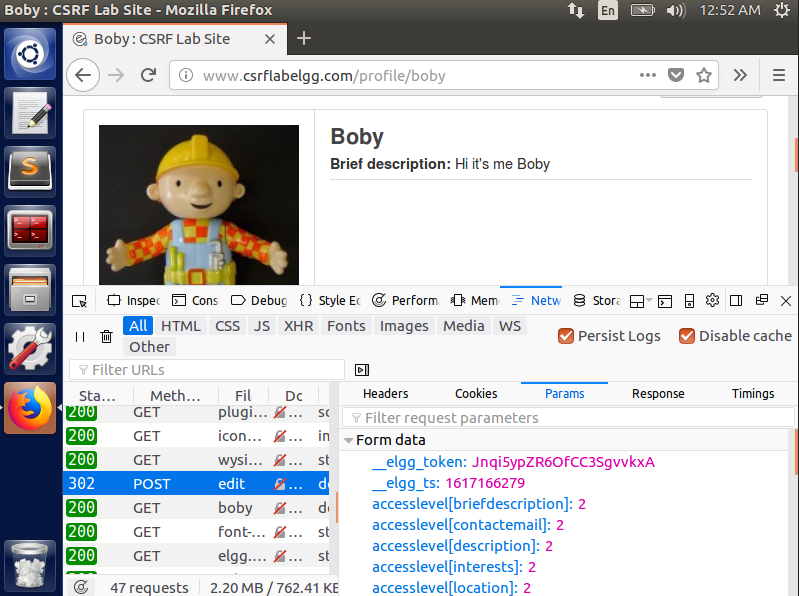
**Task 3 – Using POST Request to launch CSRF Attack:**

* The basic idea behind this task is to edit the profile description of Alice, without her knowing it, done using POST Request method.
* For this attack to be successful like the previous one, we need to know Alice’s GUID. To do so, we can search for Alice name from any account (let’s say we search from Boby’s account) and enter inside her profile page. Once there, do **Inpsect element**  and get the GUID value as **42** from the **Inspectors** tab.

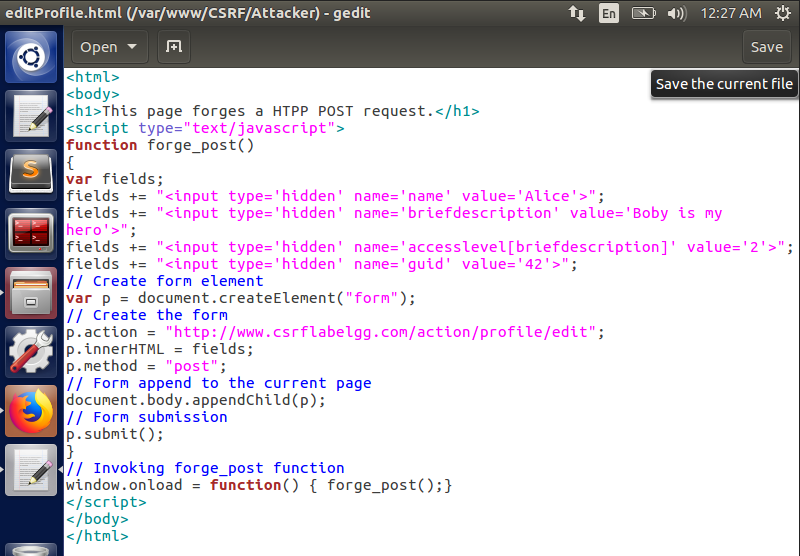


* Next, we need to know the access levels of description of Boby, which we want to overwrite onto Alice’s brief description. We then log into Boby’s account and click on **Edit Account** and update the brief description and click Save. Then we check the HTTP Request corresponding to the description change made using POST request in **Network** tab of **Developer Tools** option as seen from the below images.

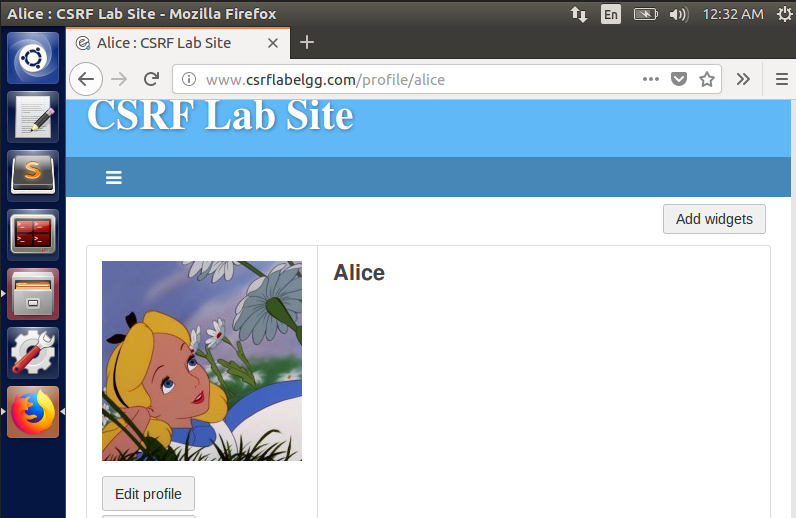




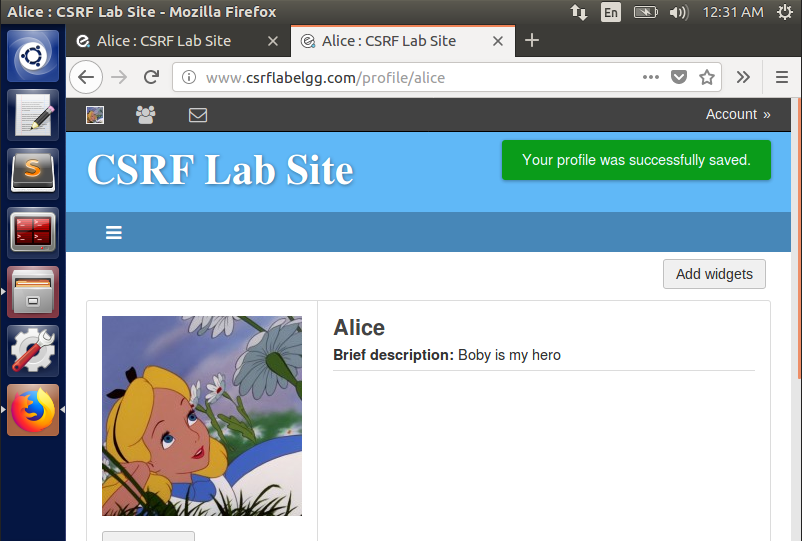
* Since we are looking to change the description, it is important to note the **accesslevel[briefdescription]:2** from the Params tab in the POST request made.
* Now that we have Alice’s GUID, Boby’s description values, we use these to construct a malicious html file, which will be sent as a link to Alice to perform the attack. The code is saved as **editProfile.html** in the **var/www/CSRF/Attacker** directory as shown.



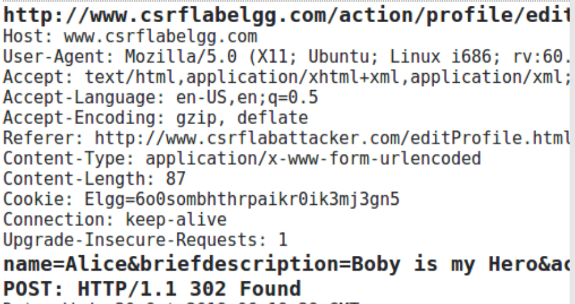
* Notice in the above html file, we have inputted description field values and used guid value of Alice, to parse her description field without her knowledge. The **p.action** is the action that will be performed by the code when Alice visits the malicious link.
* To check if the attack is carried out successfully or not, we navigate to Alice’s account by using her credentials.



* As of now, there is no description on Alice’s account. Then Alice navigates to the malicious site: [www.csrflabattacker.com/editProfile.html](http://www.csrflabattacker.com/editProfile.html)
* The updated account page of Alice can be seen below after she visits that site.



* The corresponding HTTP request for the same is shown.



* As seen from the above screenshots that we were successful in launching a CSRF attack to edit the brief description of Alice.

**Question 1 – Using Alice’s GUID without knowing credentials:**

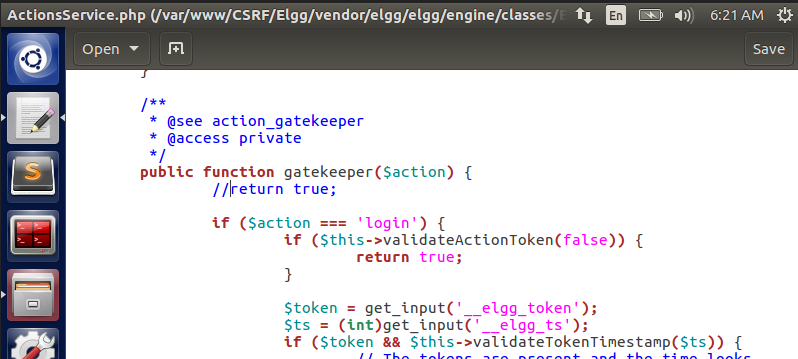
* This issue can be resolved using the same approach we had followed for the previous two tasks. By just visiting Alice’s account using the search bar and then using **Inspect Element** in her homepage, we will be able to get the **GUID** of Alice. If we manage to get the GUID, we wouldn’t need the credentials to launch an attack.
* The only drawback with this method is that, while inspecting the page elements, if the source code does not have GUID in it, then we would not be able to use this method. In such cases, we could monitor the **HTTP Request and Response headers** to see if we have GUID anywhere. If it’s not available even there, we wouldn’t be able to launch this attack.

**Question 2 – Launching CSRF attack on anybody visiting the malicious link:**

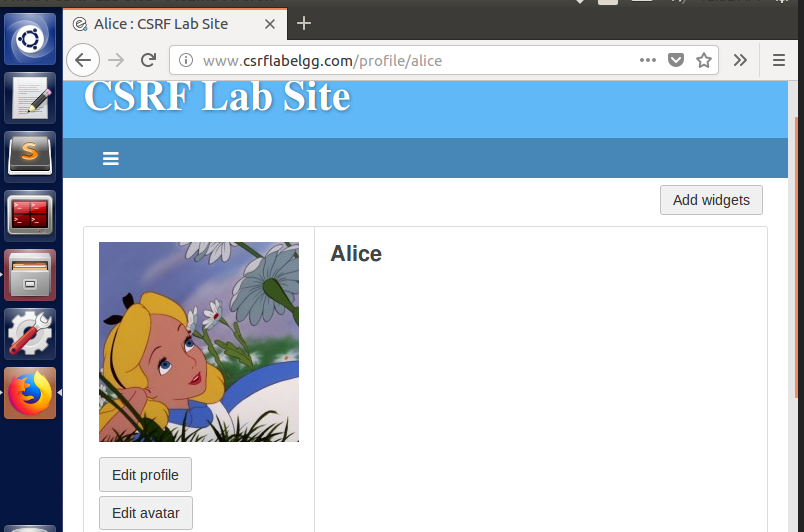
* This attack cannot be used on any random person visiting the malicious website because while constructing the contents of the malicious site’s source code HTML file, we specify the GUID and other attributes such as brief description corresponding to the person we are looking to attack. Only if this GUID present in the code matches with the GUID of the person visiting that malicious link, the attack would be successful, otherwise not.

**Task 4 – Countermeasure implementation for Elgg:**

* CSRF Attack is generally implemented through the gatekeeper function present in **ActionService.php** file, which returns true only after validating the token and timestamp of the visitor thereby assisting us in launching the attack.
* Therefore, in our task we enable this countermeasure again by commenting out the return statement and then try to launch an attack. The updated code of **ActionService.php** found in **var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg** is shown, where **return true** statement has been commented out.

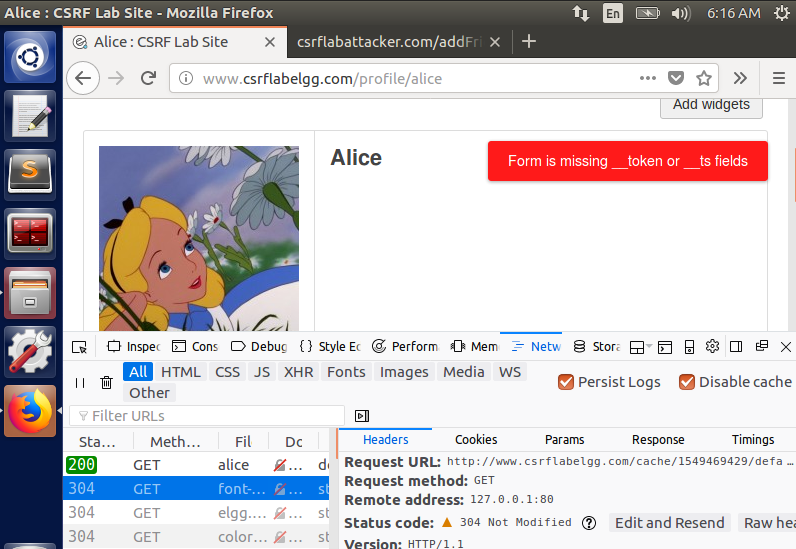


* Before proceeding to try and launch earlier attacks, we remove the results of previous attack. That is, we remove Boby from Alice’s friend list and also remove the description by navigating to Alice’s account.

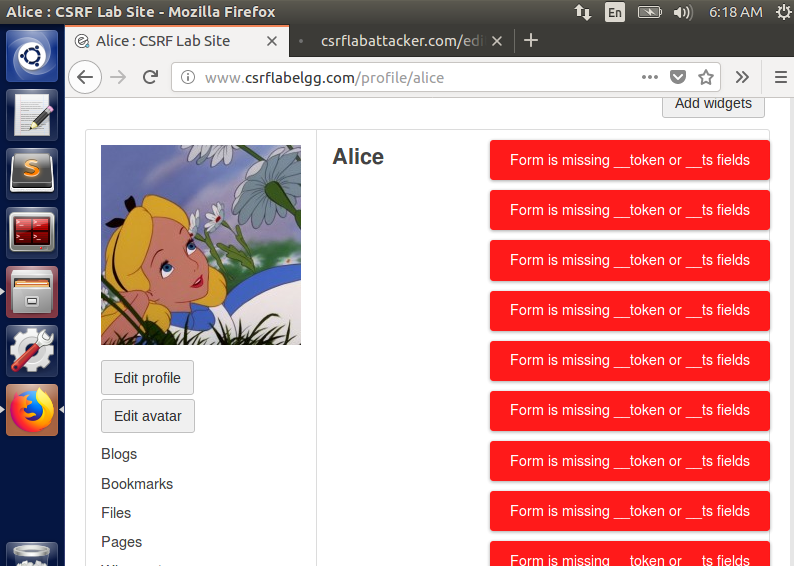


* After resetting everything, we go ahead and try to execute the previous GET and POST request attacks involving adding Boby as a friend and updating his malicious code’s description onto Alice’s profile.
* Using the GET method to launch an attack to add Boby as a friend, we get the below error indicating an unsuccessful attack.

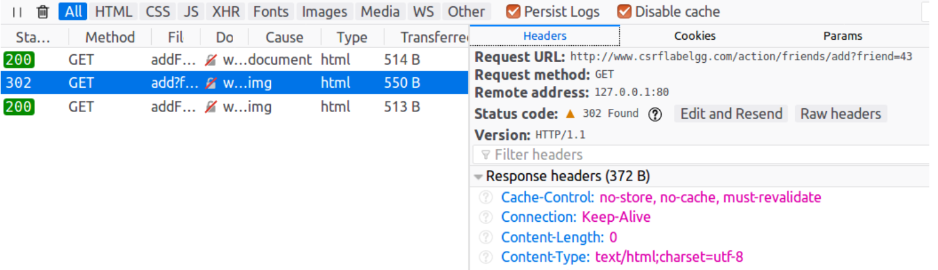




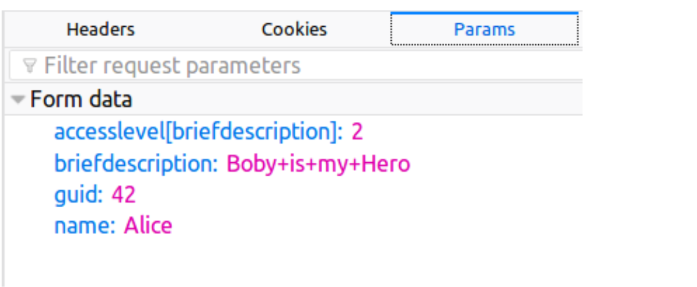
* We try the POST method to launch an attack to update Alice’s description, we get the below error indicating an unsuccessful attack.



* One possible explanation to this is because, since we commented out the **return true** statement, the code is not able to verify the token and timestamp of the visitor of the malicious site. As seen below in the GET Request URL, we only have one friend parameter.



* Also, for the POST Request method to update profile description, the token and the timestamp parameters are not present in the params tabs.



* This is because the attack is launched from a different malicious website designed by the attacker. The HTTP Request goes from that malicious site to Elgg Server. Only if a request comes from Elgg website to Elgg server, it sets the token and other parameters for the request. Otherwise, it does not set them.
* Therefore, a random user will not be able to get the token value of any user because the token gets generated only when a request is made. It is not preset in the database.
* So, the attacker would not be able to get access to those values and cannot launch a successful attack anymore.