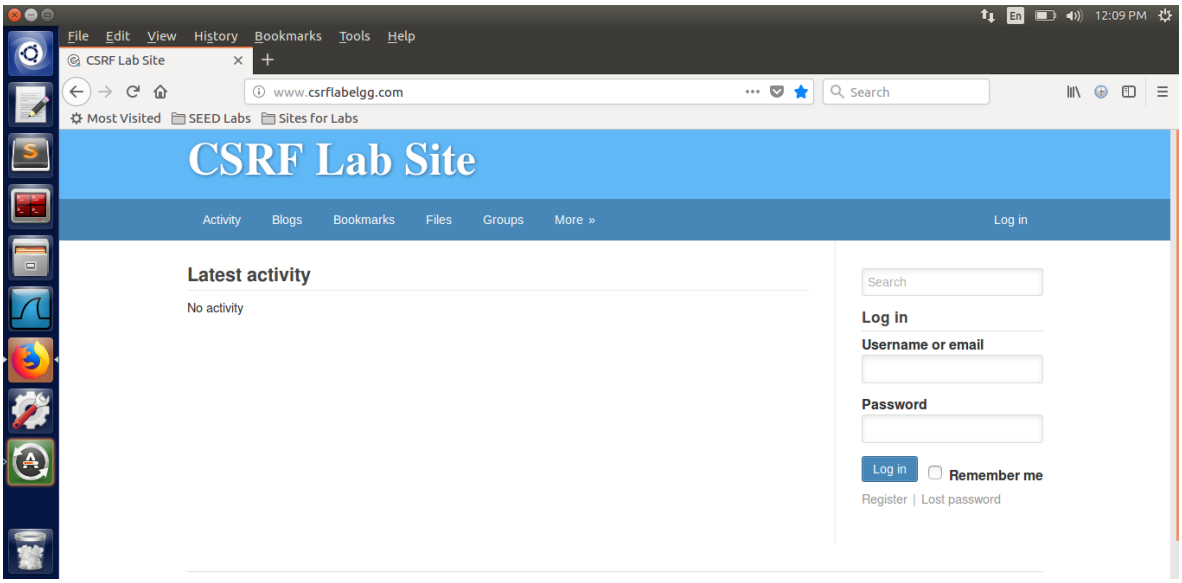
Assignment 7 :- Cross-Site Request Forgery (CSRF) Attack Lab (Web Application: Elgg)

Name :- Aparna Krishna Bhat

ID:- 1001255079

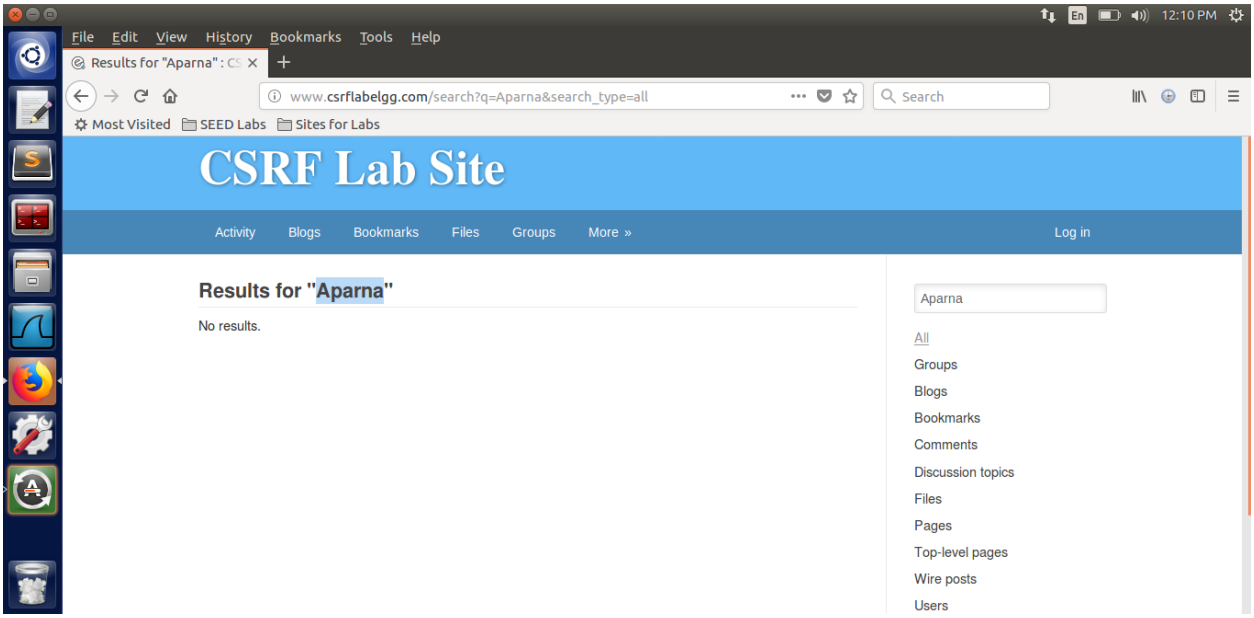
Task 0 :- Initial setup

Inside the virtual machine, the insecure Elgg site can be found at www.csrflabelgg.com. Throughout the lab, this will be our designated website.

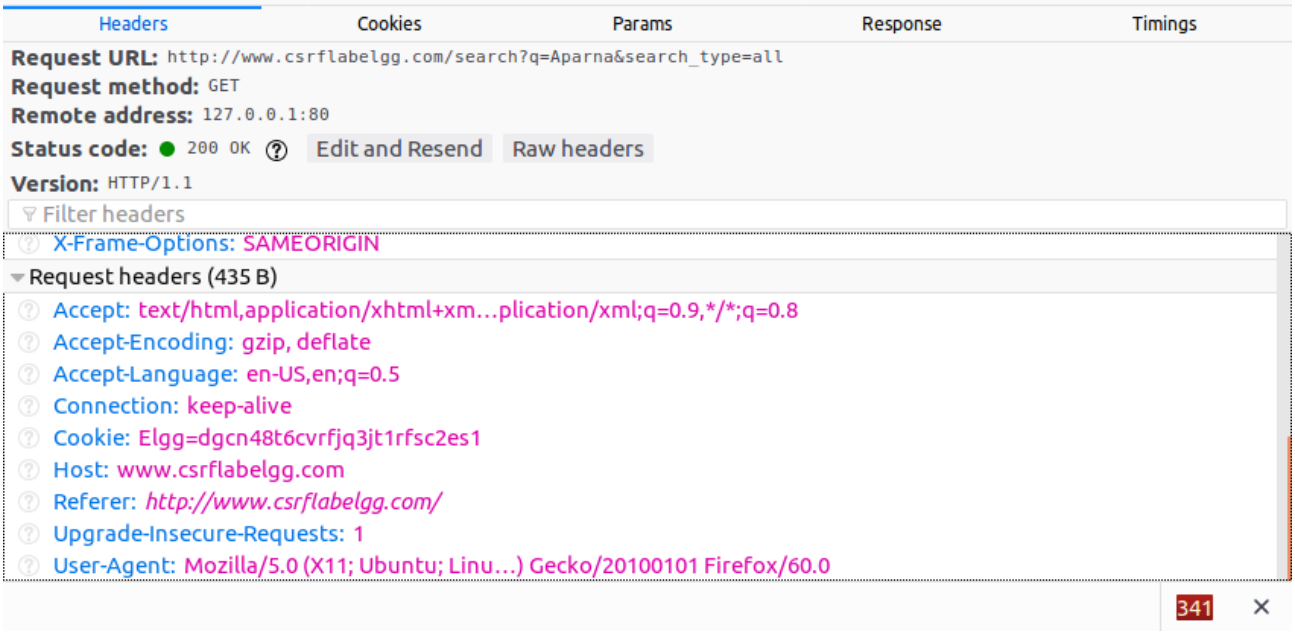


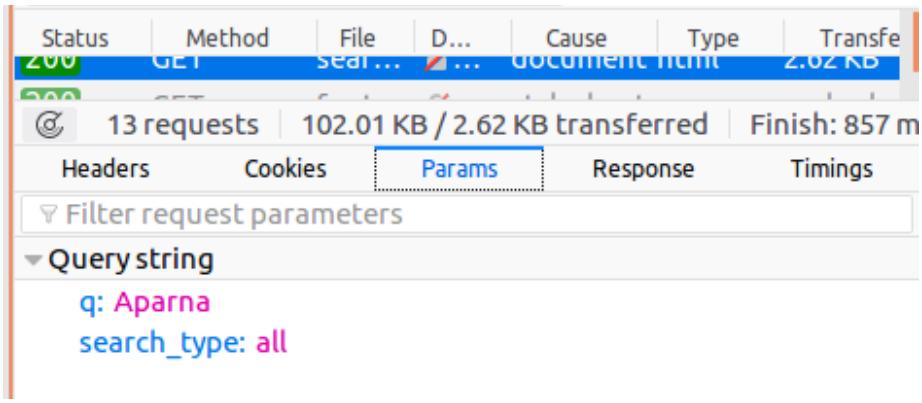
Task 1 :- Observing HTTP Request

The HTTP Request is the focus of this task. I perform some action on the website in order to see an HTTP request. Here, I use the website's search bar to look for my name “Aparna”, and using the developer tools, I can see the request that was made.

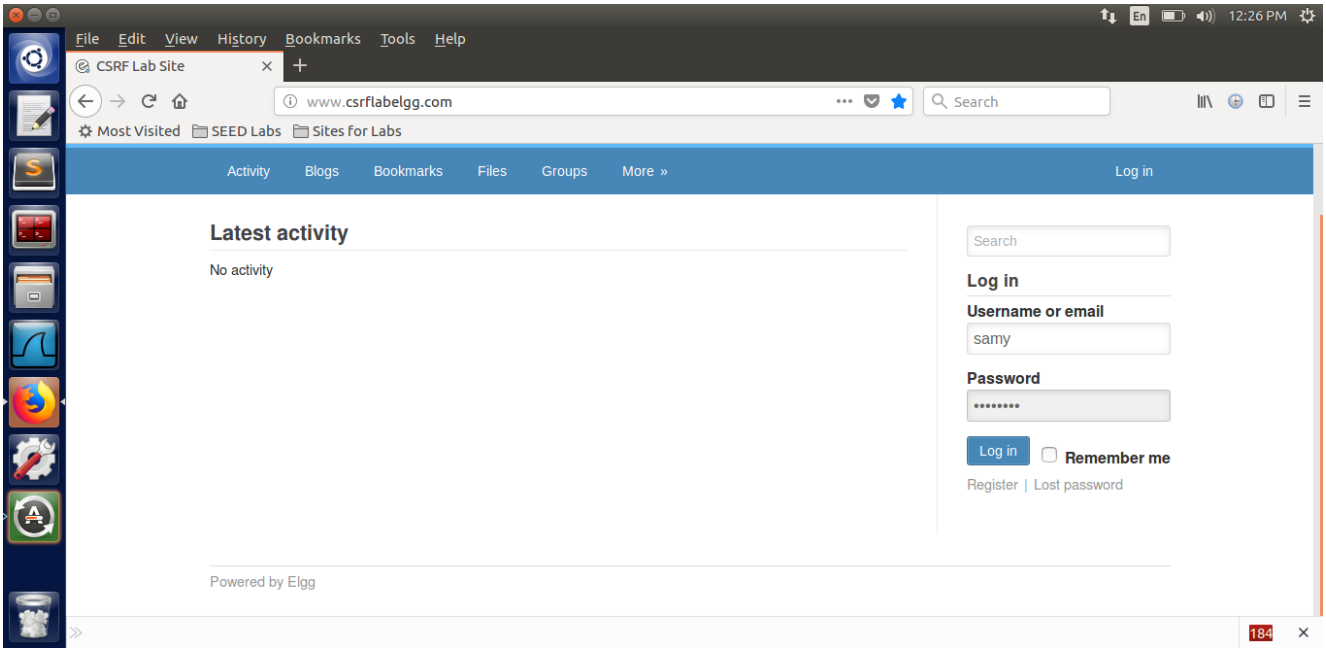


From the below screenshots we can notice that the method of the HTTP request is GET. From the Params tab, we can notice that this request contains parameters. Since All is selected as the field to search, q's value is set to 'Aparna', the string I entered, and search category is set to all. In addition, the Accept fields are sent to the server to show the type of data that the browser accepts.

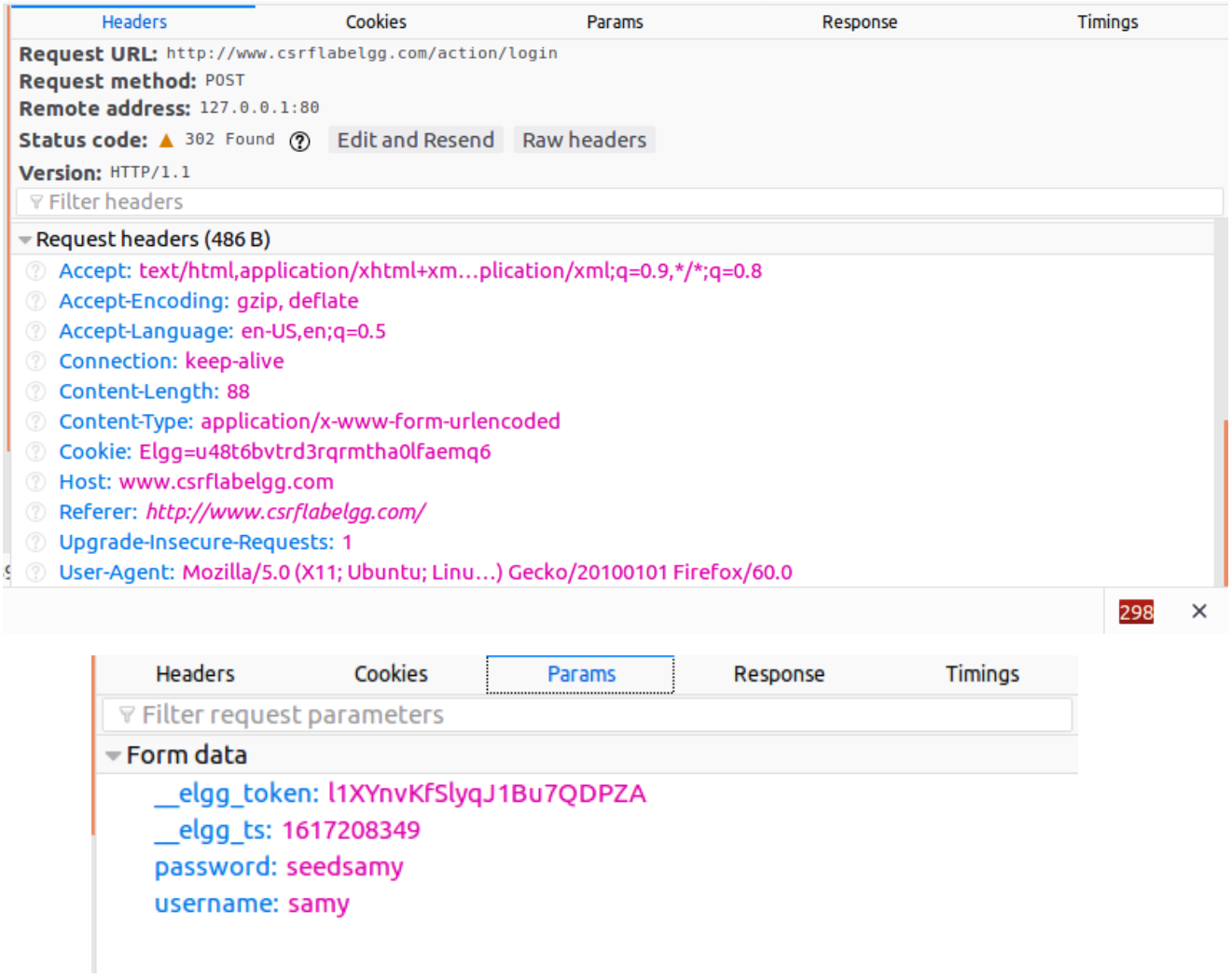




After that, we go to the browser and do something else. We know that a form will submit a post request, so we try to log in with one of the credentials provided in the task instructions, because the login will be a form eventually. Let us choose username as samy and the password provided for that would be seedsamy.



From the below screenshot we can note something interesting when we examine the HTTP request in the web developer tool. We can see a POST request with similar details in the header, as predicted. We can see that the cookie information is also present. The content-length and content-type parameters were missing from the GET request but were present in the POST request. This means that in addition to the HTTP Request header, there is some additional data sent. To do so, we will look at the Params Tab's contents.

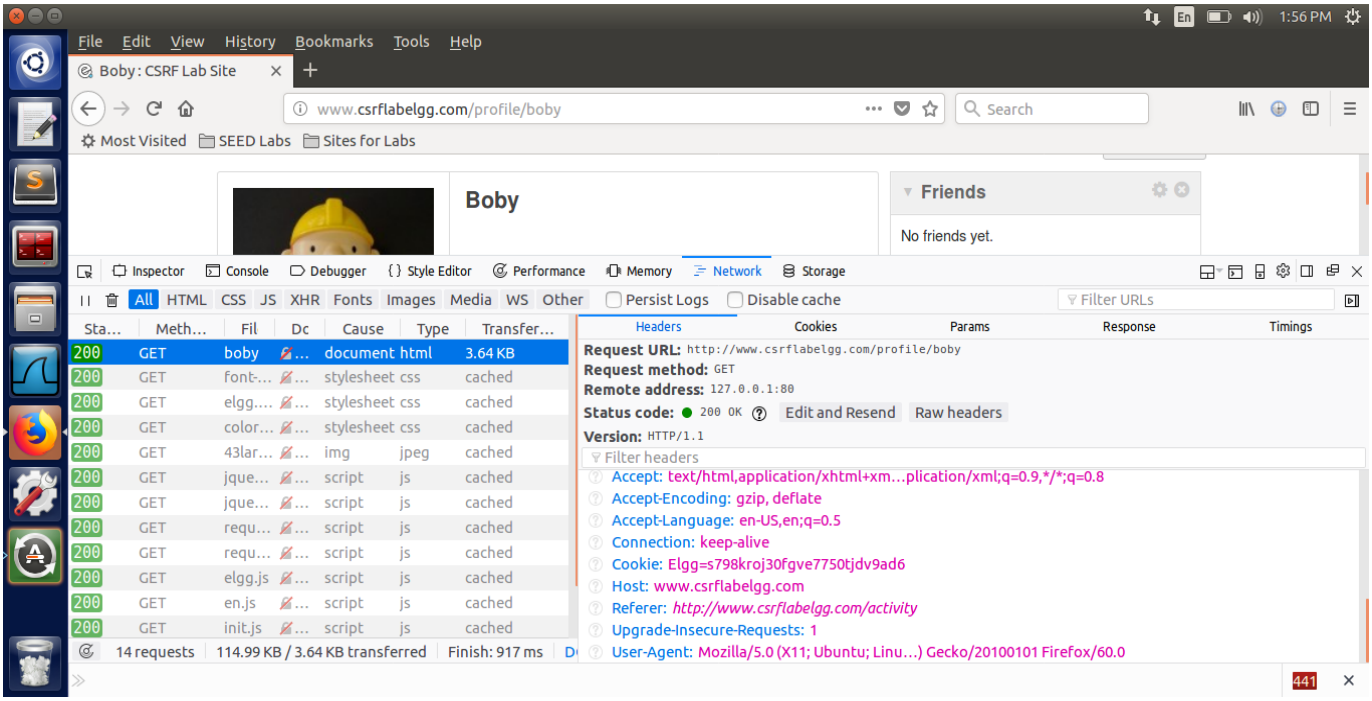


Some parameters that are sent in the HTTP request, can be noticed it the above screenshots. I filled out the fields Username and Password. Return to referrer is set to Correct, which means the result will be sent to the referrer specified in the HTTP request. The token and timestamp are the first two parameters, and they are CSRF countermeasures.

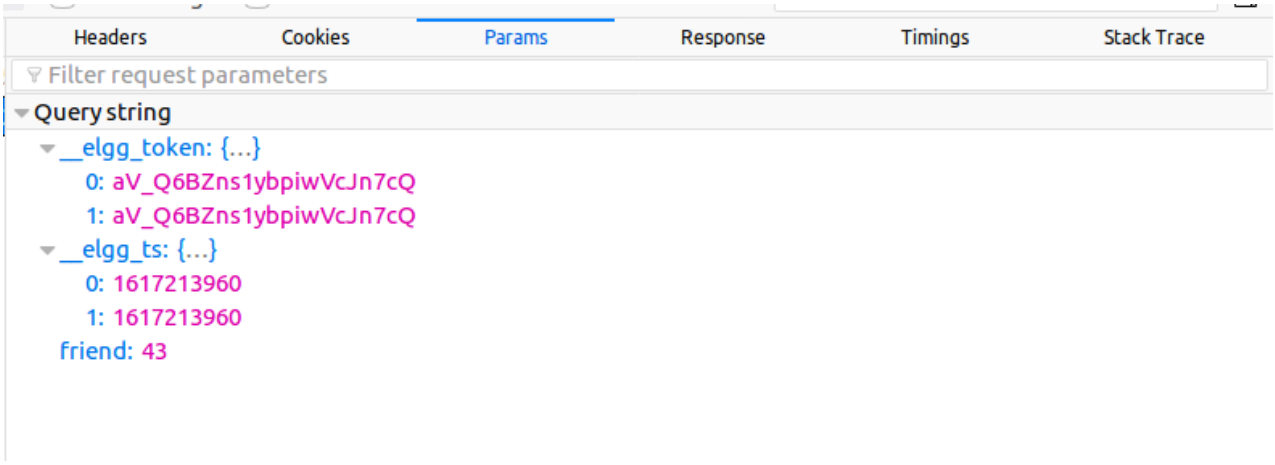
Next, one of the differences noticed between the two requests type. The params are included in the URL string in the GET request, but they are included in the request body in the POST request, resulting in the content-type and length fields in the header. A Referer field is also present, which specifies the request's source website. This field will tell the server whether the request is cross-site or same-site, and therefore can be used as a CSRF countermeasure; however, not all browsers support it, and it could be violating people's privacy.

Task 2: CSRF Attack using GET Request

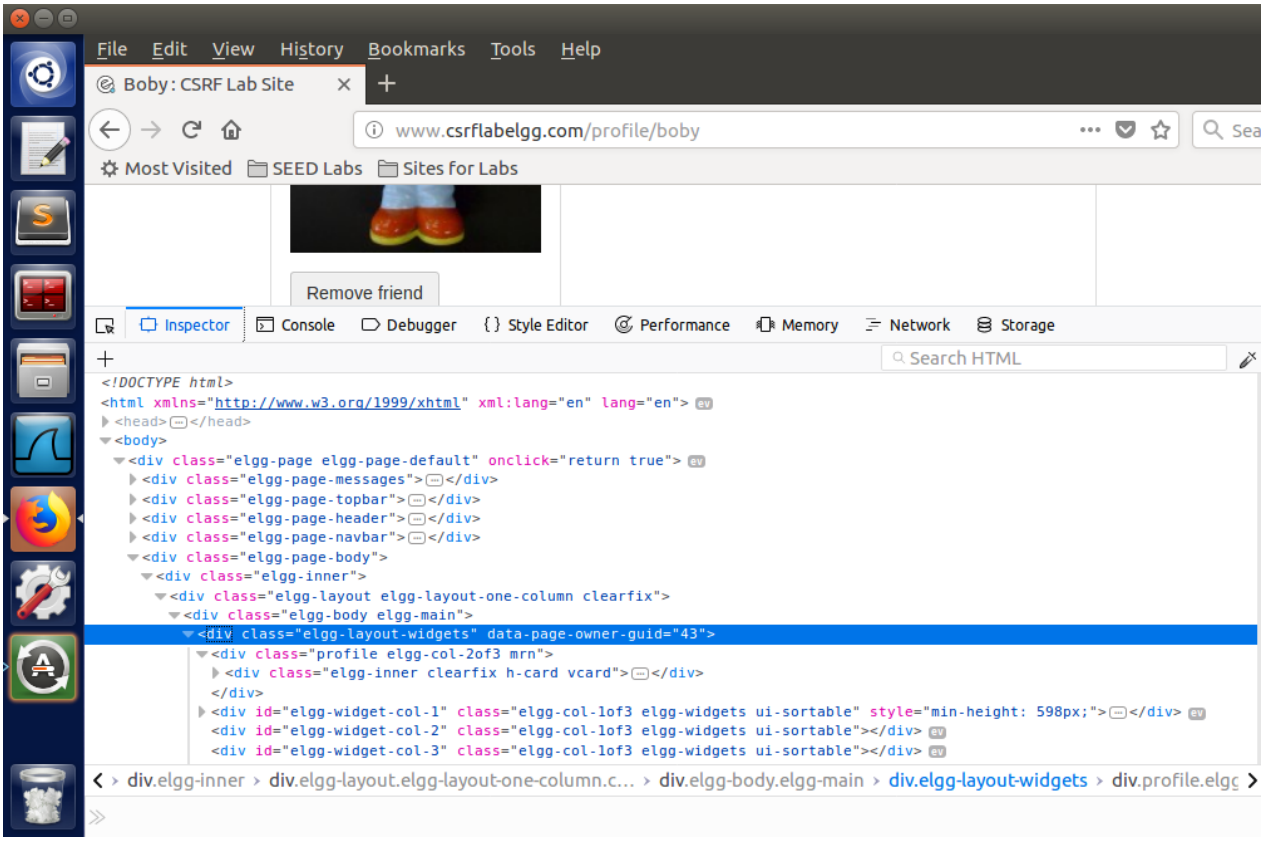
We(Boby) need to figure out how the ‘add friend' request works in order to build a request that will add Boby as a friend in Alice's account. So, let us pretend we have made a fake account called Charlie, and we first log in to Charlie's account to add Boby as Charlie's friend and look at the request conditions for adding a friend. We look for Boby and press the add friend button after logging into Charlie's account. We look for the HTTP request in the web developer tools while doing this.



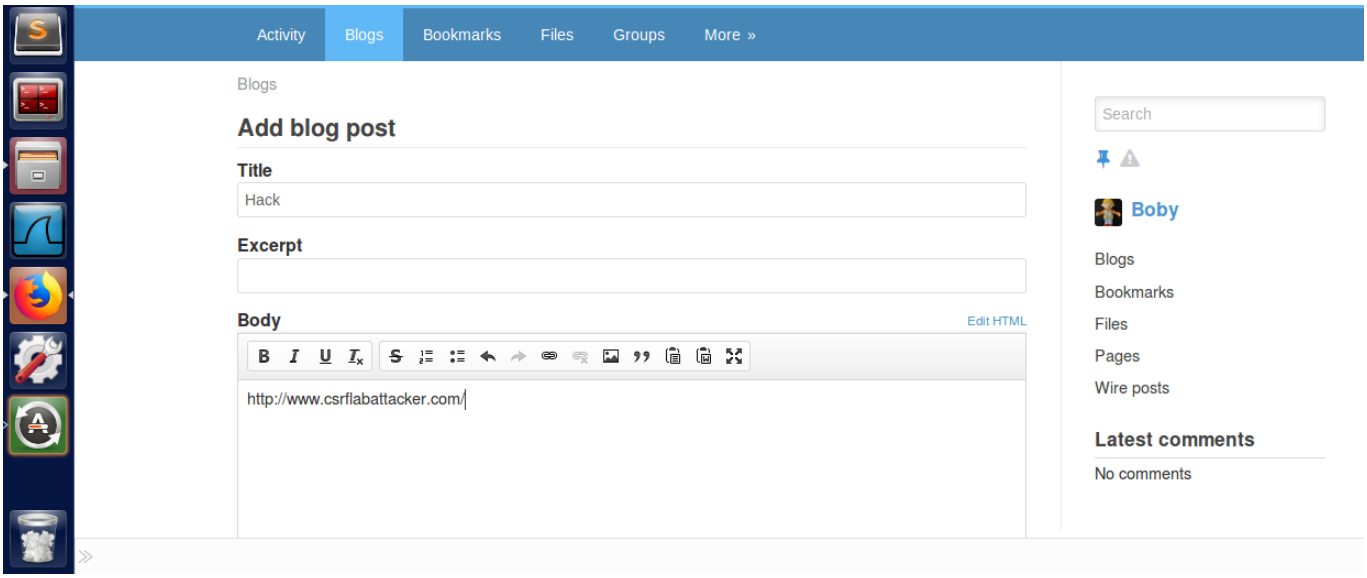
We can see the friend has a value of 43 because it is a GET request. Other parameters in the request can be seen in the Params tab.



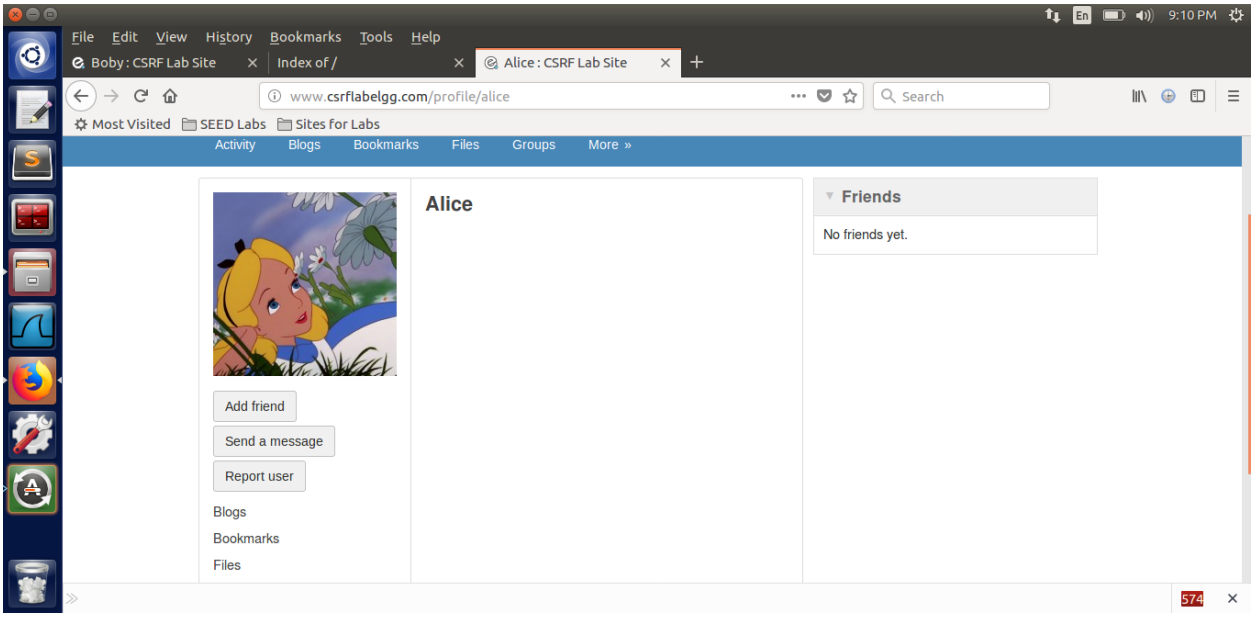
We know when Charlie tried to add Boby as a friend, a request with the friend value of 43 was sent, indicating that it was Boby. To double-check this, we can use the inspect element function to search for the website's source code.



Boby composes a blog post in which the malicious url is embedded in the body. Boby is added to Alice's friends list when she clicks on this url.

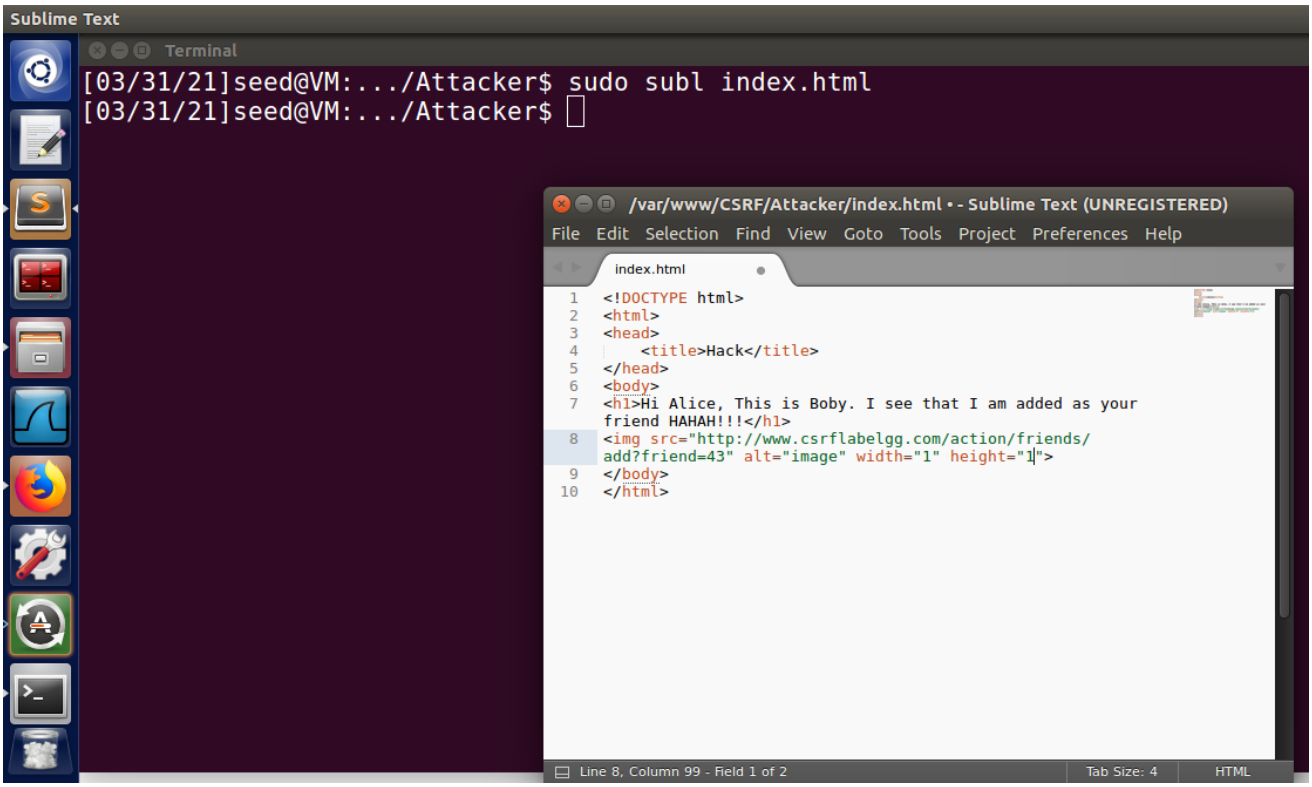


Alice currently has no friends.



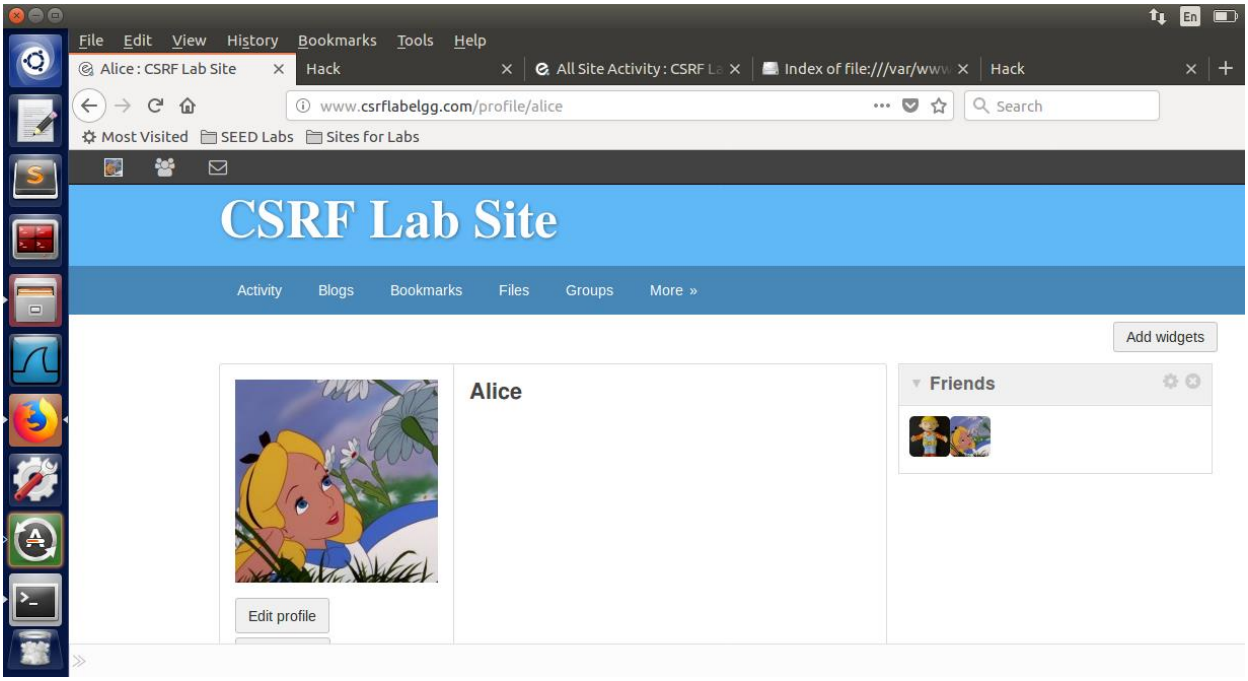
We use the img tag of HTML pages to create a GET request, which sends a GET request as soon as the web page loads in order to view the image. We set the image's width and height to 1 in order for it to be very small and invisible to Alice. This aids in concealing the purpose of the web page in this scenario, adding a friend.

The malicious url used by Boby to target Alice can be seen in the index.html code below.

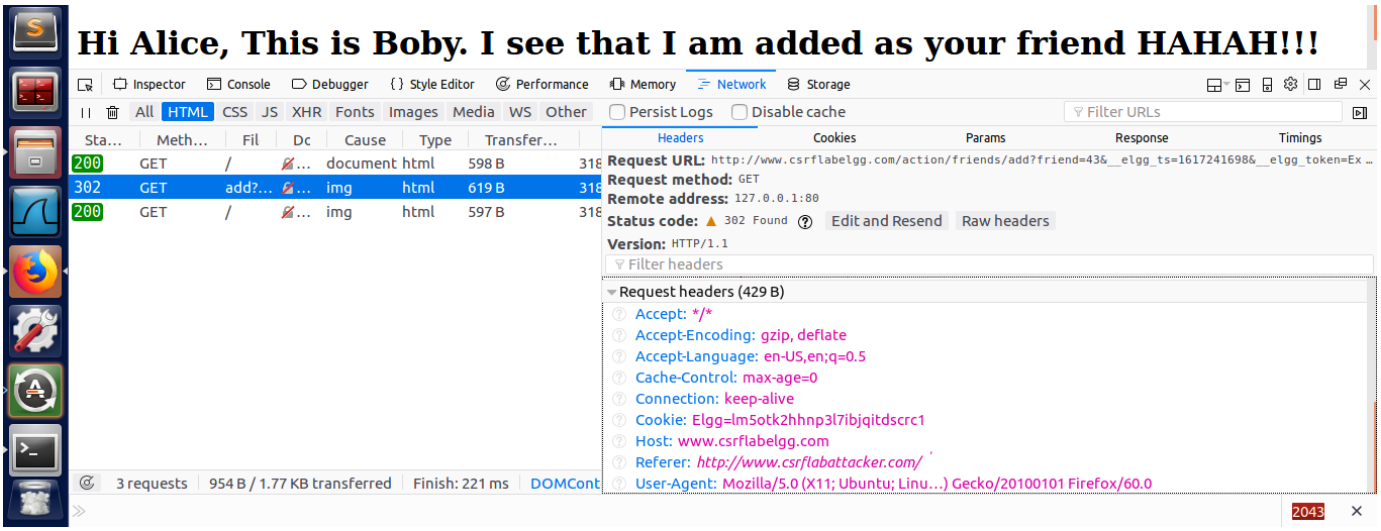


```
[03/31/21]seed@VM:.../Attacker$ cat index.html
<!DOCTYPE html>
<html>
<head>
        <title>Hack</title>
</head>
<body>
<h1>Hi Alice, This is Boby. I see that I am added as your friend HAHAH!!!</h1>
<img src="http://www.csrflabelgg.com/action/friends/add?friend=43" alt="image" w
idth="1" height="1">
</body>
</html>[03/31/21]seed@VM:.../Attacker$
```

Now, Alice visits the malicious website by opening a new tab and clicking on the link sent to her in a message created by Boby.

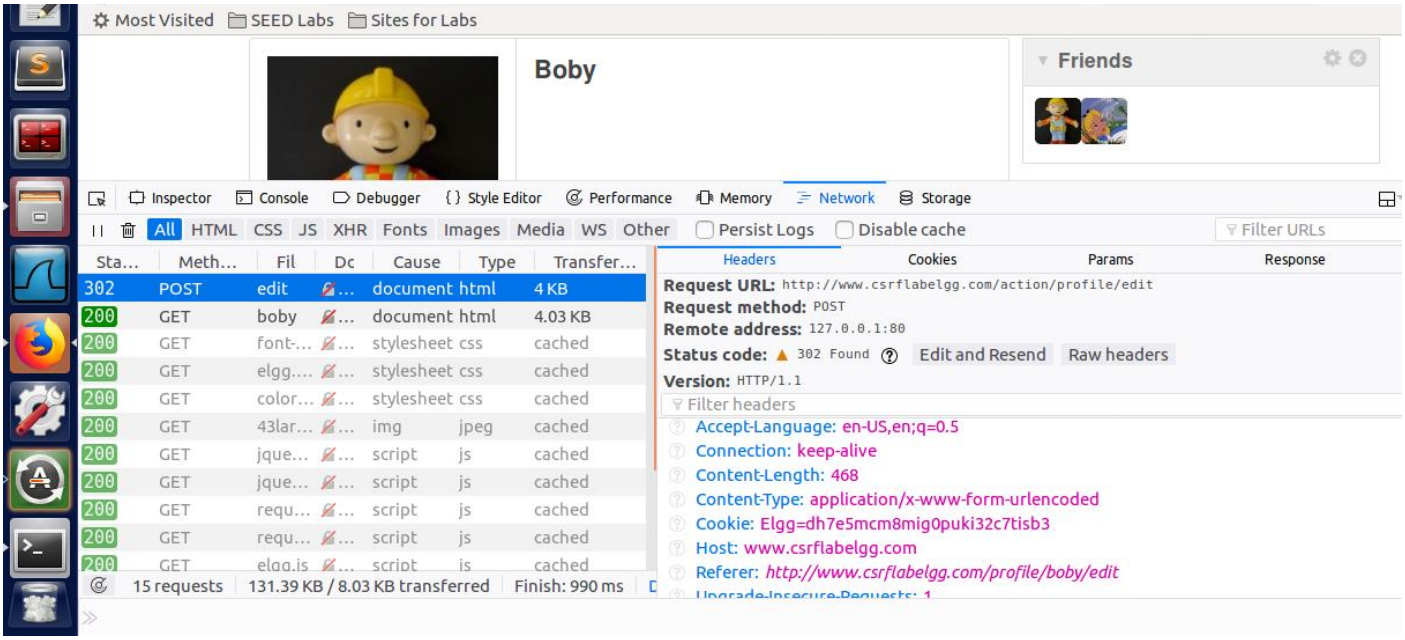


As a result, we were able to add Boby as Alice's friend without Alice's knowledge. As the malicious website loads, the following shows the content of the HTTP request. We can see that the URL we mentioned is sent in an HTTP GET request as soon as the connection is clicked, and a friend with GUID 43 is added to the current session, Alice's. The GET request is used to add Boby to Alice's friends list, which is a cross-site request forgery attack. We have a trusted site www.csrflabelgg.com, a user Alice logging into the trusted site, and a malicious website created by Boby, www.csrflabattacker.com. It appears that Alice is attempting to add Boby as a friend on the elgg website. Since the image is loaded when the page is opened, we use the img tag, but we make the image very tiny.



Task 3 :- CSRF Attack using POST Request

Next, in order to edit Alice's profile, we (Boby) must first learn how to use the website's edit profile function. To do so, we log into Boby's account and pick Edit Account from the drop-down menu. Then we click submit after making changes to the brief description area. When doing so, we use the web developer options to inspect the content of the HTTP request and find the following. We can notice from the below screenshot that it is a post request, and the content length is that of 468.



Headers | Cookies | Params | Response

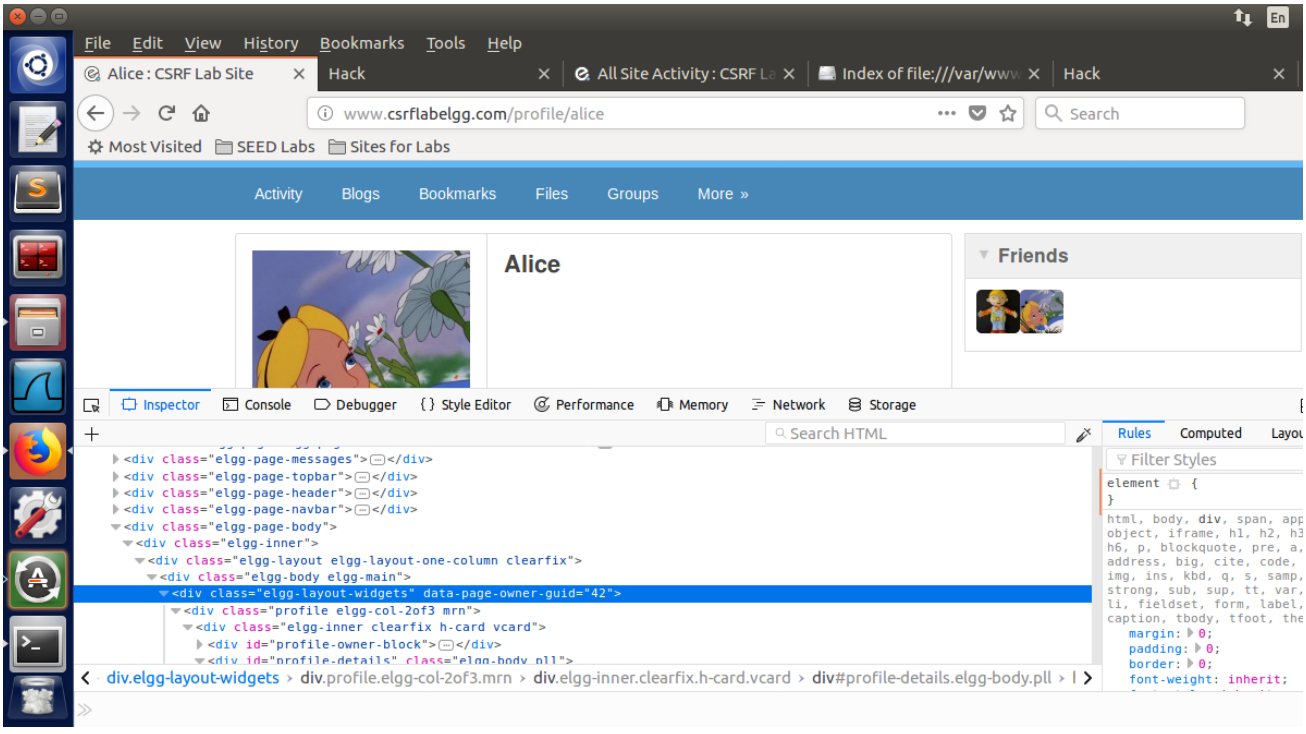
Filter request parameters

Form data

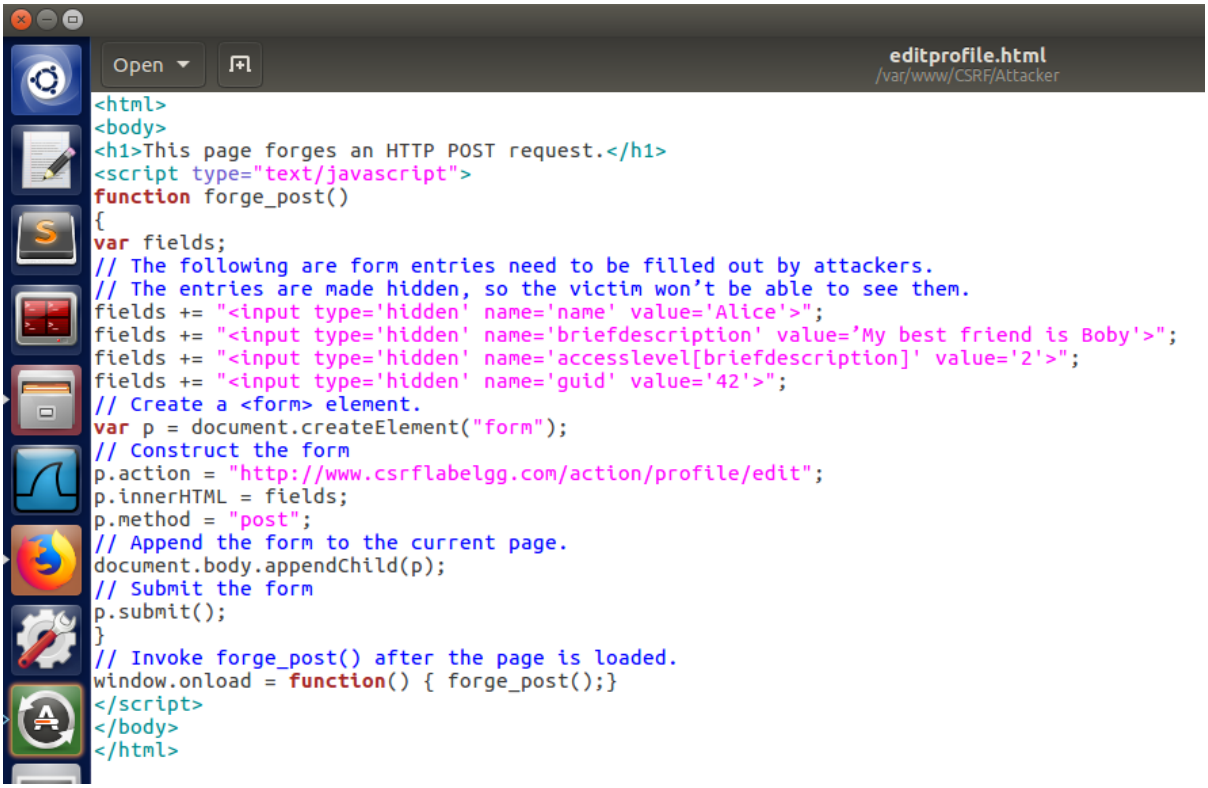
__elgg_token: 8UzSzc1XY1wODJuJRVZ3yQ
__elgg_ts: 1617333764
accesslevel[briefdescription]: 2
accesslevel[contactemail]: 2
accesslevel[description]: 2
accesslevel[interests]: 2
accesslevel[location]: 2
accesslevel[mobile]: 2
accesslevel[phone]: 2
accesslevel[skills]: 2
accesslevel[twitter]: 2
accesslevel[website]: 2
briefdescription:
contactemail:
description:
guid: 43

As previously discovered, the guid value is initially set to Boby's GUID. So, in order to edit Alice's profile, we will need her GUID, the string we want to store in the brief description parameter, and the access level for this parameter to be publicly accessible set to 2.

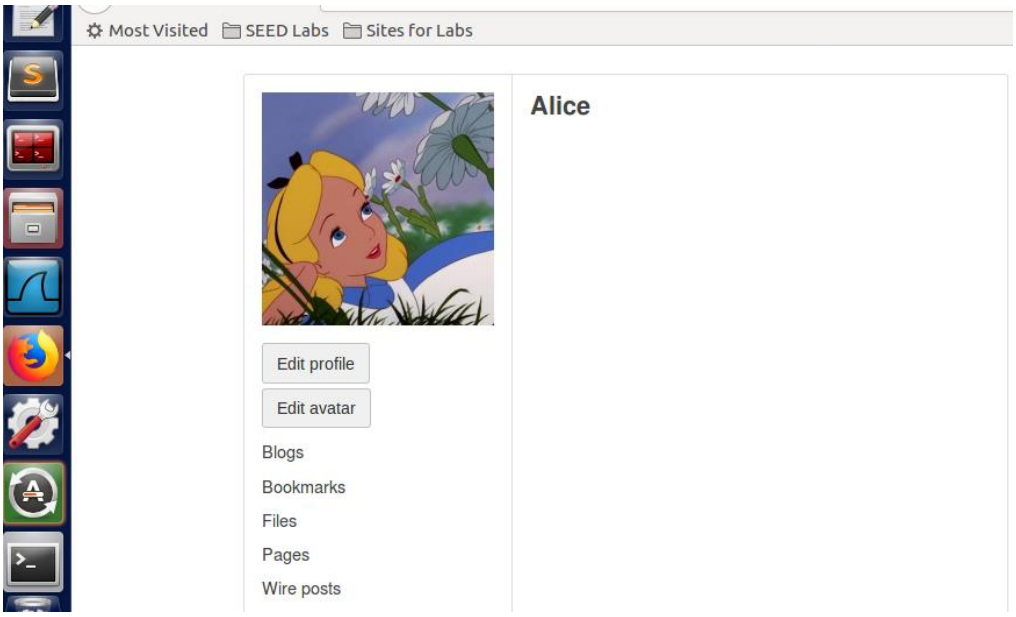
To find Alice's GUID, simply search for her profile in any user's account on the website and then use the inspect element function to look for the web page owner's guid – which will show Alice's GUID. we can notice from the below screenshot that Alice’s guid is 42.



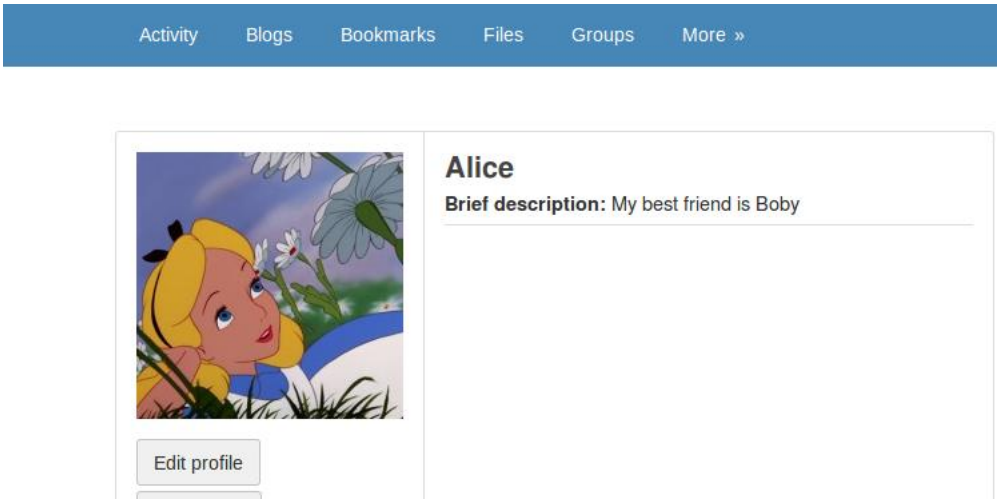
We create an editprofile.html web page in the var/www/CSRF/Attacker folder that is connected to the malicious web page using this value. The name and GUID are set to Alice's, and the Description is what we want to save, i.e. 'My best friend's name is Boby'. We also raised the description's access level to 2, enabling us to see the improvements. The POST request's URL is the one of the targeted site's edit profile tab.



Next, we log into Alice's account to show a successful attack.



Then, we think Alice clicks on the malicious website connection that was sent to her in a message.



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?

This problem can be solved the same way we found Alice's GUID: by searching Alice on the website and then inspecting the web page owner's GUID with an inspect feature. This does not necessitate having Alice's credentials. If the website's source code does not contain any GUIDs, and therefore we cannot use the first method, we can try using Alice's full name as username and a random password and inspecting the HTTP Request or Answer. We could also use Alice's GUID if any of those had it.

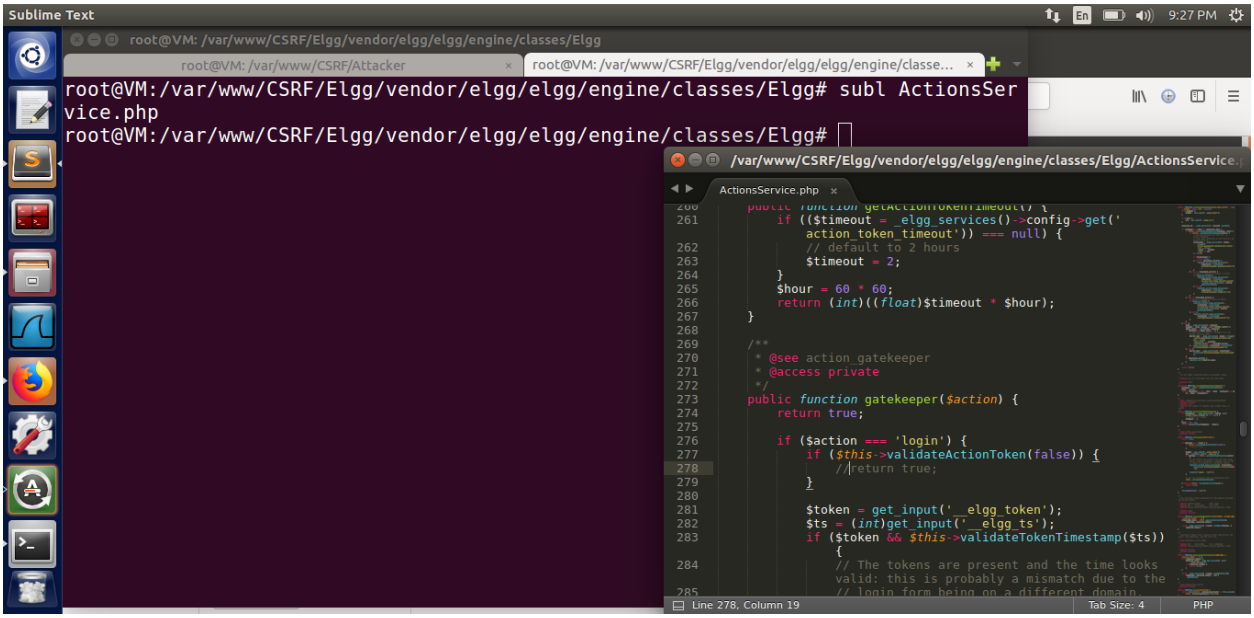
Question 2: If Boby would like to launch the attack to anybody who visits his malicious web page. In this case, he does not know who is visiting the web page beforehand. Can he still launch the CSRF attack to modify the victim's Elgg profile?

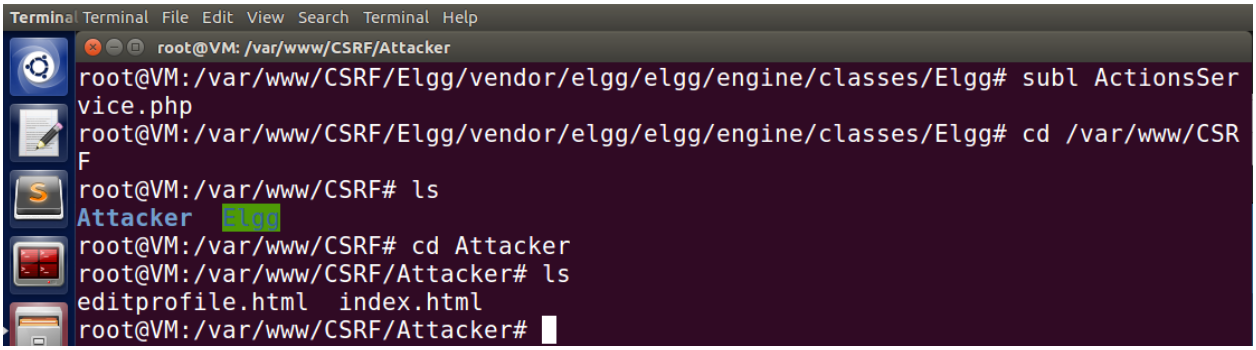
Boby will be unable to carry out the CSRF attack in this case because his malicious web page varies from that of the targeted website. We do not have access to the source code of the targeted website in that situation, so we cannot derive the GUID like we might before. Furthermore, since the GUID is sent only to the server of the targeted website and not to any other website, we will not be able to obtain it from the HTTP request from elgg to attacker's website.

Task 4 :- Implementing a countermeasure for Elgg

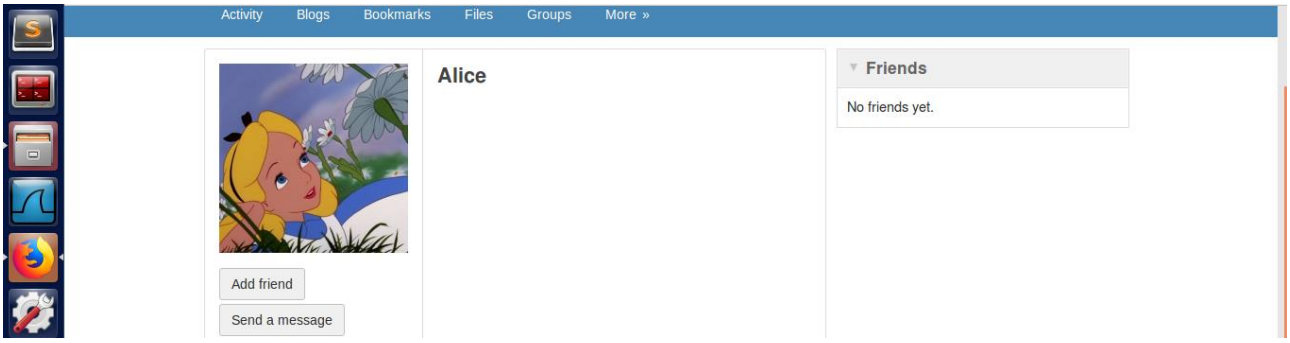
By commenting out the return True argument, the CSRF countermeasure is now allowed. The function always returned true as a result of this argument, even if the token did not fit. So, by

commenting it out, we are checking token and timestamp and returning true only if they are the same. The action will be refused, and the user will be redirected if the tokens are missing or invalid.



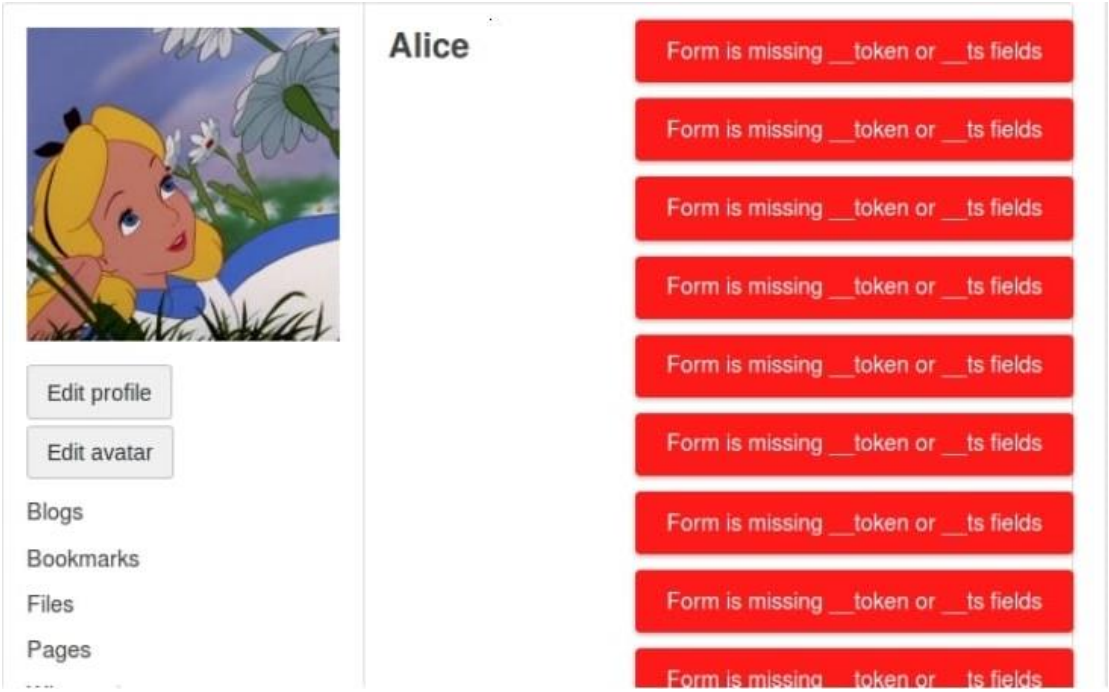


To have Alice's account as follows, I delete the effects of the previous attacks.



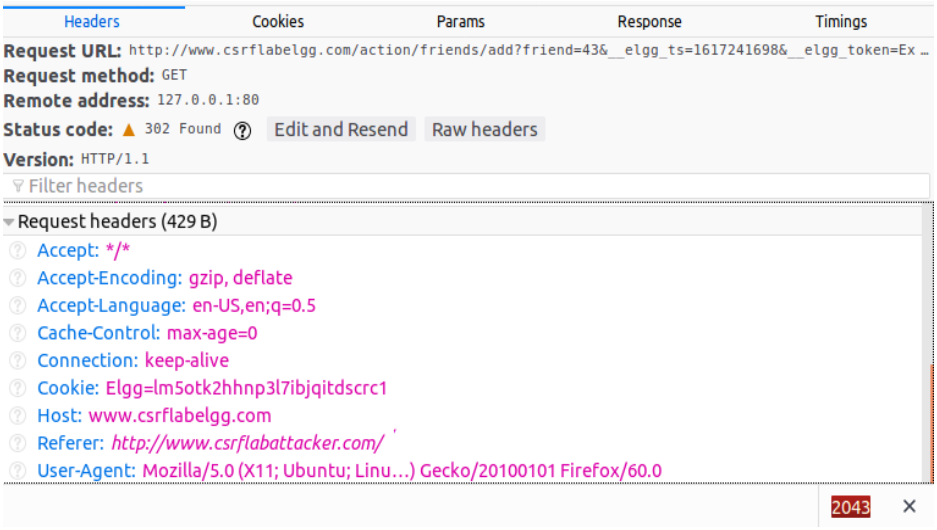
Elgg security token consists of a hash value (md5 message digest) of the site secret value, timestamp, user session ID, and a randomly generated session string. Then we replay the attacks, finding the following when executing a GET request attack.

On doing the POST request attack we can notice the following(Refer the below attached screenshot for the POST request)

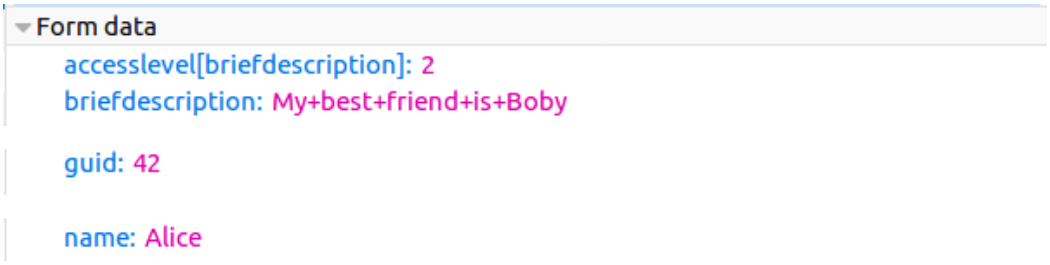


During both times, we notice a mistake, and our attack fails. The action was not completed successfully because the token and timestamp fields were lacking. We can see that no token or ts fields are sent in the HTTP headers for both GET and POST requests. This is due to the fact that we have not defined any timestamp or secret token parameters when creating the HTTP request.

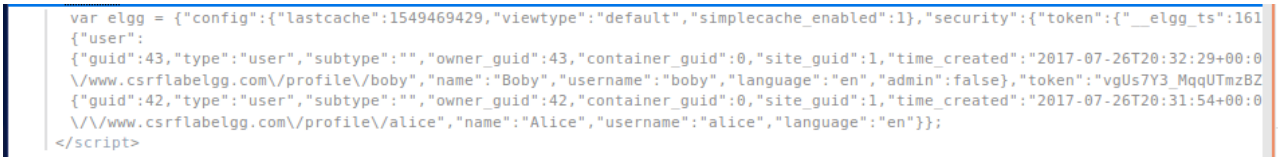
This can be noticed in the URL for the addfriend GET request, where the only parameter is friend (see the below attached screenshot).



We see all of the parameters we set in the POST request to edit the profile, but not the ts and token.



Using the inspect function, we can see the elgg ts and elgg token in the source code of the elgg website's web page



However, since the request is sent from the attacker's website to the elgg server, rather than from the server to the attacker's website, these are not included in the HTTP request. Since these tokens are set on the elgg website, these parameters will only be present in requests made from the same website. Because of the same origin policy, no other web page can access the contents of the elgg's web page, so they cannot use this token in forged requests.

When we log in to Alice's account, we will see these hidden tokens, but no other user on the network has Alice's credentials, so they will not be able to see these values. Also, even though finding the timestamp value is easy, we need two values to pass the test: the timestamp and the secret token, which is a hash value of the site secret value – which is retrieved from the database, as well as the timestamp, user session ID, and randomly created session string. Even if we knew the timestamp and user sessionID from previous activities, it is impossible to obtain the site's

secret value, which is stored in its own secret database and is a string generated at random. As a result, the attacker will be unable to guess or discover the hidden tokens – all of which require valid credentials – and the attack will fail.