DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Session: Jan 2021 – May 2021

INFORMATION SECURITY
LAB – 7

NAME : AMRUTHA BS

Lab Tasks

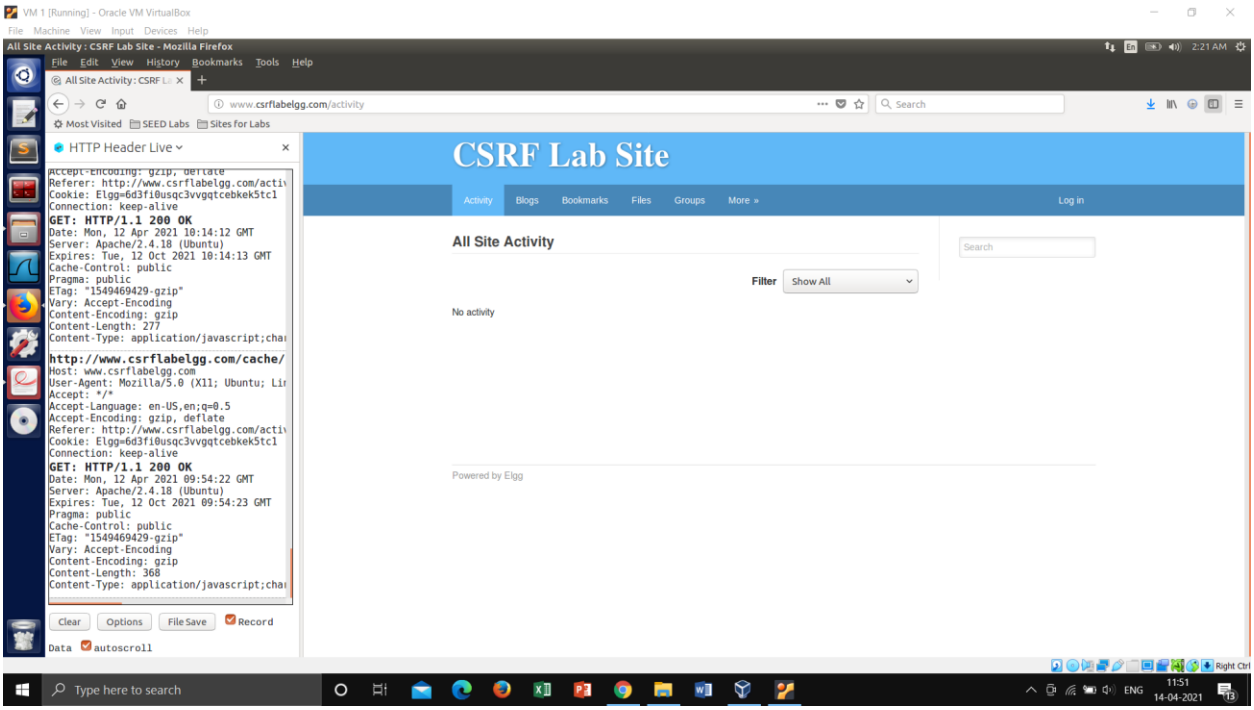
For the lab tasks, you will use two web sites that are locally setup in the virtual machine. The first web site is the vulnerable Elgg site accessible at www.csrflabelgg.com inside the virtual machine. The second web site is the attacker's malicious web site that is used for attacking Elgg. This web site is accessible via www.csrflabattacker.com inside the virtual machine.

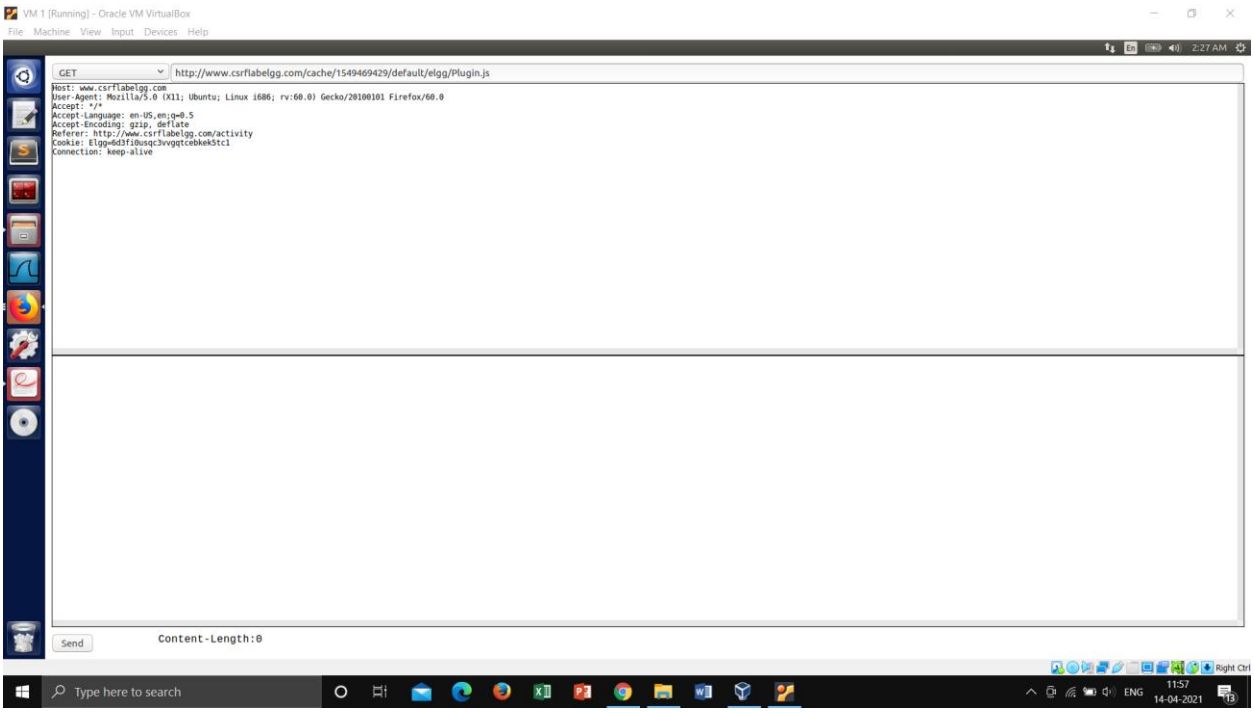
Task 1: Observing HTTP Request

In Cross-Site Request Forget attacks, we need to forge HTTP requests. Therefore, we need to know what a legitimate HTTP request looks like and what parameters it uses, etc. We can use a Firefox add-on called "HTTP Header Live" for this purpose.

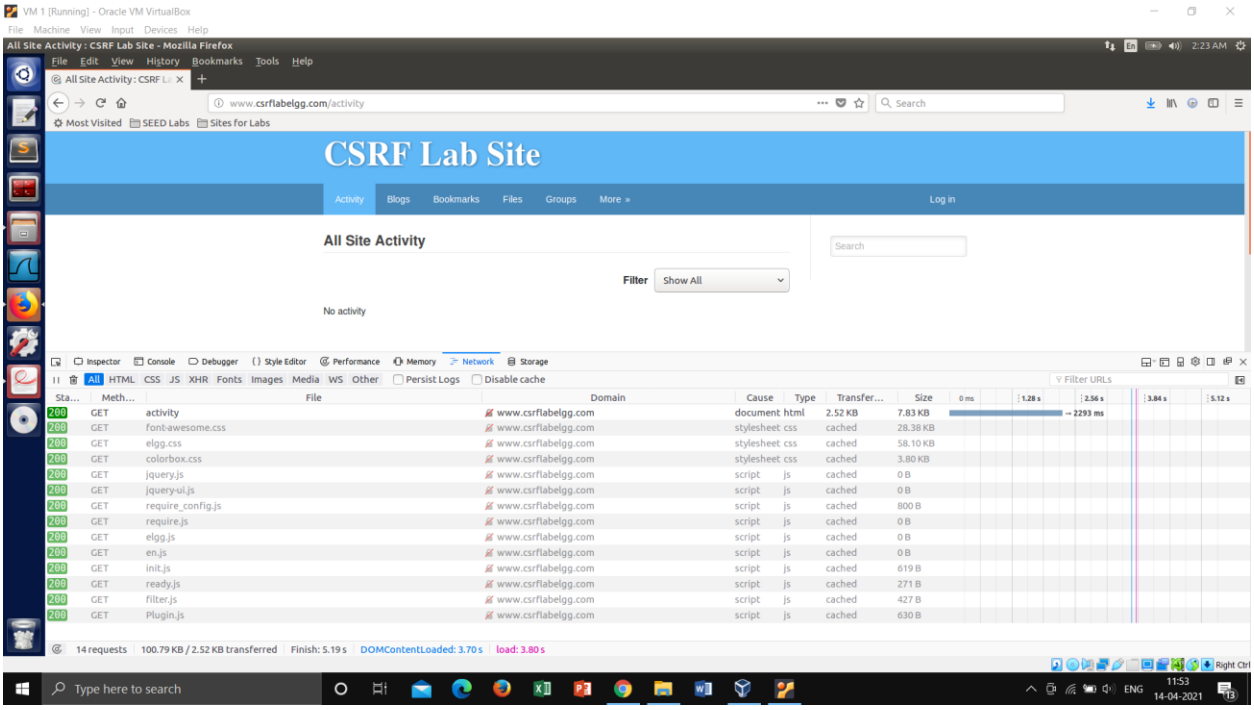
Observing HTTP GET Request:

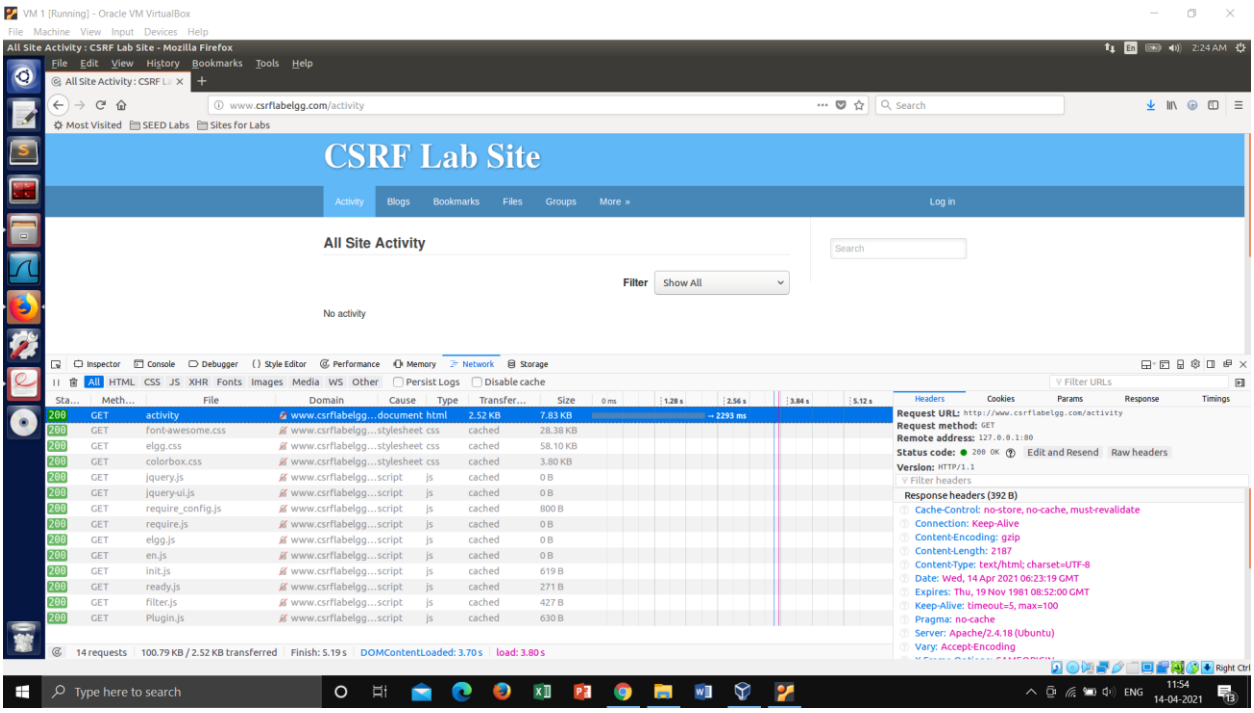
On clicking the activity tab a HTTP GET request is generated





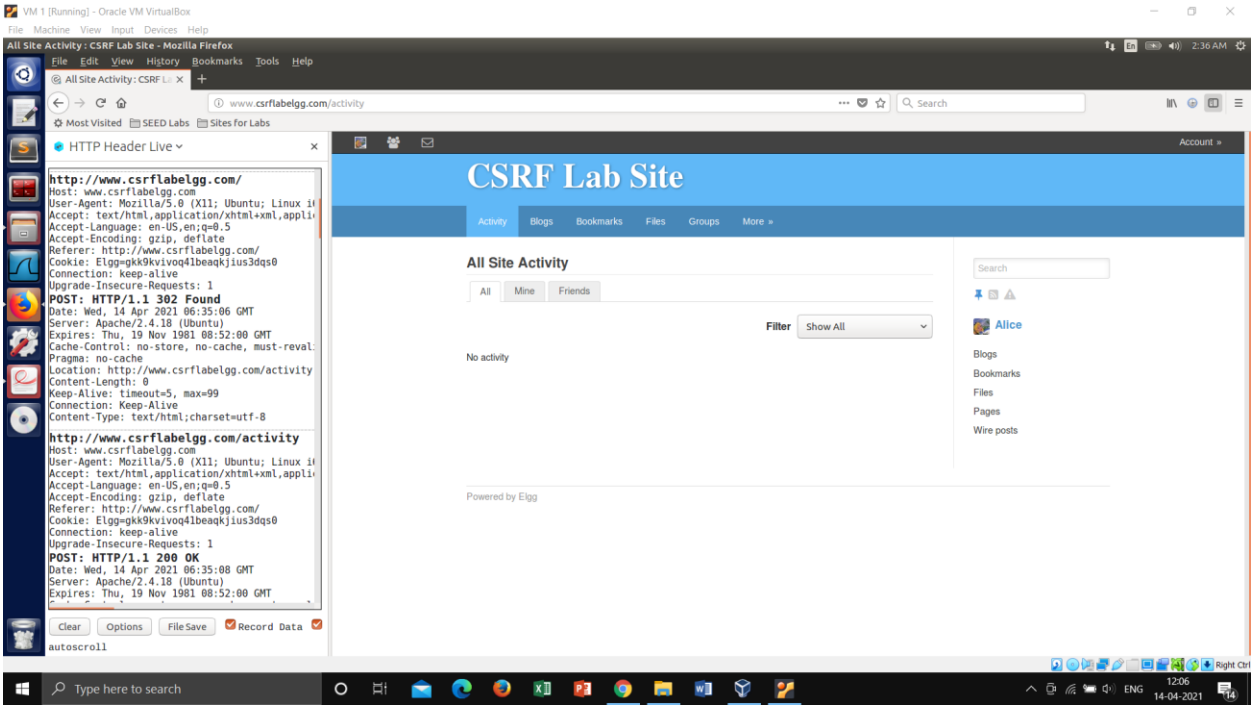
We can see the GET Request in the web developer tool:



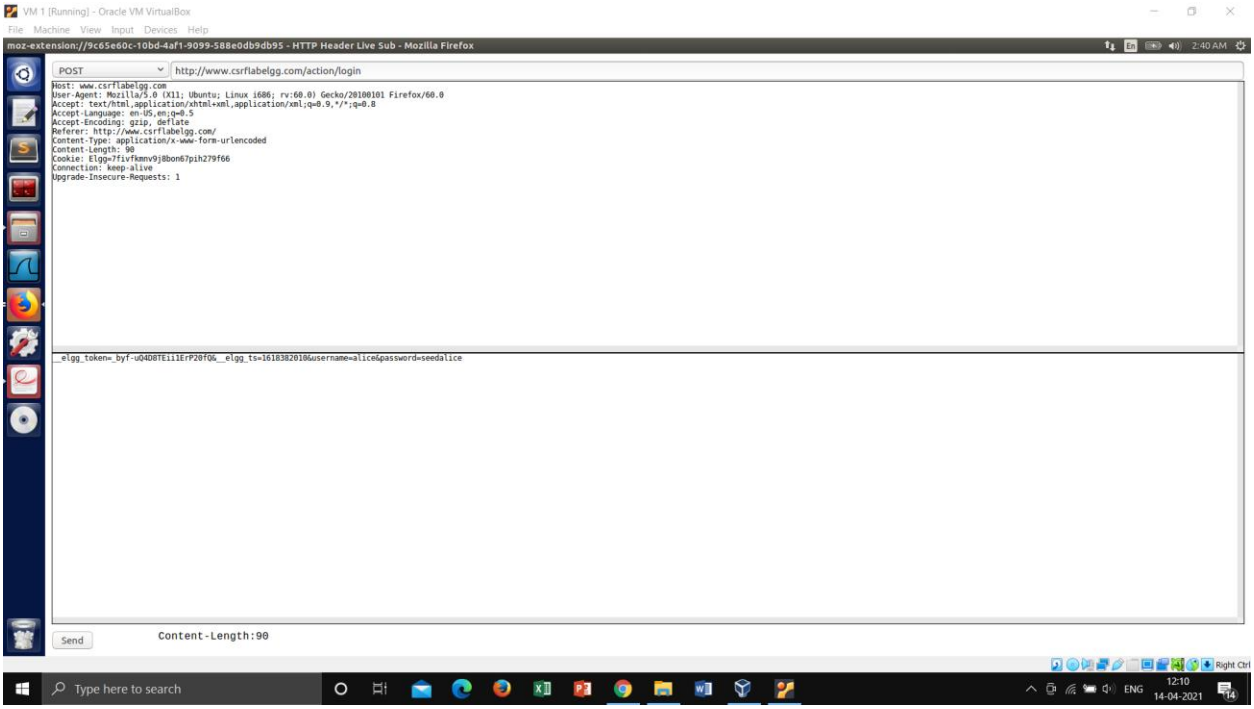


Observing HTTP POST Request:

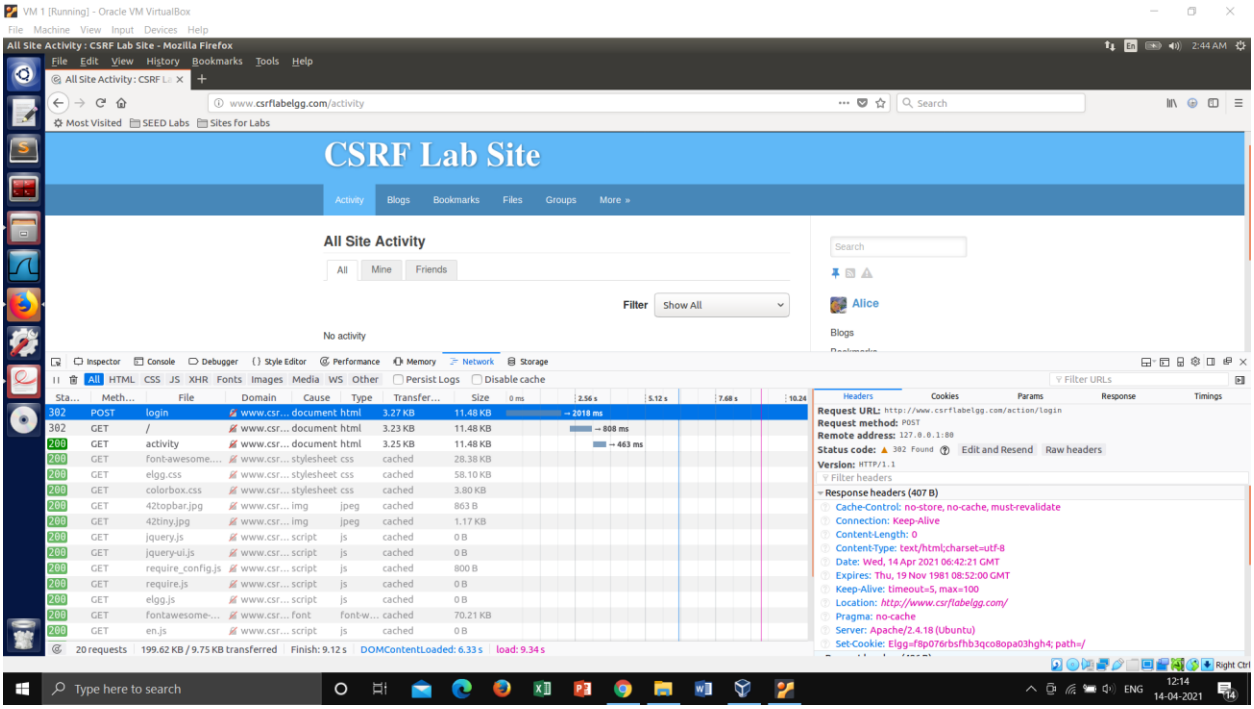
Since form will send a HTTP POST Request, we try to login as Boby:
Below screenshot shows the HTTP Live Header for the same



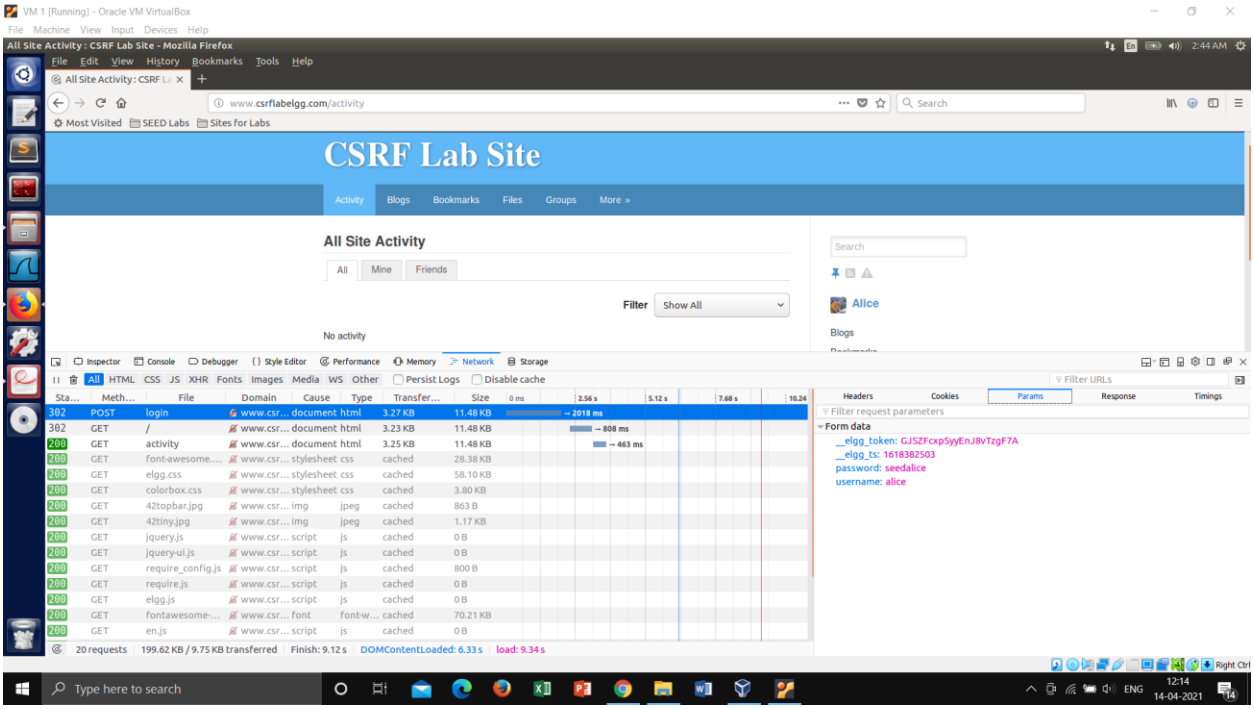
Here in the additional section we can see the username, password and the elgg token



Observing from the web developers tools:



Here in the params tab, we can find the username, password, elgg token and timestamp



Hence, we notice that the parameters in the GET request are included in the URL whereas in POST request it will be included in the request body.

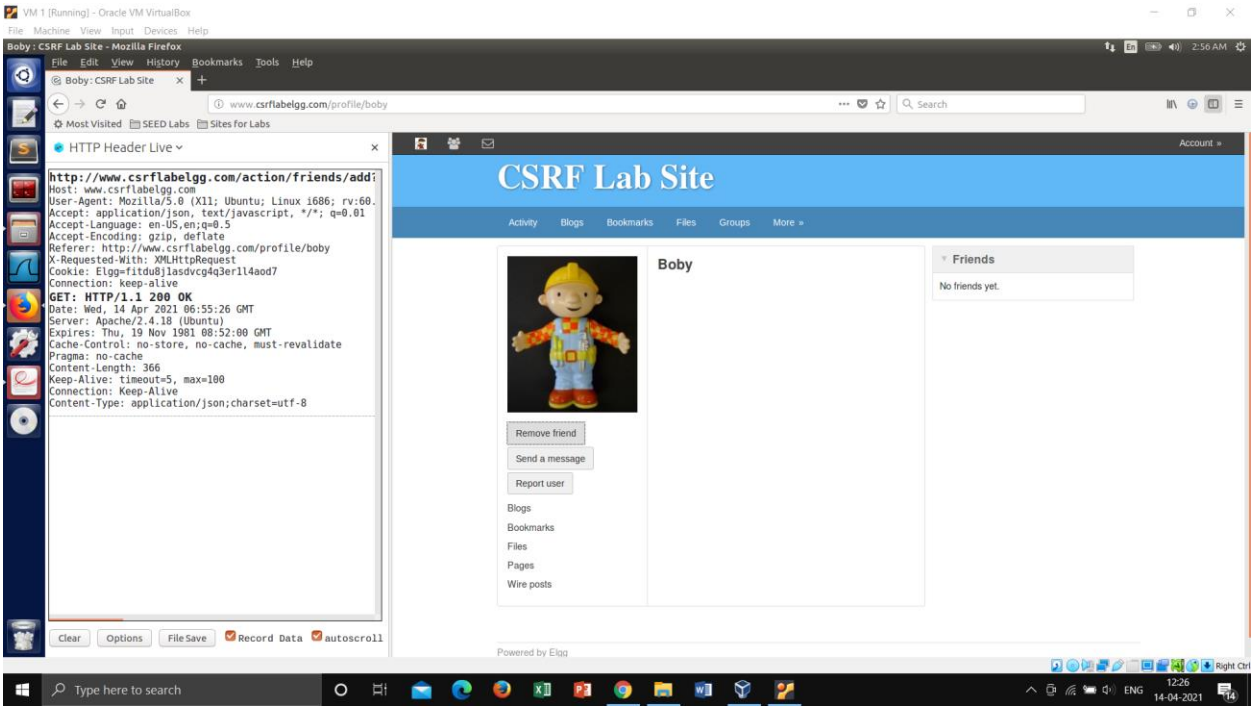
Task 2: CSRF Attack using GET Request:

In order to create a request that will add boby as a friend in Alice's account, we (Boby) need to find the way 'add friend' request works. So, we assume that we have created a fake account named Charlie and we first log in into Charlie's account so that we can add Boby as Charlie's friend and see the request parameters that are used to add a friend. After logging into Charlie's account, we search for Boby and click on the add friend button. While doing this, we look for the HTTP request in the web developer tools and see:

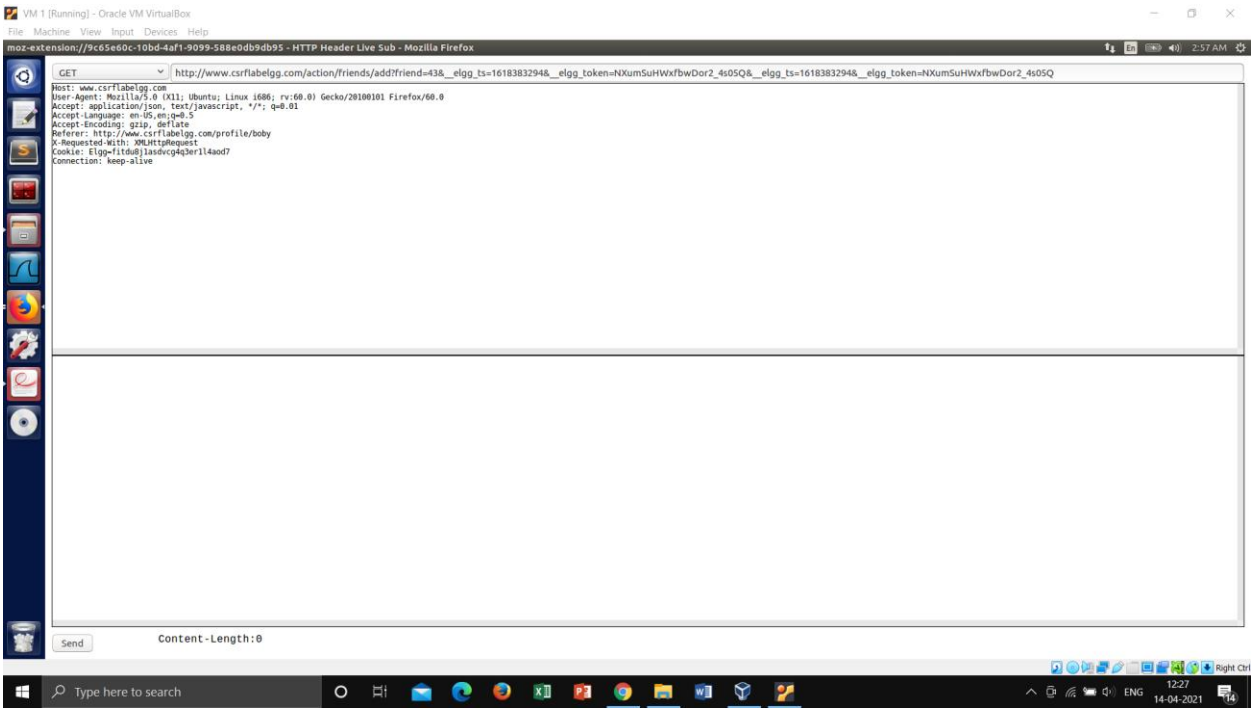
To add a friend to the victim, we need to identify what the legitimate Add-Friend HTTP request (a GET request) looks like. We can use the "HTTP Header Live" Tool to do the investigation.

1. To add Boby to Alice's friend list, we should observe the HTTP request on Add Friend button click. We use HTTP header live to view the HTTP traffic sent from the browser to the Elgg server

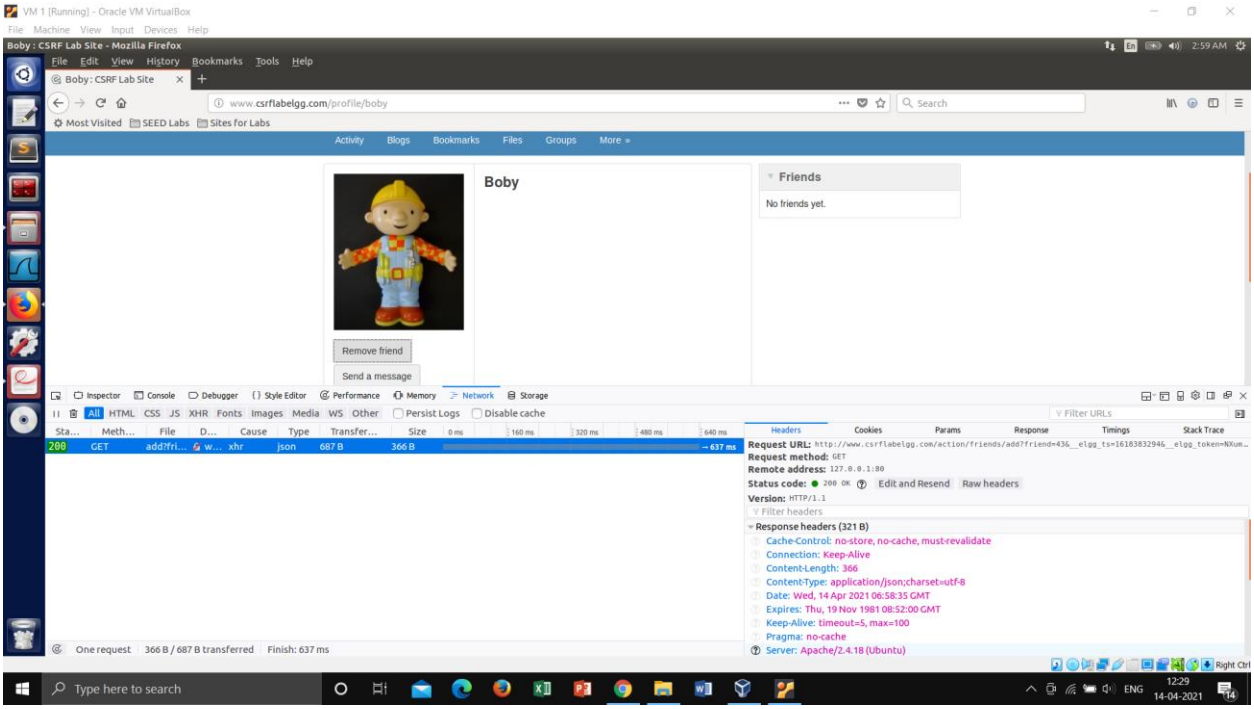
HTTP Header Live:

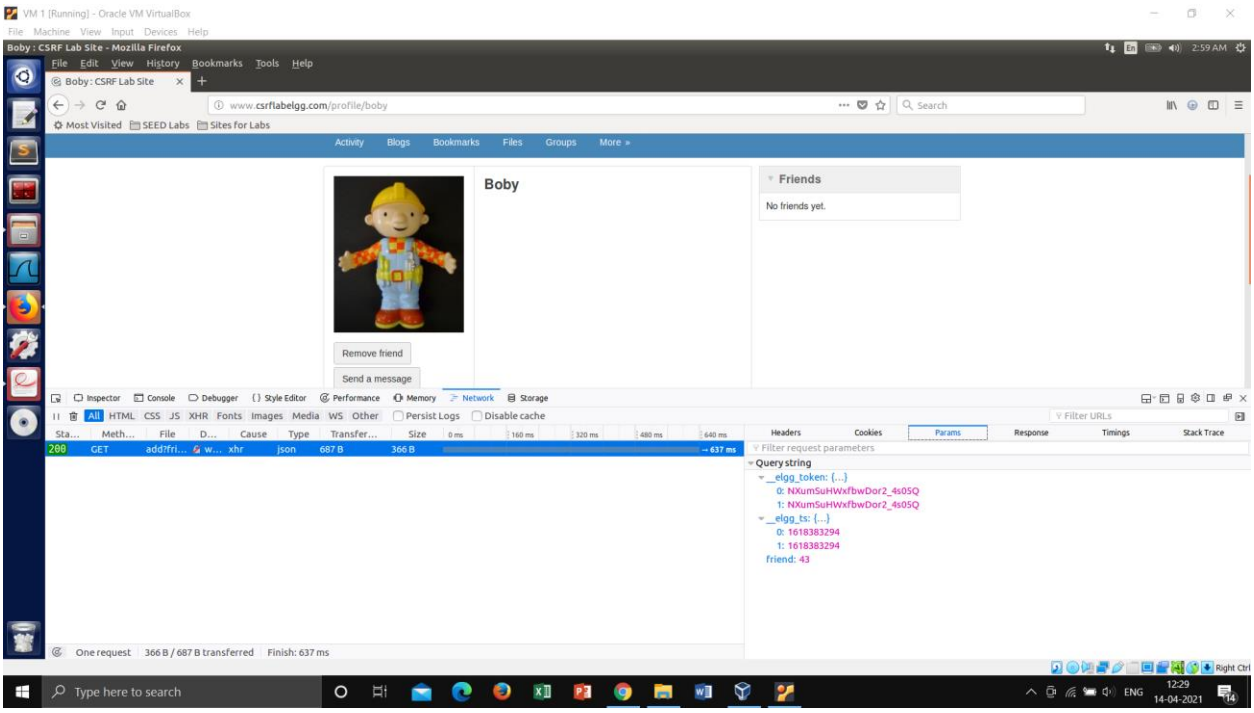


Here we can see that guid value is 43

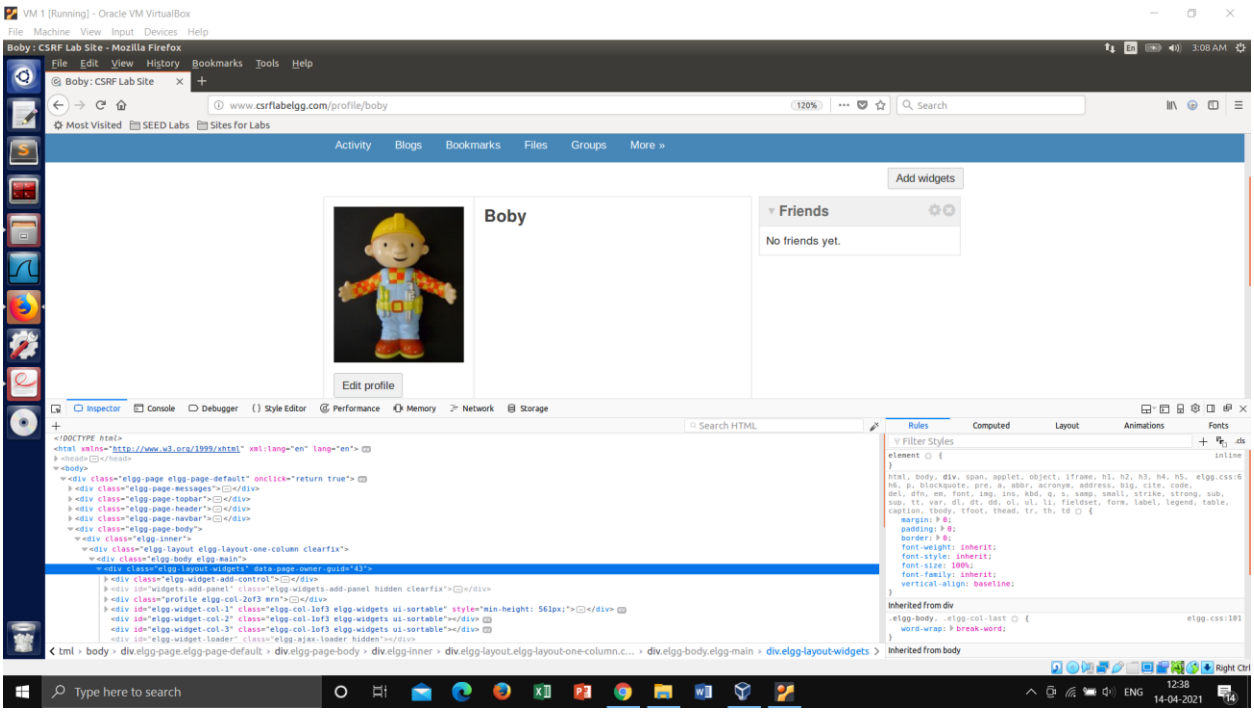


We can observe the parameters from the web developer's section.
43 is the guid value of friend



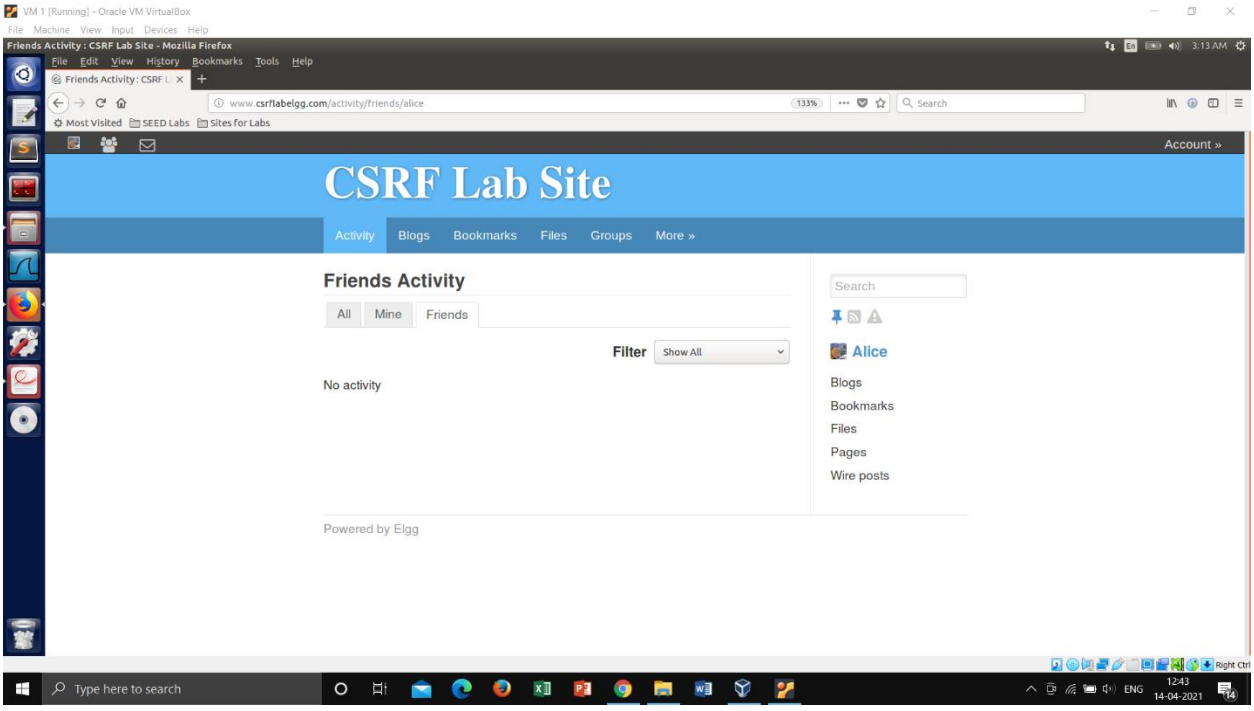


2. In the friend request which Boby sent to Alice to his friends list the parameters friend=Alice. Now to launch CSRF attack to add Boby to Alice friend list , we need to know Boby's Guid. We can get to know Boby's Guid by observing the web page source. In the Elgg web page source there is
Here we can see that guid value for Boby is 43

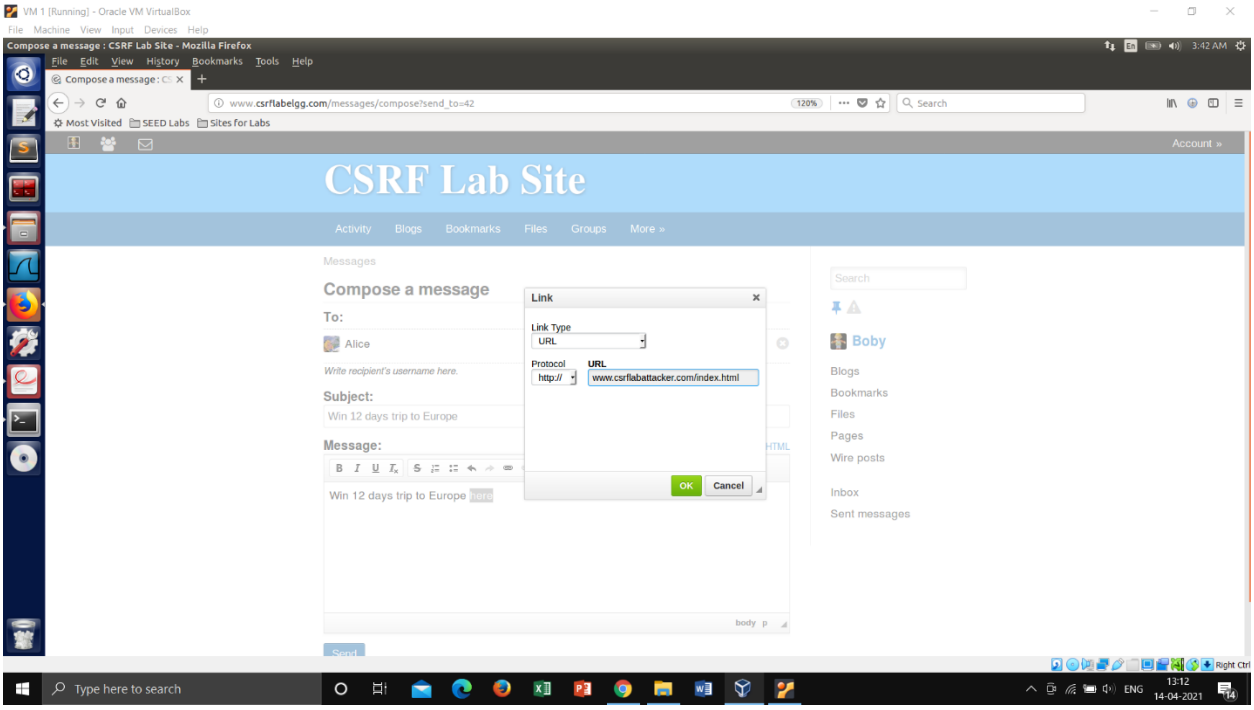


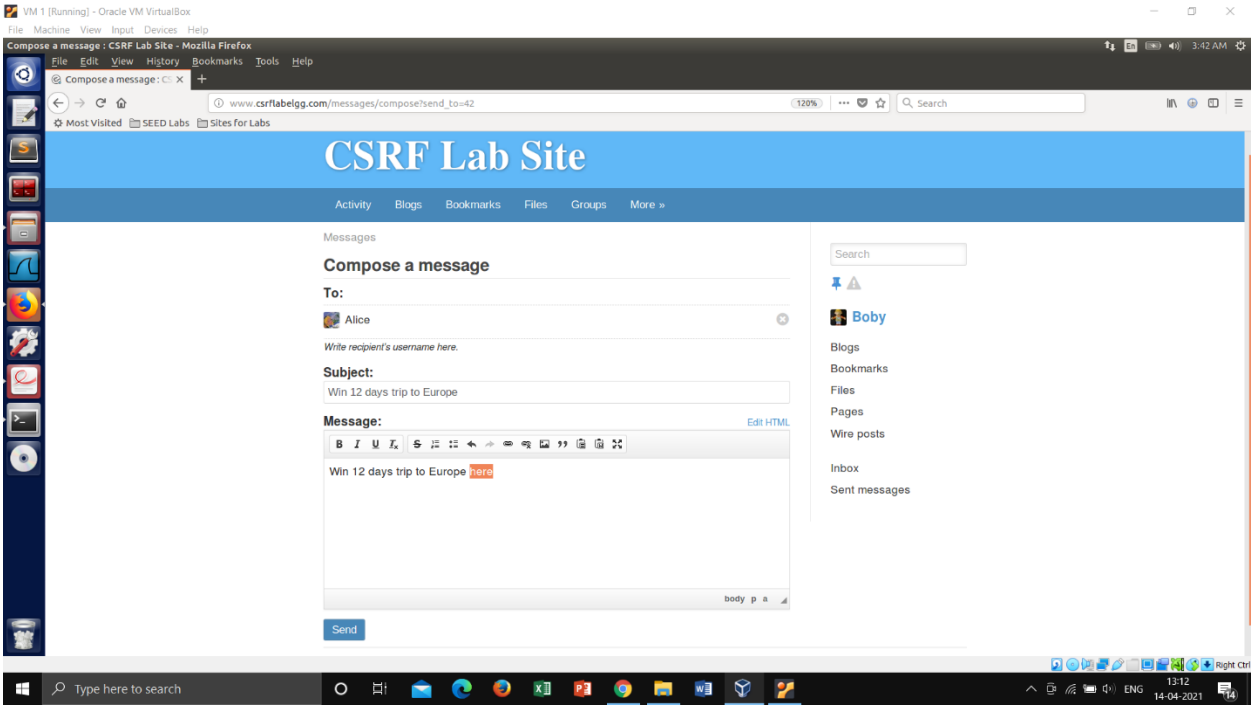
3. We can observe the friends list of Alice

Before Alice clicks on the malicious link sent by Boby she has no friends in her friends list



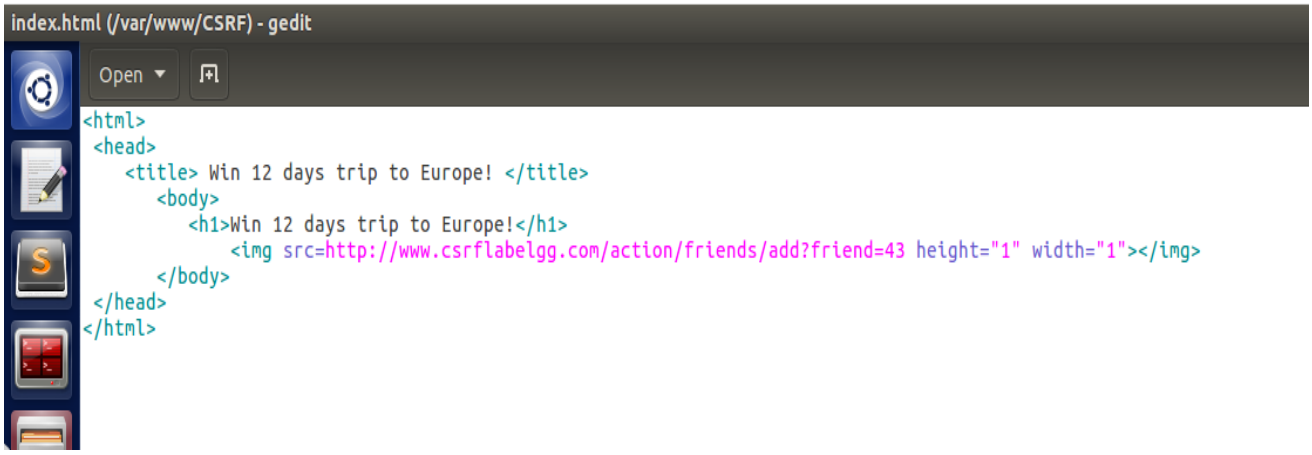
4. Now we frame the HTTP request to add Boby as a friend



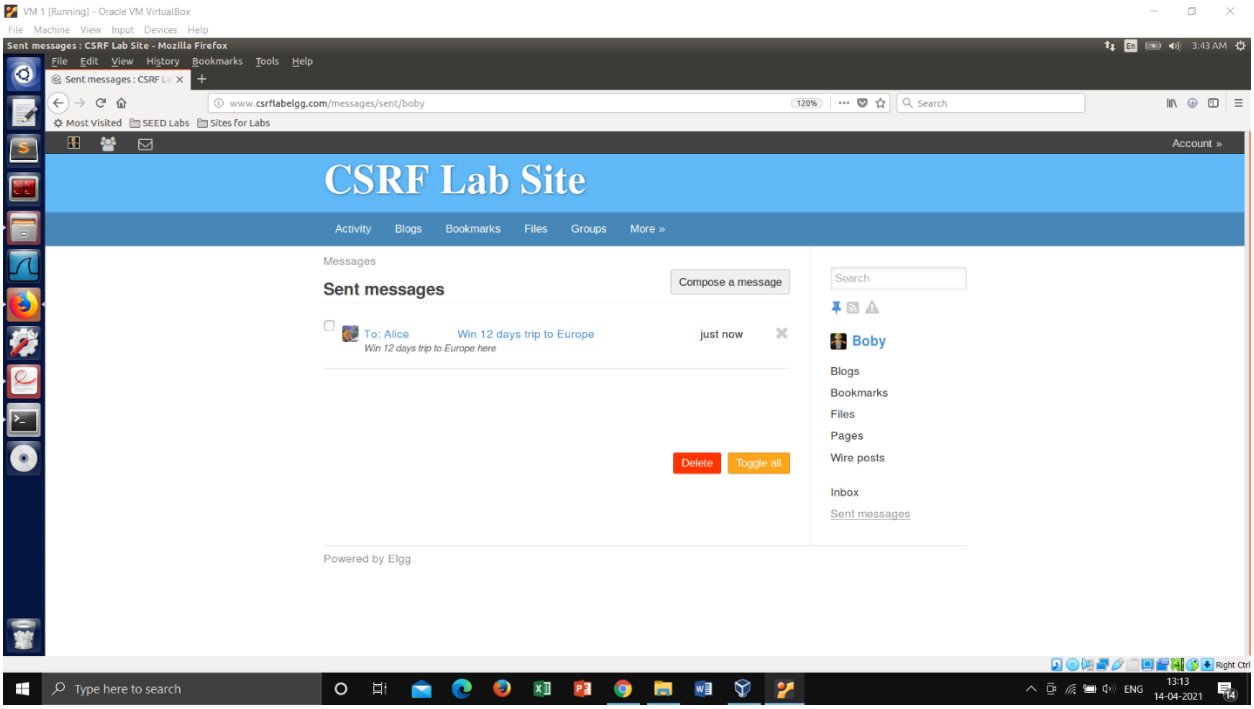


5. Now we prepare the attractive malicious web page in /var/www/CSRF/attacker/index.html which contains the above add friend request and hosts it on the URL www.csrflabattacker.com.

In order to generate a GET request we use the img tag of HTML page, 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 as 1 so that its very small and not visible to Alice. This file is created in var/www/CSRF folder

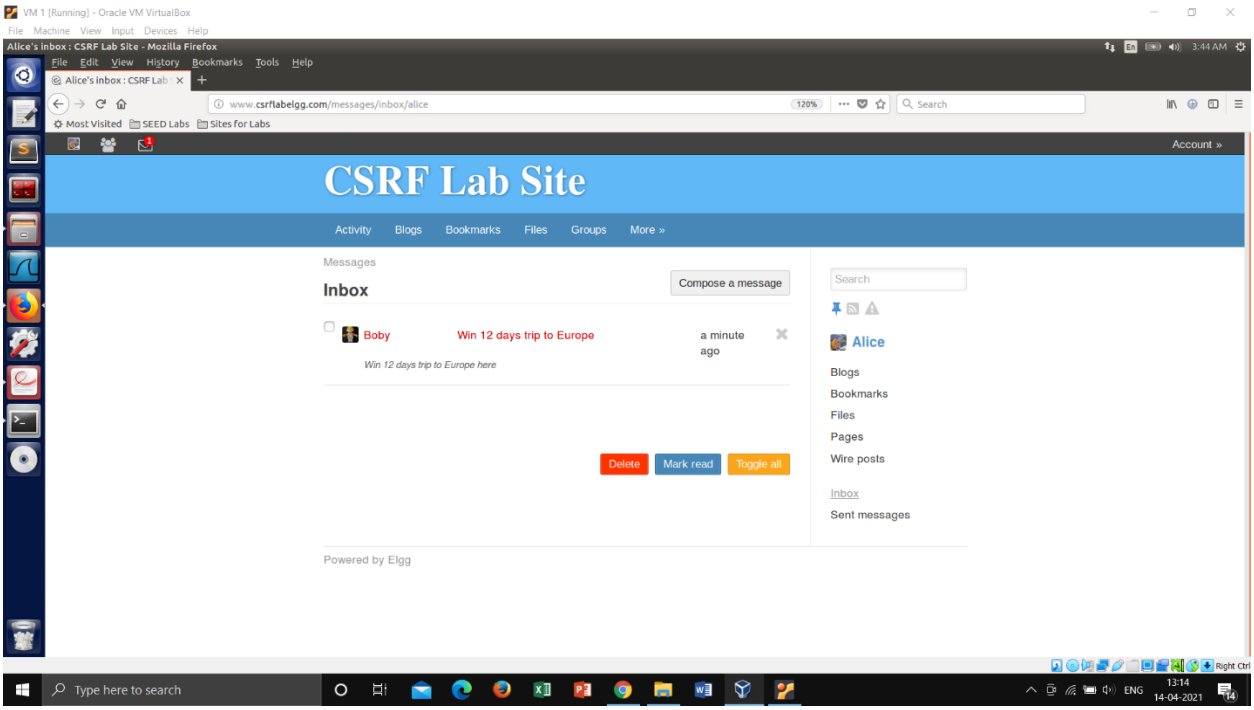


Boby's sent messages

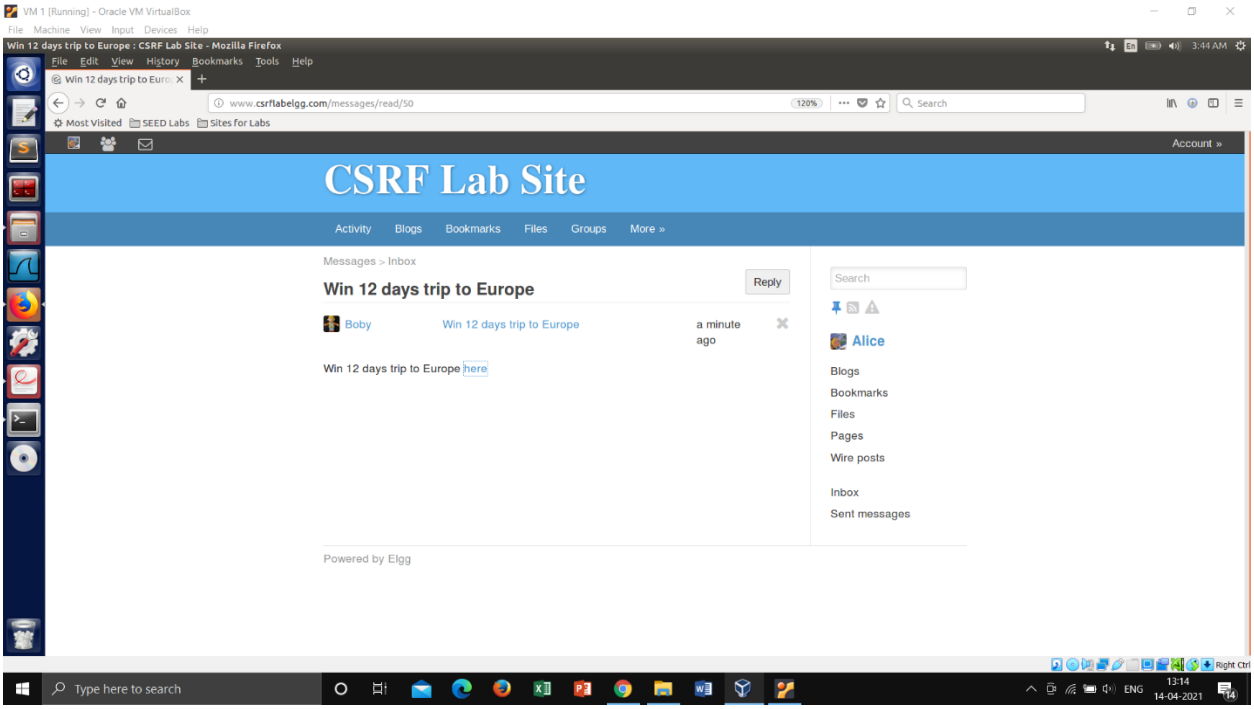


6. Alice opens and reads the message. She finds the content to be attractive and visits the URL

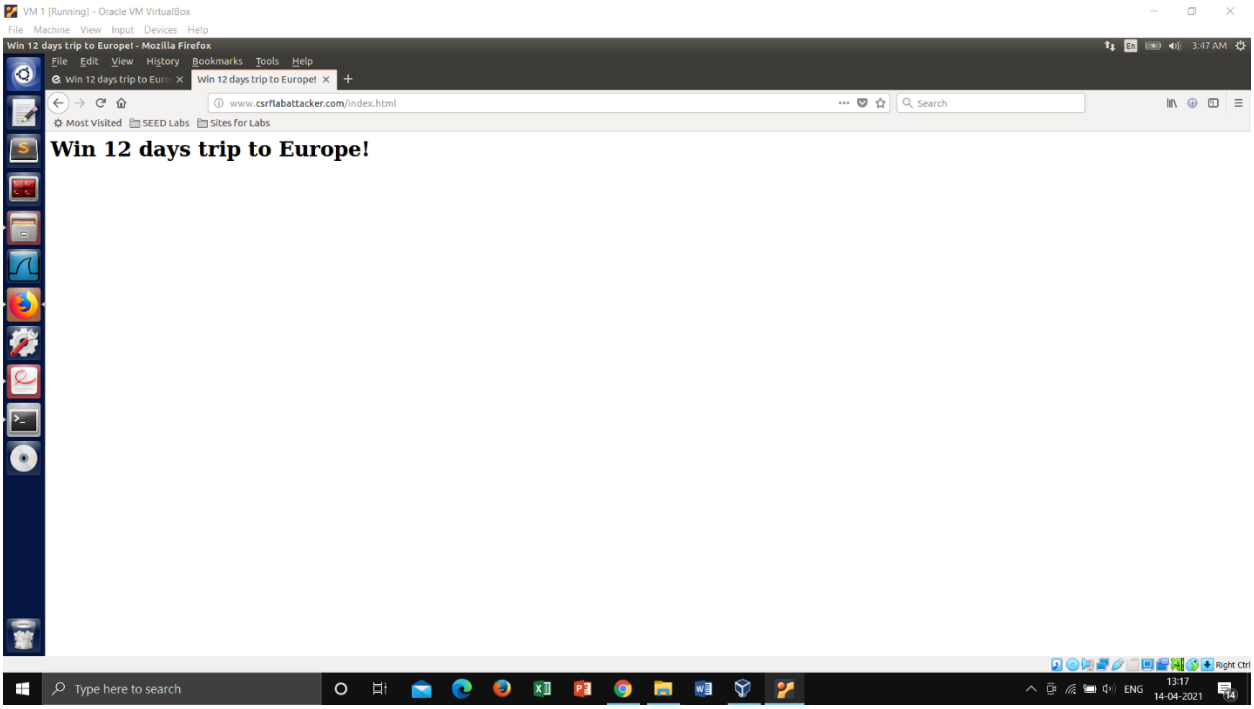
Alice gets a message from Boby



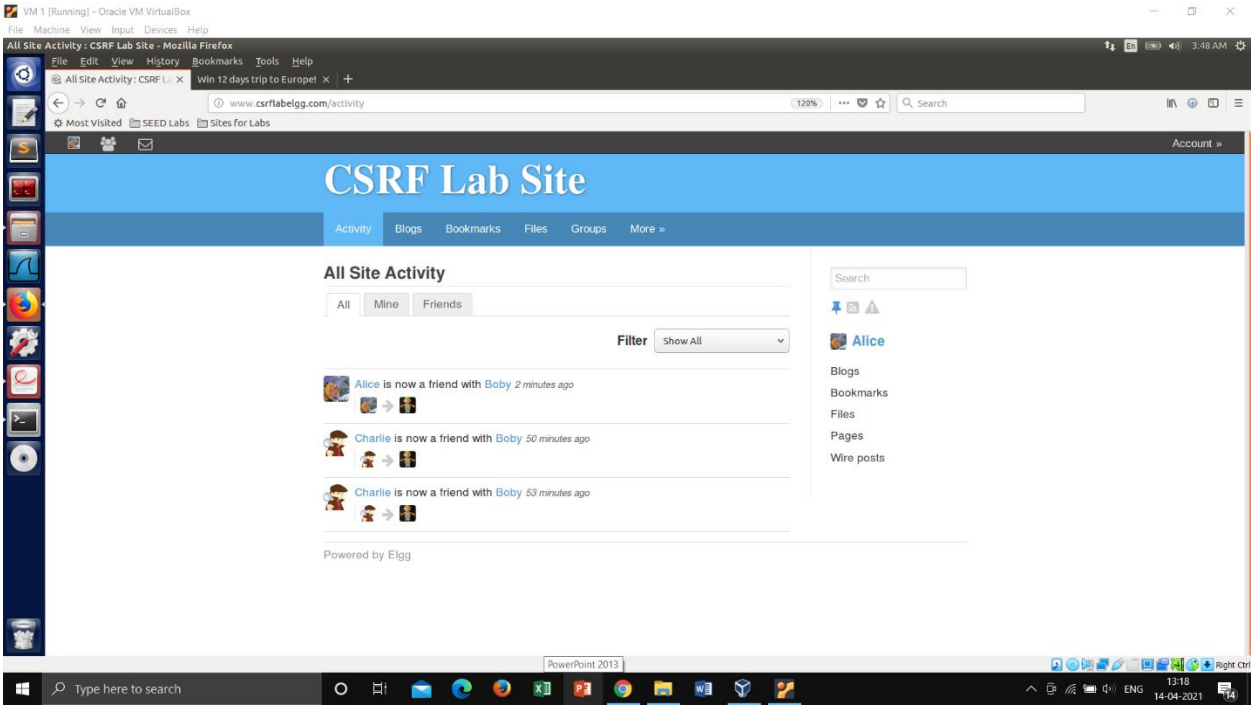
Contents of the message sent by Boby to Alice



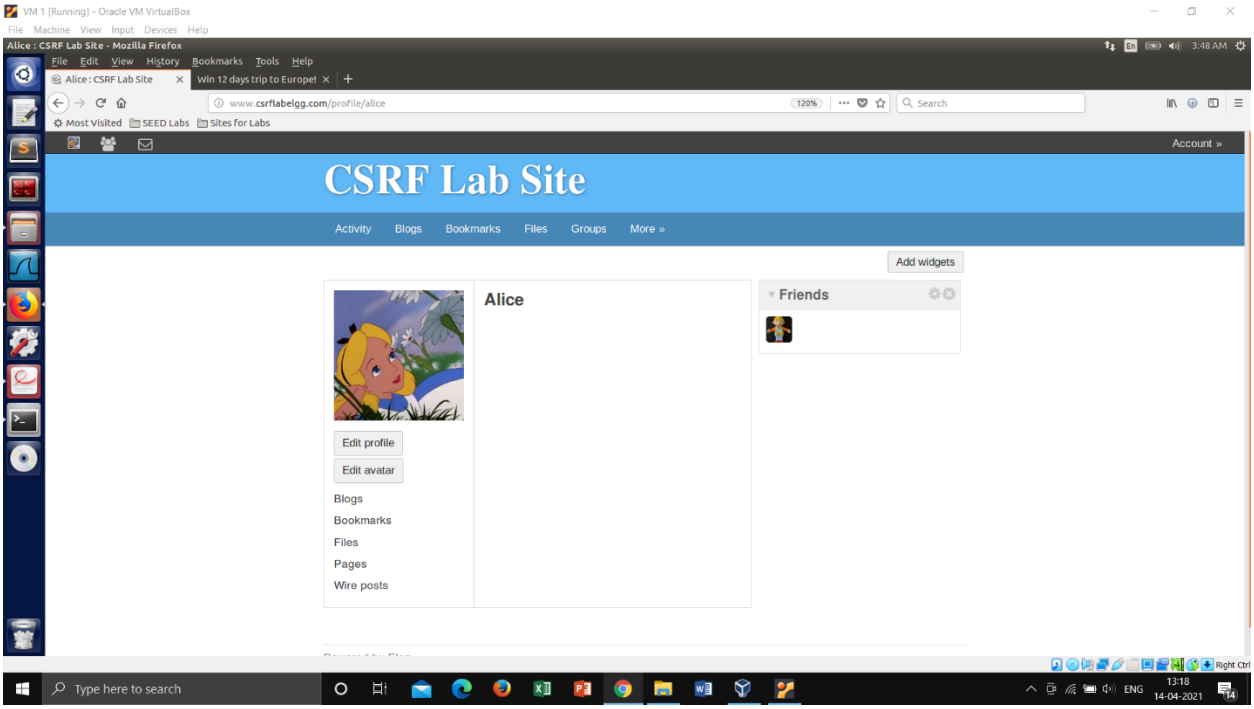
On clicking the URL, it takes Alice to a new page which has the following contents:



On returning back to the previous page, we can see that Boby has been added as a friend of Alice which shows that our attack was successful



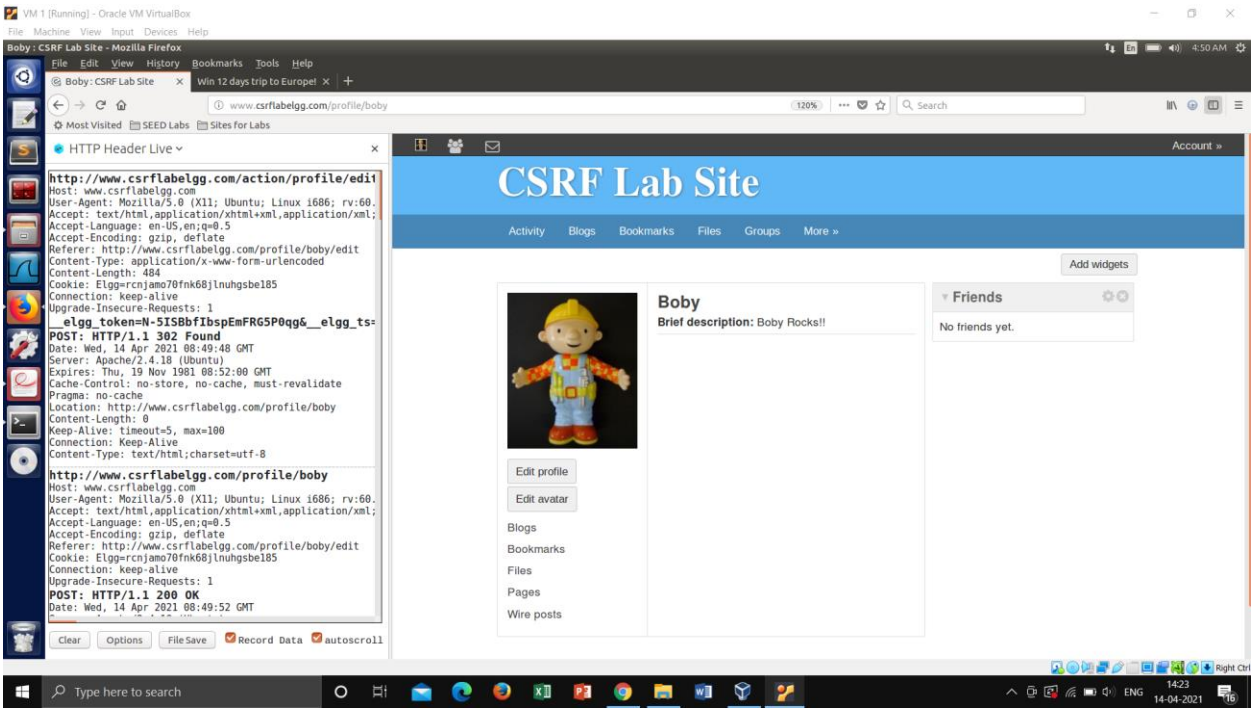
Alice's friend list after the attack:



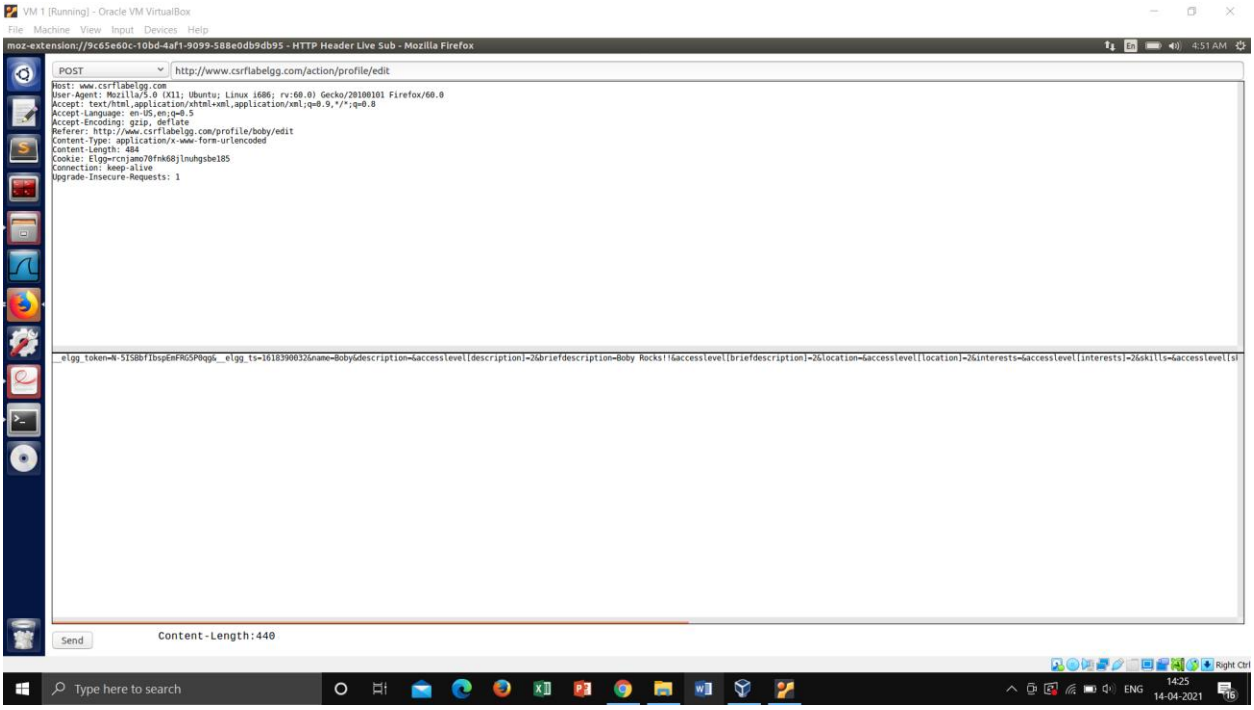
Task 3: CSRF Attack using POST Request

After adding himself to Alice's friend list, Boby wants to do something more. He wants Alice to say "Boby is my Hero" in her profile, so everybody knows about that. Alice does not like Boby, let alone putting that statement in her profile. Boby plans to use a CSRF attack to achieve that goal. That is the purpose of this task.

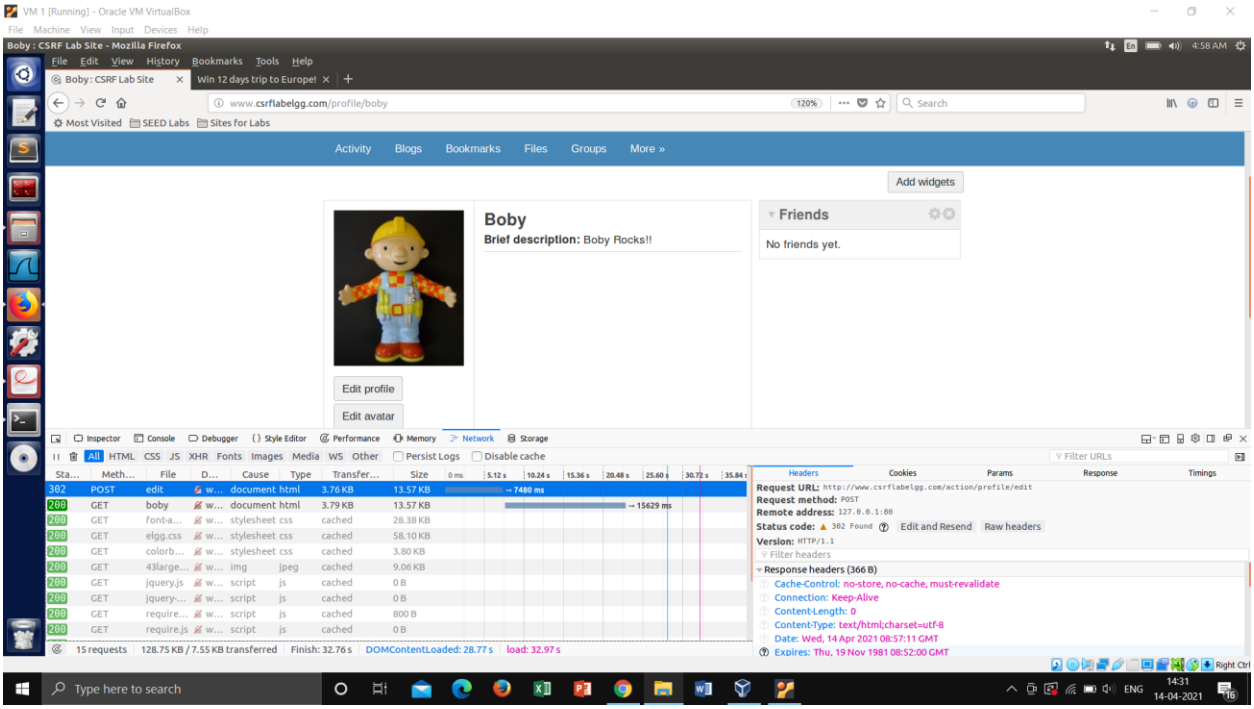
We observe the HTTP request on editing the profile

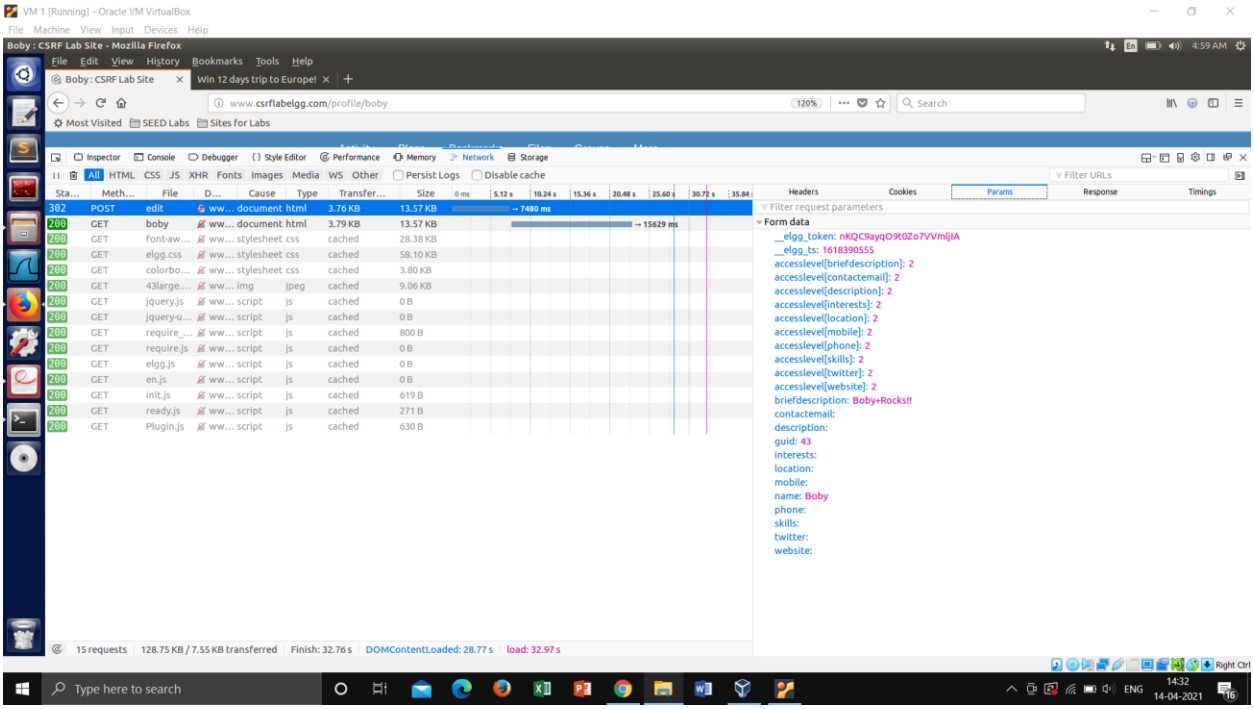


Here we can see that the content length is 484

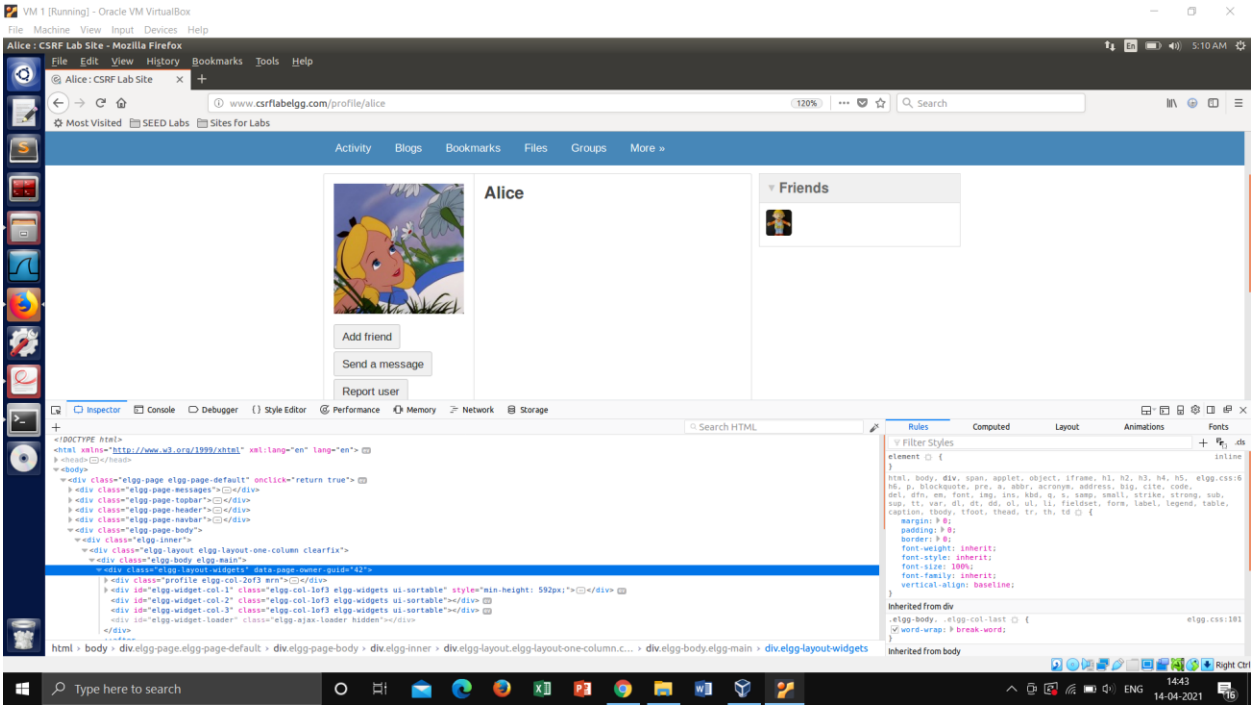


In the parameters tab, we can see that the access level is 2 indicating that it's public

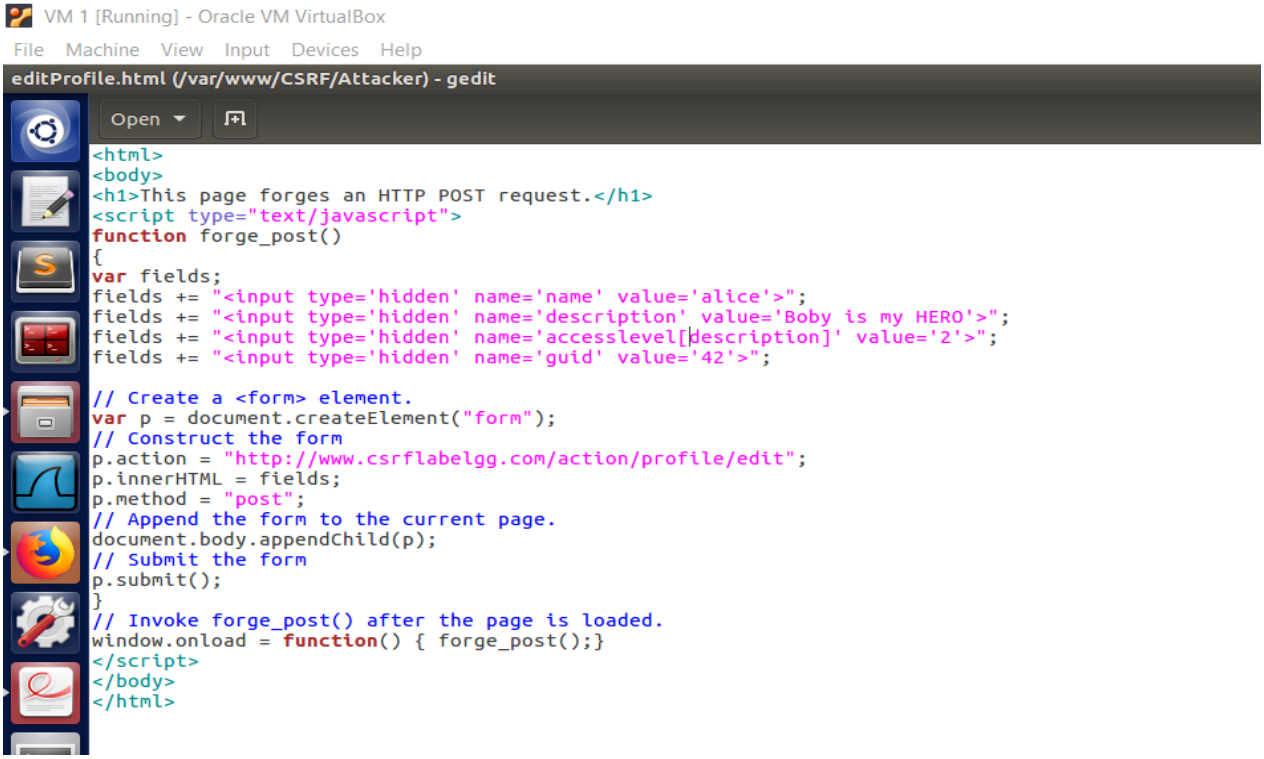




We find that Alice's guid value is 42



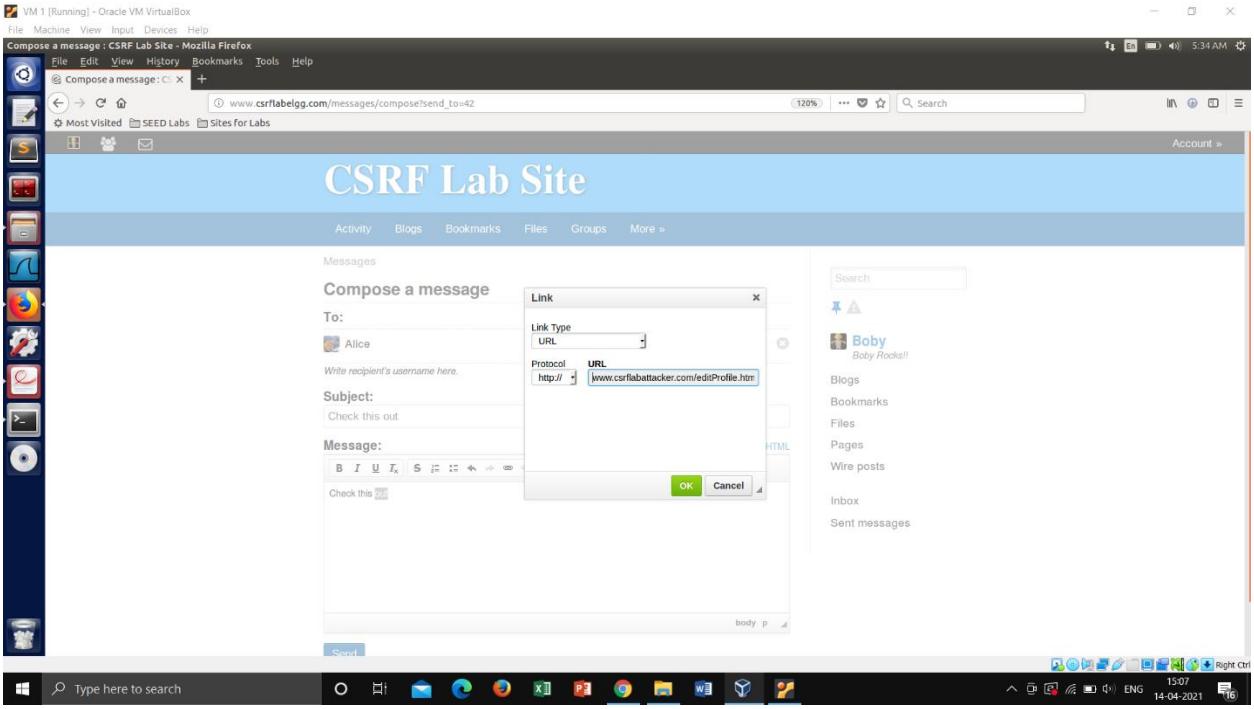
We generate the request from your attacking web page using JavaScript code
The value 2 sets the access level of a field to public. This is needed, otherwise, the access level will be set by default to private, so others cannot see this field.



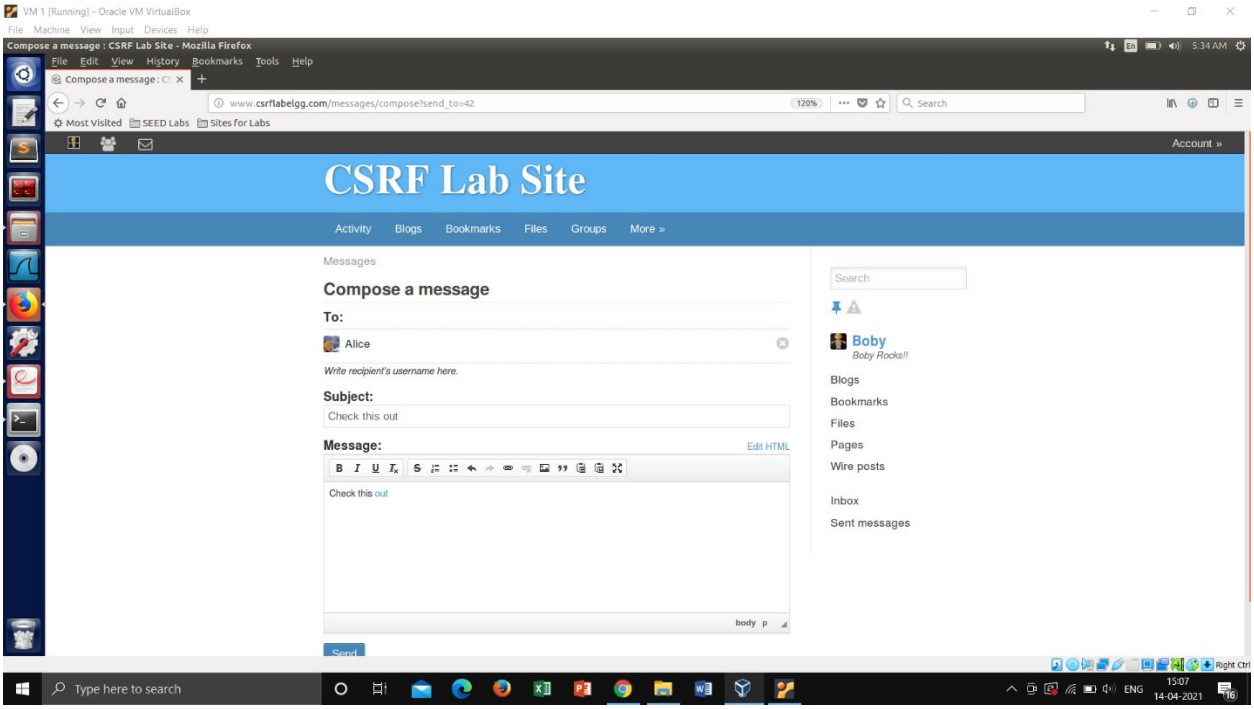

```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">
function forge_post()
{
var fields;
fields += "<input type='hidden' name='name' value='alice'>";
fields += "<input type='hidden' name='description' value='Boby is my HERO'>";
fields += "<input type='hidden' name='accesslevel[description]' 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>
```

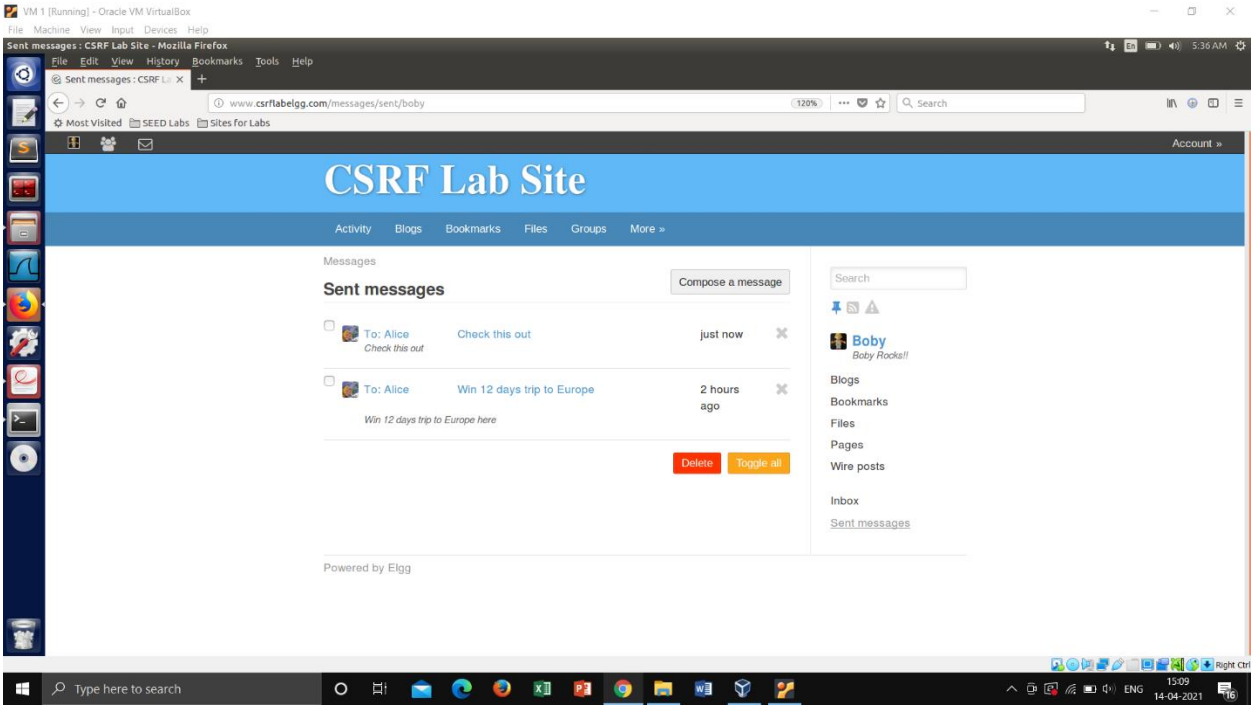
Now we frame the HTTP request to add Boby is my HERO in Alice's description



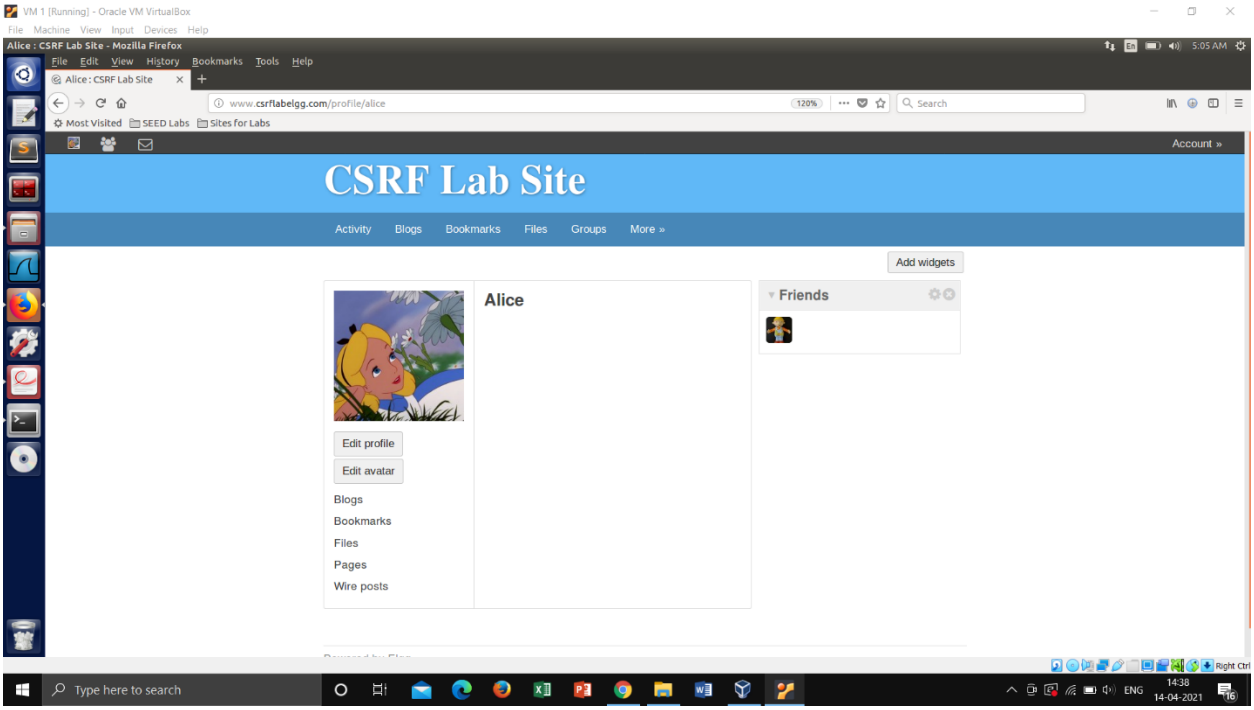
Contents of the message sent by Boby to Alice



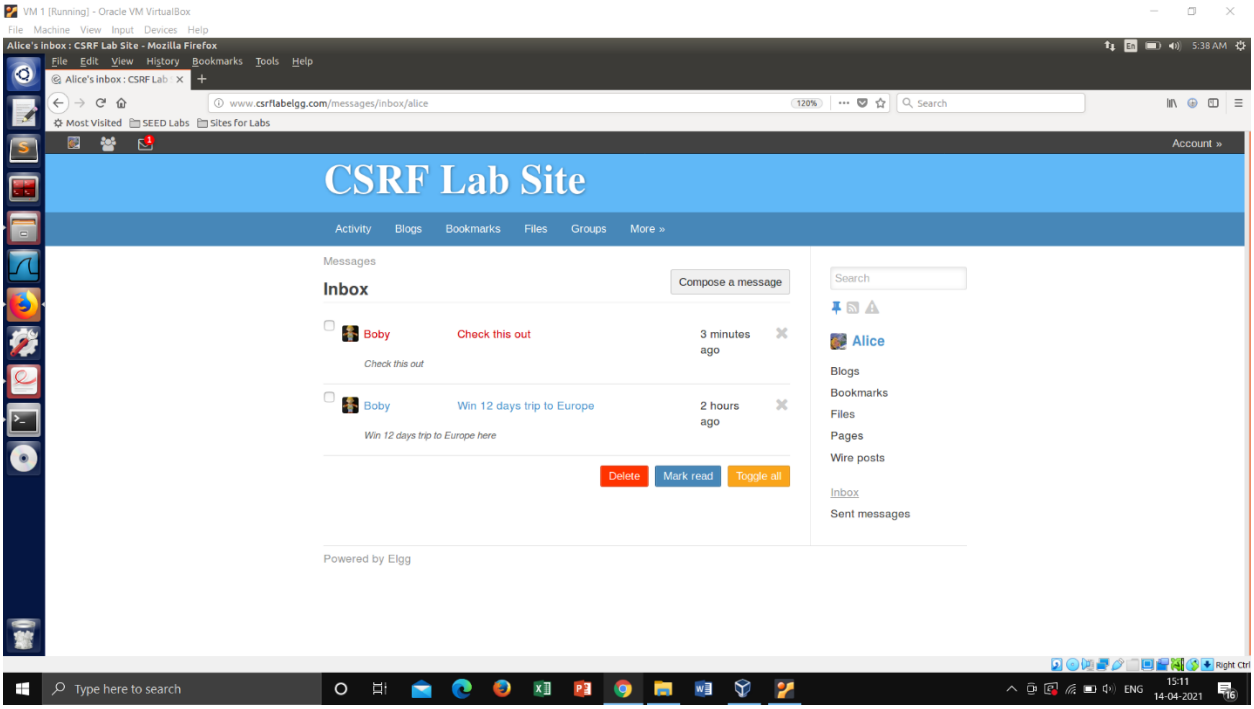
Boby's sent messages



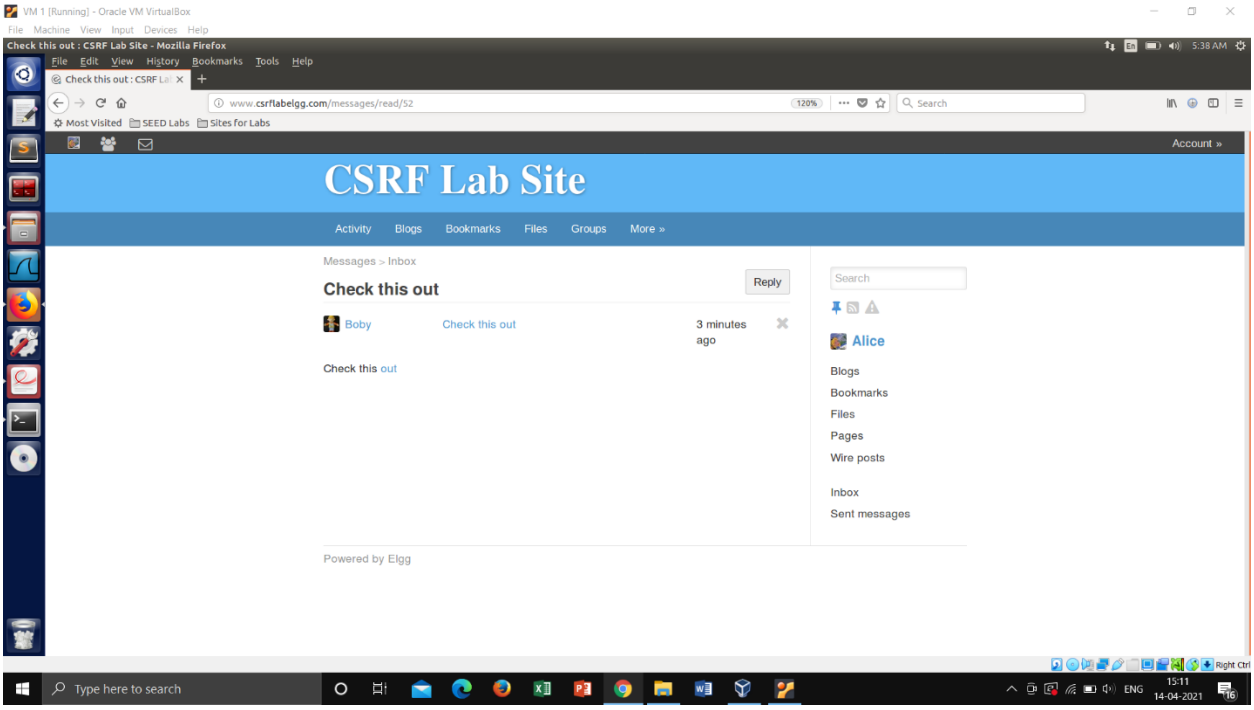
Alice's profile is empty before she opens the message



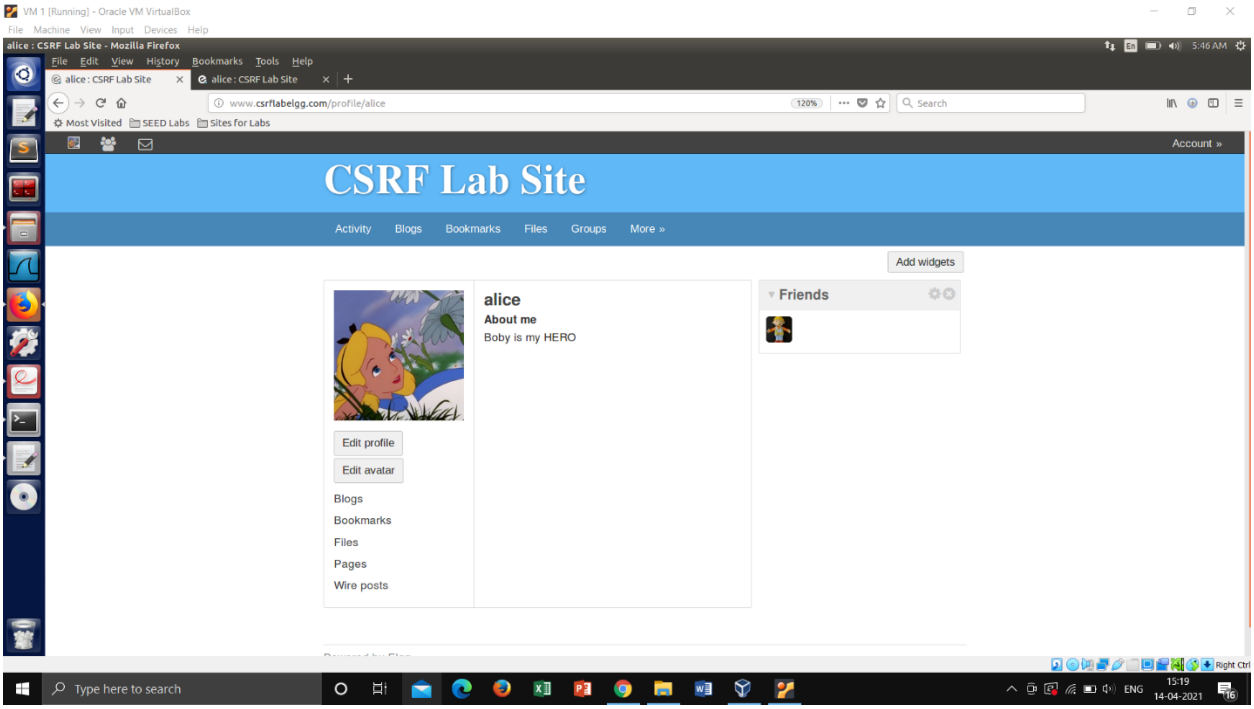
Message sent by Boby to Alice



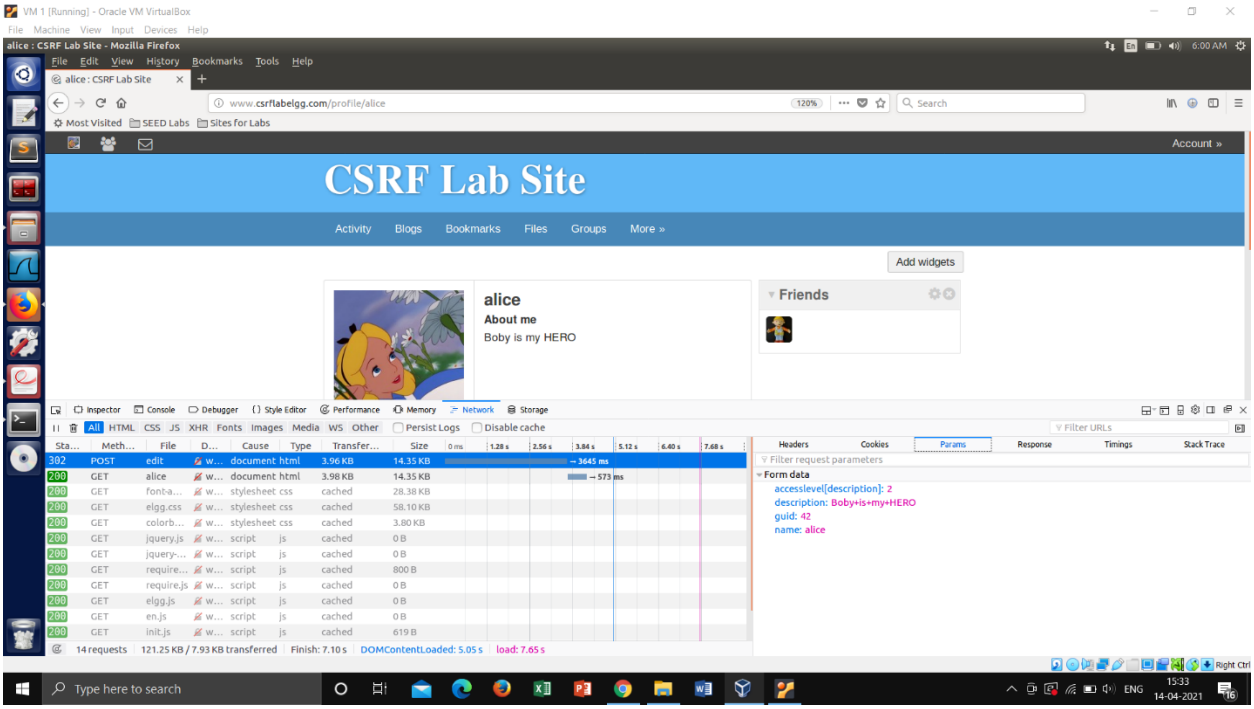
Contents of the message sent by Boby



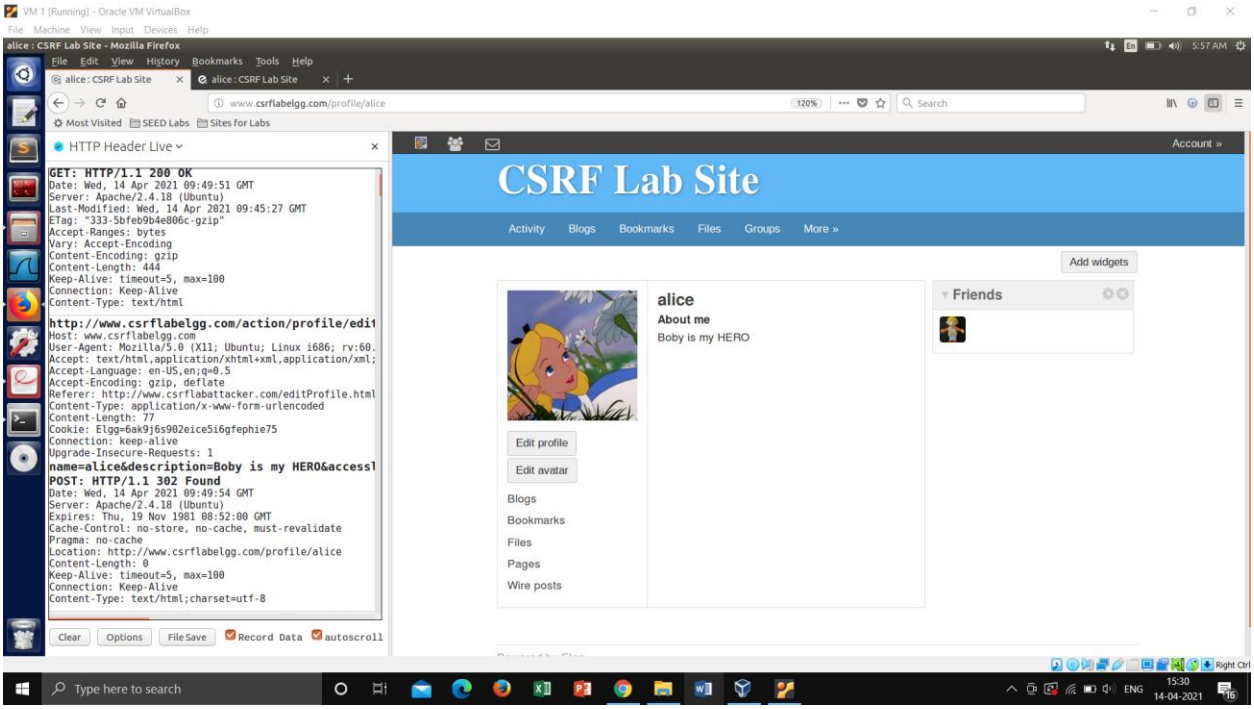
On clicking the link sent by Boby, Alice's description changes to Boby is my HERO



Observing the POST Request in the web developers tool:



Observing the Post Request in HTTP Header Live



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. Please describe how Boby can solve this problem.

Ans: We don't require Alice's credentials to get the GUID value because we can get it by searching for Alice on the webpage and clicking on inspect element to get the GUID value of Alice. If the website does not contain any GUID value in the code then we can enter random password and look at HTTP request or response and get the GUID value if it is present.

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? Please explain.

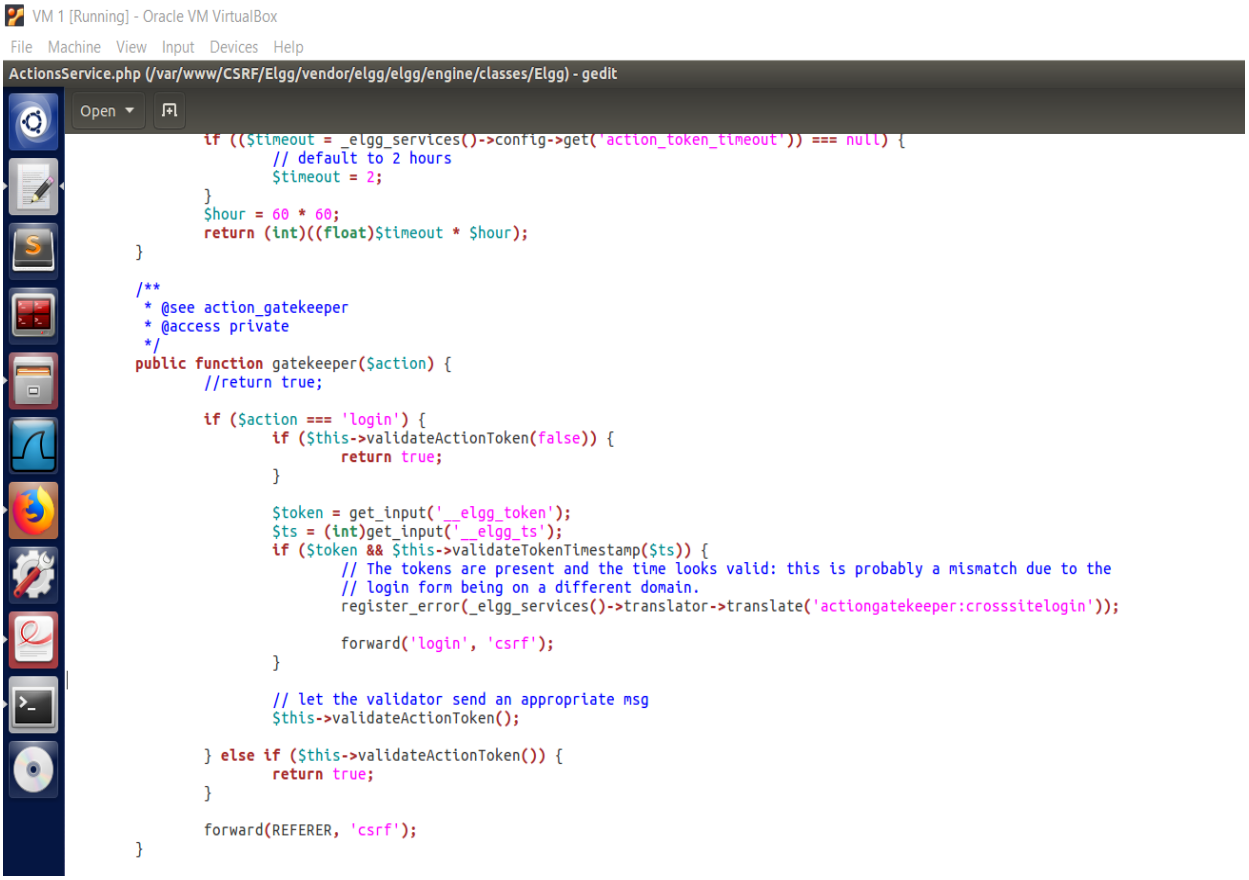
Ans: Boby will not be able to launch an attack because he needs to know the GUID value to modify the victim's profile and moreover the web page is different

than the targeted website. Also, the GUID value is sent only to the targeted website's server and not to any other website. Hence Boby cannot launch this attack.

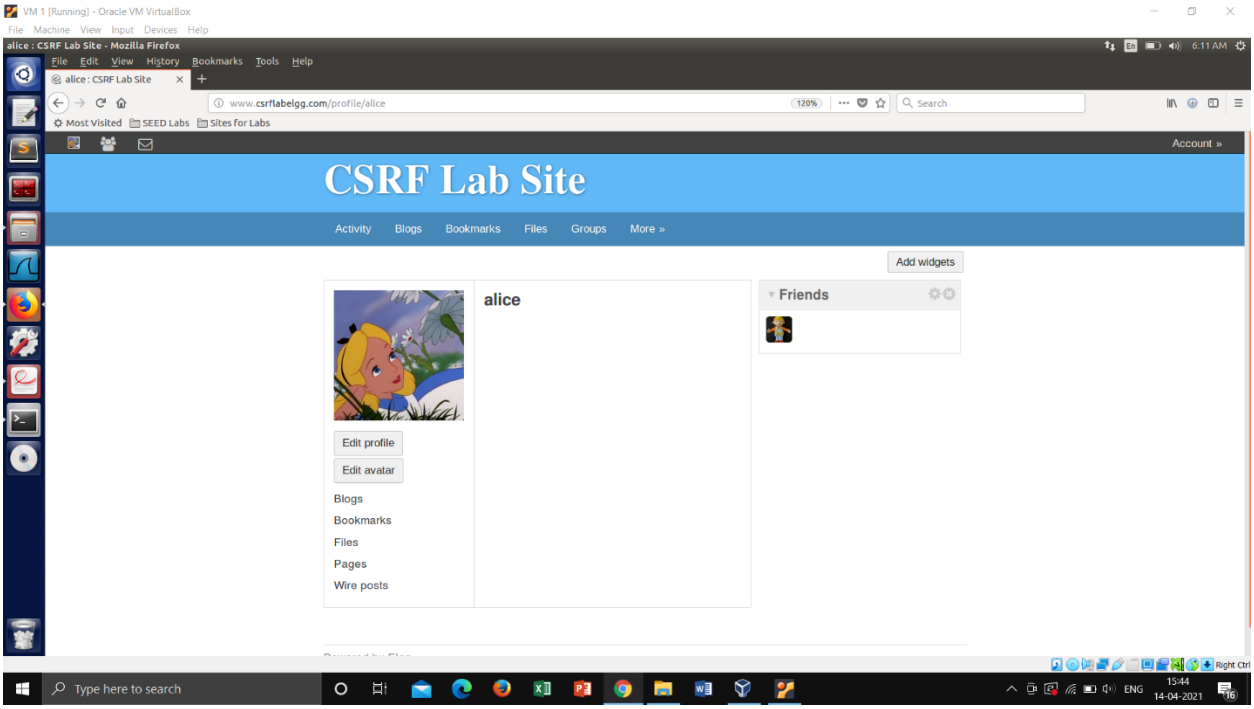
Task 4: Implementing a countermeasure for Elgg

Elgg does have a built-in countermeasures to defend against the CSRF attack. Secret-token approach: Web applications can embed a secret token in their pages, and all requests coming from these pages will carry this token. Because cross-site requests cannot obtain this token, their forged requests will be easily identified by the server. Referrer header approach: Web applications can also verify the origin page of the request using the referrer header. The web application Elgg uses secret-token approach. It embeds two parameters elgg ts and elgg token in the request as a countermeasure to CSRF attack. The two parameters are added to the HTTP message body for the POST requests and to the URL string for the HTTP GET requests.

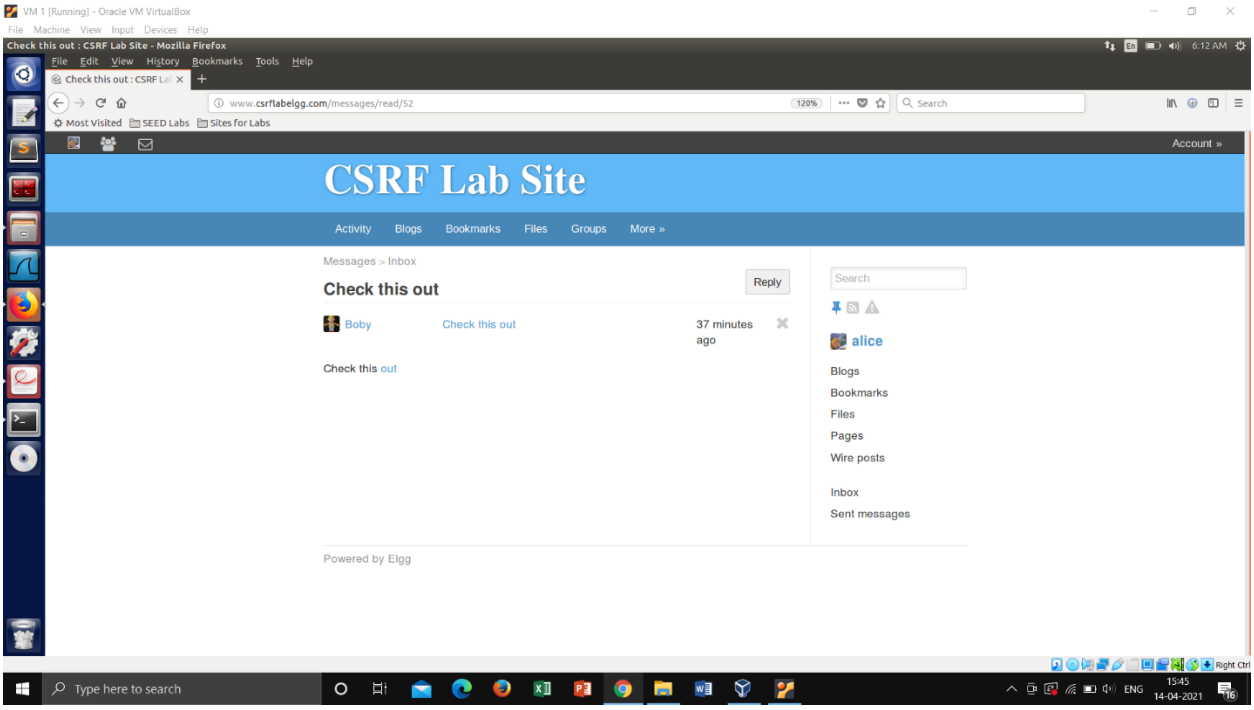
We comment out return True statement



Alice's profile before the attack:

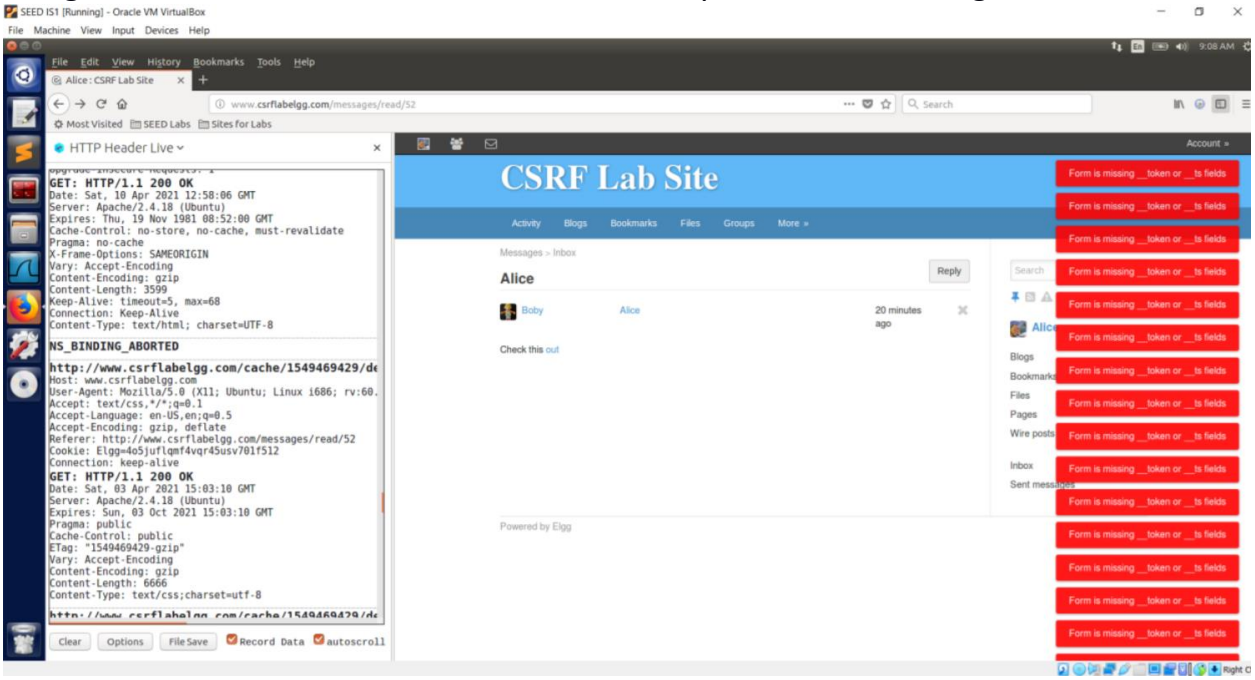


Now Alice tries to open the link sent by Boby

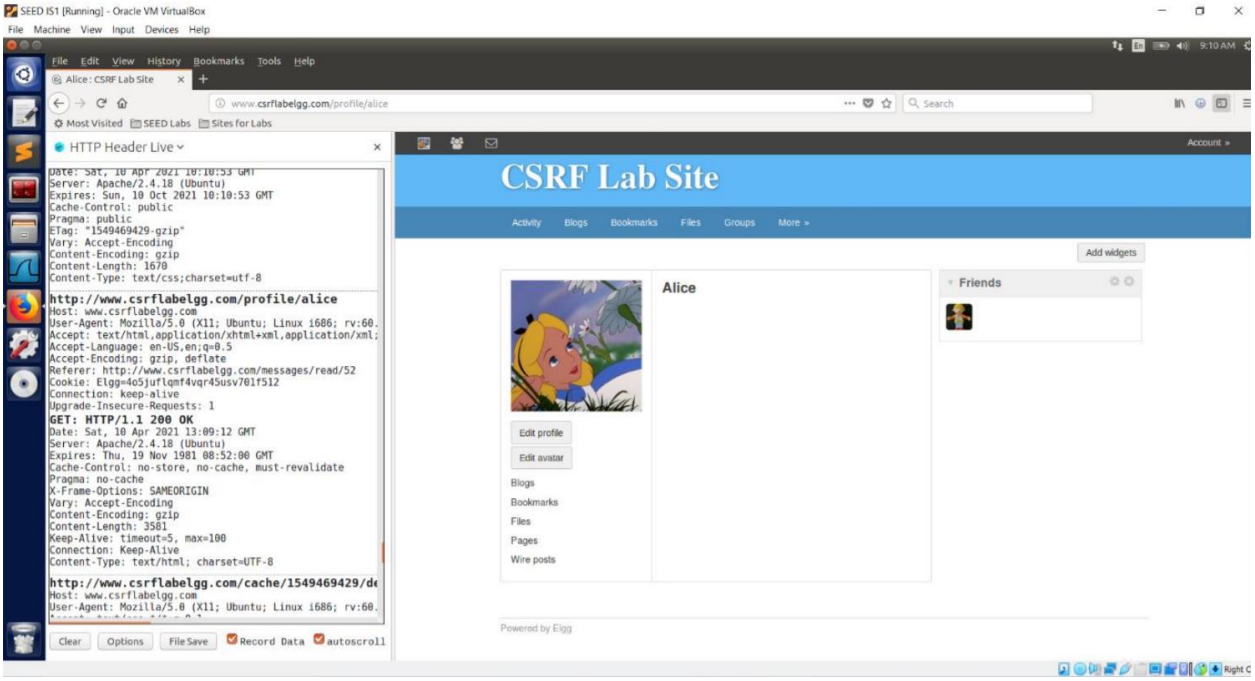


On sending the POST Request, we get an error indicating that our attack was unsuccessful.

We get the error that the token and timestamp fields are missing.



Alice's profile remains unchanged, hence our attack was not successful.



After turning on the countermeasure above, try the CSRF attack again, and describe your observation. Please point out the secret tokens in the HTTP request captured using Firefox's HTTP inspection tool. Please explain why the attacker cannot send these secret tokens in the CSRF attack; what prevents them from finding out the secret tokens from the web page?
Only GUID and Name is sent as params and elgg token and timestamp seem to be missing. They are not sent because the request is sent from attacker's website to elgg server and not from elgg website to server. Since the secret tokens are set only in the elgg's website, request going only from the same website will have these parameters