Eric Samuel Huddleston hpg103

Matthew Frierson evz006

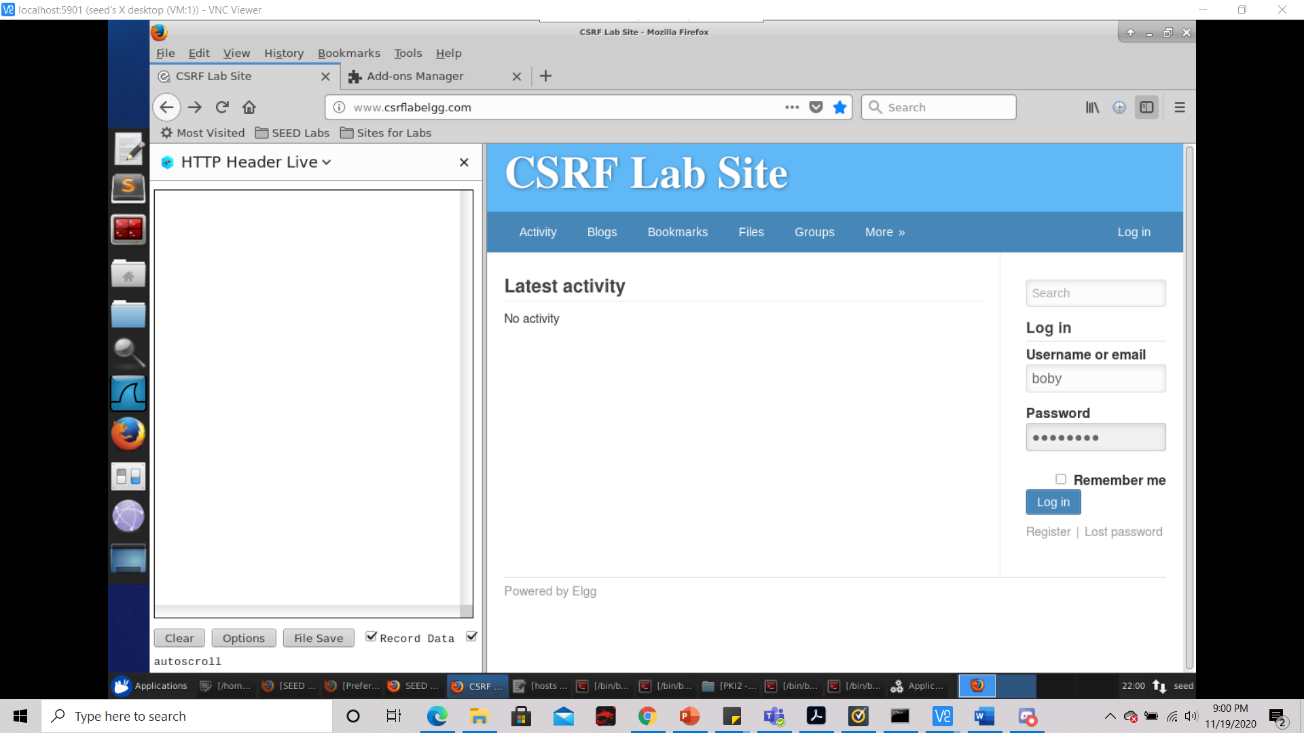
11/19/2020

**Cross-Site Request Forgery (CSRF) Attack Lab – Lab Report**

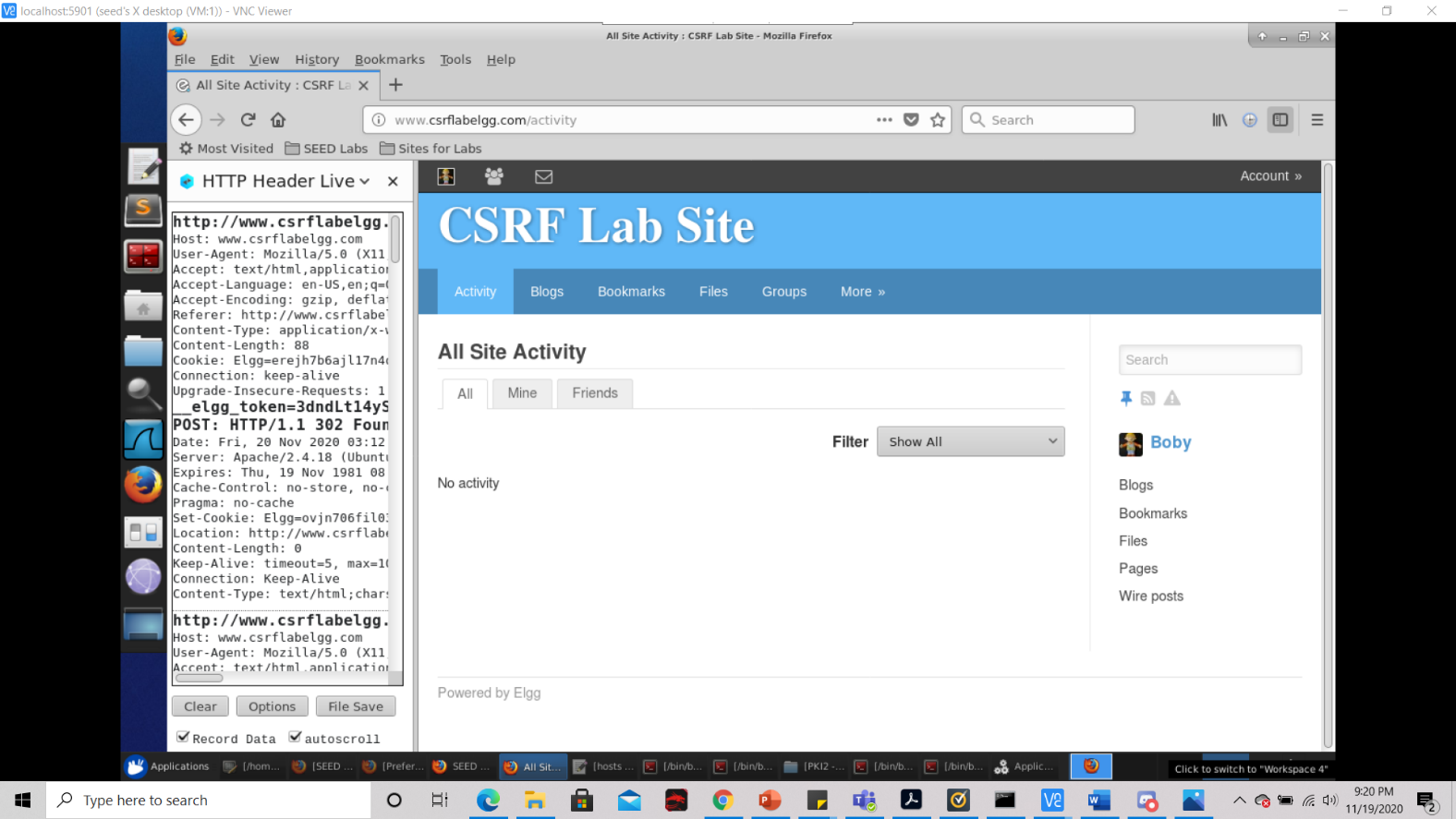
**3.1 Task 1: Observing HTTP Request.**

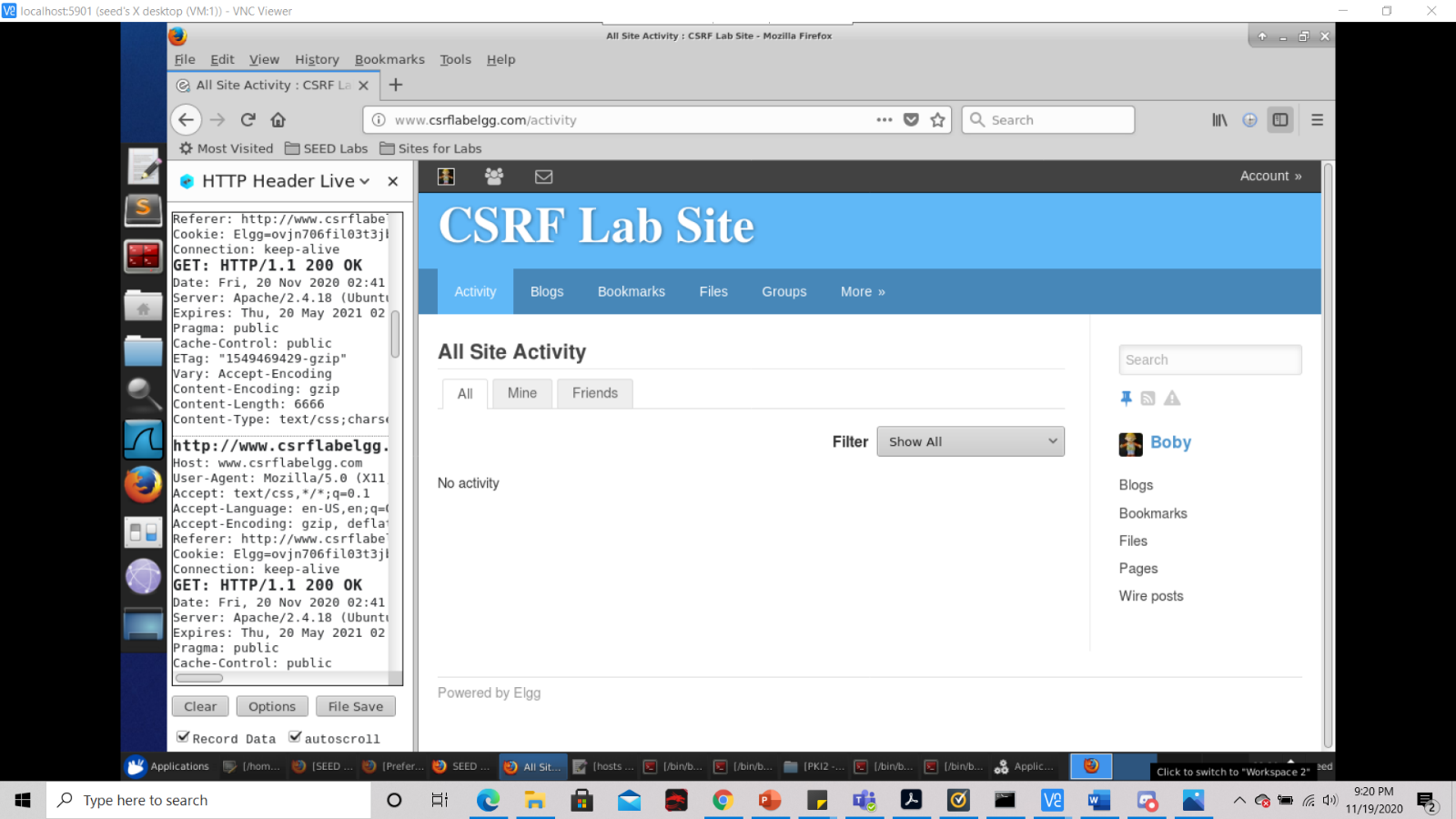
The first step was installing the HTTP Header Live tool to Firefox, and here I show It being successfully installed on the Virtual Machine’s browser:

  
Thereafter I login as the user ‘Boby’ using the ‘UserName’ and ‘Password’ credentials that were provided in the assignment description as such:

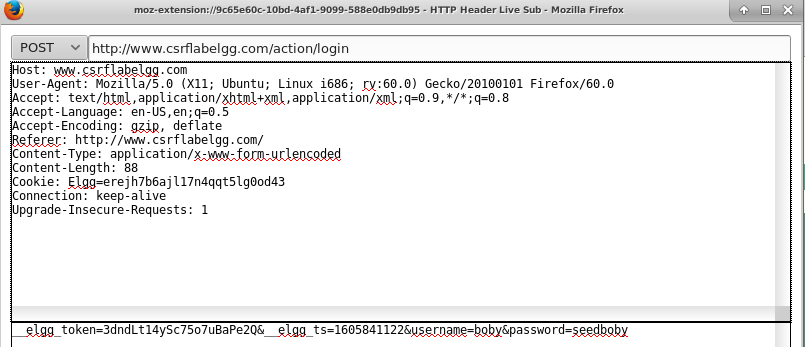


Once logged in, I observed that the tool was receiving get requests and post requests as shown below:





If I click on one of these POST requests and maximize the window, we can see that the tool displays some useful information about the request as shown below:



Here we can see the name of the Host, where the POST request is being referred to, the http secured version of the same site, and the HTTP cookie stored for the site.

The same thing can be said for a GET request if I click on the GET request and maximize the window, we can also see the same, useful information the HTTP Header Live Tool provides as seen below:



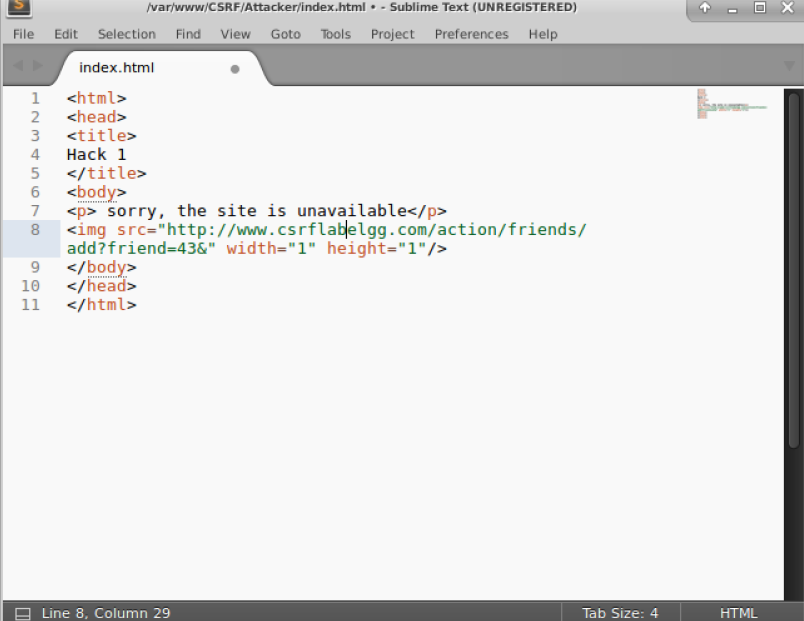
Unlike the information the tool provided for the POST request, it is obvious that much less information is provided for the GET request. Namely that the Content-Type and Content-Length and Upgrade portions are missing. This is because with a GET request less amount of data can be sent since the data is sent in the header. This is reflected here.

**3.2 Task 2: CSRF Attack using GET Request**

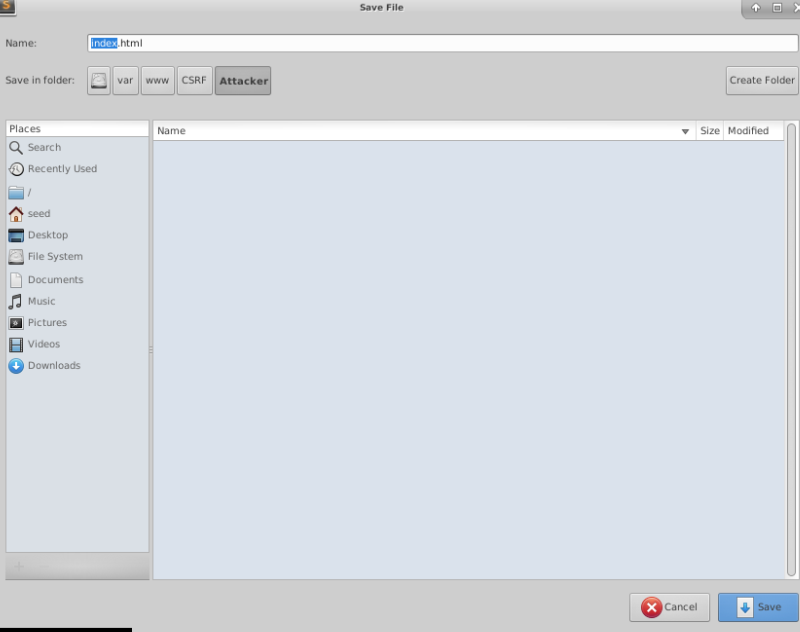
First, I open up the terminal, and then open up the index.html file with the following command:



Then I type in the following code to index.html as such:



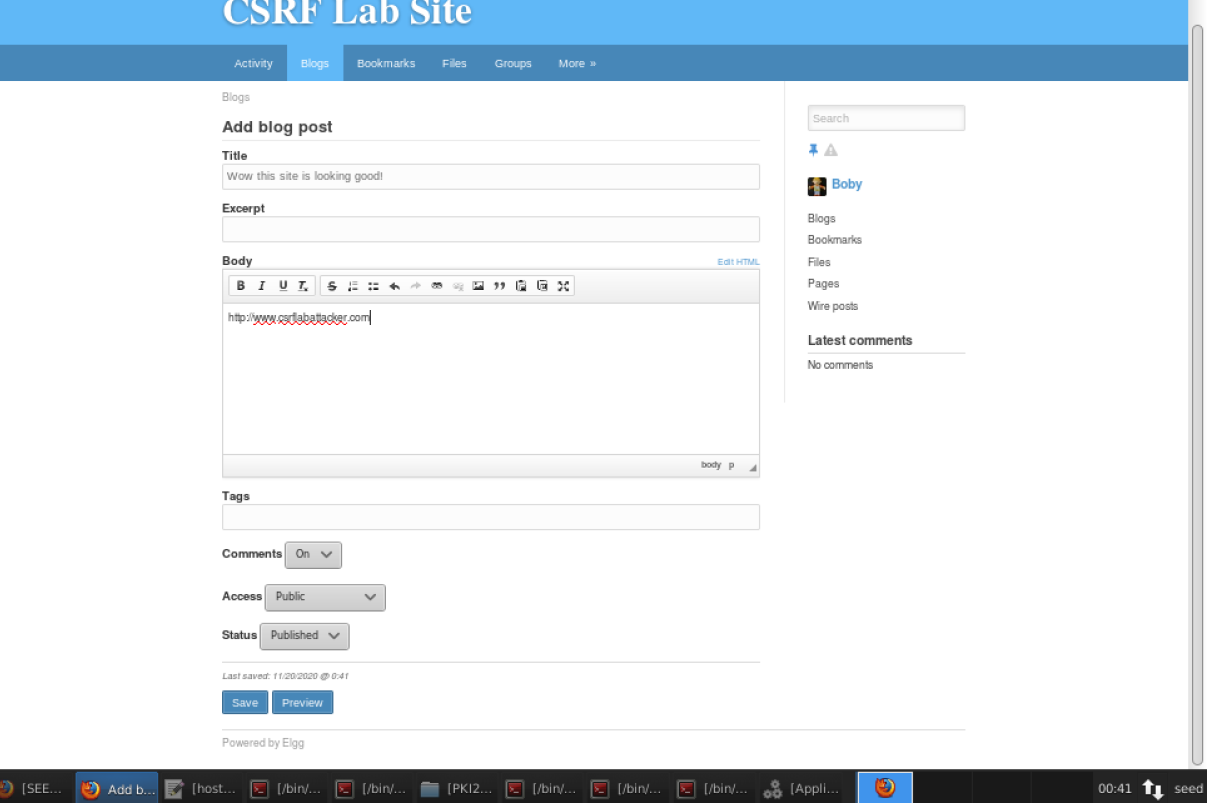
Then I click ‘File’🡪’Save-as’ and then save the file to the directory /var/www/CSRF/Attacker as so:



After saving and exiting out of the index.html file, I need to restart apache2 as shown below:



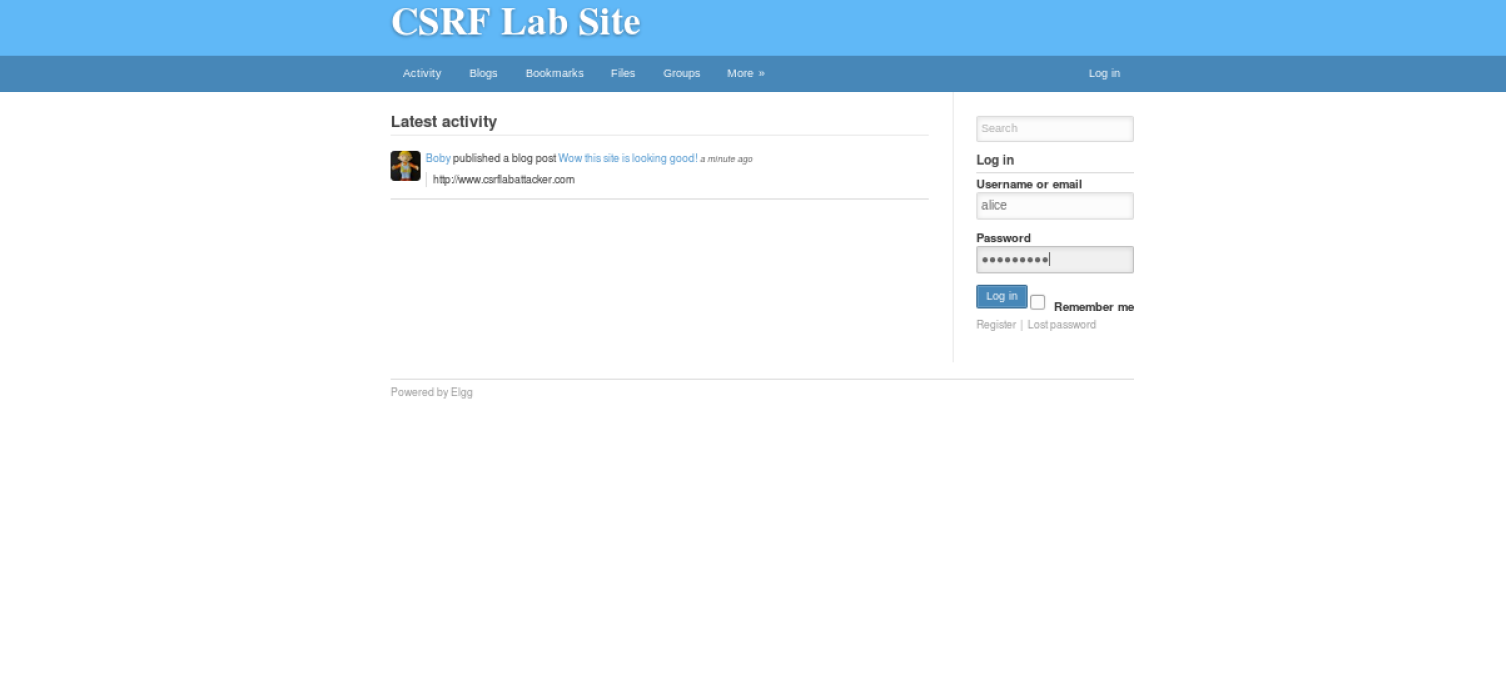
Then I go to the elgg website and create a blog on Bobby’s profile under the malicious website as so, and then I click the save button at the bottom of the page:



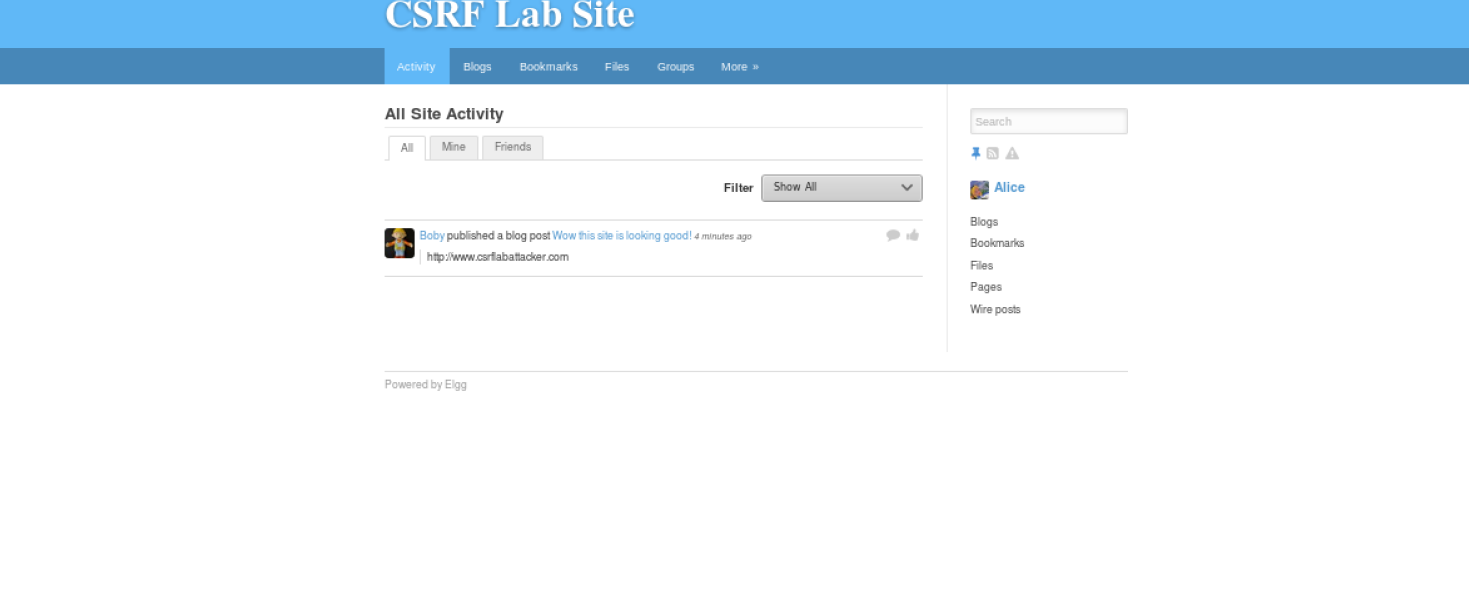
After saving, I log out of Bobby’s account as shown,



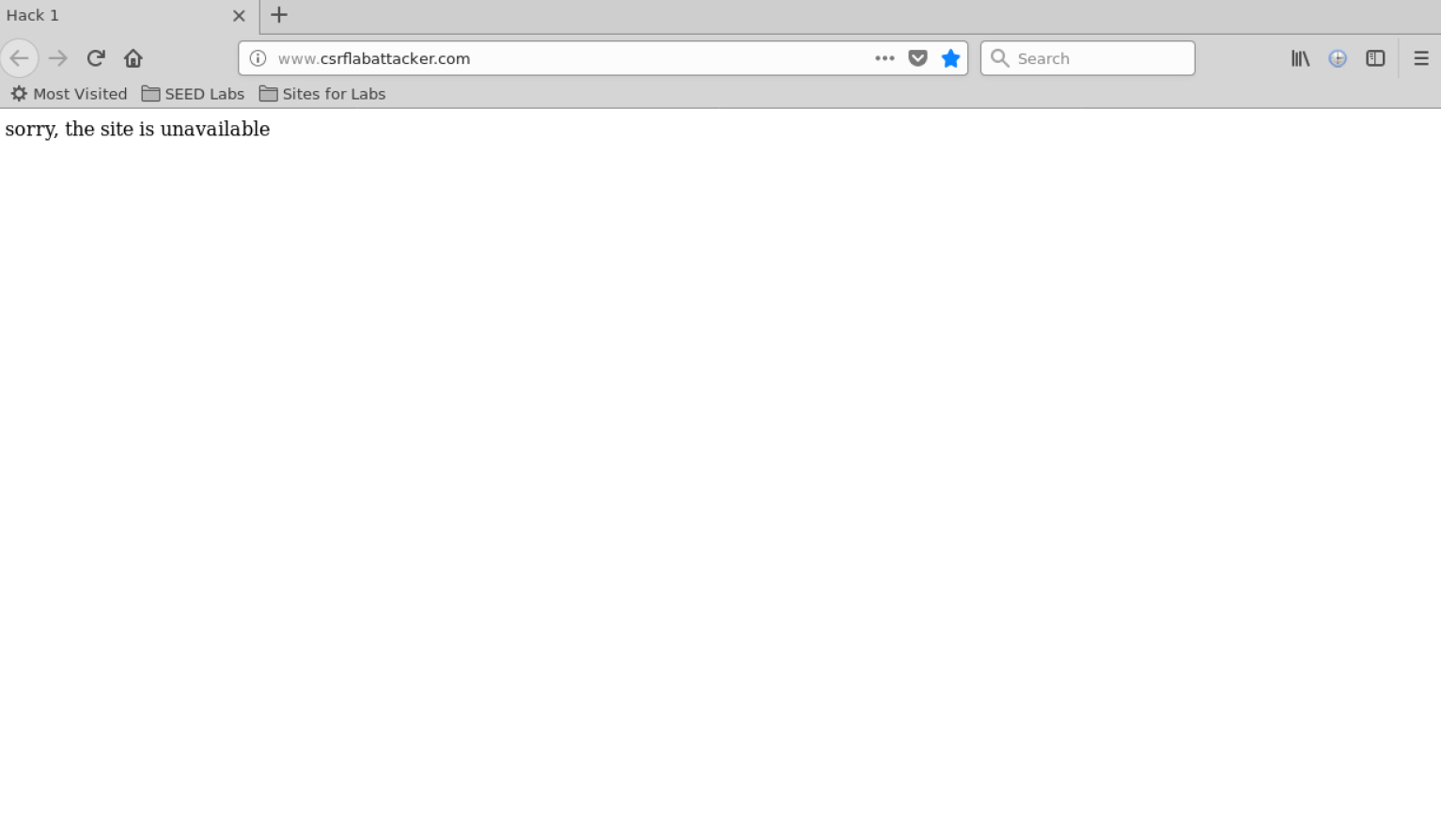
And then log into Alice’s account as shown, using the credentials supplied in the assignment page:



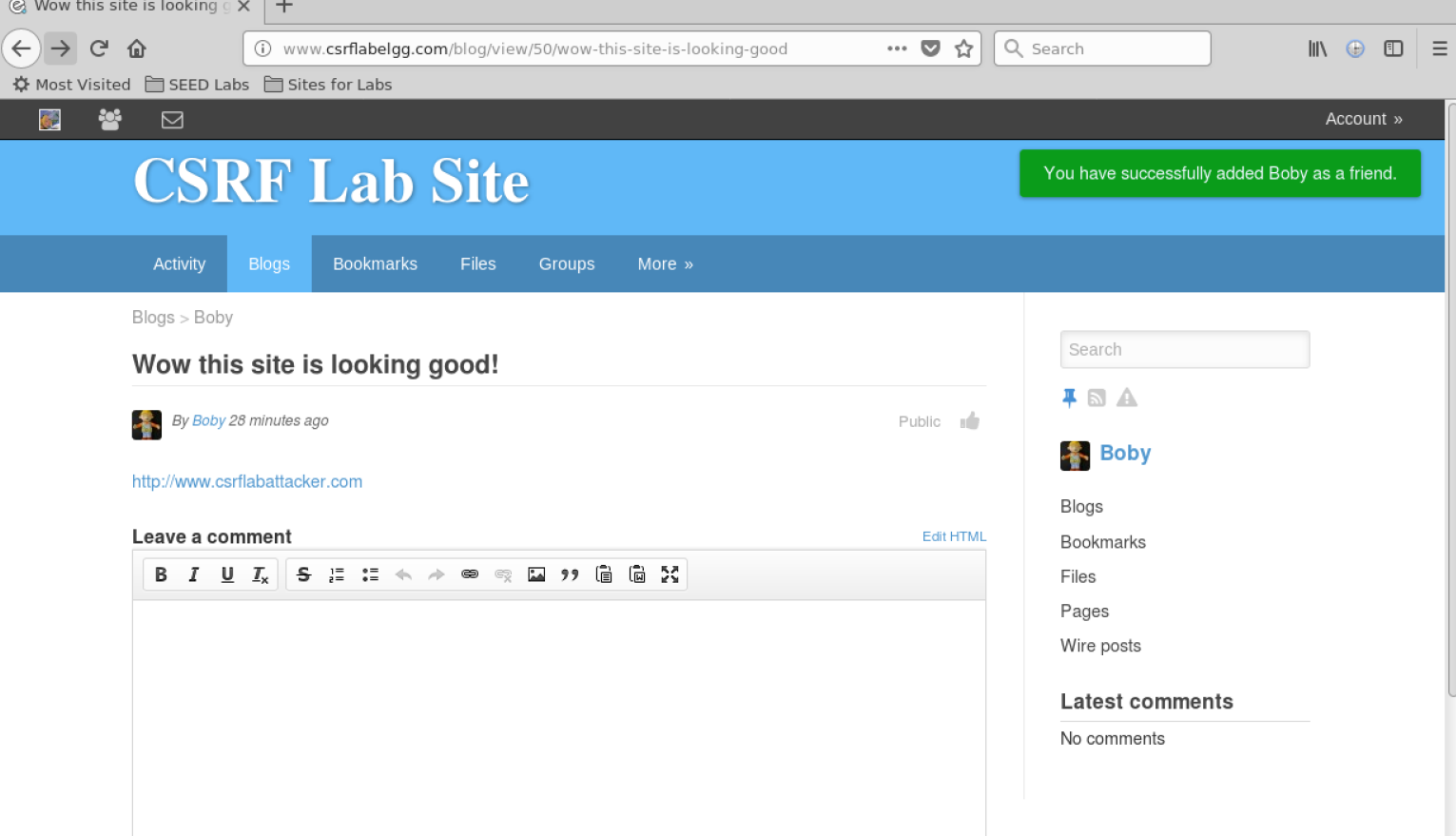
Now I’m logged in as Alice, shown below:

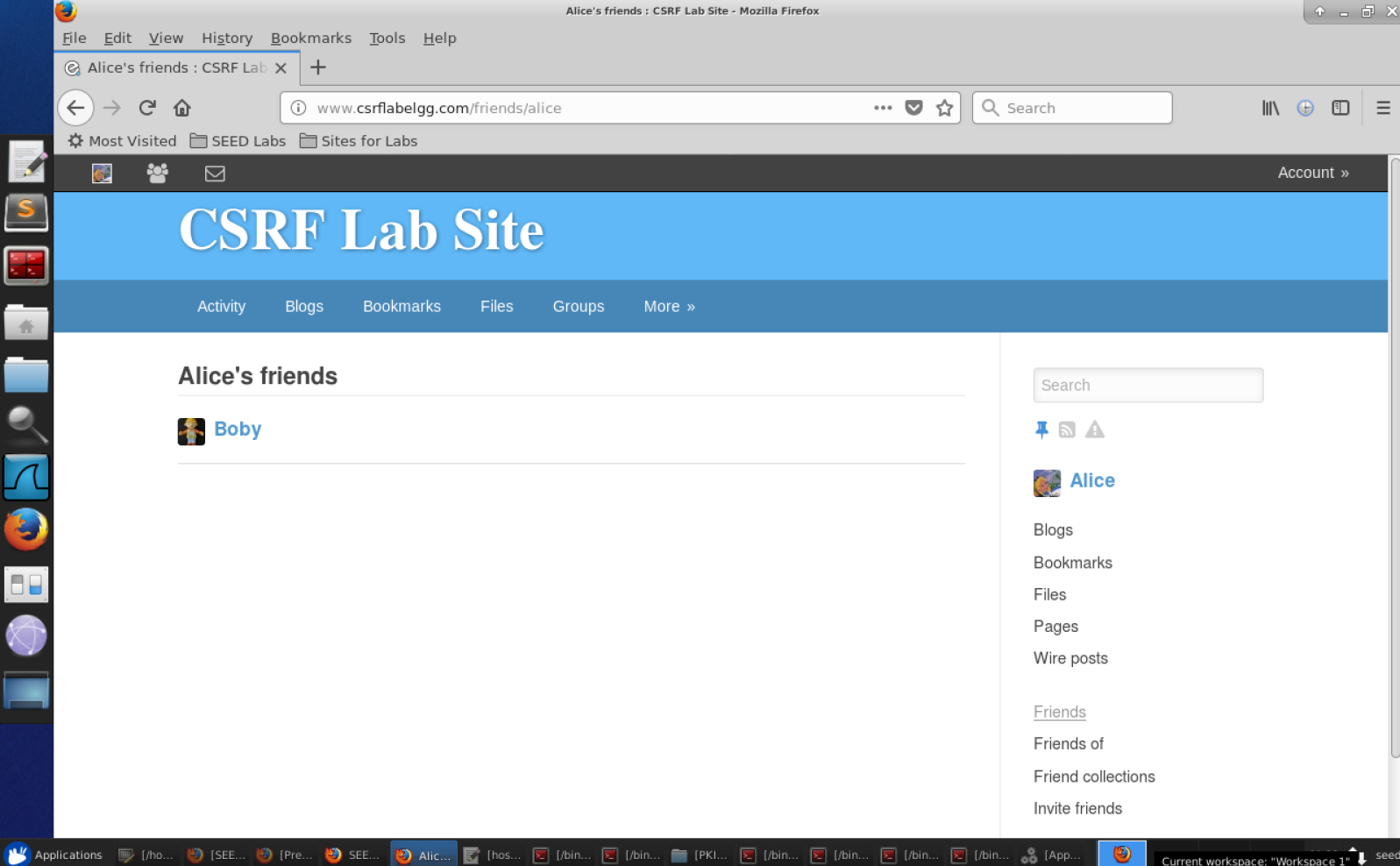


Now as alice, if I open the link under Bobby’s blog post in a new window I am redirected to the attacker’s site as shown below:



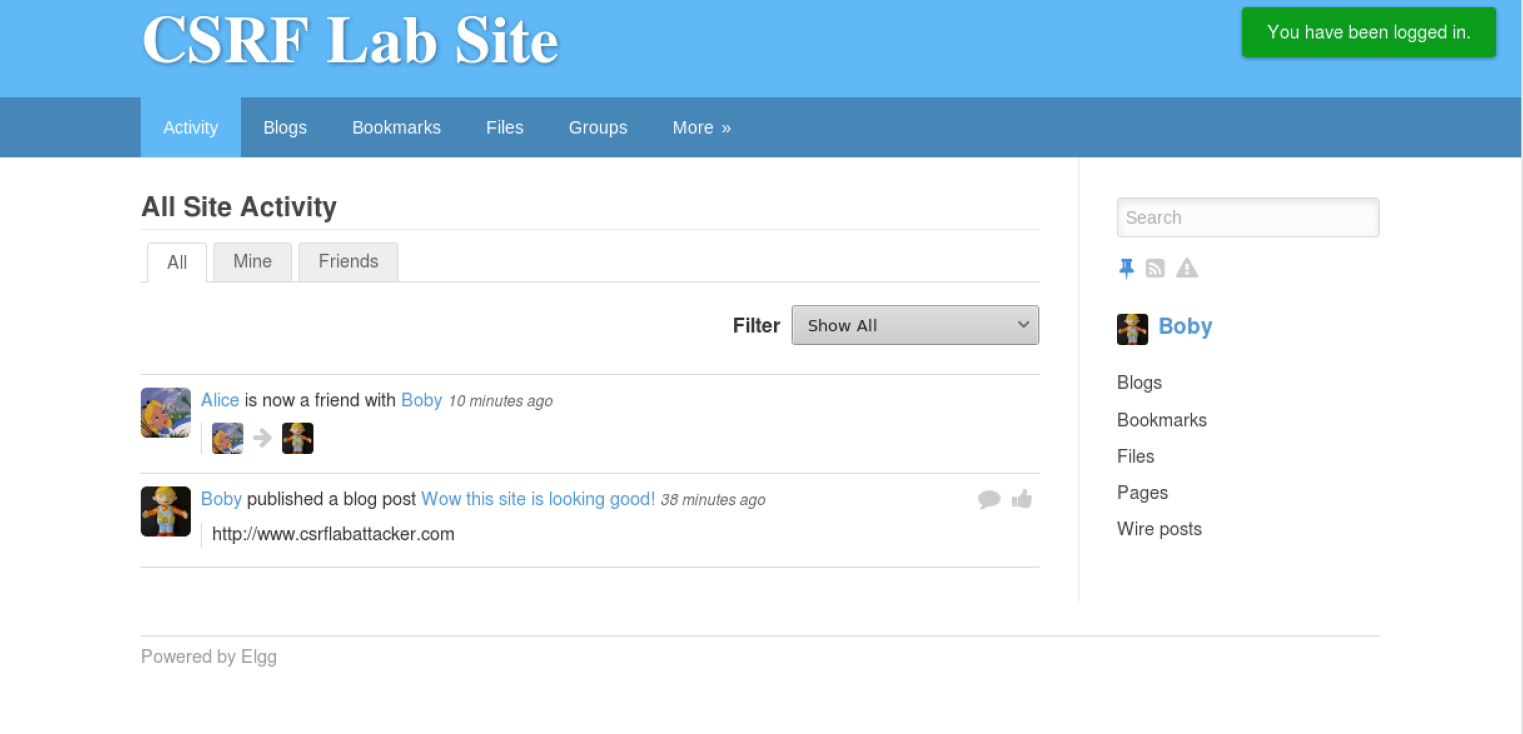
Now, when I click the back arrow on Firefox I can see when I go back to the CSRF Lab Site Blog Page I can see that Bobby was added as a friend (to the dismay of Alice):



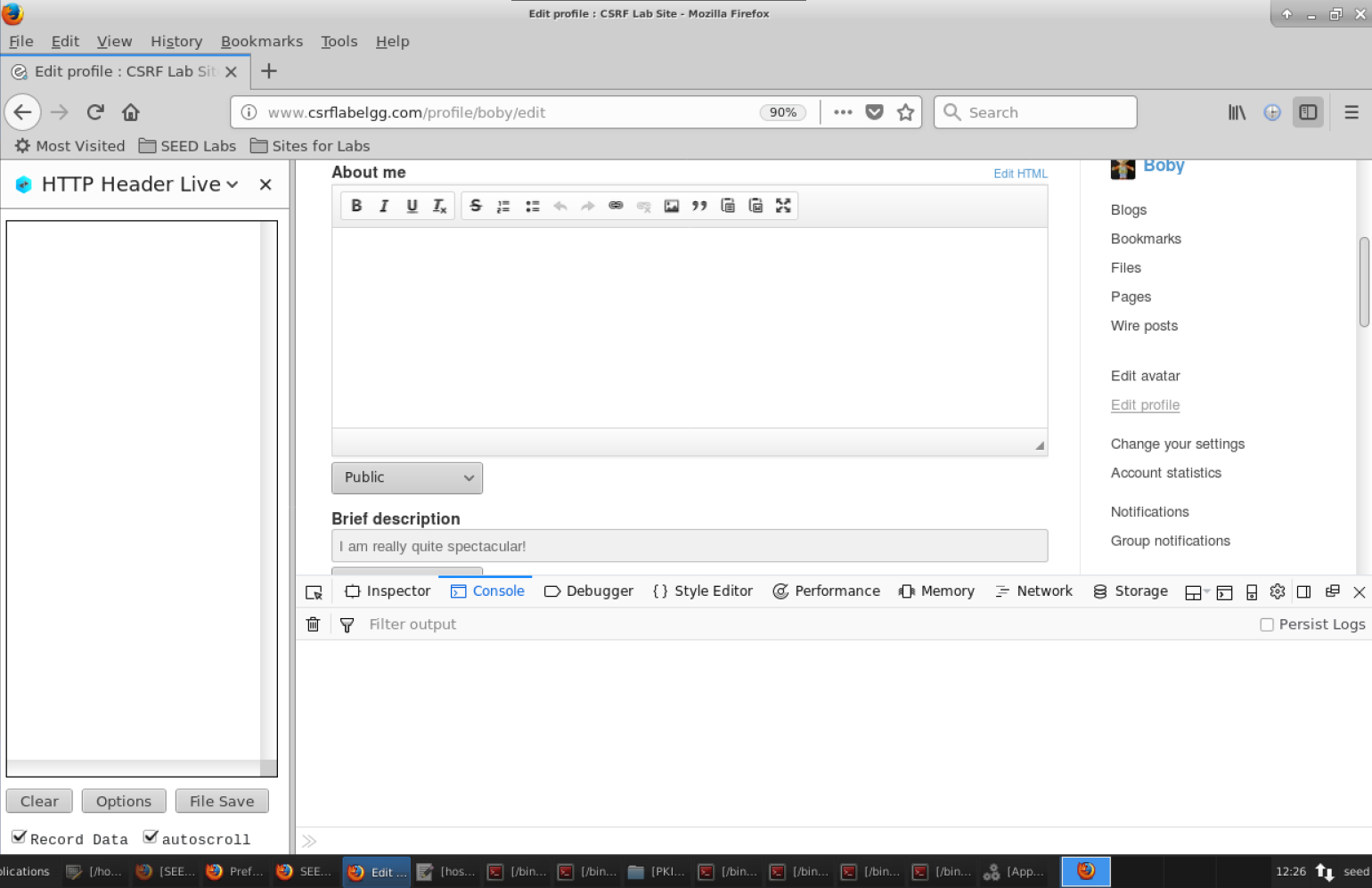


**Task 3: CSRF Attack using POST Request**

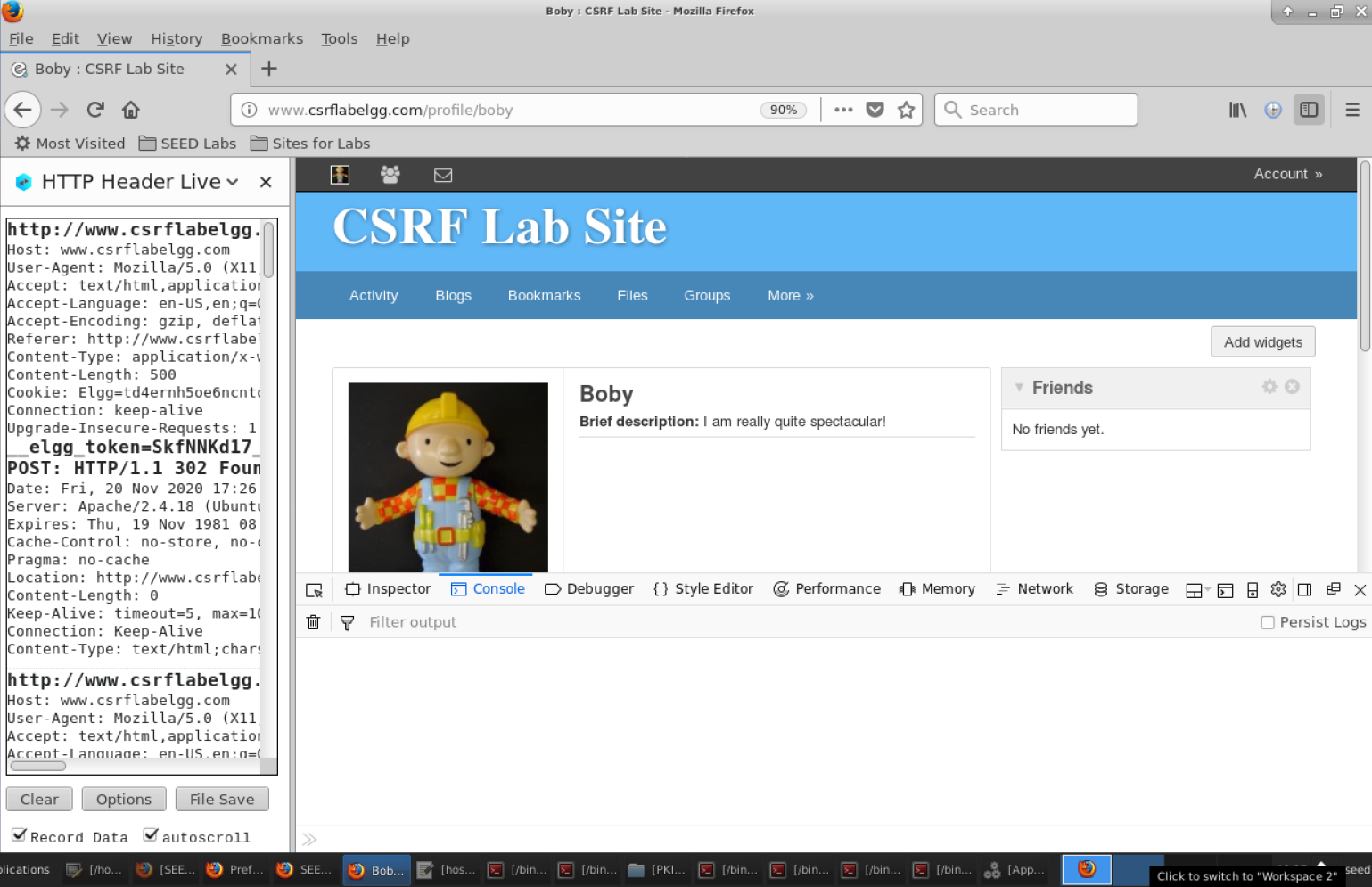
First I log out of Alice’s account and log into Boby’s account again as shown:



Now I go into Bobby’s profile and open up http header live, then write a description for myself(Boby) and click the save button, so as to see the resulting POST request as shown below:



Now I can see that the HTTP Header Live Tool was populated with POST requests as expected:



Now I click on this post request at the top of the tool’s window specifically, since this is the POST request which we can use as reference for Boby to create the malicious site and so as to change Alice’s description:



Once I click on this POST request and maximize the window, I can see that we are provided with necessary information, such as the token, the timestamp, and so on, as shown below:

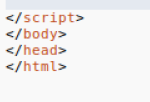


Now I simply open up a new terminal and type up the following commands as such, to edit the index file again:

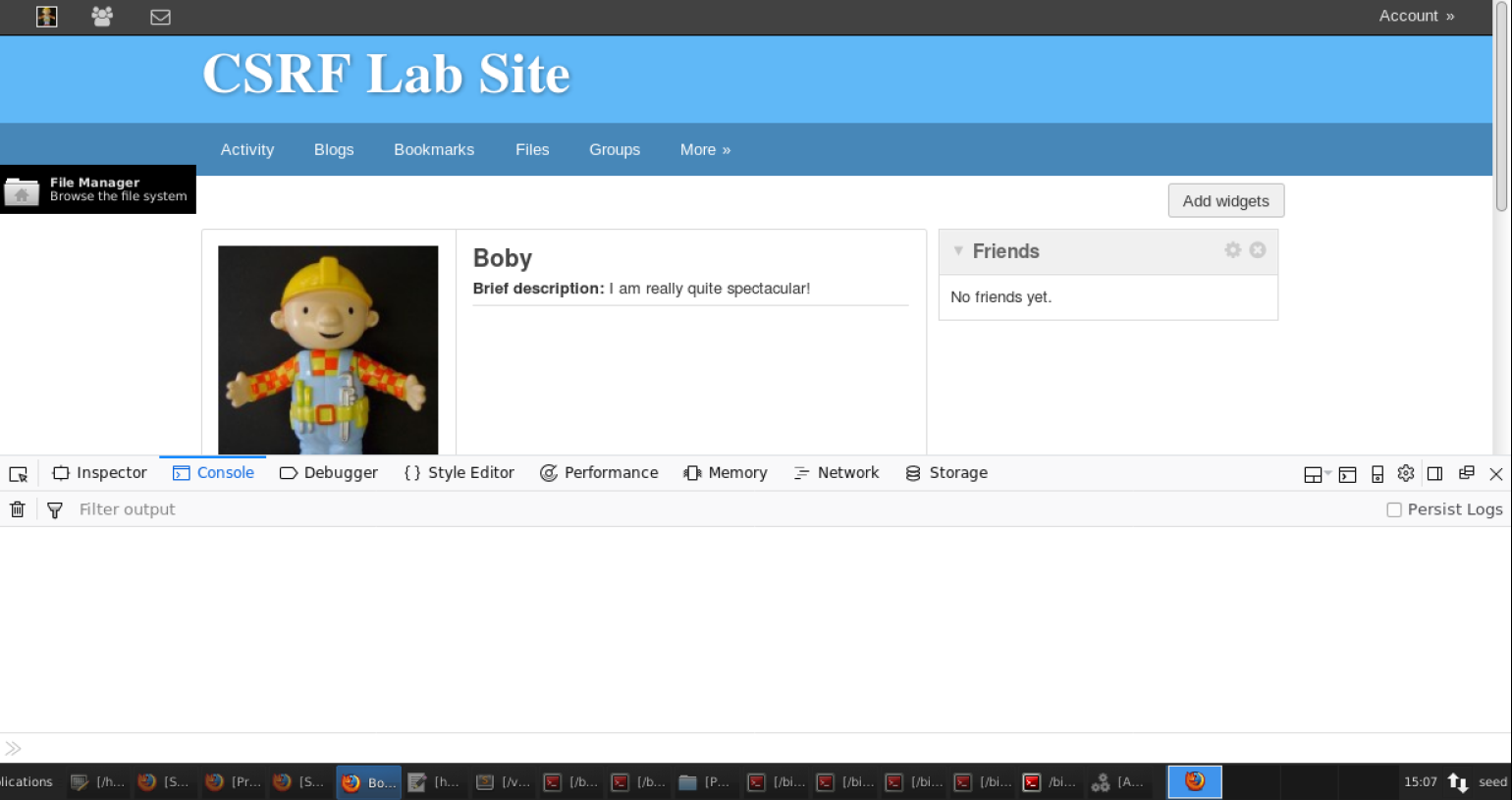


Then I edit the index.html file as shown below; this code is based on the above POST request:

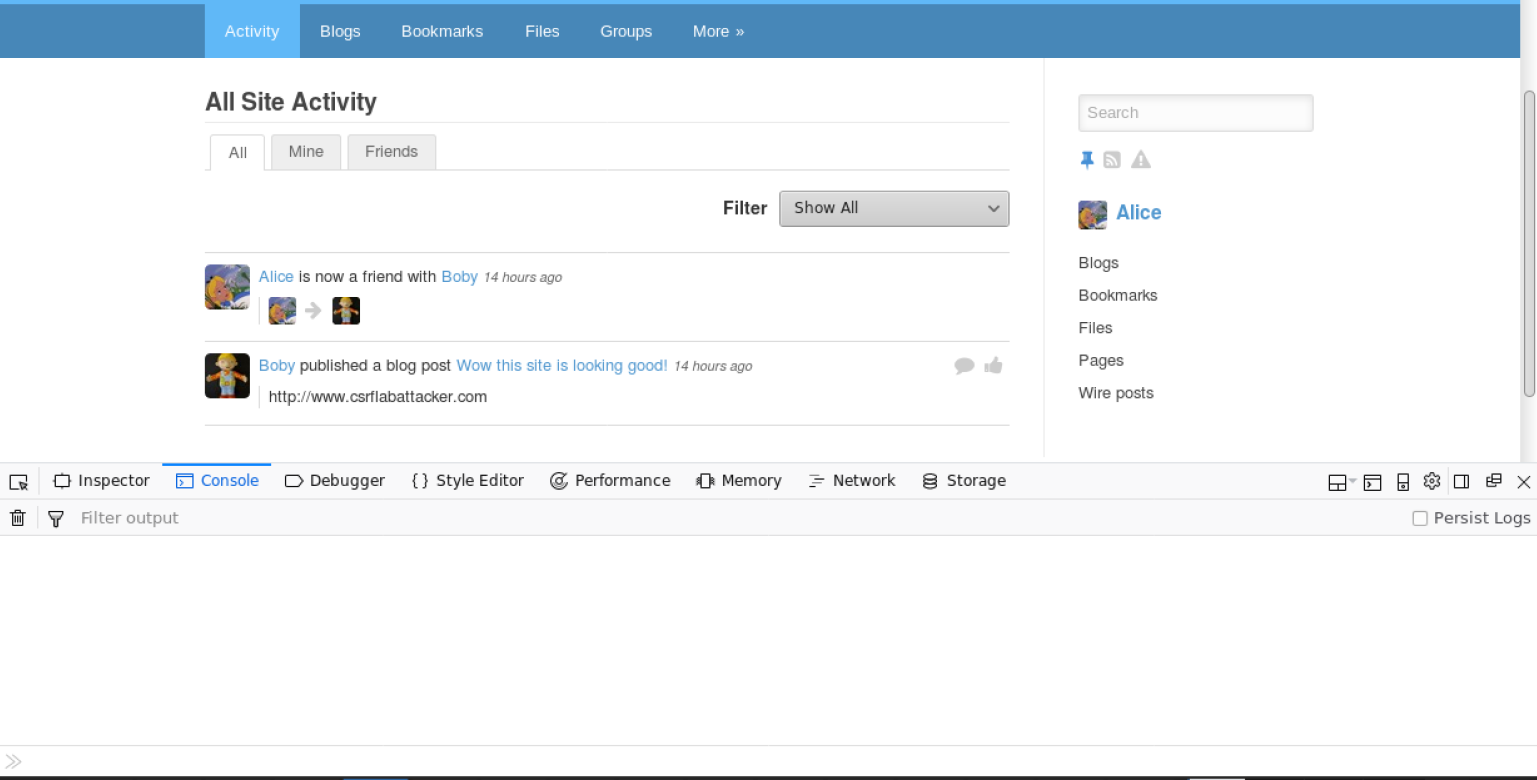




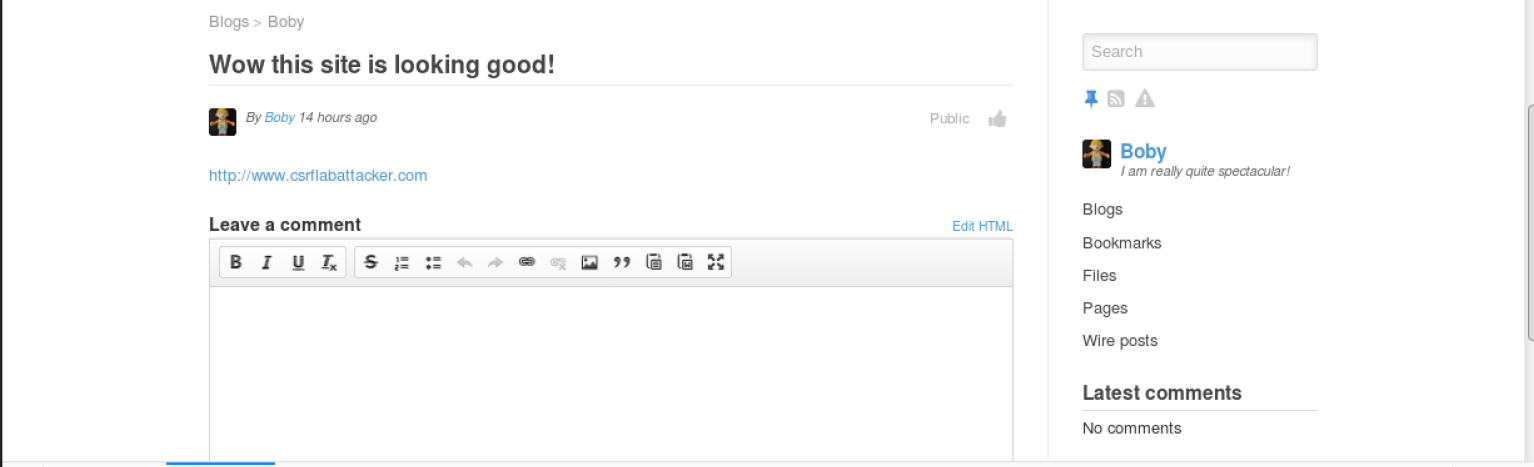
Then I save the file and go back to Firefox, where I was logged in as Boby, as seen below, and log out of his account:



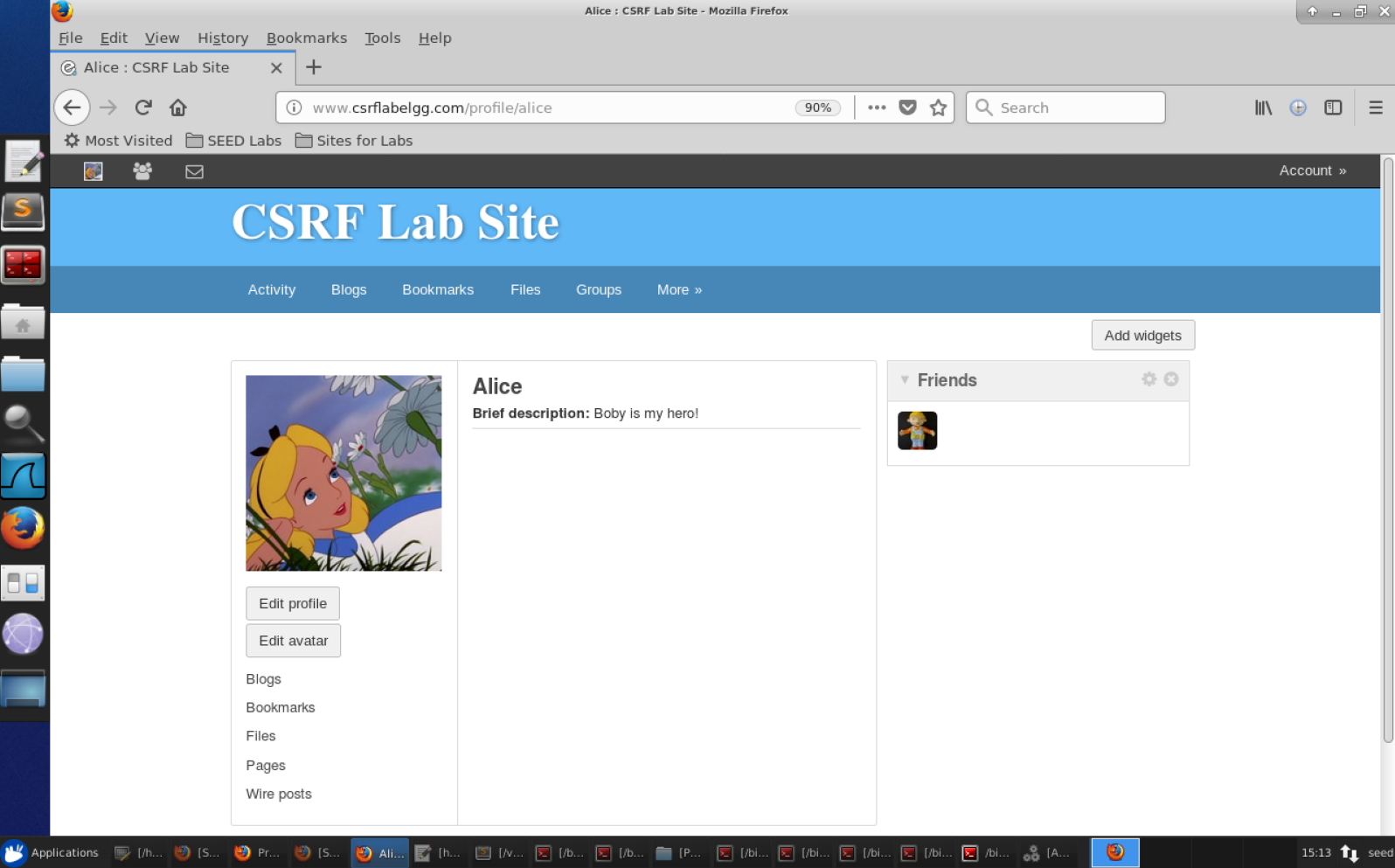
Once logged out, I log back in as Alice as shown below:



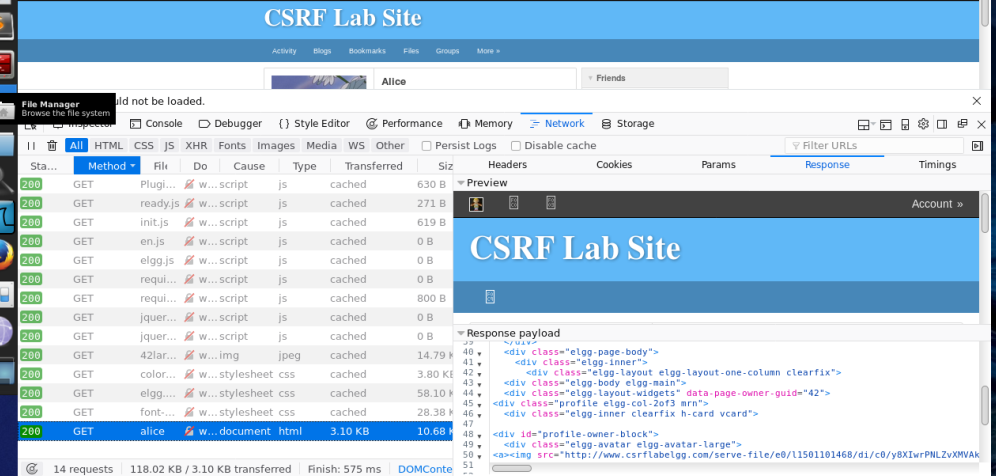
Then I click on the ‘Blogs’ tab, and click the link to Boby’s malicious site, shown below:



Now we see that the index file code was functioning correctly in that Alice’s profile was changed and her description to the description we specified in the source code, much to Alice’s dismay, as shown below:



Question 1: How do we get Alice’s user ID, guid, if Boby doesn’t know what Alice’s password is. What we did is that we used the Web Developer Tool as shown below when going into FireFox top right menu -> Web Developer -> Network. In order for Boby to be able to get Alice’s guid, he would have to go to Alice’s profile page, from his account. Activate the Web Developer’s Tool and then go into the HTTP GET request for Alic’s page as shown below on the left side. After that, Boby would comb through the code on the right until he would see “data-page-owner-guid = 42”. This is the guid of Alice that would be needed in order to launch a CSRF POST request attack.

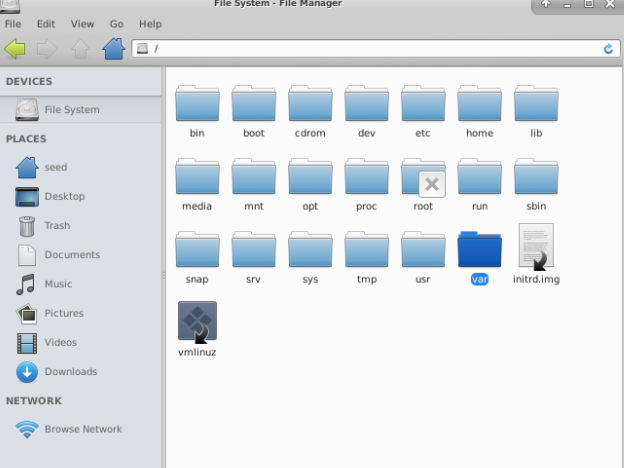
****

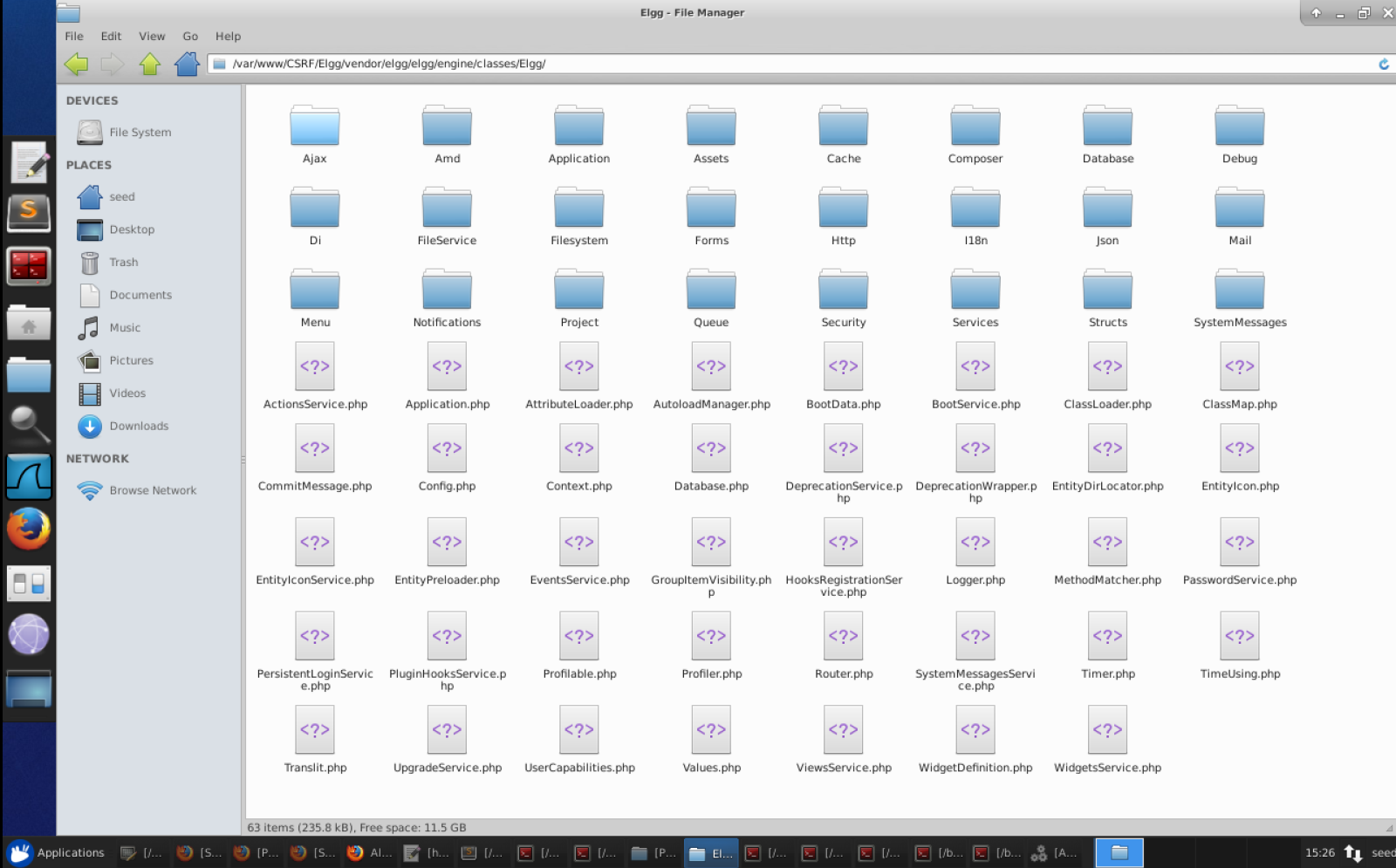
Question 2: Would Boby be able to launch the attack on anyone that visits his malicious website?

No, the reason why is because Boby would need to know the victim’s guid in order for the attack to be successful. Because Boby hard coded Alice’s guid in the Javascript program, it would only work if Alice clicks on the URL, but not for anyone else.

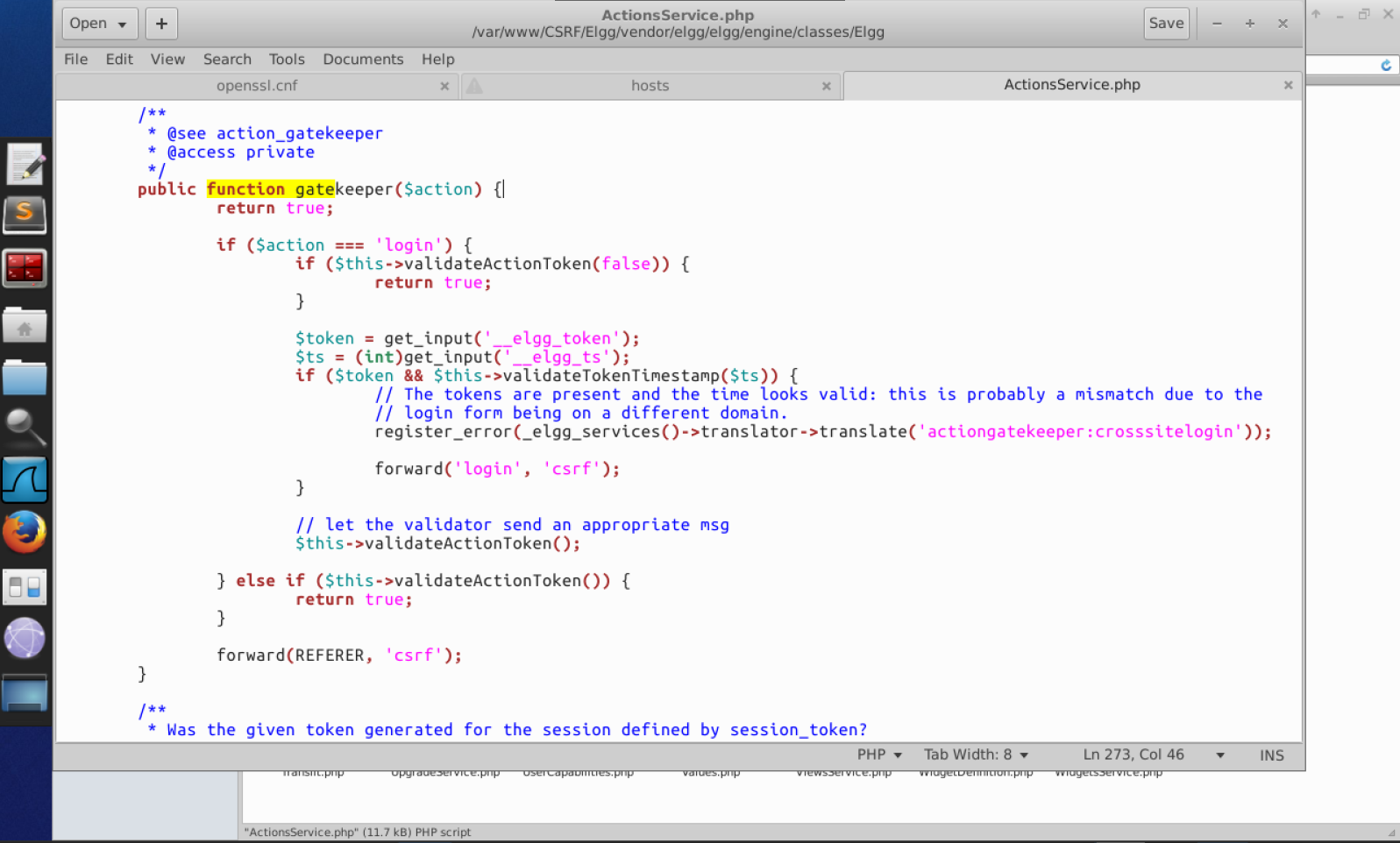
**Task 4: Implementing a countermeasure for Elgg**

Firstly, I go into files and follow the path to ActionService.php as shown below:





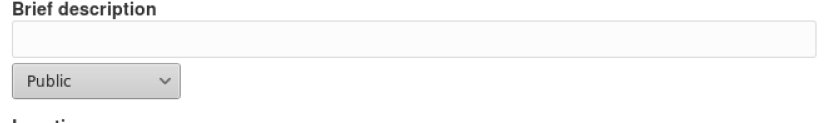
Then I open up the file ActionsService.php, and find the function ‘gatekeeper’, as shown below:



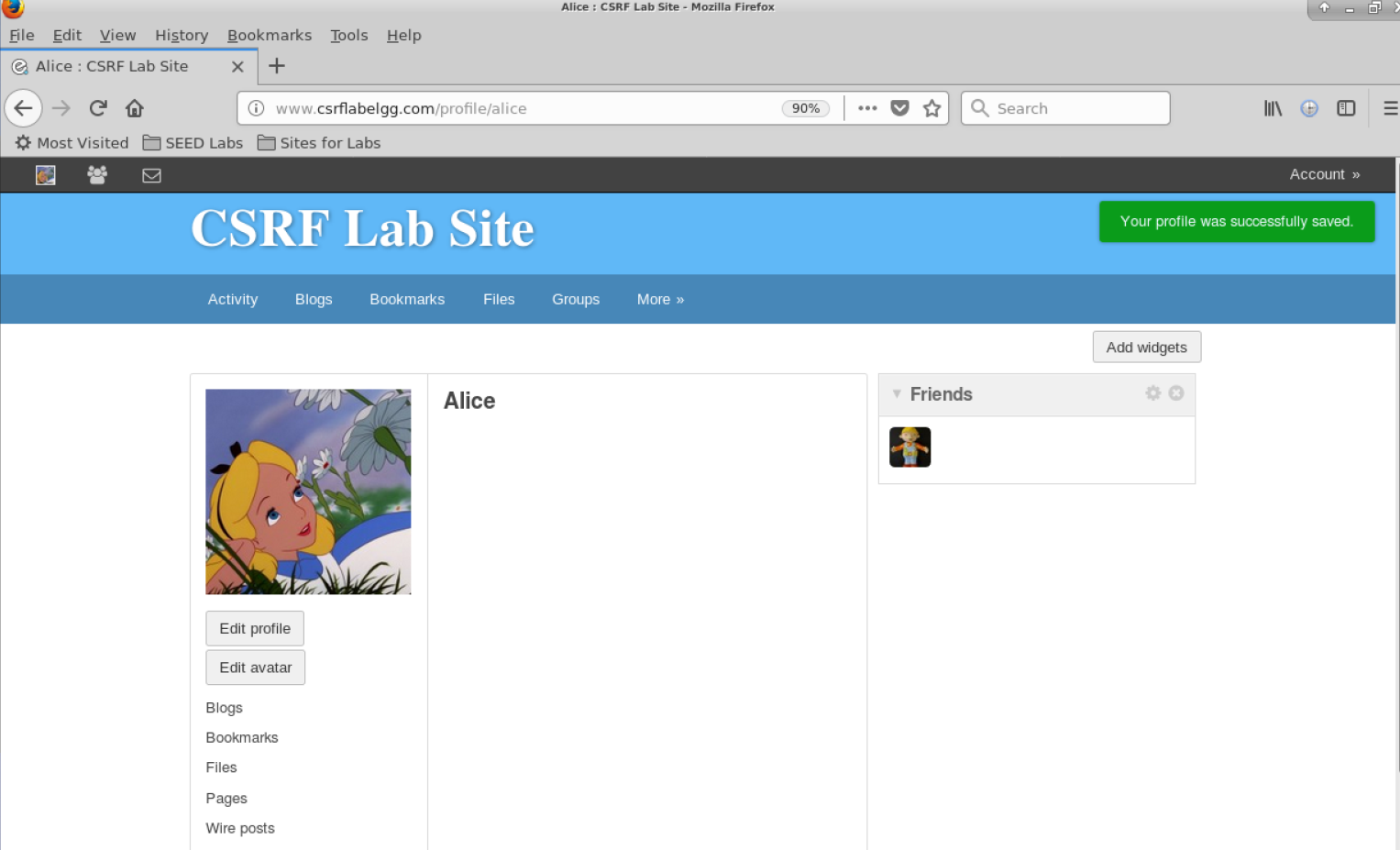
Then I comment the ‘return true’ line as shown below, and save the file using ‘File🡪Save’:

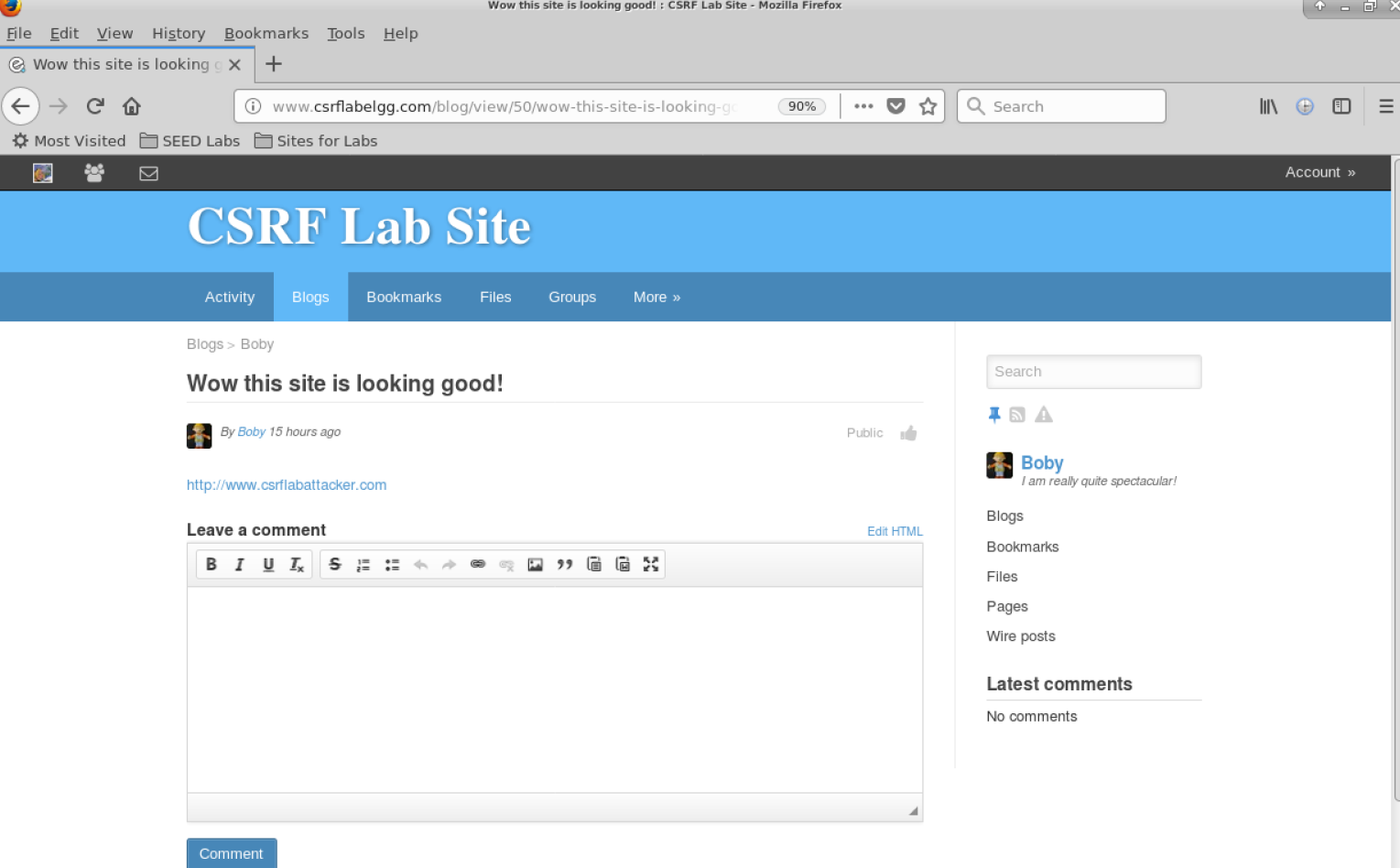


Then I go back to Firefox and run the POST request again, but first making sure to clear out Alice’s description again (Edit Profile) and saving as shown below:

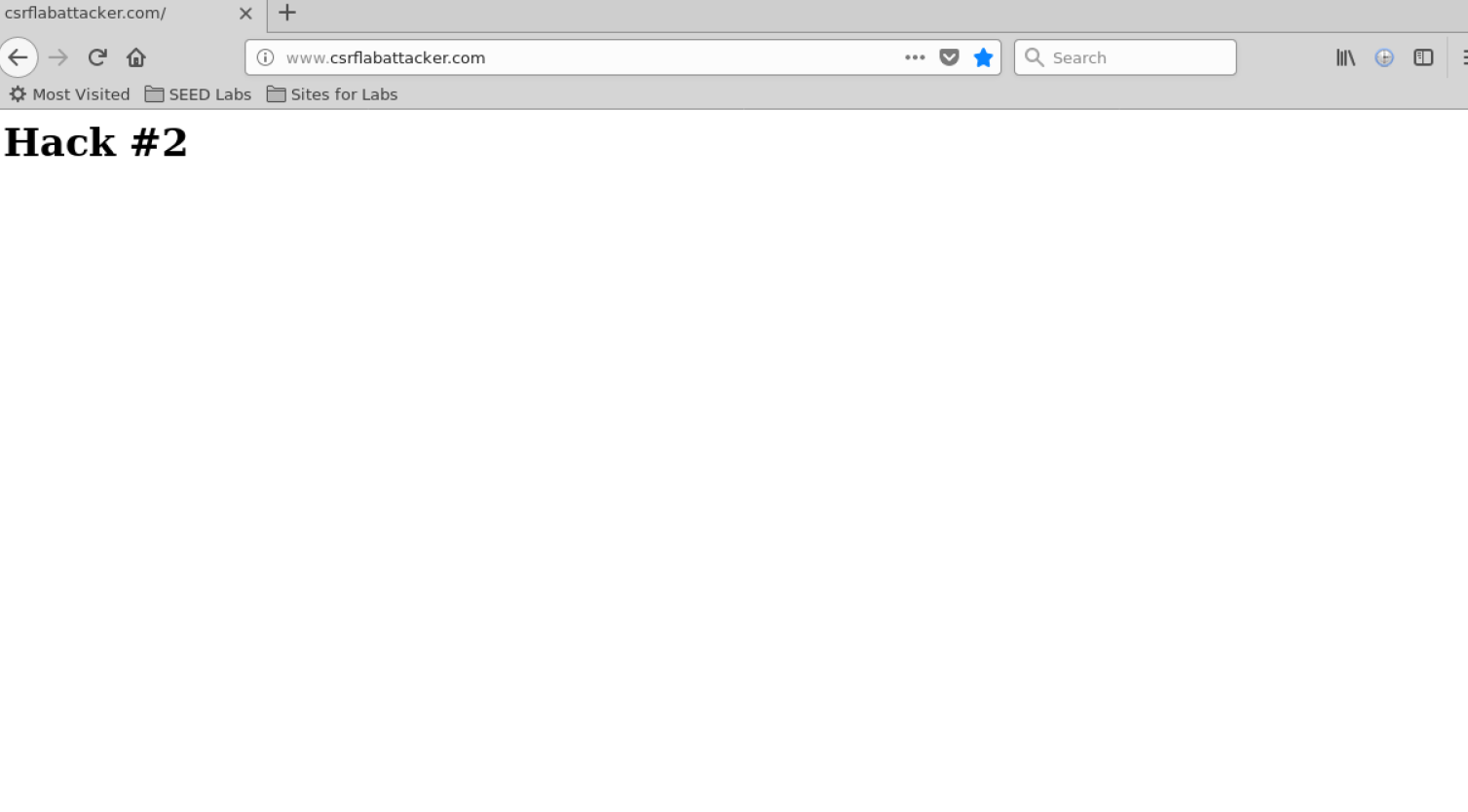


Now we see that the profile was successfully save with these setting as shown below:

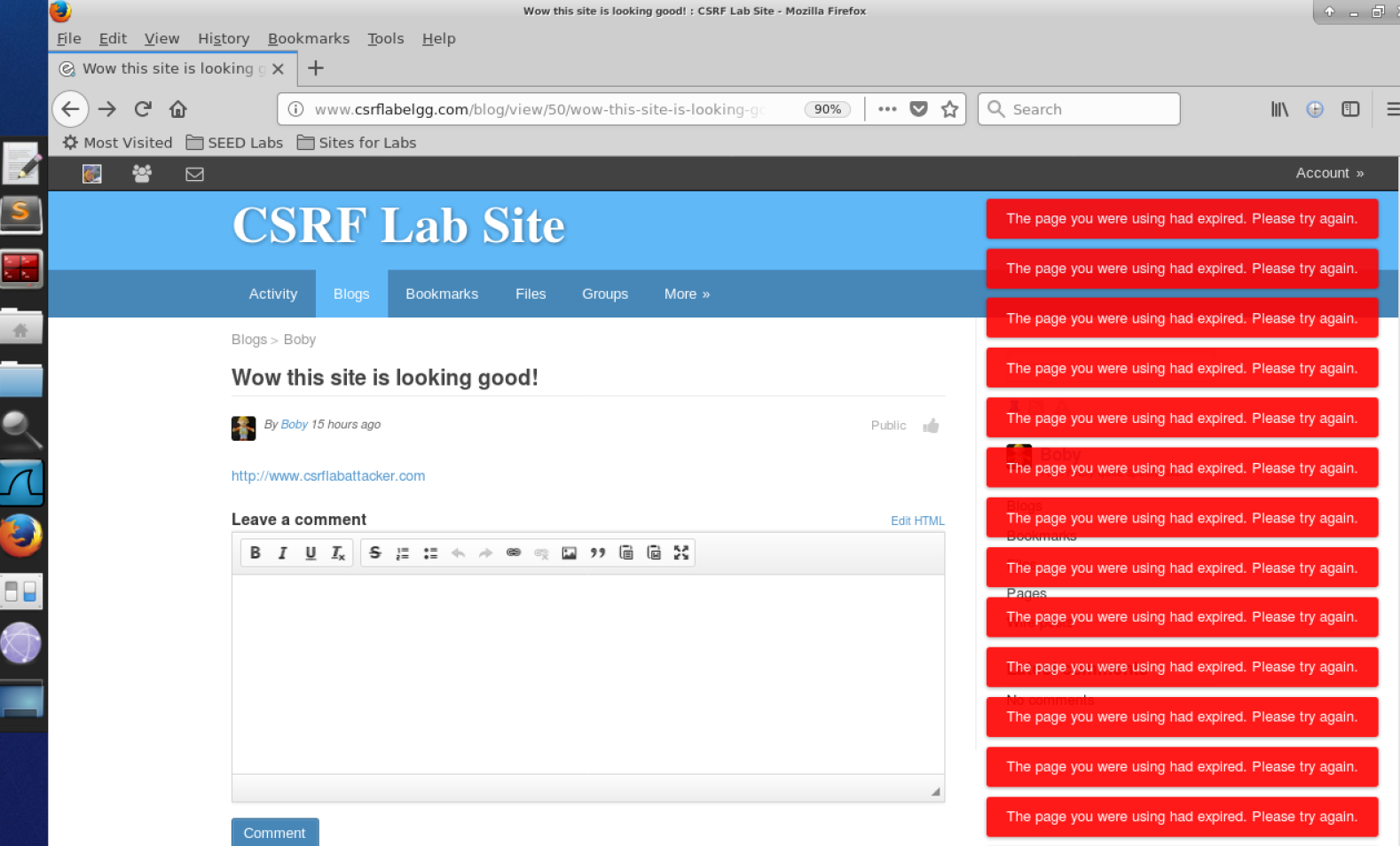
  
Then I go back to the ‘Blogs’ tab and click on Boby’s malicious site again as shown below:



After clicking on the link not only do we see that the code was executed successfully and displays what we specified in the browser as shown below:

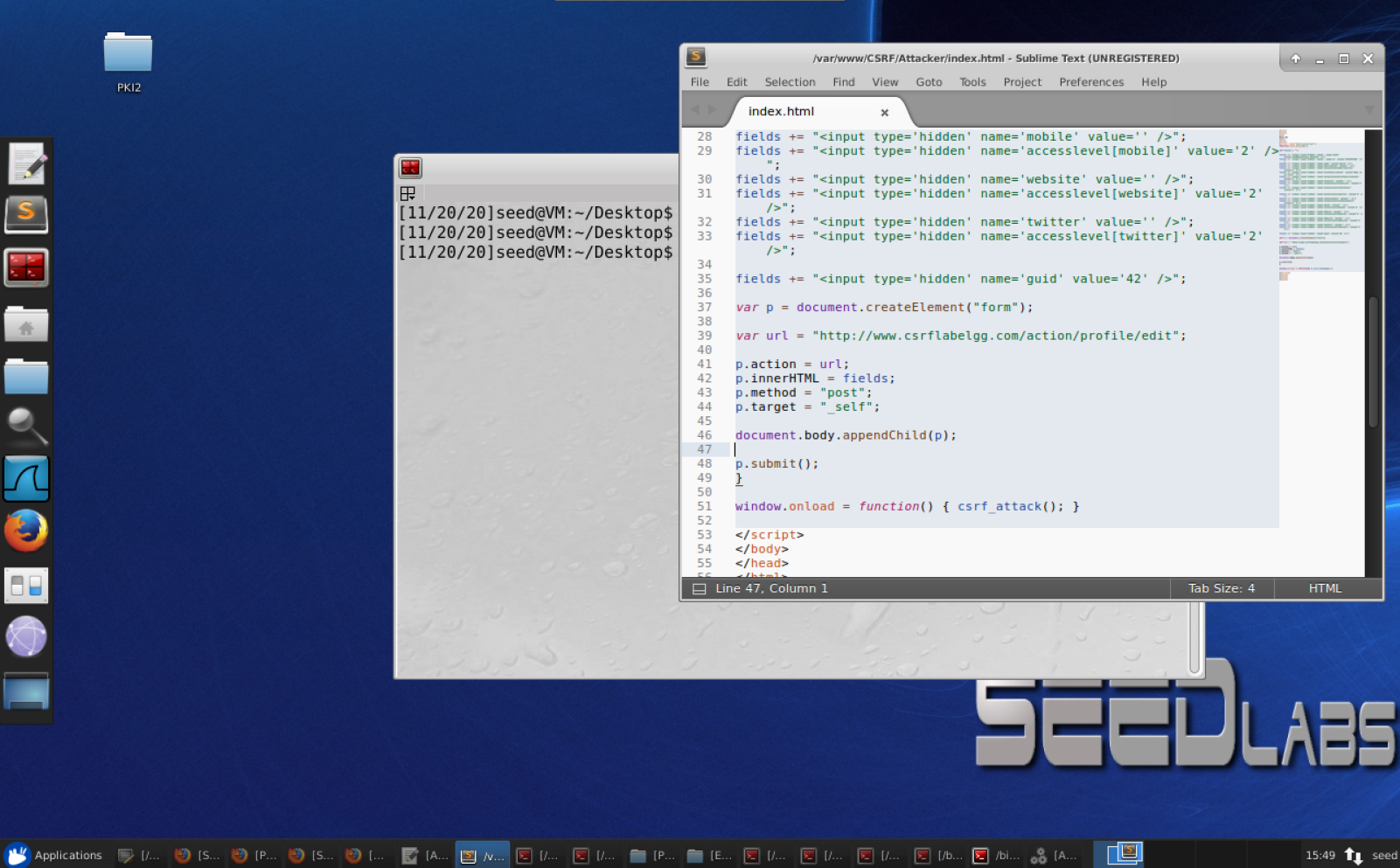


We also see these errors as seen below:

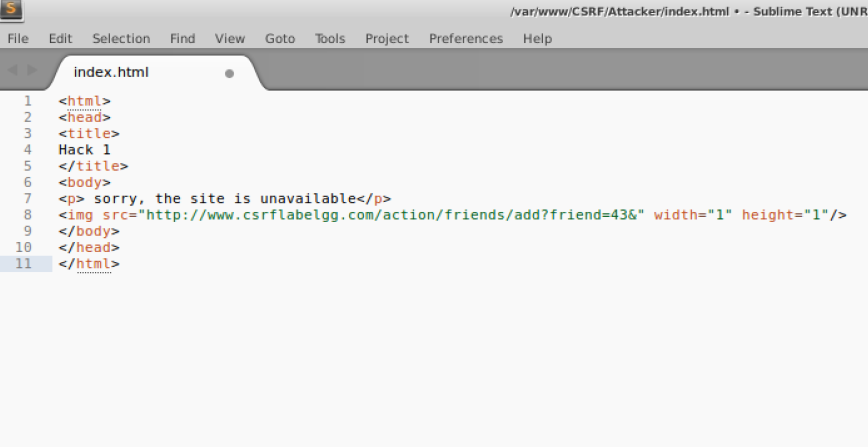


Now to do the ‘GET’ request first we go back into the index.html as shown below:

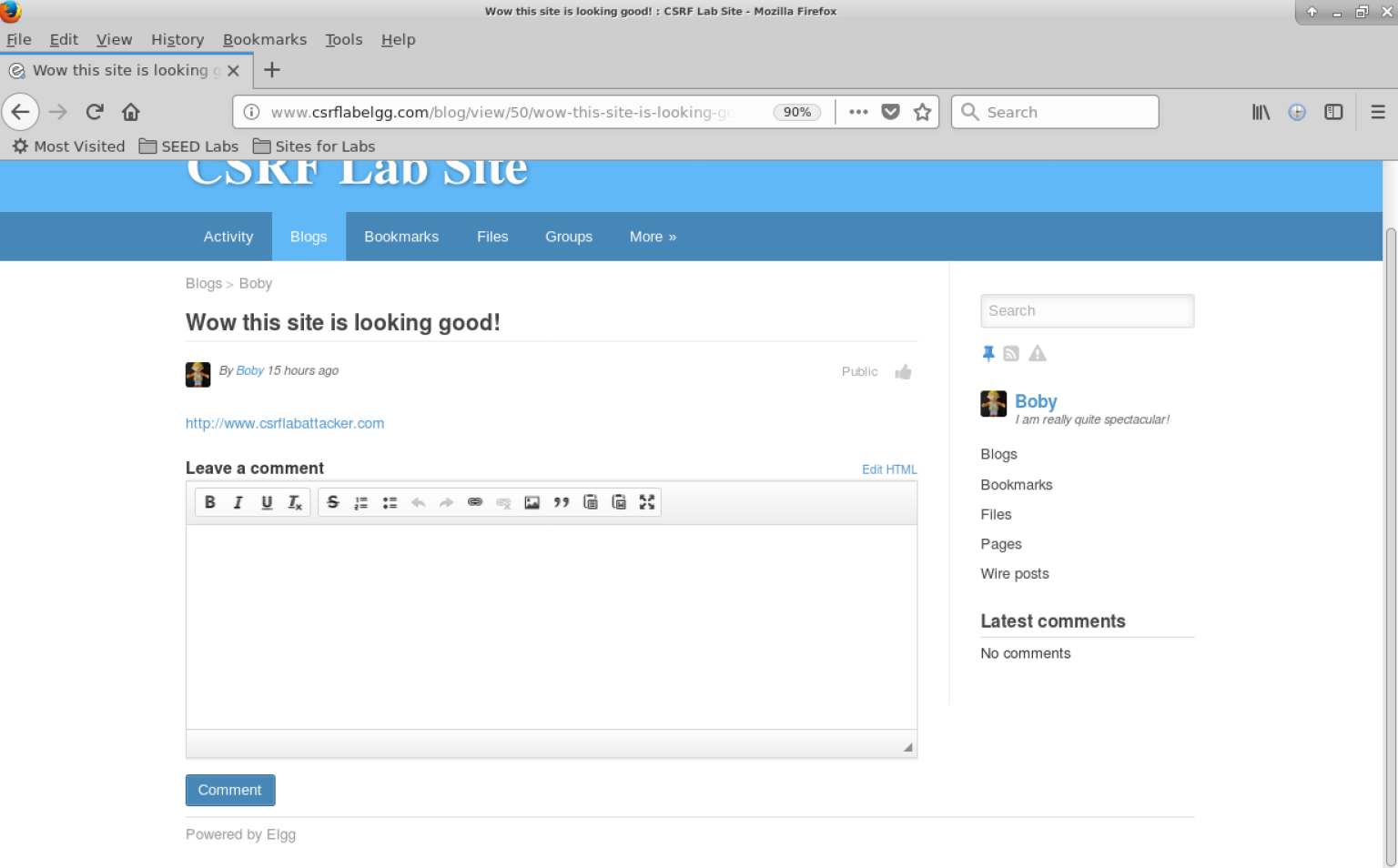




Then we put back the code that we used to do the GET request into index.html, as shown, below, and click ‘File🡪Save’ to save the code:



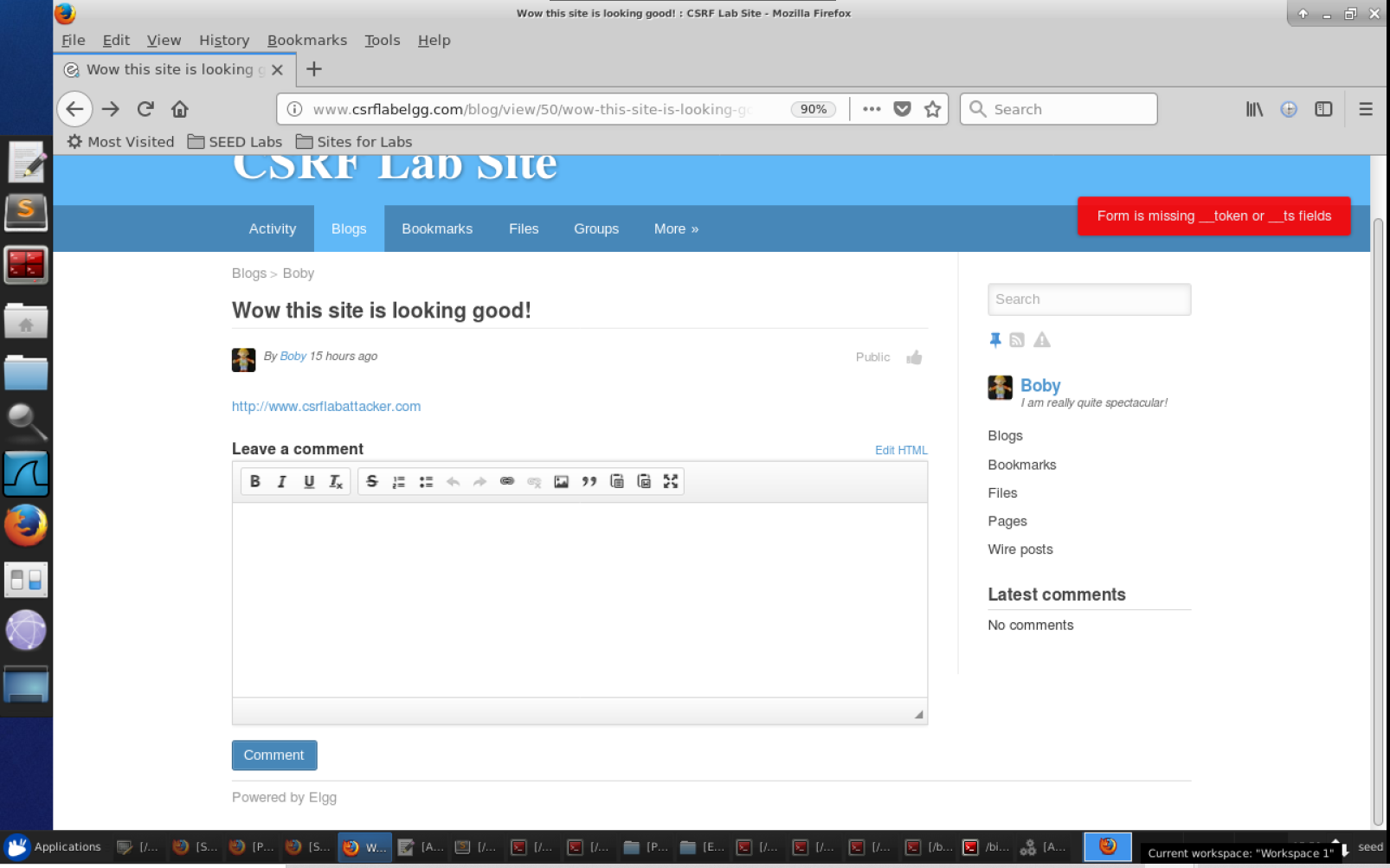
Then we go back to the elgg site and click the ‘Blogs’ tab and navigate back to Bobby’s malicious site post like earlier, as shown below:



After clicking on the link, we see the ‘site is unavailable’ message from the source code as shown below:



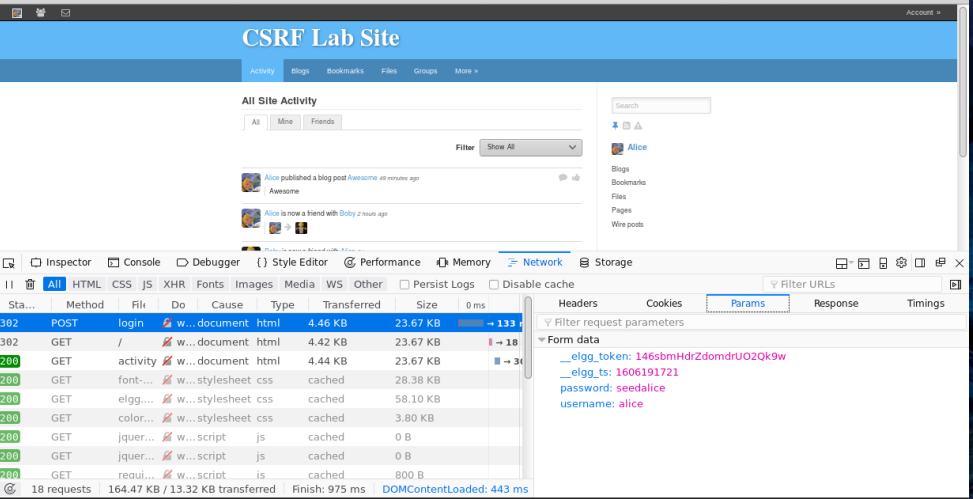
Then, if we click the back arrow to go back to the elgg site, we see the following error message on the top right corner of the screen as shown below:



This is because we are doing the GET request while the countermeasure is on, and the countermeasure requires the unique token and time stamp fields so that it may validate the information that goes with the session of the current user. Additionally, this shows that the countermeasure is functioning properly.

**Task: 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?**

After turning on the countermeasure in the gatekeeper function in ActionsService.php, we logged into Alice’s account on elgg. We then used Firefox’s inspection tool to find what the secret tokens are. We looked at the HTTP POST request in order to find out that there are now two parameters created when logging into a current session, \_\_elgg\_token and \_\_elgg\_ts, as shown below.



The attack was not able to send secret tokens in the CSRF attack since they do not know the values of the secret tokens and timestamp which are embedded in the page of the victim. The access control of the browser prevents the Javascript code in the attacker’s page from being able to access any of the content in the ‘elgg’ website’s pages. The security token makes this rather difficult for the attacker, as Elgg stores it as four individual pieces of information: the timestamp, the secret value, user session ID, as well as a random string, in other words it is stored as an MD5 digest. All of these are extremely difficult for the attacker to guess.