# Lab Assignment 8

**(Due your class date of July 21/16)**

## Background and Preparation.

In this lab, we will use three websites. The first website is the vulnerable Elgg site accessible at **www.seed-server.com**. The second website is the attacker's malicious web site that is used for attacking Elgg. This web site is accessible via **www.attacker32.com**. The third website is used for the defense tasks, and its hostname is **www.example32.com**. There websites are hosted in our simulated VMs. To enable the access, open your **/etc/hosts** file, add

```
10.9.0.5        www.seed-server.com
10.9.0.5        www.example32.com
10.9.0.105      www.attacker32.com
```

In the elgg VM (10.9.0.5), www.seed-server.com is hosted in **/var/www/elgg** while www.example32.com is hosted in **/var/www/defense**

In the attacker VM (10.9.0.105), www.attacker32.com is hosted in **/var/www/attacker**

As in other lectures, to set the VMs, do the following:

- Download Labsetup.zip from brightspace **directly through** VM firefox and unzip.
- In the folder Labsetup, you will see docker-compose.yml. That is the configuration file to set up the VMs. Then run $ **docker-compose duild** and then $ **docker-compose up**.
- Run **$ dockps** and you will see lines similar to the following:

```
[10/08/21]seed@VM:~$ dockps
bd2e0c3b273b  mysql-10.9.0.6
27b5134d1e4b  attacker-10.9.0.105
14b1f39950fc  elgg-10.9.0.5
```

- Then run **$ docksh 27** will give you a terminal of attacker VM. Other VMs are similar.

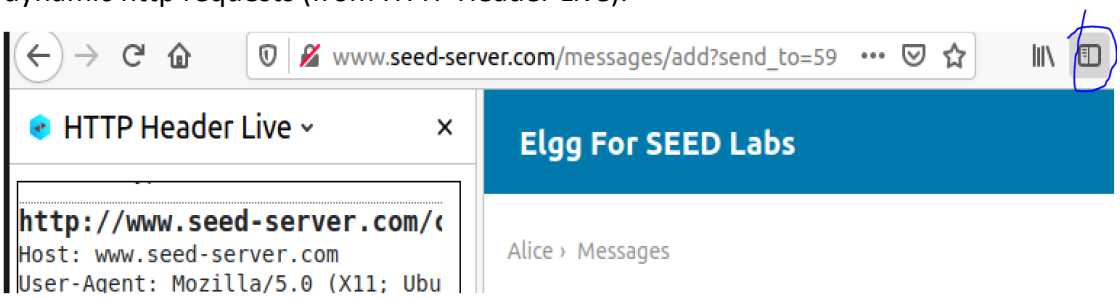
Then, on your firefox browser, access **www.seed-server.com** (the Elgg site). There are a few accounts already registered for the lab:

```
-----------------------------
UserName   | Password
-----------------------------
admin      |  seedelgg
alice      |  seedalice
boby       |  seedboby
charlie    |  seedcharlie
samy       |  seedsamy
-----------------------------
```

Please login alice's account and find out where to edit profile and where to add friends (see **members** entry and click on a member) and where to send/receive messages.

**P1.** In this problem, we will "help" Alice to add Samy to her friend list (without her consent). We will use http GET request to achieve. Follow the steps below.

1. **(Observe HTTP GET request code)** Use the method showed in class to watch the dynamic http requests (from HTTP Header Live).




2. Login Samy to find out the **guid** of samy (look the page source and search for **guid**).
3. In Samy's account, add friend Charlie and check HTTP Header Live to find out this add friend HTTP request link. Modify this link as a link for adding Samy as a friend (**ignore** the correct values for __elgg_ts and __elgg_token).
4. You go to your attacker's document root (where the attacker website is hosted) on 10.9.0.105 (attacker VM): /var/www/attacker. Use the method in class and the link in Step 3 to modify addfriend.html
5. In Samy's account, send an attractive message to alice including the link for addfriend.html (in Step 4).
6. Logout Samy and Login Alice's account and check one HTTP request. You should see the cookie item like this: Cookie: Elgg=0vso36eo6imvd9olosvfr6dtp5. This will enable Alice to send any other HTTP request (such as site update) securely (as attacker does not know this).
7. Alice checks her message from Samy and click the link in the message. You should see that Samy is now a friend of Alice. This is because the link is addfriend link and the cookie will automatically be added to the HTTP request (check this on HTTP live).

In the submission, please show the following:

- Your addfriend.html content
- Screenshot of Alice that Samy is a friend of her now after attack.
- Find out the HTTP GET request for adding Samy as a friend of Alice. Mark that Alice's cookie is automatically attached with this request.

**P2**. In this problem, we will "guide" Alice to automatically modify her profile that show Samy is hero (without her consent). We use HTTP POST request to achieve this. Follow the steps below.

1. Profile updating on Elgg is achieved through HTTP POST request. Through HTTP Header Live, check the HTTP POST request for profile update. Login Samy's account to check this. [**Note**: if you forge the request, the information you need to provide is POST command line and the content; others will be provided by the browser. In our case, the content is the filled profile edit form. ]

2. Construct a HTML program with Javascript function to implement the submit button of the above profile edit form. Remember to revise the guid to be alice's guid and change the brief description of profile as "Samy is MY HERO" (or something else you prefer).
3. Copy your html program to /var/www/attacker/editprofile.html
4. Login Samy's account and send an attractive message to Alice including the link:

   **www.attacker32.com/editprofile.html**
5. Login Alice's account and check the message from Samy and click on the link. Then, you should see that Alice's profile is now changed.

In the submission, you should provide the following:

- The content of editprofile.html
- Alice's modified profile.
- The HTTP POST request (by Alice's browser after clicking the link from Samy) to www.seed-server.com for the above profile revision.

**P3**. (**Counter Measure**: **Secret Token**) In this problem, we will study the counter measure for CSRF attack using secret token.

- The secret token is a pair: timestamp **__elgg_ts** and a security token **__elgg_token.** This pair has been added to every form in the elgg site (that needs an action). Normal HTTP request will also carry this secret token to the seed-server.com server. It will be verified to preserve the security (our attacks in P1 and P2 succeed because the verification is disabled). Check and copy a HTTP request from HTTP Header Live to verify that this secret token is carried to the server. For example, here is a friend-remove http request:

  GET http://www.seed-server.com/action/friends/remove?friend=58&__elgg_ts=1633752196&__elgg_token=

- The above secret token pair is generated in elgg folder of /var/www/elgg:

  **vendor/elgg/elgg/views/default/input/securitytoken.php**

  and added to elgg webpages.

```
$ts = time();
$token = elgg()->csrf->generateActionToken($ts);

echo elgg_view('input/hidden', ['name' => '__elgg_token', 'value' => $token]);
echo elgg_view('input/hidden', ['name' => '__elgg_ts', 'value' => $ts]);
```

  **token** is a HMAC value of secret key, timestamp, user ID and random string and is hard to guess by attacker.

- When a HTTP request (carrying this secret token pair) reaches the seed-server.com, it is eventually verified by

  **vendor/elgg/elgg/engine/classes/Elgg/Security/Csrf.php**

  based on **ts** and **token.** The partial code of verification code is as follows:

```
public function validate(Request $request) {
    return; // Added for SEED Labs (disabling the CSRF countermeasure)

    $token = $request->getParam('__elgg_token');
    $ts = $request->getParam('__elgg_ts');
    ... (code omitted) ...
}
```

The attacks in P1 and P2 are successful just because the verification returns about running the verification code. Please comment out the colored **return line**. Then run the edit-profile attack in P2. Verify that cookies is attached in the HTTP request but the secret token pair is not. So the verification can not pass. Provide the screen shot for this failed profile-edit.

P4 (counter measure: **samesite cookie**). Most browsers have now implemented a mechanism called SameSite cookie, which is a property associated with cookies. When sending out requests, browsers will check this property, and decide whether to attach the cookie in a cross-site request. A web application can set a cookie as SameSite if it does not want the cookie to be attached to cross-site requests.

To help students get an idea on how the SameSite cookies can help defend against CSTF attacks, we have created a website called **www.example32.com** on 10.9.0.5 (address mapping should be on /etc/hosts).

Once you have visited this website once, three cookies will be set on your browser, cookie-normal, cookie-lax, and cookie-strict. As indicated by the name, the first cookie is just a normal one, the second and third cookies are samesite cookies of two different types (Lax and Strict types). We have designed two sets of experiments to see which cookies will be attached when you send an HTTP request back to the server. Typically, all the cookies belonging to the server will be attached, but this is not the case if a cookie is a samesite type.

Do the following experiments. I have put the same testing.html on two different servers: **attacker32.com** (link B) and **example32.com** (link A). Testing.html has three different types of requests to www.example32.com/showcookies.php which simply displays the cookies sent by the browser. By looking at the display results, you can tell which cookies were sent by the browser. Please do the following:

- Please describe what you see with the screen shots on the displayed pages and explain why some cookies are not sent in certain scenarios.
- Please describe how you would use the SameSite cookie mechanism to help Elgg defend against CSRF attacks. You only need to describe general ideas, and there is no need to implement them.