Justin Gallagher

CSC 482

CSRF Lab

In this lab, we will exploit a cross-site request forgery (CSRF) vulnerability in a web application. 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 while visiting a malicious site. The malicious site injects an HTTP request for the trusted site into the victim user session, causing damages.

Task 1: Observing HTTP Request

In this task, we capture an HTTP GET request and a HTTP POST request. We do this to better understand what a legitimate HTTP request looks like and what parameters it uses so that we can successfully perform a CSRF attack. To do this, I used the Web Developer Network Tool provided by Firefox. To capture a GET request, I decided to make a search on the website given by the lab: 'http://www.csrflabelgg.com.' This will act as the vulnerable website and mimics a social media website. I open the Network tool in the same browser that the website is opened in, and to perform the GET request, I type 'search' in the search bar and click enter. After doing so, the Network tool filled with different web activity. The one we are interested in is the one pertaining to our search request. I find the HTTP GET request and we can see a few key things. The tool allows us to see the request URL, method, status, address, etc. There are also fields that pertain to data accepted by the browser, one that tells us whether cookies are sent and a few others. It also allows us to see the parameters of the GET request. In this case, there is a parameter 'q' defined as the search query (in our case the word 'search') and a search_type defined as 'All' because we are searching the entire database for our query. Next, we want to capture a POST request. The easiest way to do this would be to attempt to log in to the website because a login is a form and forms will send POST requests. I attempt to log in as the user 'hello' with the password 'pass'. Upon hitting the enter key, the Network tool again filled with website activity. I was able to find the POST request that was sent and looking into it we see a few things that differ from the GET request. The biggest different is the method used, POST. We also see fields 'content-length' and 'content-type' which do not show up in the GET request. These correspond with additional data being sent by the request. We can view this data in the parameter tab of the tool. The parameters used by the POST request are 'username' and 'password' and these two correspond with the entries we made ('hello' for username and 'pass' for password). Additionally, there are two other parameters called '_elgg_token' and '_elgg_ts'. These parameters act as a countermeasure to the CSRF attack and will be focused on later. Another thing to note, is that the GET request includes its parameters in the URL where the POST request includes them in the body of the request. This also explains the content length and type fields show up in the POST request but not in the GET request. The last difference I spotted between the two requests was the addition of the Referrer field within the POST request. This is another countermeasure to the CSRF attack as it lets us know whether the request comes from the same website or another one. I included pictures at the end of the lab that show the Network Tool's output for the GET and POST requests.

Task 2: CSRF Attack using GET Request

In this task, we want to add Boby as a friend of Alice by using the GET Request. We first need to observe the requests that tacke place when adding a friend on the website. I start by logging in to the user 'Charlie' using the username 'charlie' and the password 'seedcharlie'. Once logged in, I navigate to Boby's page and click on the 'Add Friend' button. While doing this, we observe for the request that pertains to the action of adding Boby and a friend of Charlie's. I notice a GET request pop up as soon as the button is pressed, and there are a few things to take not of. The request has 5 total parameters. Four of them are countermeasures to CSRF which are being ignored now, and the last parameter is 'friend' which is set to the value 43. With this value, we can assume Boby has a friend value of 43. We can confirm this by looking at the 'Inspect' tab of the Network tool to look at the source code of Boby's page. Here we see the line of code that includes: 'data-page-owner-guid="43"'. With this, we know that Boby has a GUID of 43. Now we can try to create a webpage that will add Boby as a friend to Alice as soon as Alice opens the webpage. I start by creating an HTML file called 'friendadd'. This will include the same request we saw in adding Charlie as a friend of Boby. I use the image tag in HTML because it sends a GET request upon loading the webpage. I also included a header tag with the word 'Success!' to see if I was successful. I saved the file in the /var/www/CSRF/Attacker folder, and then I navigated to the vulnerable website. In order to see if the attack works, we will need to sign into Alice's account because the target must have a valid session with the webpage for the CSRF attack to work. We do this by entering username 'alice' and password 'seedalice'. Once we are into Alice's account, I make sure she has no current friends. Then I click on the link that we created for Boby (assuming Alice would be the one to click this link). Upon doing so, the header I had included in the malicious HTML file pops up which means our attack was a success. We can confirm this by navigating to Alice's friend page and viewing her current friends. We see that Boby has been added to the list. We also see a GET request that pertains to our attack. This means that we have successfully performed the CSRF attack using the GET request.

Task 3: CSRF Attack using POST Request

In this task, we use the POST request in order to write an update to Alice's page pertaining to Boby. To do this, we will first need to see what type of request is sent while attempting to edit a profile. I start by accessing Boby's account using the username 'boby' and password 'seedboby'. I then click the 'Edit Account' button which navigates me to a screen where you can edit different parts of your profile. I change the Brief Description field to 'Boby is cool'. Before saving the changes made, I make sure the Network tool is running. We are looking out for a POST request made when we hit the 'Save Changes' button. Upon doing so, a POST request pops up and I click on it for further information. Looking at the Parameter tab, we see lots of parameters named 'accesslevel[]' with values equal to 2. This represents whether the edit to the profile is public or private, an option given on the edit profile screen. Next, we see the parameter called 'briefdescription' which is equal to 'Boby+is+cool'. This is the string we entered for the Breif Description field. We also see the GUID parameter valued at 43 as we would expect. A name parameter is also given and is equal to 'Boby'. With this information, we know that in order to edit Alice's profile using the POST request, we will need to know the value for accesslevel[], the

string we want to write to her account, her GUID, and her name. Three of these fields are already known, value of accesslevel[] is 2, the string we want to write is “Boby is my hero” and the name of the victim, ‘Alice’. To find out Alice’s GUID, we will make a search in the search bar from Boby’s profile. Then we observe the source code, and looking into it we see the line that contains the following: ‘id=”elgg_user_42” class=”elgg_item”’. With this information, we can assume that Alice’s GUID is 42. Next, we create the HTML file that will be used in the attack. I call this file ‘editprofile.html’ and it is stored at the location ‘/var/www/CSRF/Attacker’. In order to generate the request from our webpage, we need to use JavaScript. Shellcode is provided, and modifications need to be made for it to work properly. We input the values we obtained by searching for Alice’s account and by using information from the POST request for edit profile (GUID, name, briefdescription, and accesslevel[]) and save the file. To test if the attack works, we log into Alice’s account and see that she currently has no brief description. Then we click on the link as if Boby had maliciously provided it to us. Upon doing so, we check back to see if the brief description has updated. It now reads ‘Boby is my hero’ so this proves our attack was a success. We also see a POST request that pertains to our attack. This means that we have successfully performed the CSRF attack using the POST request. We also have a few questions to answer in this lab. They are listed below:

1. The forged HTTP request needs Alice’s user id (guid) to work properly. If Boby targets Alice specifically, before the attack, he can find ways to get Alice’s user id. Boby does not know Alice’s Elgg password, so he cannot log into Alice’s account to get the information. Please describe how Boby can solve this problem.
    - Boby can solve this problem like the way we did in the lab. Performing a search on Alice will result in the source code telling us what Alice’s GUID is. If the website did not contain this information in its source code, then we would have to attempt logging in to Alice’s account multiple times with a random password while observing the requests with the hope that one contains Alice’s GUID.
2. If Boby would like to launch the attack to anybody who visits his malicious web page. In this case, he does not know who is visiting the web page beforehand. Can he still launch the CSRF attack to modify the victim’s Elgg profile? Please explain.
    - Boby could not, because his website is different than the one he is attacking. The GUID is only sent on that site's server, meaning that this information will never be sent to an outside website. This makes it impossible for Boby to do.

Task 4: CSRF Attack using POST Request

In the final task, we enable the countermeasure to CSRF by commenting out the ‘return True’ statement inside the ‘gatekeeper()’ function of the ‘ActionServices.php’ file In the ‘/var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg’ directory. After doing so, we try to perform the same attacks from tasks 2 and 3. We login to Alice’s account and navigate to her homepage. We then try to open the webpage that Boby would have sent from task 2 in order to add himself to Alice’s friend list. After opening the link, we see that we get an error for our GET request. The error is ‘form is missing __token or __ts field’. This is because the timestamp and token fields are missing thus the attack is unsuccessful. Looking at the GET request, we see that these values are empty. This is because we have not defined values for these fields. This shows that the attack is unsuccessful

against the countermeasures using the GET request. We now try to open the webpage from task 3 that adds a brief description to Alice's profile pertaining to Boby. Upon doing so, we get the same error as before but this time we get it multiple times. This is because we have not defined the timestamp or token fields. This shows that the attack is unsuccessful against the countermeasures using the POST request. Had we provided these parameters in our 'addfriend.html' or 'editprofile.html' files then we would have been successful. But this information is not sent in the HTTP request because the request comes from the attacker's webpage to the victim's server. Since these parameters are only set on the victim's server, they will never be sent to any other servers. Likewise, they cannot be changed by incoming requests. The only way to figure out these values is to have the valid login information for the user whose information you need.