Project 4:

Web Security Report Entry

Fall 2020

Task 1 – Warm Up Exercises

Activity 1 - The Inspector & Console tabs

1. What is the value of the 'CanYouSeeMe' input? *Do not include quotes in your answer.*
   TheCakeIsALie

2. The page references a single JavaScript file in a script tag. Name this file including the file extension. *Do not include the path, just the file and extension. Ex: "ajavascriptfile.js".*
   cs6035.js

3. The script file has a JavaScript function named 'runme'. Use the console to execute this function. What is the output that shows up in the console?
   42

Activity 2 - Network Tab

1. What request method (http verb) was used in the request to the server?
   post

2. What status code did the server return? *Include both the code and description. Ex: "200 Ok"*

418 I'm a teapot

3. The server returned a cookie named 'coffee' for the browser to store. What is the value of this cookie? *Do not include quotes in your answer.*
   With_Cream

Activity 3 - Built-in browser protections

1. You can do more than just echo back text. Construct a URL such that a JavaScript alert dialog appears with the text cs6035 on the screen. Upload **activity3.html** and paste in a screenshot of the page with the dialog as your answer below. Be sure to include the URL of the browser in your screenshot.

   http://cs6035-warmup.gatech.edu:5000/tools/echo?payload=<script>alert('cs6035')<\/script>

Activity 4 - Submitting forms

1. Copy and paste below the entire output message you see and submit that as your answer to this activity. Upload **activity4.html** which is the form that you constructed.
   Congratulations!, you've successfully finished this activity. The answer is Birthday Cake

Activity 5 - Accessing the DOM with JavaScript

1. Upload **activity5.html** which is the form that you constructed. No other answers are required for this activity.

Task 5 – Epilogue Questions

Target 1 Epilogue

1. List the PHP page and lines that should be changed to fix the vulnerability.
   The page was account.php and the vulnerable lines were the account and routing number input lines and the xsrf token input line too. Lines 63, 65, 66.

2. Describe in detail why the code listed in the line numbers above are vulnerable. You're free to use generalized concepts to help show your understanding but we also need to know details that pertain to this target and assignment. A definition of XSRF is not what we're looking for.

   Those lines are vulnerable because it allows the attacker to use any account and routing number they want and save it as the user's new account info. Line 66 is vulnerable because it allows an attacker to go into another page without passing along a session token. In this case, the page stores the token in a cookie and the next page it goes to just grabs the session token from the cookie. For example, in auth.php, the line <?php echo $_SESSION['csrf_token'] ?>, it uses php sessions which is similar to cookies as it only persists during the active session, if a session has been

opened already and the client opens another php file that uses the session, all the values in that session will be passed along. Now if crsf_token is not available in the session, it means an user has never logged on before. If it is available, any php file with need for that token can use that available session. The php session, like a cookie persists across the pages during a session and allows an attacker to login any page that uses that session token and authenticates them.

3. Explanation of how to fix the code. Feel free to include snippets and examples. Be detailed!

   The code can be fixed by not setting the session token in the cookies or php session but rather passing them as another hidden input as part of the form. This way, an attacker will have a harder time trying to find out what the token value is and will be harder for them to get or post a page. Another option is to use the same-site attribute for the cookies (Kirsten, S.). If set, this attribute can prevent a cookie being sent along if the request is of a third party. If someone tries to access a website page without going directly through the website, the cookie wouldn't be passed along therefore preventing the user from being authenticated and logging in. The attribute can be assigned the values of "lax", "strict", or "none" (MDN). Setting the attribute to "strict" can prevent this XSRF attack. This attribute can be set when the cookie is first created.

Target 2 Epilogue

1. List the PHP page and lines that should be changed to fix the vulnerability.
   The vulnerability is in the index.php file and line 34.

2. Describe in detail why the code listed in the line numbers above are vulnerable. You're free to use generalized concepts to help show your understanding but we also need to know details that pertain to this target and assignment. A definition of XSS is not what we're looking for.

   The input uses php code to grab the login name that was posted to the client side by the server. An attacker can inject script code easily by escaping the quotes early and closing the input tag, so the login value is just seen as "". Once the input tag has been closed, the attacker is free to add any sort of code they want the page to run. For example, <input value="<? echo @$_POST['login'] ?>" /> , in our post request for the field login we can input ""/> <br class="" and the input tag would look like <input value=""/><br class=""/> without any major differences in the webpage ui. An attacker can input a script in between the input and br tags and have the script perform malicious acts.

3. Explanation of how to fix the code. Feel free to include snippets and examples. Be detailed!
   a. Be careful with your explanation here. There are wrong ways to fix this vulnerability. Hint: Never write your own crypto algorithms. This concept extends to XSS sanitization.
   b. Warning: Removing site functionality will not be accepted here.

Filtering exists already, but the filtering can be more robust. The existing filter only checks for SQL commands to SQL injections. It can be extended to look for any closing tags such as "/>" and deem the login input no good. So instead of filtering that particular string out, just let the user know that the login input is no good. Also instead of writing your own filtering and sanitizing function, use an existing library that can do that already (OWASP). This existing library should be able to filter the input better than the one created. The filtering should remove any instances of characters that can cause an XSS attack. Always treat user input as bad and potentially harmful to your webpage. With this mindset, it'll help develop code that tries to prevent this situation.

Target 3 Epilogue

1. List the PHP page and lines that should be changed to fix the vulnerability.
   The auth.php file and the lines are 30-51 and line 68.

2. Describe in detail why the code listed in the line numbers above are vulnerable. You're free to use generalized concepts to help show your understanding but we also need to know details that pertain to this target and assignment. A definition of SQL Injection is not what we're looking for.
   The lines 30-51 make up the sqli_filter function in the auth.php file. This function is vulnerable because it doesn't work correctly enough. It has strings to filter out, but it filters it in a linear fashion making it possible for an attacker to hide an SQL injection attack. The attacker can find the order of the filtering and then include the command you want to run and then put the filtered commands inside the command you want to run. So for example, the filtering function filters "--" before "||", so if we input "-||-" it would skip that "--" filter because that doesn't exist yet, then it gets to filter the "||" line and it'll remove that and leave the input with the "--" SQL command which will just comment out everything after it. Line 68 was vulnerable because that's where the sql command to get the user is and the SQL injection value would allow the attacker to get the user info without needing a password. So the command SELECT * FROM users WHERE eid='username' AND password='hash'; can be escaped with SELECT * FROM users WHERE eid='username'--AND password='hash'; if the username is inputted as username'--. Everything after -- becomes a comment.

3. Explanation of how to fix the code. Feel free to include snippets and examples. Be detailed!
    a. Be careful with your explanation here. There are wrong ways to fix this vulnerability. Hint: Never write your own crypto algorithms. This concept extends to SQL sanitization.

The code can be fixed by using an existing library that has a SQL sanitization method (OWASP). The existing sqli_filter function that is being used is no good because the filtering is done linearly. Meaning that there is a sequence of filters that is used in order, one after another. This makes it vulnerable as mentioned before, an attacker can hide the SQL attack by using the filtered inputs as a decoy to get the SQL command the attacker really wants after the filtering. Another way to fix code is to parameterize the SQL command (OWASP). Parameterizing meaning using variables with prepared SQL commands, meaning that all the SQL command is defined first before inputting the user inputs or inputs to complete the command. When you parameterize the SQL query, the user inputs won't escape any SQL query. That is due to the user input being taken at face value and used as such. For example, like in this task if the user inputs "username'--", without parameterizing the SQL command, this input can comment out everything after the command gets the username. If we parameterize it, the SQL command would find a username that has the string value of "username'--".

Additional Targets

1. Describe any two additional issues (they need not be code issues) that create security holes in the site.

   In the first task, there was a hidden input with name response that uses the challenge, account and routing inputs. The response is calculated from those three values, without that response input, the user or attacker can't save the updated information. However, if that is the only input that is incorrect, the page shows a banner that prints out the correct value for response. The attacker can take that value and input it into their attack and successfully update the information.

   Another issue is in the account.php file. Lines 63, 65, 66 have additional issues besides being vulnerable to a CSRF attack. It is also vulnerable to a XSS attack in similar fashion to task 2. In each of those inputs, an attacker can input a script that can do malicious things. In those lines the php code just echos the accounting and routing variables, so an attacker can write scripts that inserts itself in between those input tags.

2. Provide an explanation of how to safely fix the identified issues. Feel free to include snippets and examples. Be detailed!

   The first issue can simply be fixed by not showing that hidden response input value in the banner, and simply say that the information couldn't be updated. It is bad practice to show the user the client side error or any too specific error, that can lead to potential attacks as in task 1. Show some generic error in the banner and an attacker will have a harder time in figuring out the response value.

   For the second issue, treat all user inputs as malicious and sanitize or filter the input so that all malicious commands or symbols are filtered out before using the inputs (OWASP). Instead of writing your own filtering function, use an existing library that has a filtering function to handle the

user inputs. Using that should help prevent any malicious attempts to attack a webpage using XSS. Another way is since the account and routing inputs are to be numbers, a check can be done to check that inputs are numbers and not of a different type. In addition, a character length limit can be implemented to prevent long strings that can contain scripts. The shorter the input, the harder it is to input a script as an input. Using these methods should help prevent an XSS attack on those inputs.

Works Cited

1. **Kirsten, S., Cross Site Request Forgery (CSRF),** https://owasp.org/www-community/attacks/csrf
2. **MDN, SameSite cookies,** https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite#Values
3. **Kirsten, S., Cross Site Scripting (XSS),** https://owasp.org/www-community/attacks/xss/
4. **OWASP, CheatSheet Series Team, Cross Site Scripting Prevention Cheat Sheet,** https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
5. **OWASP, CheatSheet Series Team, SQL Injection Prevention Cheat Sheet,** https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

When complete, please save this form as a PDF and submit with your HTML files as "report.pdf". Do not zip up anything!