**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

1. 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”.*

/js/cs6035.js

1. 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?

I’m a teapot

**Activity 2 - Network Tab**

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

POST

1. What status code did the server return? *Include both the code and description. Ex: “200 Ok”*

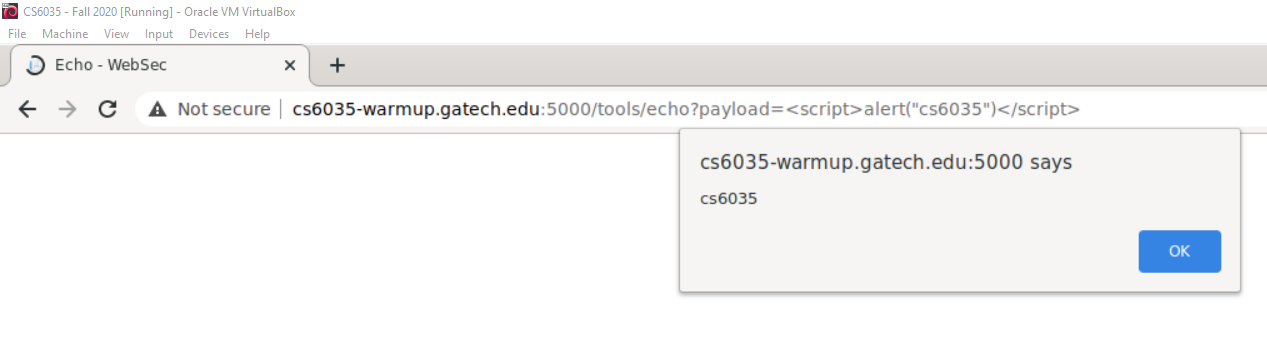
418 I’m a teapot

1. 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.



**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.

Within the file account.php, the vulnerability appears in lines 20-33, which contains the following code:

else {

// verify CSRF protection

$expected = 1;

$teststr = $\_POST['account'].$\_POST['challenge'].$\_POST['routing'];

for ($i = 0; $i < strlen($teststr); $i++) {

$expected = (13337 \* $expected + ord($teststr[$i])) % 100000;

}

if ($\_POST['response'] != $expected) {

notify('CSRF attempt prevented!'.$teststr.'--'.$\_POST['response'].' != '.$expected, -1);

} else {

$accounting = ($\_POST['account']).':'.($\_POST['routing']);

$db->query("UPDATE users SET accounting='$accounting' WHERE user\_id='".$auth->user\_id()."'");

notify('Changes saved');

}

1. 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.

The vulnerability that is present in the lines of code above is within the CSRF protection that is implemented within the PHP. Cross-Site Request Forgery works here (and is not caught by the implemented safeguard) due to the check that is made for “response != expected”.

This check does not work as desired because the attacker is able to pass any arbitrary value to the webpage for the ‘challenge’ value, find out what the ‘expected’ value is (from the notification on the page that is created containing this value within line 28), and then inject the expected value in order to bypass the CSRF. Because the ‘challenge’ value is static, the ‘expected’ value is also static, therefore the attacks works every login.

For this webpage, within the t1.html I crafted, I set the values of the input fields accordingly, passing the account and routing numbers as desired, but also inputting custom values for ‘challenge’ and ‘response’, which were really the key values for the vulnerability. After inputting arbitrary values for those two fields, as the attacker I was able to have the notification populate the calculated ‘expected’ value. At this point I then had to populate the ‘expected’ value within my HTML file for the ‘response’ input, which allowed the attack to succeed.

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

The implemented CSRF protection only checks if the calculated integer using the passed ‘challenge’ value is correct, rather than calculating using a unique token that is generated each time the webpage is loaded (or the login button is pressed). There is already a CSRF session token that appears within the PHP code (‘csrf\_token’), which should be utilized for the inequality check at lines 27-29, which is currently the following:

if ($\_POST['response'] != $expected) {

notify('CSRF attempt prevented!'.$teststr.'--'.$\_POST['response'].' != '.$expected, -1);

Not only should the inequality be changed, the notification that is sent to the webpage should also be altered to be sure that the information cannot be easily accessed by an attacker. Instead of the current code, a potential fix could be achieved by using the ‘csrf\_token’ in the calculation for the ‘expected’ value.

if ($\_POST['response'] != $\_SESSION['csrf\_token']) {

notify('CSRF attempt prevented!');

}

Alternatively, if the ‘csrf\_token’ is used in the calculation for the ‘expected’ value, then the original code could be sufficient in order to prevent the CSRF attempt.

**Target 2 Epilogue**

1. List the PHP page and lines that should be changed to fix the vulnerability.

The vulnerability for the XSS attack is in the file index.php, specifically relating to line 34 of the HTML portion, which contains the following:

<input type="text" name="login" value="<?php echo @$\_POST['login'] ?>">

1. 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.

This code is vulnerable due to the fact that it is posting data straight to the server without any kind of validation from the user. There are no implemented checks to validate the input, which allows for users to input whatever they desire, malicious or not, and it gets sent over without being cleansed (<https://owasp.org/www-community/attacks/xss/>). Because of this, in this target, the XSS attack is successful because tick marks can be used to inject a script containing any malicious functions that are desired.

1. Explanation of how to fix the code. Feel free to include snippets and examples. Be detailed!
   1. 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.
   2. Warning: Removing site functionality will not be accepted here.

In order to fix this, there should be a function added to ‘index.php’ that will validate the input before it is sent to the server. This will ensure that any input is actually valid and is not changing anything server-side that it shouldn’t (or executing any javascript that will alter the webpage). This fix can be achieved by using a regular expression to filter out any characters that are not expected (<https://www.sitepoint.com/php-security-cross-site-scripting-attacks-xss/>).

Also, the data that is being passed through to the server should be sanitized, which can potentially be done at the same step. In the case of this ‘login’ value, a function could be added to ensure that only alphanumeric values are passed through. The value could also be compared to a whitelist of values that are to be accepted by the function, and if there are any characters/values that are not on that whitelist, then it will throw an error (<https://www.troyhunt.com/understanding-xss-input-sanitisation/>).

**Target 3 Epilogue**

1. List the PHP page and lines that should be changed to fix the vulnerability.

The vulnerability for the SQL Injection is in auth.php, lines 57 and 68, should be changed in order to fix the vunlerability. The code at these two lines are as follows:

$sql = "SELECT \* FROM users WHERE eid='$escaped\_username'";

$sql = "SELECT \* FROM users WHERE eid='$escaped\_username' AND password='$hash'";

In order to protect the vulnerable code, lines 30-51, the filter function, should be changed.

function sqli\_filter($string) {

$filtered\_string = $string;

$filtered\_string = str\_replace("admin'","",$filtered\_string);

…

$filtered\_string = str\_replace("||","",$filtered\_string);

return $filtered\_string;

}

1. 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.

When sending forms that interact with a database, code should never rely on dynamic queries that have user input included as parameters. What this allows for is for an attacker to use this weakness to insert SQL statements that can manually change the database that is being accessed. SQL sanitization was the prevention method that was attempted to be used within auth.php within the function ‘sqli\_filter’, but it was not successful (<https://www.acunetix.com/websitesecurity/sql-injection/>) .

The reason that this was not successful was because not enough values were stripped from the input, so while many basic SQL commands were cleansed from the input, some input (including tick marks) were not filtered using that function. The attacker is then able to input SQL commands to change the original function of the query (<https://www.w3schools.com/sql/sql_injection.asp>) .

Within this specific PHP page, there was no other protection against SQL injection, which is why the attack is successful. Because of this, were able to inject [] within the username input field, which allowed our attack to work (reference for where I got the code from to inject <https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/> .

1. Explanation of how to fix the code. Feel free to include snippets and examples. Be detailed!
   1. 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.

While SQL sanitization was unsuccessful in preventing SQL injection in this specific example, there are some other methods of avoiding this vulnerability including using prepared statements, or using stored procedures (<https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html> ). However, if we were to only want to implement an improved method of SQL sanitization, then we would have to improve the ‘sqli\_filter’ function.

Instead of only targeting specific SQL statements and other values that one would assume to be used in a SQL injection attack, the filter should really cleanse all characters that are not alphanumeric. If the username field is limited at registration to only include alphanumeric values at account creation, then there would be no reason to not filter out all other values within the ‘sqli\_filter’ function (<https://www.acunetix.com/blog/articles/prevent-sql-injection-vulnerabilities-in-php-applications/>) .

While that would make the filter more secure, it is still not bulletproof. It would be beneficial to include prepared statements in ‘auth.php’ for the queries at lines 57 and 68. The prepared statements would allow for the SQL statement to be prepared before the user input is included in the statement. Therefore, if the user input is incorrectly escaped, it will not affect the SQL query (<https://www.w3schools.com/php/php_mysql_prepared_statements.asp>).

**Additional Targets**

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

// in the password field, sql injection also?

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

**Works Cited**

**You are required to include works cited**

[**https://www.w3schools.com/sql/sql\_injection.asp**](https://www.w3schools.com/sql/sql_injection.asp)

<https://www.acunetix.com/websitesecurity/sql-injection/>

<https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html>

<https://www.veracode.com/security/xss>

<https://www.w3schools.com/html/html_forms.asp>

<https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_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!**