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".*
   /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?
   I'm a teapot

Activity 2 - Network Tab

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

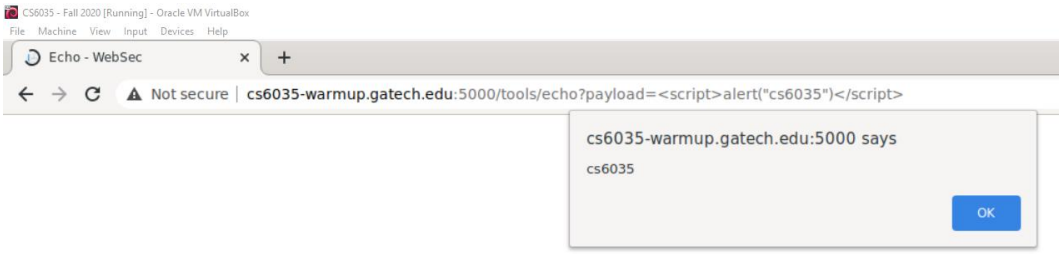
2. What status code did the server return? *Include both the code and description. Ex: "200 Ok"*
   418 I'm a teapot

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.



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');
}
```

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.

The vulnerability that is present in the lines of code above is within the CSRF protection function that is implemented within the 'account.php' page. Cross-Site Request Forgery works here (and is not caught by the implemented safeguard) due to the check that is made for "response != expected" at line 27.

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 protection. Because the 'challenge' value is static (as per the attacker's input), the 'expected' value is also static, therefore the attack will work with every login.

For this webpage, within the 't1.html' file crafted, I set the values of the input fields accordingly, passing the account and routing numbers as desired, but also inputted custom values for 'challenge' and 'response', which were truly the key values for this 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 populated that 'expected' value for the 'response' input within my HTML file, which then allowed the attack to succeed ("*How to demonstrate a CSRF attack*", 2018).

3. 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 in order to use the 'csrf_token' within the calculation for the 'expected' value, but the notification that is sent to the webpage should also be altered to ensure 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, as demonstrated below.

```
if ($_POST['response'] != $_SESSION['csrf_token']) {
        notify('CSRF attempt prevented!');
}
```

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'] ?>">
```

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.

   This code is vulnerable due to the fact that it is posting data straight to the server without any kind of validation after retrieving the input from the user. There are no implemented checks for input validation, which allows for users to input whatever they desire, malicious or not, and it gets sent over without being cleansed ("*Cross Site Scripting (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.

   For example, for our purposes of performing an XSS attack on the payload website, I was able to inject JavaScript functions within one string into the 'login' input. Because the literal value is taken without any alterations, the string was interpreted as a part of the input variable and I was able to inject '<script>' tags and their contents into the HTML code on the webpage (Doyle, 2017).

   This vulnerability allowed for the value of 'login' to be a string with a function that ran when the window was loaded, and another function that executed when the login button was pressed. At this point is when I was able to capture the inputs for the username and password in order to email them to the desired location.

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.

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 valid and is not changing anything server-side that it should not (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 and sanitize the input before it is passed on to the server (Fekete, 2013).

In the case of this 'login' value within 'index/php', a function could be added to ensure that only alphanumeric values are passed through. Or, the value could 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 (Hunt, 2013). This will ensure that values such as tick marks and HTML elements/JavaScript function are not accidentally included in the variable.

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 within 'auth.php', lines 57 and 68, and 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, the filter function within lines 30-51 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;
}
```

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.

   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 ("*What is SQL Injection (SQLi) and How to Prevent Attacks*", 2020).

The reason for this is that 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 ("*SQL Injection"*).

Within this specific PHP page, there was no other protection against SQL injection, which is why the attack is successful. Because of this, I was able to inject "' Or '1'='1" within the username input field, which allowed the attack to work (Mavituna, 2019). This input was not caught by the filter because the filter was not case sensitive; it only cleansed (and removed) "OR" and "or" from the user input, therefore the mixed capitalization bypassed the filter, allowing for the query to be altered.

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.

While the SQL sanitization that is in place was unsuccessful in preventing SQL injection in this specific example, there are some other methods of avoiding this vulnerability that are functional, including using prepared statements ("*SQL Injection Prevention Cheat Sheet*"). 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 values that one would assume to be used in a SQL injection attack, the filter should 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 (Muscat, 2020).

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 ("*PHP MySQL Prepared Statements*").

Additional Targets

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

   There is not any protection around the session ID within 'account.php', leaving the ID vulnerable to session hijacking (Shirey, 2013). There are also many data elements that are passed directly to the server without any data cleansing, particularly within the inputs used to register a new user.

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

   The session ID within 'account.php' is not protected and can easily be

hijacked for an attacker to impersonate a legitimate user. Session hijacking is a dangerous vulnerability because it can be used to impersonate an administrator account, or any account with higher privileges than a base user. Within 'account.php', the session id ('csrf_token') is only set using the following line of code (line 12).

```
if (!isset($_SESSION['csrf_token'])) $_SESSION['csrf_token'] = mt_rand();
```

There is no other check done regarding the session, therefore an attacker only needs this id to impersonate the user. In order to prevent this, there should be a function that will check if the session expired, when the last access was, if the IP address is correct, or any other form of identification that could be used to cross reference the user ("*PHP Security Vulnerabilities*" 2020). This would make it much more difficult to impersonate a user because it would be harder for an attacker to know all the values.

As for the 'register' function within the 'auth.php' file, there are no validation functions that would ensure that the passed data is not malicious, which leaves it open for an XSRF or SQL Injection attack. It also allows for the creation of an account without any kind of password strength, which makes it more likely that a user's information can get stolen simply by guessing simple passwords.

In order to solve this, there should be safeguards put in place from the user registration in order to only allow certain whitelisted characters in the username and password fields. If there were safeguards in place for every user input on the page, it would fill many holes that are currently wide open for attackers to exploit. Adding a list of whitelisted values to

be used when creating accounts would not only make the inputs more secure and less vulnerable to these attacks, but enforcing mandatory variation within passwords would make user accounts more protected overall.

Works Cited

Cross-Site Scripting (XSS) Tutorial: Learn About XSS Vulnerabilities, Injections and How to Prevent Attacks. (n.d.). Retrieved November 22, 2020, from https://www.veracode.com/security/xss

Doyle, R. (2017, July 17). XSS Password Stealing - Who needs cookies?! Retrieved November 23, 2020, from https://www.doyler.net/security-not-included/xss-password-stealing

Fekete, G. (2013, October 28). Cross-Site Scripting Attacks (XSS). Retrieved November 22, 2020, from https://www.sitepoint.com/php-security-cross-site-scripting-attacks-xss/

FreeCodeCamp.org. (2020, January 27). PHP Security Vulnerabilities: Session Hijacking, Cross-Site Scripting, SQL Injection, and How to Fix Them. Retrieved November 23, 2020, from https://www.freecodecamp.org/news/php-security-vulnerabilities/

How to demonstrate a CSRF attack. (2018, August 23). Retrieved November 22, 2020, from https://stackoverflow.com/questions/6812765/how-to-demonstrate-a-csrf-attac

HTML Forms. (n.d.). Retrieved November 22, 2020, from https://www.w3schools.com/html/html_forms.asp

Hunt, T. (2013, May 29). Understanding XSS – input sanitisation semantics and output encoding contexts. Retrieved November 22, 2020, from https://www.troyhunt.com/understanding-xss-input-sanitisation

Mavituna, F. (2019, August 26). SQL Injection Cheat Sheet. Retrieved November 22, 2020, from https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/

Muscat, I. (2020, May 13). Prevent SQL injection vulnerabilities in PHP applications and fix them. Retrieved November 22, 2020, from https://www.acunetix.com/blog/articles/prevent-sql-injection-vulnerabilities-in-php-applications/

PHP MySQL Prepared Statements. (n.d.). Retrieved November 22, 2020, from https://www.w3schools.com/php/php_mysql_prepared_statements.asp

S, K. (n.d.). Cross Site Scripting (XSS). Retrieved November 22, 2020, from https://owasp.org/www-community/attacks/xss/

Shirey, D. (2013, August 20). Top 10 PHP Security Vulnerabilities. Retrieved November 23, 2020, from https://www.sitepoint.com/top-10-php-security-vulnerabilities/

SQL Injection Prevention Cheat Sheet¶. (n.d.). Retrieved November 22, 2020, from https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

SQL Injection. (n.d.). Retrieved November 22, 2020, from https://www.w3schools.com/sql/sql_injection.asp

What is SQL Injection (SQLi) and How to Prevent Attacks. (2020, September 10). Retrieved November 22, 2020, from https://www.acunetix.com/websitesecurity/sql-injection/

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