

## Project 4:

# Web Security Report Entry

Spring 2022

### 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.*

**Answer:** A\_Value\_Between\_One\_And\_Ten

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".*

**Answer:** project4.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?

**Answer:** There are 42 bugs on the wall.

#### Activity 2 - Network Tab

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

**Answer:** POST

2. What status code did the server return?. *Ex: "200"*

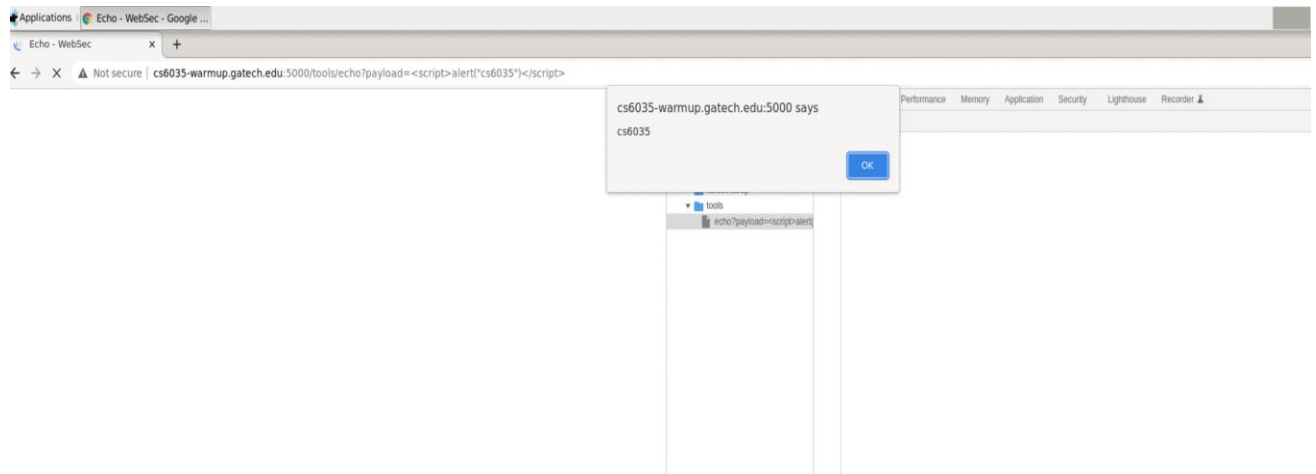
**Answer:** 499

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

**Answer:** but\_most\_of\_all\_samy\_is\_my\_hero

### Activity 3 - Built-in browser protections

- 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

- 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.

**Answer:** Congratulations!, you've successfully finished this activity. The answer is Stuxnet (2010)

## 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.

#### Answer:

The PHP page can be retrieved via the root directory

/var/www/payroll/account.php

The vulnerability exists between line 21 and 28 in the code. The snippet as shown below.

```

21     // verify CSRF protection
22     $expected = 1;
23     $teststr = $_POST['account'].$_POST['challenge'].$_POST['routing'];
24     for ($i = 0; $i < strlen($teststr); $i++) {
25         $expected = (13337 * $expected + ord($teststr[$i])) % 100000;
26     }
27     if ($_POST['response'] != $expected) {
28         notify('CSRF attempt prevented!'.$teststr.'--'.$_POST['response'].' != '.$expected
29     } else {
30         $accounting = ($_POST['account']).':'.($_POST['routing']);
31         $db->query("UPDATE users SET accounting='$accounting' WHERE user_id='".$_auth->user
32         notify('Changes saved');
33     }

```

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.

**Answer:**

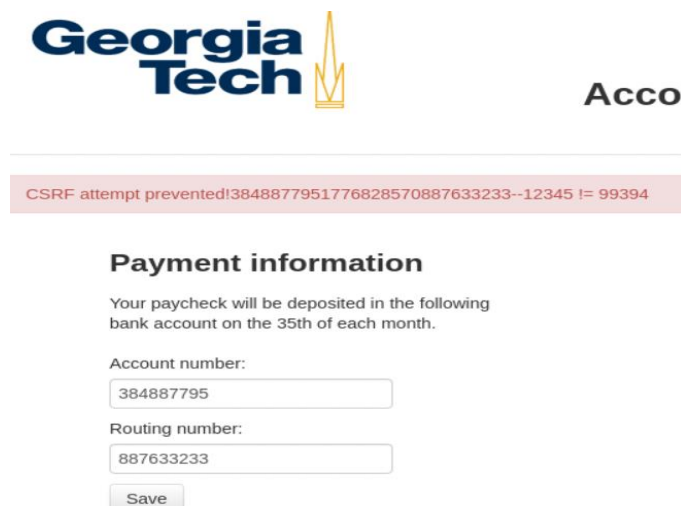
The purpose of the code snippet is to generate an expected value **\$expected** via account number, csrf token, which is a random number generated each session and routing number and compare it to the response value **\$\_POST['response']**, if the two does not match, notify the user with both values (line 28), otherwise, save it to the database.

There are two problems with the code:

- 1) The value used for the check all came from the post, i.e., the csrf token is static, being, **\$\_POST['challenge']**, but it should use the token generated when session was initiated, aka **\$\_SESSION['csrf\_token']**. Therefore, if the one has a pair of **\$\_POST['challenge']** and **\$expected**, the site can be attacked via hardcoded values in the html code.
- 2) It notifies the user of the pair **\$\_POST['challenge']** and **\$expected**. A hacker can easily launch a session with a random number as the **\$\_POST['challenge']**, get the notification to obtain **\$expected** and plug it back as the value for **\$\_POST['response']** to pass the check.

Below is how I finish the attack:

- 1) I first hardcoded a random number 1776828570 as my csrf value (aka **\$\_POST['challenge']**) in the html file, and a random number 12345 as my csrf value (aka **\$\_POST['response']**).
- 2) Launch a session and trigger the notification showing the expected value, 99394. Screenshot presented below.



The screenshot shows the Georgia Tech Accolade interface. At the top, the Georgia Tech logo and 'Accolade' text are visible. A red error message states: 'CSRF attempt prevented!3848877951776828570887633233--12345 != 99394'. Below this is a 'Payment information' section. It includes a message: 'Your paycheck will be deposited in the following bank account on the 35th of each month.' The form contains two input fields: 'Account number:' with the value '384887795' and 'Routing number:' with the value '887633233'. A 'Save' button is located at the bottom of the form.

- 3) Open the t1.html, update my csrf value to 99394 and relaunch to finish the attack.

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

**Answer:**

To address the two problems mentioned above, we can first replace the current post challenge value in the `$teststr` with the randomly generated csrf token at line 23.

**Original:**

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

**Updated:**

```
$teststr = $_POST['account'].$_SESSION['csrf_token'].$_POST['routing'];
```

Then update the logic in the if statement between line 27 and line 33 to the only when the response value matches the expected, db query will be executed, and if not, do nothing. Code update as below:

**Original:**

```
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' WH
ERE user_id='".$auth->user_id()."'");
    notify('Changes saved');
}
```

**Updated:**

```
if ($_POST['response'] == $expected) {
    $accounting = ($_POST['account']).':'.($_POST['routing']
);
    $db->query("UPDATE users SET accounting='$accounting' WH
ERE user_id='".$auth->user_id()."'");
```

```
        notify('Changes saved');  
    }
```

## Target 2 Epilogue

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

### Answer:

The PHP page can be retrieved via the root directory

/var/www/payroll/index.php

The vulnerability exists between line 34 in the code. The snippet as shown below.

```
34      <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.

### Answer:

Code at line 34 has no input check to prevent from user or hacker inputting malicious scripts. When the 'Login' button is clicked, a POST request is sent to the server with the input value as the parameter. After the server processes the information, it returns with a response. Since the browser sees the entire response data as HTML, further processes it and renders it. Therefore, if the input contains any malicious code, the code will be executed when being rendered.

In this task, I tricked the browser to inject the malicious script by wrapping it with single quotes and double quotes as below. The content between the single

quotes will be read as the value of the input and be rendered later to finish the attack.

```
<input type = "hidden" name = "login" value = ' ">
```

```
<script> //malicious script </script>
```

```
<br class = " '>
```

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.

#### Answer:

As the reason why line 34 is vulnerable is because it lacks the guarding on the input, the solution would be either patch it with input checking or detect and transform special characters that have special significance in HTML, such as single quotes, double quotes, less than and greater than, to HTML entities, for all input fields from the client side.

The PHP function htmlspecialchars() can solve the problem by addressing the second proposal above by converting the input string.<sup>1</sup> Line 34 can be updated as below:

#### Original:

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

#### Updated:

```
<input type="text" name="login" value="<?php echo htmlspecialchars(@$_POST['login']) ?>">
```

## Target 3 Epilogue

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

### Answer:

The PHP page can be retrieved via the root directory

/var/www/payroll/includes/auth.php

The vulnerability exists between line 30-52 and line 57-60 in the code. The snippet as shown below.

```

29      //filters any fishy input
30      function sql_filter($string) {
31          $filtered_string = $string;
32          $filtered_string = str_replace("admin'", "", $filtered_string);
33          $filtered_string = str_replace("ADMIN'", "", $filtered_string);
34          $filtered_string = str_replace("or", "", $filtered_string);
35          $filtered_string = str_replace("collate", "", $filtered_string);
36          $filtered_string = str_replace("drop", "", $filtered_string);
37          $filtered_string = str_replace("and", "", $filtered_string);
38          $filtered_string = str_replace("OR", "", $filtered_string);
39          $filtered_string = str_replace("COLLATE", "", $filtered_string);
40          $filtered_string = str_replace("DROP", "", $filtered_string);
41          $filtered_string = str_replace("AND", "", $filtered_string);
42          $filtered_string = str_replace("union", "", $filtered_string);
43          $filtered_string = str_replace("UNION", "", $filtered_string);
44          $filtered_string = str_replace("/*", "", $filtered_string);
45          $filtered_string = str_replace("*/", "", $filtered_string);
46          $filtered_string = str_replace("//", "", $filtered_string);
47          $filtered_string = str_replace("; ", "", $filtered_string);
48          $filtered_string = str_replace("||", "", $filtered_string);
49          $filtered_string = str_replace("--", "", $filtered_string);
50          $filtered_string = str_replace("#", "", $filtered_string);
51
52          return $filtered_string;
53      }

```

```

57      $escaped_username = $this->sql_filter($username);
58      // get the user's salt
59      $sql = "SELECT * FROM users WHERE eid='$escaped_username'";
60      $result = $this->db->query($sql);

```



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.

**Answer:**

Code in line 57-60 intends to retrieve data of the filtered input by the user via querying the database, while code in line 30-52 intends to filter the input string to strip off the risky characters/substrings. The problems with both snippets are shown below:

- 1) The filter function lacks completeness in transforming the string. How I bypassed the filter is simply putting an extra 'O' and 'R' besides the usual 'OR' in the injection because the string is only filtered once. SQL Injection code as below:

```
8      var username = document.getElementById("targetlogin").value + "' OORR '1=1';
```

- 2) The query to obtain \$result to check whether the user exists seems redundant to me. By querying the database with username only, it's even more vulnerable to SQL injection because it's relatively easy to attack with just 1 field, just like how I injected the "OR" keywords to make anything after "WHERE" always true, compared to both username and passwords are needed for a query. To avoid it, we can simply check user existence right below line 70 (snippet below).

```
70      $sql = "SELECT * FROM users WHERE eid='$escaped_username' AND password='$hash';
```

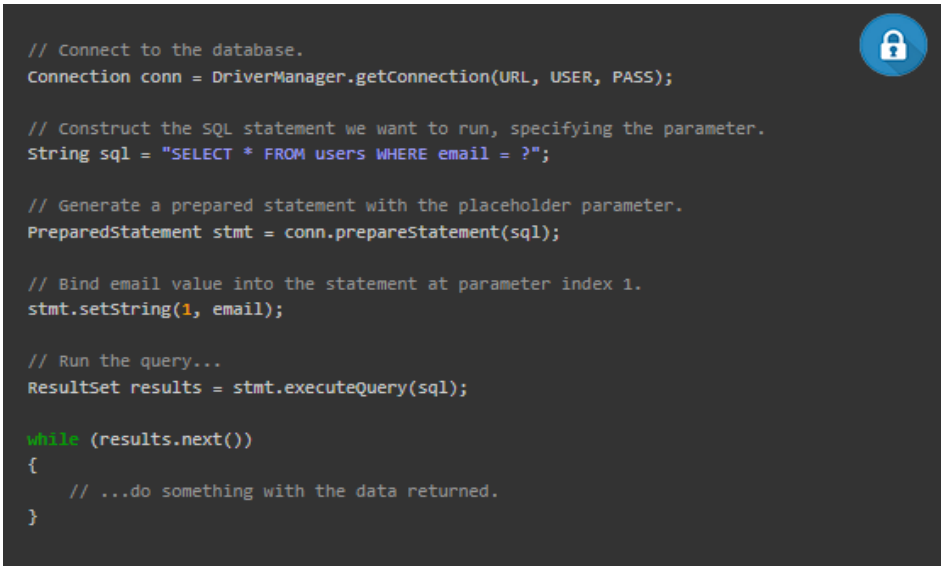
3. Explanation of how to fix the code. Feel free to include snippets and examples. Be detailed!
  - a. Be careful with your explanation here. There are wrong ways to fix this vulnerability. Hint: Never write your own crypto algorithms. This concept extends to SQL sanitization.

**Answer:**

For `sqli_filter` function, we can rewrite the function by using a while loop to ensure it escapes all user input.

We can also use parameterized statements instead of string concatenation to prevent SQL Injections.<sup>3</sup> It can be implemented via PDO with strongly typed parameterized queries using `bindParam()`.<sup>3</sup> When malicious SQL code was passed to parameterized statements, such as “username’ OR ‘1=1”, the query would not be vulnerable because it looks for the whole statement and try to find a match.

Sample code snippet as below.



```
// Connect to the database.
Connection conn = DriverManager.getConnection(URL, USER, PASS);

// Construct the SQL statement we want to run, specifying the parameter.
String sql = "SELECT * FROM users WHERE email = ?";

// Generate a prepared statement with the placeholder parameter.
PreparedStatement stmt = conn.prepareStatement(sql);

// Bind email value into the statement at parameter index 1.
stmt.setString(1, email);

// Run the query...
ResultSet results = stmt.executeQuery(sql);

while (results.next())
{
    // ...do something with the data returned.
}
```

4

### Additional Targets

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

#### Answer:

- 1) The login and registration process lacks an important step – authentication. There's no notification or authentication request sent to the user to conduct further verification of identity.
  - 2) Besides the CSRF the vulnerability discovered in Target 1, the CSRF token generation algorithm is too simple so that it's easy to break in via brute force attack. With my 'challenge' value of 1776828570, the 'expected' value is 99394. In real world, this could be cracked instantly.
2. Provide an explanation of how to safely fix the identified issues. Feel free to include snippets and examples. Be detailed!

#### Answer:

- 1) There are 3 types of authentications – something one knows, such as a password or a PIN; something one has, such as a token or verification code; something one is, such as fingerprint or voice. For a payroll site, the system should be equipped with dual factor authentication at least to ensure security.
- 2) Use the built-in CSRF protection if it's available in the framework. Or incorporate SameSite Cookie Attribute and custom request headers or verify the origin with standard headers to enhance the security. <sup>5</sup>

## Works Cited

1. PHP.net (2021-2022). htmlspecialchars. Retrieved Mar 31, 2022, from <https://www.php.net/manual/en/function htmlspecialchars.php>
2. Refsnes Data (2022). SQL Injection. Retrieved Mar 31, 2022, from [https://www.w3schools.com/sql/sql\\_injection.asp](https://www.w3schools.com/sql/sql_injection.asp)
3. CheatSheets Series Team (2021). SQL Injection Prevention Cheat Sheet. Retrieved Mar 31, 2022, from [https://cheatsheetseries.owasp.org/cheatsheets/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html)
4. Hacksplaining (2022). Protecting against SQL Injection. Retrieved Mar 31, 2022, from <https://www.hacksplaining.com/prevention/sql-injection>
5. CheatSheets Series Team (2021). Cross-Site Request Forgery Prevention Cheat Sheet. Retrieved Mar 31, 2022, from [https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site\\_Request\\_Forgery\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html)