Elsa Rodriguez Plaza
GT CS 6035: Introduction to Information Security

# Project 4:

# Web Security Report Entry

*Summer 2020*

## Task 1 – Warm Up Exercises

Complete the following five activities:

### Activity 1 - The Inspector & Console tabs

1. What is the value of the 'CanYouSeeMe' input?
   The input's value is: TheCakeIsALie

2. The page references a single JavaScript file in a script tag. Name this file including the file extension.
   The JavaScript referenced by the page is: 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?
   The function returns 42

### Activity 2 - Network Tab

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

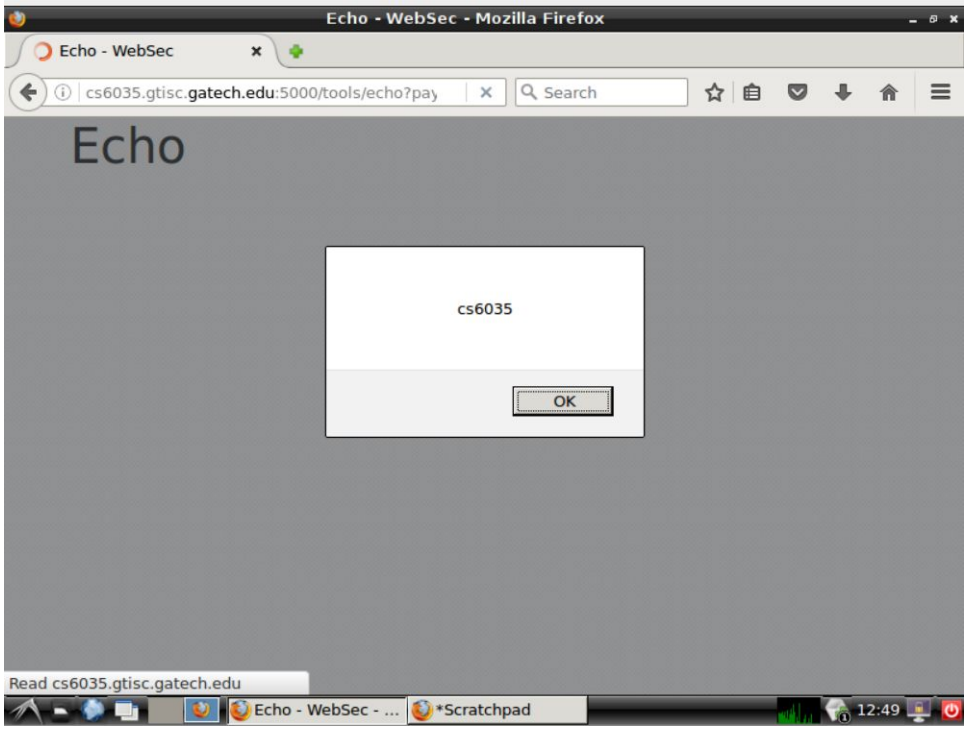
2. What status code did the server return? The status code including the

description is:
418 I'm a teapot

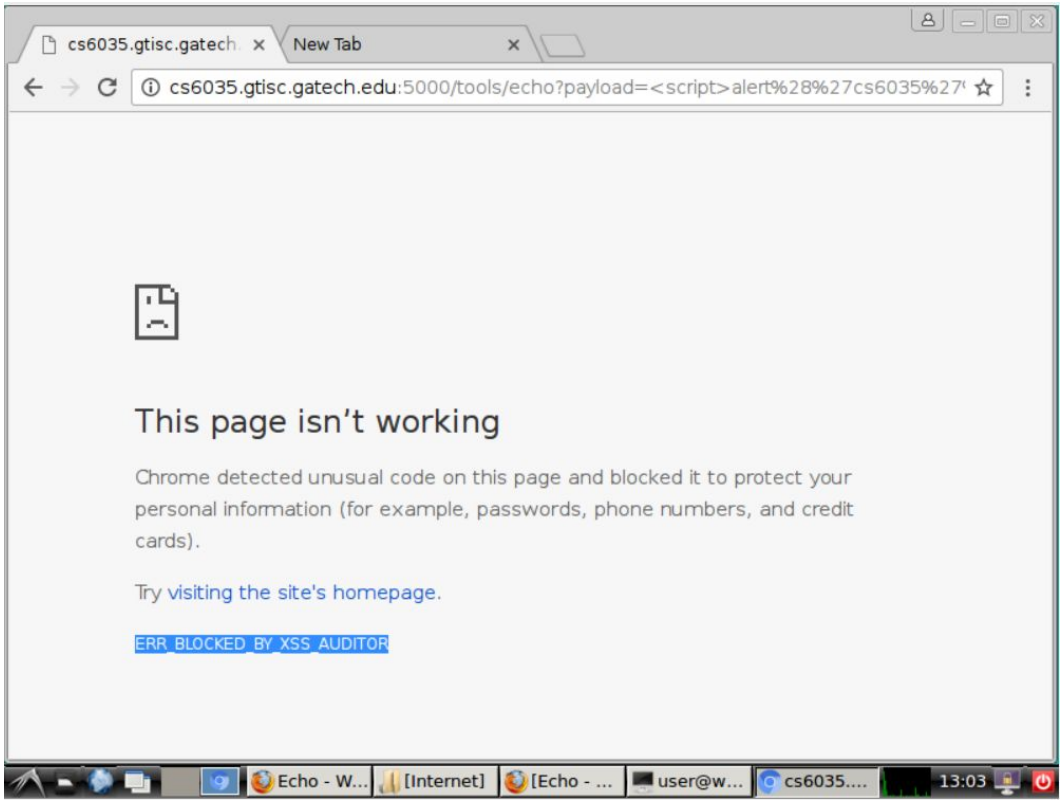
3. The server returned a cookie named 'coffee' for the browser to store. What is the value of this cookie?
   The Cookie's value is: With_Cream

## Activity 3 - Built-in browser protections

1. You can do more than just echo back text. Construct a URL such that a JavaScript alert message appears with the text cs6035 on the screen. Submit your constructed URL and a screenshot of the page as your answer.
   I used this URL:
   http://cs6035.gtisc.gatech.edu:5000/tools/echo?payload=<script>alert('cs6035')</script> to generate the following screenshot:



2. Open chromium and try your same exploit. What error message do you see on the page that begins with ERR…?
   The error I see is: ERR_BLOCKED_BY_XSS_AUDITOR

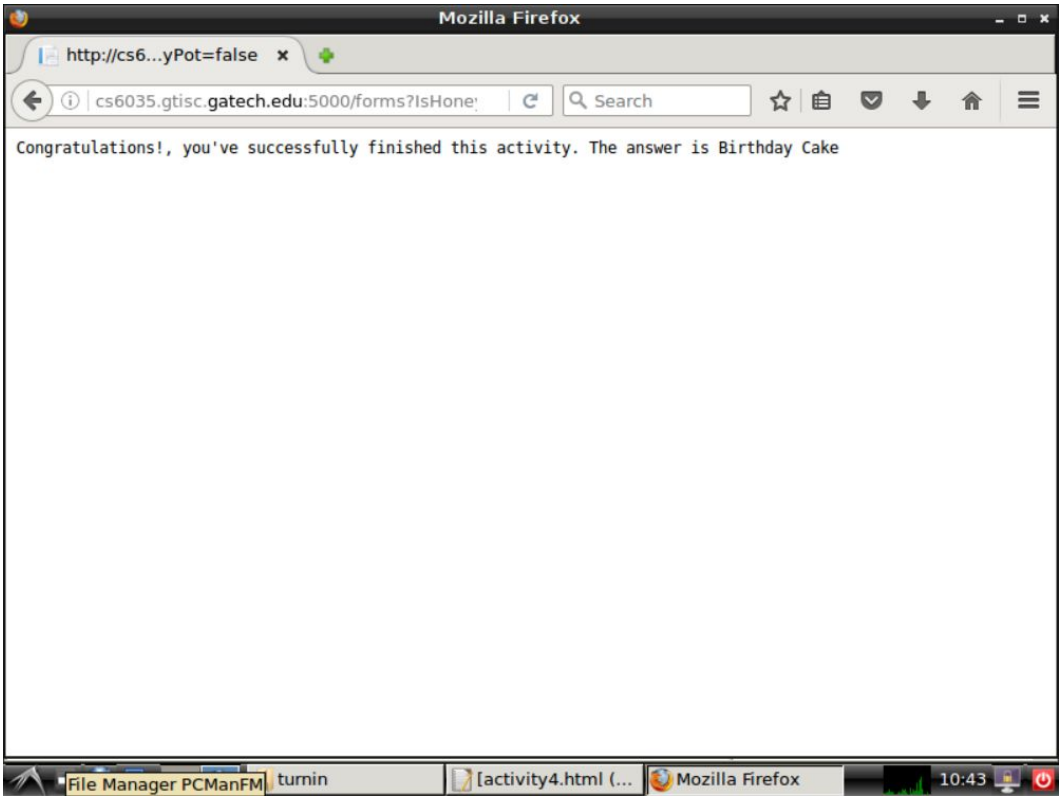



3. Research this feature and then in your own words, describe what this security feature in chromium is and how it can help protect against attacks.
   This security feature prevents XSS exploits by blocking the XSS requests submitted via HTML form values. The security feature protects user's personal information such as: passwords, credits

cards, phone numbers, etcetera. [1][2]

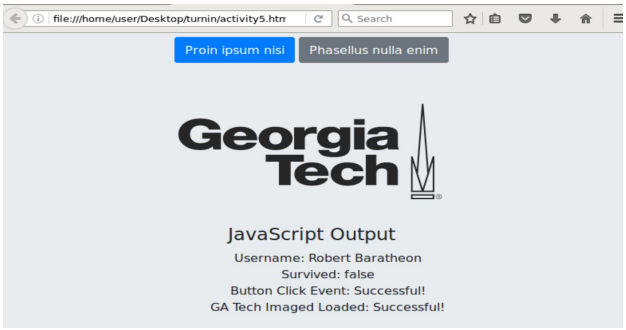
### Activity 4 - Submitting forms

The answer I got when submitting this activity was: Congratulations! you've successfully finished this activity. The answer is Birthday Cake. As depicted on the screenshot:




### Activity 5 - Accessing the DOM with JavaScript

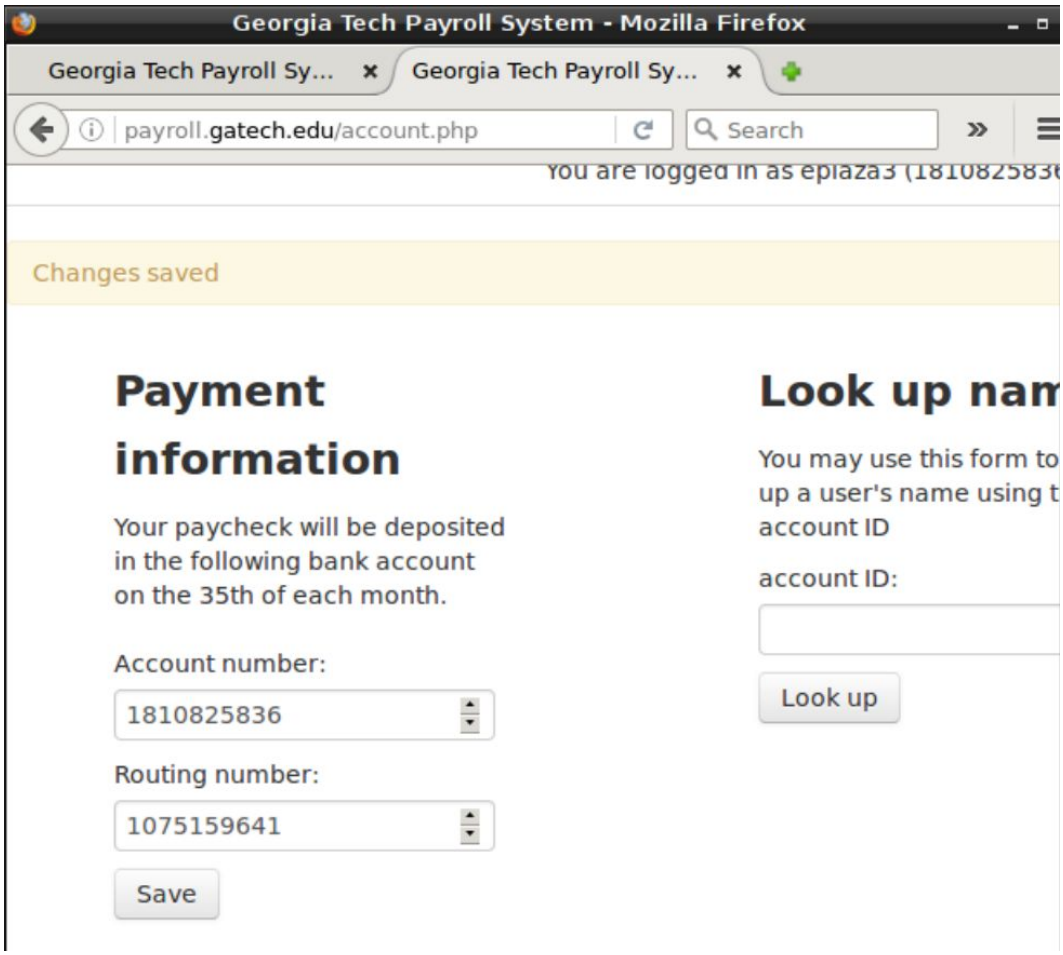
Uploaded activity5.html, here is the screenshot that I got from the VM:




## Task 5 – Epilogue

### Target 1 Epilogue

When I implemented target 1, I got the following results:



1. List the PHP page and lines that should be changed to fix the vulnerability.
    a. The vulnerability that I found is implemented in the includes/account.php file from lines 17 to 36.
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.
3. Vulnerable lines of code in account.php:

```
17  if ($action == 'save' && $_POST['Tm9PbmVXaWxsS25vdz0pLUVhc3RlckVnZ0ZvclRhcmdldDFTdW1tZXIyMDIw'] == 'Tm9PbmVXaWxsS25vdz0pLUVhc3RlckVnZ0ZvclRhcmdldDFTdW1tZXIyMDIw') {
18      if($_POST["varId"] != "42"){
19          notify("Missing the answer to life the universe and everything");
20      } else {
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, -1);
29          } else {
30              $accounting = ($_POST['account']).':'.($_POST['routing']);
31              $db->query("UPDATE users SET accounting='$accounting' WHERE user_id='".$auth->user_id()."'");
32              notify('Changes saved');
33          }
34      }
35
36  }
```

I was able to bypass the XSRF protection by:

- Implementing the following hidden fields in my form: **Tm9PbmVXaWxsS25vdz0pLUVhc3RlckVnZ0ZvclRhcmdldDFTdW1tZXIyMDIw** and **save action**. These two hidden fields allowed me to bypass the if condition on line 17.
- Then I created the hidden field: **varId** with a value of **42**. This allowed me to jump to the else statement on line 20.
- Next, I was able to execute the else statement on line 29 by implementing the **challenge (csrfc)**, **account** and **routing** hidden fields (that I got from the browser and the get_bank_info script). The value of these fields are used to compute the expected string. The expected string is then compared to the response (**csrfr**) field's value (which I got from the Browser's developer tools).
- The issue with this XSRF verification is that if the hacker passes in correct **csrfc** and **csrfr** values (he/she can get these from the browser's developer tools), then lines 30 to 32 will always execute, saving his/her account and routing numbers into the database.

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

   Based on my research from reading [3] and [4]. The idea is to generate a one-time token by using a non-deterministic approach. Then, at the time of logging in, we need to compare the generated token with the **challenge** input field. If all matches, then we know that the user is legitimate.

   The token is already generated in account.php, line 12:

```
// initiate csrf prevention

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

We can use this token to verify the authenticity of the session by adding the following if statement account.php, line 20:

```
if (isset($_POST[‘challenge’]) &&
   isset($_SESSION['csrf_token']) && $_POST[‘challenge’] !=
   $_SESSION['csrf_token'])
```

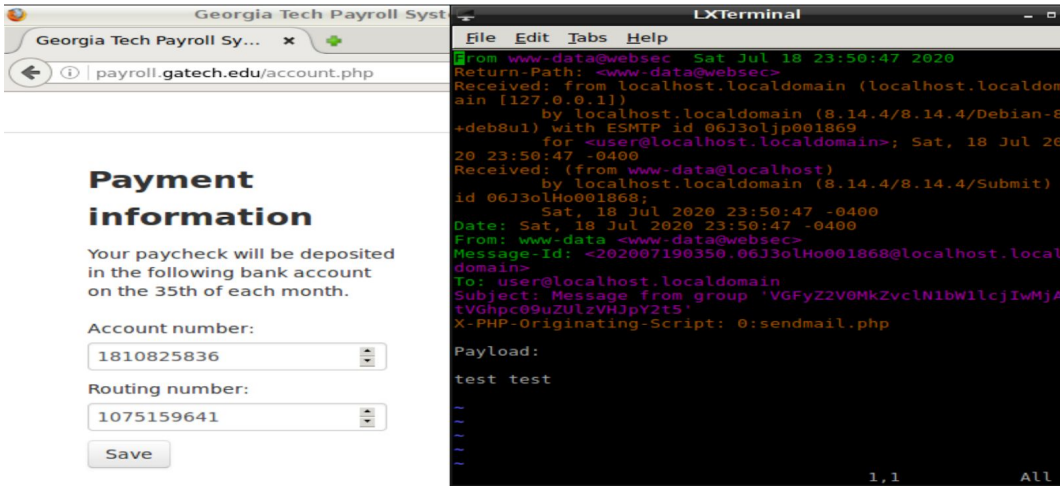
The complete logic for verification:

```
if ($action == 'save' &&
    $_POST['Tm9PbmVXaWxsS25vdz0pLUVhc3RlckVnZ0ZvclRhcmdldDFTdW1tZXIyMDIw
    '] == 'Tm9PbmVXaWxsS25vdz0pLUVhc3RlckVnZ0ZvclRhcmdldDFTdW1tZXIyMDIw') {
  if($_POST["varId"] != "42"){
      notify("Missing the answer to life the universe and everything");
  }
if ( isset($_POST[‘challenge’]) && isset($_SESSION['csrf_token']) && $_POST[‘challenge’] !=
    $_SESSION['csrf_token']){
notify(“Incorrect challenge…”)
}
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');
    }
}
```

## Target 2 Epilogue

I was able to send the email (on the right) and redirect the user to the account.php page (on the left) as depicted in the screenshot:




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

   There are multiple lines that contain this vulnerability throughout the project. But, in terms of this exploit. Line 34 in index.php: <input type="text" name="login" value="<?php echo @$_POST['login'] ?>">

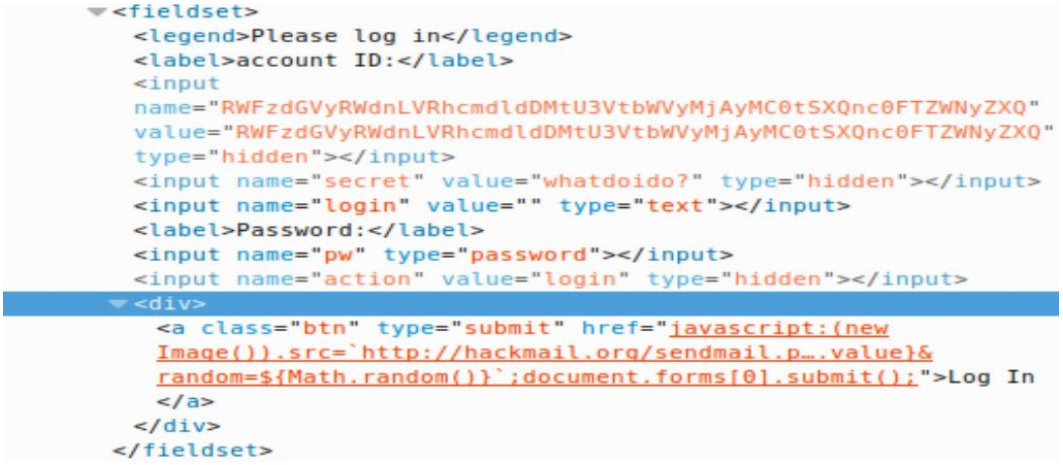
2. Describe in detail why the code listed in the line numbers above are vulnerable. You're free to use generalized concepts to help show your understanding but we also need to know details that pertain to this target and assignment. A definition of XSS is not what we're looking for.

   The issue is that the form fields' input are not sanitized, so we can easily embed malicious scripts. I crafted my script by implementing an extra hidden form that encapsulated the real **password** input field. Then, I added my malicious script as input value to the **login** field. By using the Browser's Developer tools, I realized that if I added the fields contained in the original form (**secret**, **password**, etc) and modified the login button's click event to send the email, I would be able to steal the

username and password and cover my tracks. So, basically, when the user clicks the Login button, the email is sent containing the user's input from my custom form and the user gets redirected to the account.php page as usual.

The original page looked like:


The modified page with my embedded script:




The extra hidden (the very last) form that hides the original password field:

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.

    One of the ways that we could prevent this attack from happening is adding a regular expression that ensures that no invalid input is entered. PHP has out of the box functions that could clean the input for us. We could also use sanitation libraries to make sure this input is validated before it gets processed and added into the DOM [8]. A simple way to prevent this attack, would be to escape the value of the login field by using htmlspecialchars, which is a php-provided function. [9]. For example in auth.php line 34, instead of:

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

We could do:

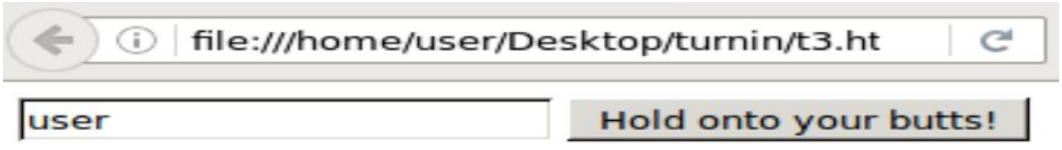
```
<input type="text" name="login" value="<?php
   htmlspecialchars(echo @$_POST['login'], ENT_QUOTES) ?>">
```

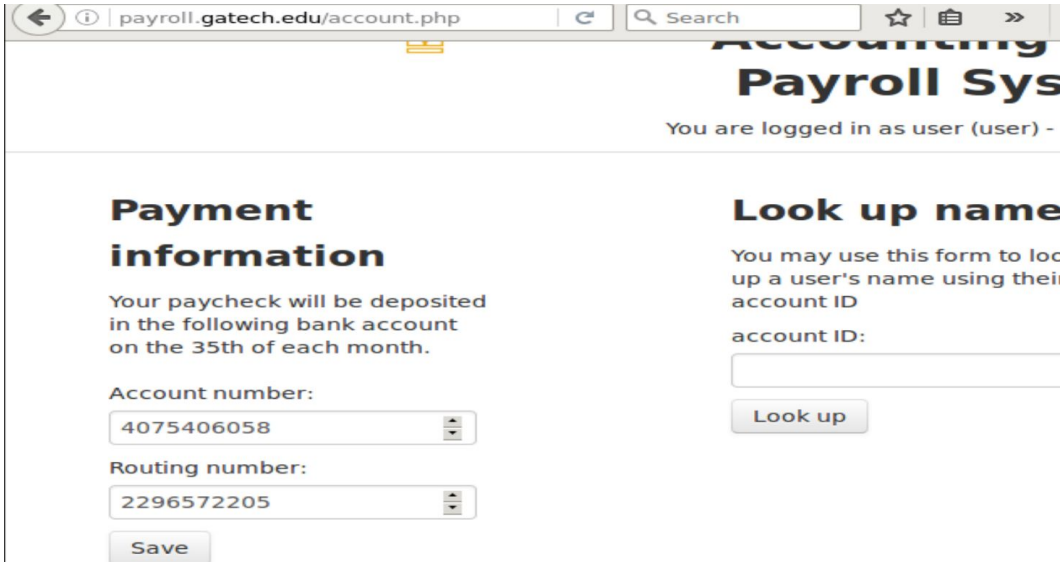
**ENT_QUOTES**: Converts quotes into HTML entities.

More advanced techniques could be used, but using htmlspecialchars is a simple, yet efficient way to prevent XSS attacks.

### Target 3 Epilogue

SQL Injection Exploit results:






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

I found the vulnerability in the includes/auth.php, from lines 30 - 79.

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.

Vulnerable code in auth.php:

```
29          //filters any fishy input
30          function sqli_filter($string) {
31          $filtered_string = $string;
32          $filtered_string = str_replace("admin'","",$filtered_string);
33          $filtered_string = str_replace("or","",$filtered_string);
34          $filtered_string = str_replace("collate","",$filtered_string);
35          $filtered_string = str_replace("drop","",$filtered_string);
36          $filtered_string = str_replace("and","",$filtered_string);
37          $filtered_string = str_replace("OR","",$filtered_string);
38          $filtered_string = str_replace("COLLATE","",$filtered_string);
39          $filtered_string = str_replace("DROP","",$filtered_string);
40          $filtered_string = str_replace("AND","",$filtered_string);
41          $filtered_string = str_replace("union","",$filtered_string);
42          $filtered_string = str_replace("UNION","",$filtered_string);
43          $filtered_string = str_replace("/*","",$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          return $filtered_string;
51          }
```

```
54      function login($username, $password) {
55          $escaped_username = $this->sqli_filter($username);
56          // get the user's salt
57          $sql = "SELECT * FROM users WHERE eid='$escaped_username'";
58          $result = $this->db->query($sql);
59          $user = $result->next();
60          // make sure the user exists
61          if (!$user) {
62              notify('User does not exist', -1);
63              return false;
64          }
65          // verify the password hash
66          $salt = $user['salt'];
67          $hash = md5($salt.$password);
68          $sql = "SELECT * FROM users WHERE eid='$escaped_username' AND password='$hash'";
69          $userdata = $this->db->query($sql)->next();
70          if ($userdata) {
71              // awesome, we're logged in
72              $_SESSION['user_id'] = $userdata['user_id'];
73              $_SESSION['eid'] = $userdata['eid'];
74              $_SESSION['name'] = $userdata['name'];
75          } else {
76              notify('Invalid password', -1);
77              return false;
78          }
79      }
```

The main problem with the code is that the input fields' values are not sanitized properly. So, I entered the following string: v**alidUser"'oR'" + validUser + "'='" + validUser** as input to the login field and I was able to bypass the authentication. The main issue is the function **sqli_filter** on line 30. This function uses **str_replace** to filter the input that could cause a SQL Injection. The problem is that strings like "Or" and "dROp" , etcetera are perfectly valid when constructing queries. So, **sqli_filter** fails to clean up input that contains any of those keywords, making possible SQL Injection attacks.
The string that I passed as user, **validUser"'oR'" + validUser + "'='" + validUser**, generated the following query using the test user in auth.php, line 57:
**SELECT * FROM users WHERE eid='test' oR 'test' = 'test';**
This query returns the test user row from the database, bypassing the if statement on auth.php, line 61. Then, the MD5 hash of the plain password is computed using the row returned by the previous query. Then, the following SQL query on auth.php, line 68 is executed: **SELECT * FROM users WHERE eid='test' oR 'test' = 'test'" AND password='password that was computed in line 66 and 67';**
This query returns the test user and lines 72-74 start a new valid session.

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.

One way to fix this code is to make sure that strings analyzed by the **sqli_filter** function on line 30 are treated as uppercase letters or lowercase letters to make sure that all the SQL query keywords are identified. For example if we replaced auth.php, line 31 by: **$filtered_string = strtoupper($string);**
The final function would look like:

```
//filters any fishy input
function sqli_filter($string) {
//$filtered_string = $string;
```

```
$filtered_string = strtoupper($string);
$filtered_string = str_replace("admin'","",$filtered_string);
$filtered_string = str_replace("or","",$filtered_string);
$filtered_string = str_replace("collate","",$filtered_string);
$filtered_string = str_replace("drop","",$filtered_string);
$filtered_string = str_replace("and","",$filtered_string);
$filtered_string = str_replace("OR","",$filtered_string);
$filtered_string = str_replace("COLLATE","",$filtered_string);
$filtered_string = str_replace("DROP","",$filtered_string);
$filtered_string = str_replace("AND","",$filtered_string);
$filtered_string = str_replace("union","",$filtered_string);
$filtered_string = str_replace("UNION","",$filtered_string);
$filtered_string = str_replace("/*","",$filtered_string);
$filtered_string = str_replace("*/","",$filtered_string);
$filtered_string = str_replace("//","",$filtered_string);
$filtered_string = str_replace("#","",$filtered_string);
$filtered_string = str_replace("--","",$filtered_string);
$filtered_string = str_replace(";","",$filtered_string);
$filtered_string = str_replace("||","",$filtered_string);
return $filtered_string;
}
```

## Additional Targets

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

- One issue that I identified is that the **csrf_token** is generated in a deterministic and predictable manner. [3] This should be avoided, because a hacker could easily generate the token and bypass authentication.

  Vulnerable line:

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

- The second issue that I identified is associated with the registration. Basically, the registration is too permissive when it comes to user passwords. A strong authentication mechanism should enforce strong

user passwords using a combination of special characters, letters and numbers. [5]

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

- Regarding the first issue, the best way to generate the token would be to use a function that uses entropy. For example, **random_int** is the php wrapper to **/dev/urandom**, which generates cryptographically secure random numbers. [6] One potential fix is to replace line:

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

By:

```
if (!isset($_SESSION['csrf_token'])) $_SESSION['csrf_token']
= random_int(PHP_INT_MIN, PHP_INT_MAX)
```

- Regarding the second issue, we can implement a regular expression checker to make sure that the user's password is strong enough and alert the user that he/she needs to create a stronger password before registration happens. A potential way to fix the weak password issue. On line 95 of the auth.php file, we could include an if statement that checks if the password is strong enough [7]

```
if (!preg_match("/[a-zA-Z!$@^0-9 ]*$/",$password1)) {

            notify("Your password is too weak. Please enter
a password that      contains upper, lower case letters,
number, and symbols"); }
```

## Works Cited

1. Lemons, M. "Chrome ERR_BLOCKED_BY_XSS_AUDITOR PHP Solution to Avoid Bogus Cross-Site Scripting Detection - Secure HTML parser and filter package blog". May, 2012. Accessed: July 18, 2020. [Online]

Available: https://www.phpclasses.org/blog/package/5614/post/1-How-to-Handle-Chrome-HTML-Editor-Form-Submission-Block-Due-to-Bogus-XSS-Detection-Causing-ERRBLOCKEDBYXSSAUDITOR-Error.html#:~:text=account%20without%20permissions.-,The%20Chrome%20Form%20Submission%20Error%20ERR_BLOCKED_BY_XSS_AUDITOR,eventual%20XSS%20exploits%20are%20avoided.

2. "ERR_BLOCKED_BY_XSS_AUDITOR Google Chrome error"
3. . February, 2019. Accessed: July 15, 2020. [Online] Available: https://www.thewindowsclub.com/err_blocked_by_xss_auditor-chrome/#:~:text=This%20error%20occurs%20if%20Chrome,ERR_BLOCKED_BY_XSS_AUDITOR%E2%80%9D
4. "How to properly add cross-site request forgery (CSRF) token using PHP"
5. .July, 2011. Accessed: July, 16, 2020. [Online]. Available: https://stackoverflow.com/questions/6287903/how-to-properly-add-cross-site-request-forgery-csrf-token-using-php
6. "Preventing csrf in PHP". April, 2010. Accessed: July, 17, 2020. [Online] Available: https://stackoverflow.com/questions/1780687/preventing-csrf-in-php
7. "Guidelines for Strong Passwords". February, 2018. Accessed: July, 17, 2020. [Online]. Available: https://its.lafayette.edu/policies/strongpasswords/
8. "random_int". Accessed: July 18, 2020. [Online]. Available: https://www.php.net/random_int
9. "Preg_match". Accessed: July 18, 2020. [Online]. Available: https://www.php.net/manual/en/function.preg-match.php
10. Vonnegut, Sarah. "3 Ways to Prevent XSS". October, 2017. Accessed: July 16, 2020. [Online]. Available: https://www.checkmarx.com/2017/10/09/3-ways-prevent-xss/
11. "htmlspecialchars"Accessed: July 18, 2020. [Online]. Available: https://www.php.net/manual/en/function.htmlspecialchars.php