Nathalie Ruano
NRUANO3

Target 1 Epilogue

1. The vulnerability can be found from line 16-32 with mainly line 25 being the major issue.

```
15 $action = @$_POST['action'];
16 if ($action == 'save' && $_POST['U3VtbWVyMjAxOFRhcmdldDFFYXN0ZXJFZ2c='] == 'U3VtbWVyMjAxOFRhcmdldDFFYXN0ZXJFZ2c=') {
17         // verify CSRF protection
18         notify('Action is: '.$action);
19         $expected = 1;
20         $teststr = $_POST['account'].$_POST['challenge'].$_POST['routing'];
21         for ($i = 0; $i < strlen($teststr); $i++) {
22                 $expected = (13337 * $expected + ord($teststr[$i])) % 100000;
23         }
24         print $expected;
25         if ($_POST['response'] != $expected) {
26                 notify('CSRF attempt prevented!'.$teststr.'--'.$_POST['response'].' != '.$expected, -1);
27         } else {
28                 $accounting = ($_POST['account']).':'.($_POST['routing']);
29                 $db->query("UPDATE users SET accounting='$accounting' WHERE user_id='".$auth->user_id()."'");
30                 notify('Changes saved');
31         }
32
33 }
```

2. The code mention above is a vulnerability because of how a check is created to see if someone is attempting a CSRF attack. As we can see above, we first have a check to see whether the save button has been submitted and the other random string is set. If a user has set action and the random string to the values expected then we are allowed to enter this if statement. This if statement then retrieves account, challenge, and routing information and uses them in order to create an $expected variable. The response and $expected variable must match, if they don't this means a notification will be shown informing the user of what the expected value should be. But because these values can be overwritten (which are used statically to initialize the $expected variable) a malicious user can now figure out the required value for the response. This will given the malicious user the ability to overwrite the users banking information with their own.

3. In order to fix the vulnerability mention above, one should first of all never print out the actual value the system is expecting. Even though this won't stop a malicious user from calculating the response in our case but by doing this we at least make them work for it a bit. The main way to combat this would be to remove the way the crsf check is done. Using account, routing, and challenge (which values all can be statically set) to calculate this rather important check is unwise. Using session tokens would be better method; possibly using hash of sha256 for each new session, will help ensure some security.

Target 2 Epilogue

1. The vulnerability can be found within line 24-59 specifically line 32.

```
24    <div class="row">
25     <div class="span4 offset1">
26      <form method="post">
27       <fieldset>
28        <legend>Please log in</legend>
29        <label>account ID:</label>
30         <input type="hidden" name="U3VtbWVyMjAxOFRhcmdldDNFYXN0ZXJFZ2c=" value="U3VtbWVyMjAxOFRhcmdldDNFYXN0ZXJFZ2c=">
31        <input type="hidden" name="secret" value="whatdoido?">
32        <input type="text" name="login" value="<?php echo @$_POST['login'] ?>">
33        <label>Password:</label>
34        <input type="password" name="pw">
35        <div>
36         <button class="btn" type="submit" name="action" value="login">Log In</button>
37        </div>
38       </fieldset>
39      </form>
40     </div>
41     <div class="span4 offset1 register">
42      <form method="post">
43       <fieldset>
44        <legend>First time? Register here:</legend>
45        <label>Name:</label>
46        <input type="text" name="name">
47        <label>account ID:</label>
48        <input type="text" name="login">
49        <label>Password</label>
50        <input type="password" name="pw1">
51        <label>Repeat password</label>
52        <input type="password" name="pw2">
53        <div>
54         <button class="btn" type="submit" name="action" value="register">Register</button>
55        </div>
56       </fieldset>
57      </form>
58     </div>
59    </div>
```

2. The code mention above is a vulnerability because of our login field that takes in a value. This leaves the page open for DOM-based XSS attacks in which a malicious user finds a vulnerability in the client-side such as the one in line 32 and overrides it in away such that they can launch a script. For example, in this project the script will retrieve out username and password after a user logins and sends an email to the malicious user with the gathered information. DOM-based XSS attacks are much harder to track because it will usually never be visible to the server-side.

3. In order to fix the vulnerability mentioned above one can implement input handling checks on the server side such as filtering html values. This should be handle by blacklisting tags such as <script> or even javascript: command. Whitelisting in a field such as username would be impossible to handle. Another method would be to encode inputs by the user. Encoding forces browser to interpret the malicious insertion as text rather than code. There are some weakness in encoding. If the malicious insertion for t3.html had used javascript:void for insertion then encoding would have failed.

Target 3 Epilogue

1. The vulnerability can be found from lines 28-69 with line 58 being the main issue.

```
28          function sqli_filter($string) {
29                  $filtered_string = $string;
30                  $filtered_string = str_replace("--","",$filtered_string);
31                  $filtered_string = str_replace(";","",$filtered_string);
32                  $filtered_string = str_replace("/*","",$filtered_string);
33                  $filtered_string = str_replace("*/","",$filtered_string);
34                  $filtered_string = str_replace("//","",$filtered_string);
35                  $filtered_string = str_replace(" ","",$filtered_string);
36                  $filtered_string = str_replace("#","",$filtered_string);
37                  $filtered_string = str_replace("||","",$filtered_string);
38                  $filtered_string = str_replace("admin'","",$filtered_string);
39                  $filtered_string = str_replace("UNION","",$filtered_string);
40                  $filtered_string = str_replace("COLLATE","",$filtered_string);
41                  $filtered_string = str_replace("DROP","",$filtered_string);
42                  return $filtered_string;
43          }
44          function login($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                  $sql = "SELECT user_id, name, eid FROM users WHERE eid='$escaped_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          }
```

2. The code mention above is a vulnerability because of how the $escaped_username fails to be filtered properly when its setup in line 45. Therefore, when $sql is setup in line 58 having an improperly filtered sql command causes the information after $escaped_username to drop the password field once the attacker applies a malicious insertion such as ‘or’1'='1.

3. In order to fix the vulnerability mention above the admin of the database should restrict usernames from having special characters that could lead to injection. We could also add a string replacer to add backslashes when spotting any quotes as SQL attacks will depend on quotes. For example: The attack: SELECT * FROM database WHERE name = ‘or’1’=‘1 should be forced to now be SELECT * FROM database WHERE name = \’or\’1\’=\’1 causing the sql not to be injected. Apart from that another possibility is to move from using sqllite and use PHP Data Objects (PDO) which have functions such as quote that will handle the example dilemma mentioned in the previous line (adding the backslashes when spotting quotes). Apart from that, if we use PDOs we can use prepared statements which when set help catch unwanted special characters by having place holder values if an injection is detected.