< Yingshan CHANG >
< ychang363 >

Target 1 Epilogue

```
<?php
// initialize global variables, authentication and database connections
include('includes/common.php');

// if the user is NOT logged in, redirect him to login page
if (!$auth->user_id()) {
        header('location: /');
}


// initiate csrf prevention
if (!isset($_SESSION['csrf_token'])) $_SESSION['csrf_token'] = mt_rand();

// handle the form submission
$action = @$_POST['action'];
if ($action == 'save' && $_POST['U3ByaW5nMjAxOVRhcmdldDFFYXN0ZXJFZ2c'] == 'U3ByaW5nMjAxOVRhcmdldDFFYXN0ZXJFZ2c') {
        // 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');
                                                                        7.1-8         Top
```

1. The vulnerable code is in account.php and the specific line is highlighted in the screenshot.
2. The expected value is calculated by a hash function, taking a challenge as input. This line will check if the response equal to the expected value, for the purpose of preventing XSRF attack. However, we can get the challenge value from the webpage by document.getElementById('csrfc') and compute its response. To be more specific, once we get the challenge value, we can re-run the same hash function and change the response value to the "expected value". The code of the hash function is easy to find in the source code of the webpage, which is shown in the screenshot below.

```
      <input id="route" type="number" name="routing" value="4275340171">
      <input id="csrfc" type="hidden" name="challenge" value="159691411">
      <input id="csrfr" type="hidden" name="response" value="">
    <input type="hidden" name="U3ByaW5nMjAxOVRhcmdldDFFYXN0ZXJFZ2c" value="U3ByaW5nMjAxOVRhcmdldDFFYXN0ZXJFZ2c">
       <div>
        <button class="btn submit" name="action" value="save">Save</button>
       </div>
      </form>
     </div>
     <script>

// fairly trivial string hashing function
String.prototype.hashCode = function(){
  var hash = 1;
  for (i = 0; i < this.length; i++) {
    hash = (13337 * hash + this.charCodeAt(i)) % 100000;
  }
  return hash;
}

var a = document.getElementById('account');
var r = document.getElementById('route');
function change() {
    var challenge = document.getElementById('csrfc').value;
    document.getElementById('csrfr').value = (a.value+challenge+r.value).hashCode()
}
a.onkeyup = change;
r.onkeyup = change;
change();
     </script>
     <div class="span4 offset2">
      <h3>Look up name</h3>
      <p>You may use this form to look up a user's name using their account ID</p>
      <form method="get">
       <label>account ID:</label>
       <input type="text" name="eid" value="">
       <div>
        <button class="btn submit">Look up</button>
       </div>
```

3. To prevent XSRF attack, we can include a random token with each request. The token would be a unique string generated for the user's unique "session ID" and would be included in every form as a hidden input.

```
public function get_token_id() {
        if(isset($_SESSION['token_id'])) {
                return $_SESSION['token_id'];
        } else {
                $token_id = $this->random(10);
                $_SESSION['token_id'] = $token_id;
                return $token_id;
        }
}
```

The get_token_id function retrieves the token from the user's session ID. If the session ID has

not been created yet, it will generate a random token.

```
public function get_token() {
        if(isset($_SESSION['token_value'])) {
                return $_SESSION['token_value'];
        } else {
                $token = hash('sha256', $this->random(500));
                $_SESSION['token_value'] = $token;
                return $token;
        }

}
```

This function retrieves the token value. If it has not been generated yet, a random token value will be created.

```
public function check_valid($method) {
        if($method == 'post' || $method == 'get') {
                $post = $_POST;
                $get = $_GET;
                if(isset(${$method}[$this->get_token_id()]) && (${$method}[$this->get_token_id()] == $this->get_token())) {
                        return true;
                } else {
                        return false;
                }
        } else {
                return false;
        }
}
```

This function determines whether the token id is valid and whether the token id of the GET or POST request is the same as the token value stored with the user's session ID.

< Example code coming from https://www.wikihow.com/Prevent-Cross-Site-Request-Forgery-(CSRF)-Attacks-in-PHP >

Target 2 Epilogue

```
<?php
// initialize global variables, authentication and database connections
include('includes/common.php');

// if the login or registration form has been submitted, handle it
$action = @$_POST['action'];
if ($action == 'login') {
        if($_POST['U3ByaW5nMjAxOVRhcmdldDNFYXN0ZXJFZ2c'] == 'U3ByaW5nMjAxOVRhcmdldDNFYXN0ZXJFZ2c') {
                $auth->login($_POST['login'], $_POST['pw']);
        }
} elseif ($action == 'register') {
        $auth->register($_POST['name'], $_POST['login'], $_POST['pw1'], $_POST['pw2']);
}

// if the user is logged in, redirect him to home.php
if ($auth->user_id()) {
        header('location: /account.php');
}


// otherwise, display a login page
include('includes/header.php');
?>
```

1. The vulnerable code is in index.php and the specific lines are highlighted in the screenshot.
2. XSS vulnerabilities can occur when the input is not properly escaped. In this case, there is no validation of the input we provide of any kind. This allows an attacker to write some malicious code which can steal sensitive data in a form like the bank related information in this particular case. Then, whatever entered in the account label via POST request is sent without any check, leading to the execution of arbitrary script to steal the credential information.
3. To prevent XSS attack, one way is to escape any character that can affect the structure of the document. That is, to make sure that every string is interpreted as a string literal, not as a control order or a segment of code. For example, we can use htmlspecialchars() function or htmlentities() function to convert special characters to HTML entities. Below are examples of the conversions.

```
'&' (ampersand) becomes '&amp;'
'"' (double quote) becomes '&quot;'
'<' (less than) becomes '&lt;'
'>' (greater than) becomes '&gt;'
```

Below is an example code that can escape all html, css or javascript.

```php
<?php
/**
 * Escape all HTML, JavaScript, and CSS
 *
 * @param string $input The input string
 * @param string $encoding Which character encoding are we using?
 * @return string
 */
function noHTML($input, $encoding = 'UTF-8')
{
    return htmlentities($input, ENT_QUOTES | ENT_HTML5, $encoding);
}

echo '<h2 title="', noHTML($title), '">', $articleTitle, '</h2>', "\n";
echo noHTML($some_data), "\n";
```

< Example code coming from https://www.virtuesecurity.com/preventing-cross-site-scripting-php/ , https://paragonie.com/blog/2015/06/preventing-xss-vulnerabilities-in-php-everything-you-need-know >

Target 3 Epilogue

```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);
        $sql = "SELECT user_id, name, eid FROM users WHERE eid='$escaped_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;
        }
}
```

1. The vulnerable code is in auth.php, login() function. The specific line is highlighted in the screenshot.
2. Even though the code used sqli_filter($username) function to escape the username, it didn't escape the password. We can add "' OR'1'='1" string, making the correctness of the password unnecessary to check, since "1=1" makes the OR statement always true.
3. There are several ways to prevent SQL injection or mitigate its impact
   a) Escaping strings.
   Escaping strings filters out special characters for use in SQL statements from user inputs.

```php
<?php
$username = mysqli_real_escape_string($conn,$_POST["username"]);
$password = mysqli_real_escape_string($conn,$_POST["password"]);
mysqli_close($conn);
?>
```

b) Using trim() and strip_tags()

Trim() is used for removing whitespaces at the beginning or end of an input string. Strip_tags() is used for removing html and php tags. Both of them can help remove additional codes and whitespaces commonly used by attackers.

```php
<?php
     $username = strip_tags(trim($_POST["username"]));
     $password = strip_tags(trim($_POST["password"]))
?>
```

c) Prepared statement

Prepared statement helps sanitize the input. It forces the user input to be handled as a string literal, and not as part of an SQL command. Below is an example of using prepared statements in php.

```php
<?php
    $username = $_POST["username"];
    $password = $_POST["password"];
    $stmt = $mysqli->prepare("SELECT FROM login WHERE user=? AND pa
    $stmt->mysqli_bind_param("ss",$username,$password);
    $stmt->execute();
    $stmt->close();
    $mysqli->close();
?>
```

Here what happens is that the SQL command you pass to prepare() is compiled by the server beforehand. Then you specify the parameter (such as a username), telling the server on which part of the database you want to execute the SQL command. After you call execute(), the prepared SQL command is combined with the parameters you give. The key is that the parameters are combined with the already compiled SQL statement, not an SQL command string. And whatever provided in the parameter will only be treated as input rather than an SQL command.

< Example code coming from https://www.lionblogger.com/ways-to-prevent-sql-injection-attacks-in-php/ >