CS 4235 Project 4 Report

Wenqi He, whe47

April 2, 2019

1 Target 1 Epilogue

1.1

The vulnerability lies in Line 18-29 of `/var/payroll/www/account.php`:

```
// 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) {
   ...
} else {
    ...
    $db->query(..);
    notify('Changes saved');
}
```

1.2

The given PHP code attempts to prevent CSRF by checking the submitted value of the hidden `csrfr` field against the value computed from `account`, `routing` fields and the CSRF token, which is first inserted into the `csrfc` field when the form is created and later submitted in the POST request. The vulnerability is that the server never checks the validity of the submitted CSRF token, probably under the assumption that a normal user would never tamper with a hidden field. The purpose of a randomly generated CSRF token is that it is a piece of information that an attacker cannot forge, therefore not validating the token completely defeats the purpose of having such a token. An attacker who knows the algorithm, perhaps by having access to the source code, can use essentially any value for the CSRF token, as long as the `csrfr` field is computed using the correct algorithm, thereby bypassing the CSRF prevention.

1.3

Since there is already a CSRF token in place, the easiest step to fix the vulnerability is simply to validate the submitted token first. For example, the conditional statement

```
if ($_POST['response'] != $expected) {
    // Reject request
}
```

can be changed to

```
if (
    $_POST['challenge'] != $_SESSION['csrf_token'] or
    $_POST['response'] != $expected
) {
    // Reject request
}
```

2 Target 2 Epilogue

2.1

The vulnerability lies in line 32 (and also 6-13) of `/var/payroll/www/index.php`

2.2

- The first vulnerability is in line 6-13 in the handling of POST requests:

  ```
  $action = @$_POST['action'];
  if ($action == 'login') {
      if($_POST['U3B...Z2c'] == 'U3B...Z2c') {
          $auth->login(...);
      }
  } elseif ($action == 'register') {
      $auth->register(...);
  }
  ```

  This is problematic because if the `action` field is omitted or does not match either `'login'` or `'register'`, or if `action` is set to `'login'` but the `U3B...Z2c` field does not match, the page still renders even though no login or registration is performed, which makes exploiting the following vulnerability possible.

- The *main* vulnerability in on line 32:

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

  Here the `'login'` field value from POST request is *directly* echoed into the rendered HTML, therefore an attacker can simply put whatever malicious HTML that he/she wants to inject at this location in the `'login'` field of the POST request, as long as all opening tags that come before will be closed and all closing tags the come after will have corresponding opening tags.

2.3

The special characters such as `<`, `>` and `"` which have special meanings to HTML parsers need to be escaped using HTML encoding. For example, `<` should be replaced with `<`, and `"` with `&quot;`. This will ensure that user inputs can never alter the structure of the document when inserted directly.

3 Target 3 Epilogue

3.1

The vulnerability lies in line 28-43 of `/var/payroll/www/includes/auth.php` in function `sqli_filter`

3.2

The given PHP code does have SQL injection prevention implemented, however the problem is that the list of forbidden character sequences is not exhaustive. The loophole that makes this particular exploit possible is that single quotes are not checked. Thus, it is possible to alter the `WHERE` clauses in the SQL statements. Suppose the `login` value is entered as follows `username' OR ''='`, then the first SQL statement for looking up salt value becomes

```
SELECT salt FROM users WHERE eid='username' OR ''=''
```

Since `''=''` always evalues to true, this is equivalent to

```
SELECT salt FROM users WHERE eid='username'
```

which will not cause any problem. The second SQL statementm, however, becomes

```
... WHERE eid='username' OR ''='' AND password='$hash'
```

Since `AND` will be evaluated before `OR`,this is the same as

```
... WHERE eid='username' OR password='$hash'
```

Therefore, even if the password is incorrect, this query will still returns the desired row, provided that `eid` exists in the databases. Since the success of login depends solely on the result of this query, as shown in the following code:

```
$userdata = $this->db->query($sql)->next();
if ($userdata) {
    ... // Logged in
} else {
    ...
}
```

this exploit guarantees that the user, even with incorrect password, can be logged in successfully.

3.3

One can simply add the following line to function `sqli_filter($string)` to prevent this exploit:

```
$filtered_string = str_replace("'","",$filtered_string);
```

However, in order to prevent SQL injections in general, the best practices for PHP include:

- Use prepared statements provided by PDO, MySQLi and other libraries.
- Check if the given input has the expected data type, using built-in input validating functions such as `is_numeric()`, `ctype_digit()` and Perl compatible Regular Expressions support.
- If numerical input is expected, silently change its type using `settype()` or `sprintf()`.
- Quote each non numeric user supplied value with the database-specific string escape function (e.g.`mysql_real_escape_string()`, `sqlite_escape_string()`, etc.).