CS 4235 Project 4 Report

Wenqi He, whe47

April 9, 2019

1 Target 1 Epilogue

1.1

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

```
$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 PHP server attempts to prevent CSRF by checking the submitted `csrfr` value against the value computed from `account`, `routing` and `csrfc` fields. The vulnerability is that, although the `csrfc` field is supposed to contain the CSRF token, the server never checks its validity, and therefore an attacker who has figured out the verification algorithm can put *any* value in the `csrfc` field, as long as the `csrfr` value is computed correctly based on `csrfc`.

1.3

Since the server is already issuing CSRF tokens, the easiest step to fix the vulnerability is to simply check the submitted token. For example, the conditional statement

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

can be changed to

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

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 control falls through (for example, if the `action` field is omitted), the page still renders, thereby exposing the XSS vulnerability.

- The XSS vulnerability is on line 32:

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

  The user submitted `'login'` value is echoed as is into the rendered document. Therefore, if an attacker puts malicious HTML instead of a valid username in the POST request, he/she can alter the content of the document at will, as long as the syntax of the document after injection is still valid.

2.3

The special characters such as `<`, `>` and `"` which have syntactic meanings in HTML must be escaped using HTML encoding. For example, `<` should be replaced with `<`, and `"` with `&quot;`. This can prevent user inputs from being recognized as HTML.

3 Target 3 Epilogue

3.1

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

3.2

The server does have SQLi prevention, but the list of filtered substrings in function `sqli_filter` is not exhaustive. Suppose the `login` value is entered as `username' OR ''='`. Because single quotes are not forbidden, this input will pass the filter. The first SQL statement then becomes

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

Since `''=''` always evaluates to true, this is still the same as

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

The second SQL statement, however, becomes

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

Because `AND` has a higher precedence than `OR`, this statement is equivalent to

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

This query will return the desired row even if the password is incorrect, because `eid='username' OR FALSE` is equivalent to `eid='username'`. Thus, by appending `' OR ''='` to a valid username, login can be achieved without knowing the password.

3.3

As a countermeasure to this particular exploit, one can simply add in function `sqli_filter`:

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

However, in order to prevent SQL injections in general in PHP, it is better to:

- Use prepared statements provided by libraries such as PDO and MySQLi.
- Check if the given input has the expected data type using built-in input validating functions such as `is_numeric()`, `ctype_digit()`, or 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, such as `mysql_real_escape_string()` and `sqlite_escape_string()`.