Project 4.5: Web Security Optional

mmendiola3

Task 1: XSRF Prevention

1 Patched code

The vulnerability exists because the server-side CSRF check is using the challenge value supplied by the form submission to create the hash that is then compared to the supplied response value. This allows an attacker to bypass the CSRF prevention mechanism by supplying the expected response for an attacker provided account, challenge, and routing input.

```
$teststr = $_POST['account'].$_POST['challenge'].$_POST['routing'];
```

This vulnerability was patched with the following changes:

```
$teststr = $_POST['account'].$_SESSION['csrf_token'].$_POST['routing'];
```

Furthermore, the expected response value is displayed on a failed CSRF attempt. This should be removed.

2 Improved Security

The security of the site has improved since requests to the server issued after initial login will be validated to ensure they are part of the same session.

3 Same Origin Policy

Same Origin Policy would not prevent this attack because the exploit page is not attempting to get the victim to run code from another origin. Rather, the exploit is simply having the user make a legitimate POST request from a pre-authorized browser.

Task 2: XSS Prevention

1 Patched code

In this case index.php sends unvalidated user input back to the client browser. This can be used to inject malicious code that is then run in the victim's browser:

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

The simplest fix for this issue is to remove the code that sends back the login data back to the user. Another option is to filter any embedded code before sending it back. The fix can be seen below:

```
<input type="text" name="login" value="">
```

2 Improved Security

Security is improved since an attacker will not be able to using this vector to submit malicious code to the victim's browser.

3 Same Origin Policy

XSS attacks are specifically intended to bypass the Same Origin Policy protection by getting the trusted server to send the malicious script to the victim. This makes the code seem as if it was acquired from the same origin.

Task 3: SQL Injection Prevention

1 Patched code

This vulnerability is exploitable because unfiltered data is passed directly into an SQL query. This allows an attacker to craft their input to subvert the intention of the query. In this case, the authentication process can be bypassed by ending the query before the section where the password is verified.

```
$sql = "SELECT user_id, name, eid FROM users
        WHERE eid='$username' AND password='$hash'";
```

To patch this vulnerability, the submitted username should be passed through the sqli_filter function before being used in the query:

```
$escaped_username = $this->sqli_filter($username);
$sql = "SELECT user_id, name, eid FROM users
        WHERE eid='$escaped_username' AND password='$hash'";
```

2 Improved Security

Security is improved by preventing arbitrary SQL statements from being submitted by an attacker.

Task 4: Further Security Fortification

Hashing

The simplistic hashing for CSRF prevention could be insecure, as it may result in a high number of hash collisions. If so, this would make it feasible to try a smaller number of response values when attempting to submit malicious form data. Fixing this would require implementing or importing a cryptographically secure hashing algorithm such as md5.

SQL Injection

The sqli_filter function will only stop a handful of injection attacks. It is still possible to modify the logic of queries without using the characters on the filter list. For example, an attacker could add "OR 1 = 1" which would disable the username matching section of the authentication query.

A better patch would be to use prepared sql statements. This is a mechanism that disconnects the instructions of the SQL statement from the data being processed. The following query would make the authentication query much more robust to attack:

```
$escaped_username = $this->sqli_filter($username);
$sql = $this->db->database->prepare("SELECT user_id, name, eid FROM users
                                    WHERE eid=:user AND password=:hash");
$sql->bindValue(':user', $escaped_username, SQLITE3_TEXT);
$sql->bindValue(':hash', $hash, SQLITE3_TEXT);
$result = $sql->execute();
$userdata = $result->fetchArray();
```

Payroll has many such instances of SQL input from data that hasn't been sanitized. Each of these should be refactored to utilize prepared statements along with using this PHP version's SQL special character escape function.