## Section 2: Moving On

### 1.1 Weaponize Vulnerability

In order for an attacker to create a post which appears to be from the user `dirks` with minimal user interaction, our exploit will rely on tricking the server into thinking we are `dirks`. Specifically, in **`server.py`**, the username is retrieved from **`@auth_helper.get_username`**. Then, after SQL-escaping the contents of the post, both the `username` and the contents of the `post` are inserted into the **`posts`** table via the command **`database.execute(“INSERT INTO posts VALUES (‘{}’, ‘{}’);”.format(username, post)`**.

The exploit here is that if **`@auth_helper.get_username`** returns `dirks` instead of `xoxogg`, then any post we make will be inserted as though it came from `dirks`. We can accomplish this by changing the cookie value, appropriately. Specifically, **`@auth_helper.get_username`** calls **`get_username_from_session`** which retrieves the `username` associated with the browser’s current `SESSION_ID` cookie via **`session = request.cookies.get(‘SESSION_ID’, ‘’)`**. Then, the username is pulled from the sessions table via **`database.fetchone(“SELECT username FROM sessions WHERE id=‘{}’;”.format(session)`**. Since the cookie value is not SQL-escaped, we can change the value of the cookie in the browser and perform a SQL injection attack. Specifically, we can edit the cookie value to be `“irrelevant’ OR username=‘dirks”`. Then, the SQL query becomes **`SELECT username FROM sessions WHERE id=‘irrelevant’ OR username=‘dirks’`**. Since the **`sessions`** table schema comprises both an `id` column and a `username` column, we are essentially bypassing the first half of the WHERE condition (since there is no `id=‘irrelevant’`), and simply selecting all string instances of `dirks` in the **`sessions`** table. Finally, since **`get_username_from_session`** returns the first value in this list of strings via the line **`username = found_session[0] if found_session else None`**, it will simply return the username `dirks`. So, the `post()` function in **`server.py`** will insert into the **`posts`** table the following record: (username=“`dirks`”, post=“...”), thereby making it appear as thought the post came from `dirks`.

### 1.2 Vulnerability Writeup

#### 1.2.1 Test 1

This vulnerability is essentially a stored XSS attack when a user submits a post to his or her wall. Specifically, if a user places malicious JavaScript between two script tags, then anytime someone visits the user’s wall, the script will run. For instance, *xoxogg* can input a new HTML element as follows: *<input type="checkbox" name="arbitrary" onclick="console.log(document.cookie)">*. This will log a visiting user’s cookie value into *xoxogg*’s console. As you can see, the consequences of this attack can be pretty extensive, as any malicious code placed between the script tags will execute. This vulnerability occurs in lines 194 and 195 of **`server.py`**…

```
194   post = escape_sql(request.form['post'])
195   database.execute("INSERT INTO posts VALUES ('{}',
      '{}');".format(username, post))
```

As we can see, the contents of the post are SQL-escaped but not HTML-escaped, which means that < is not replaced with &lt; and > is not replaced with &gt;. This vulnerability can be addressed by calling the `escape_html()` function on the content of the post. Then, the script tags will be treated as strings instead of valid tags. In general, to protect against this vulnerability, one should use existing libraries and frameworks which provide input sanitation.

### 1.2.2 Test 5

This vulnerability hinges on a SQL injection attack on the SESSION_ID cookie. Since this cookie value is not properly sanitized before being used in the SQL query, we can change the value of the cookie in the browser and perform a SQL injection attack. The way we do this is by opening the HTML console and setting `document.cookie="SESSION_ID=irrelevant' OR username='dirks"`. This vulnerability occurs in lines 19 and 20 of `auth_helper.py`…

```
19   session = request.cookies.get('SESSION_ID', "")
20   found_session = database.fetchone("SELECT username FROM sessions WHERE
     id = '{}';".format(session))
```

By exploiting this vulnerability, an attacker can gain unauthorized information about another user and even impersonate him or her. The attacker could also perform serious harm on the backend servers, possibly even deleting the database via the DROP TABLE command in SQL.

The vulnerability can be fixed by adding a line in between lines 19 and 20 which escapes SQL queries, akin to `escape_sql` in `server.py`, such that the quotes in `dirks` get replaced, and the user is unable to supply the OR condition. In general, SQL injection attacks can be addressed by sanitizing user input and ensuring that the value does not have commands of any sort. One can and should also use existing tools or frameworks, such as the Django web framework, which has built in sanitization and protection for other common vulnerabilities.

### 1.2.3 Test 6

This vulnerability hinges on exploiting directories and subdirectories under a given domain. Specifically, if we look at lines 52-56 of `avatar_helper.py`…

```
52   user_dir = init_user_dir(username)
..
54   assert allowed_path(avatar_filename)
55   assert os.path.isfile(avatar_filename)
56   assert user_dir in avatar_filename
```

we can see that the code ensures that an attacker cannot navigate to directories that is not his or her own. However, so long as an attacker has a valid username, he or she can can set the image source to be any value within the filename directory and can, therefore, pretend to be a different user with a different avatar image. To address this vulnerability, one can design a system which requires authentication tokens to delete an image. To avoid this vulnerability in general, it is important to design one's directories in a way that secures least privilege for an authenticated user. In other words, an authenticated user should not be allowed to navigate to directories that are not in his or her own jurisdiction.

## 1.3 Other Issues

### 1.3.1 Issue 1

The first issue pertains to how passwords are stored in the users table. Based on lines 11 and 12 in **`auth_helper.py`**…

```
11   hashed_password = sha256(password.encode()).hexdigest()
12   correct_hash = database.fetchone("SELECT hash FROM users WHERE
     username='{}';".format(username))
```

we can see that the SHA256 hash of each user’s password is stored in the users table. The `check_login()` function simply verifies the hash of the inputted password against the stored hash value in the database. However, this password storing technique is vulnerable to an amortized password guessing attack, where an attacker attacker scans through pre-computed hashes of a large number of common passwords as opposed to plain-text versions of the passwords, in order to recover the original password. To prevent this attack, Snapitterbook should hash each user’s password with a unique salt, such as the username, as discussed in lecture. This severely limits the capacity at which the attacker can launch an attack.

### 1.3.2 Issue 2

The second issue with Snapitterbook revolves around the handling of the SESSION_ID cookie when a user logs out. Currently, upon logout, the cookie value is set to an empty string, as we can see from line 115 in **`server.py`**.

```
11   resp.set_cookie(‘SESSION_ID’, ‘’)
```

However, because the corresponding record in the **`sessions`** table that pairs `username` with `SESSION_ID` is not deleted, old cookies do not expire. Therefore, if an attacker obtains a user’s cookie, he or she will be able to access the user’s account, even if the user is logged out, and can use that cookie value to impersonate the user. This vulnerability can be addressed by removing the corresponding (`session_id`, `username`) record from the **`sessions`** table upon logout, so that an attacker cannot use an expired cookie value as it will not appear in the table.

### 1.3.3 Issue 3

Another issue with the website centers around the validity of the avatar image upload. The server only checks to see whether the image upload is of type .jpg. However, naming the file in an obscure manner, such as including numbers and spaces, renders the avatar image on the wall as invalid. In order to protect against this attack, the server should check to see whether the profile image file points to an actual image file, as opposed to naming conventions of the image.