Jonathan Briggs
3032095561
Project Three Writeup
CS161 – Fall 2018

Weaponized Vulnerability

Two of the vulnerabilities on the "Profile" page for the user enable us to change the password of other accounts. Then, we can takeover the account by logging in with the new credentials we supplied the database, which allows us to post or make changes on behalf of the compromised user. Using this, we will gain access to the account belonging to Dirks and make posts on his wall.

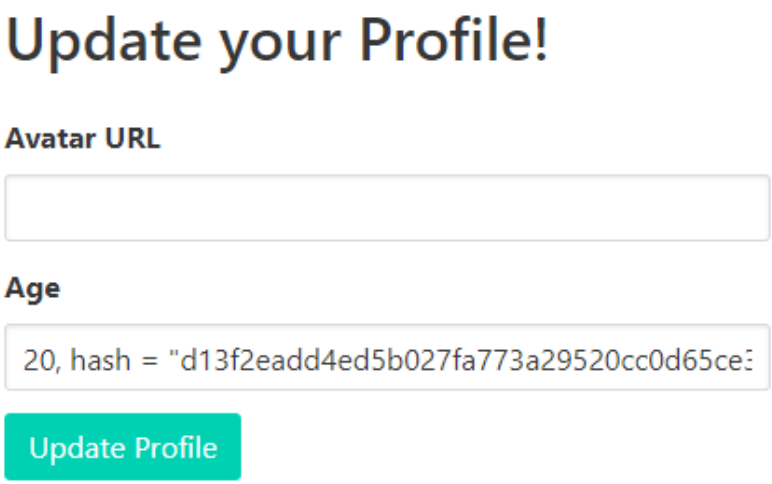
The first vulnerability is in a hidden field within the profile page's HTML. We can use this hidden field to change the username that is affected by the post request that we send to the database. Combining this with the second vulnerability on the profile page, which is a SQL injection, means that we can change the password for any desired user. Here is an example, which we use to change dirk's password:

```
▶<div class="field">…</div>
▼<div class="field">
    <label class="label" for="age">Age</label>
    <input class="input" id="age" type="text" name="age" valu
    "0">
  </div>
  <input type="text" class="input" style="display:none" id=
  "username" name="username" value="dirks"> == $0
```

Here, within the HTML, we change the text in the 'value' field from "xoxogg" to "dirks". Then, when we send a POST request, the code in server.py will execute a SQL statement and query the database for dirks' parameters to update any changes that were made to his profile.

```
database.execute("UPDATE users SET age={} WHERE username='{}';".format(age, username))
```

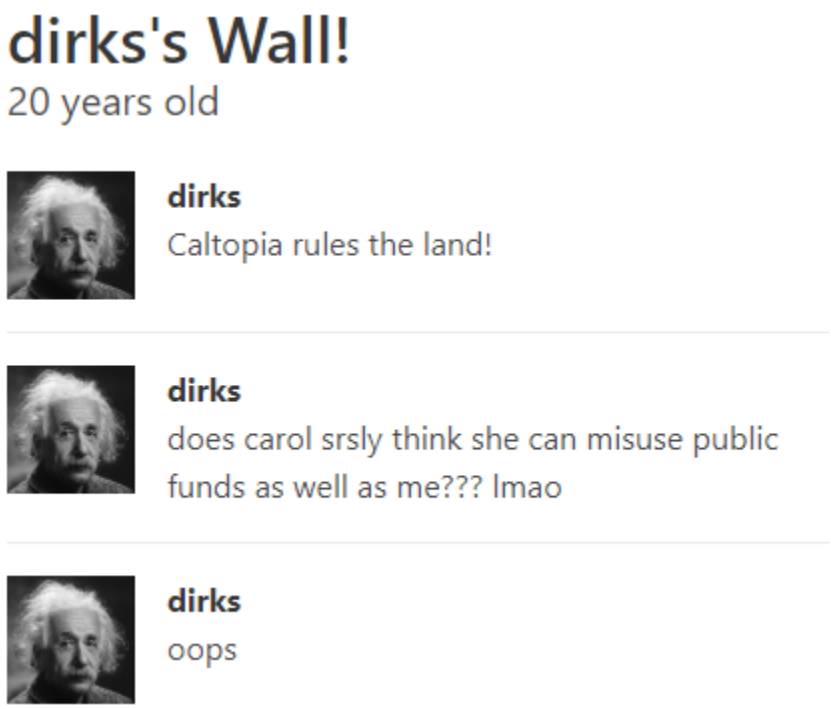
Also, as we can see in the modified HTML above, we can change the 'type' field for the age parameter on the profile page from "number" to "text." This allows us to enter the following string into the field, which will cause the password (which is stored in the field 'hash' within the users table) for dirks to be updated to "oops" (which is the SHA256 hash of "d13f2eadd4ed5b027fa773a29520cc0d65c-e374365d641112de786f8a029c2fe).



This SQL injection not only enables us to change dirk's age, but also his password! Now, we simply logout of our account and login as dirks with the following parameters:

Username: dirks
Password: oops

Now, we can post whatever we would like to dirk's wall. Oops! That's what you get for misusing public funds, I guess!!!

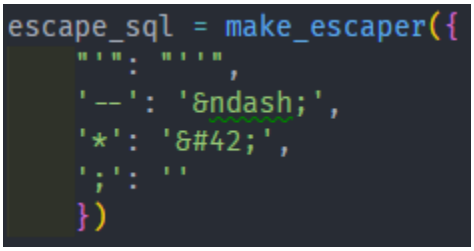


Vulnerability Writeup

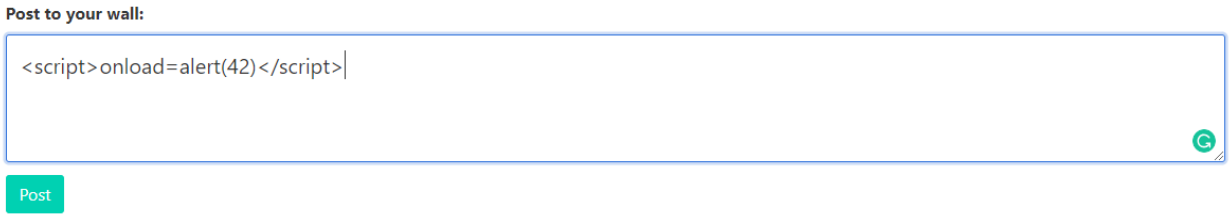
1. Stored XSS (Posting to Wall):
   The first vulnerability that I will discuss is a simple example of stored XSS.

```
@app.route('/post', methods=['GET', 'POST'])
@auth_helper.get_username
@auth_helper.csrf_protect
def post(username):
    if not username:
        return render_template('login.html', error='Please log in.')

    if request.method == 'GET':
        return render_template('post.html', username=username)

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

As we can see in the screenshot above, the server executes the INSERT statement after a call to the ‘escape_sql’ function. While this serves to prevent (some more naïve and less complicated) SQL injections, it does nothing to escape any HTML tags.

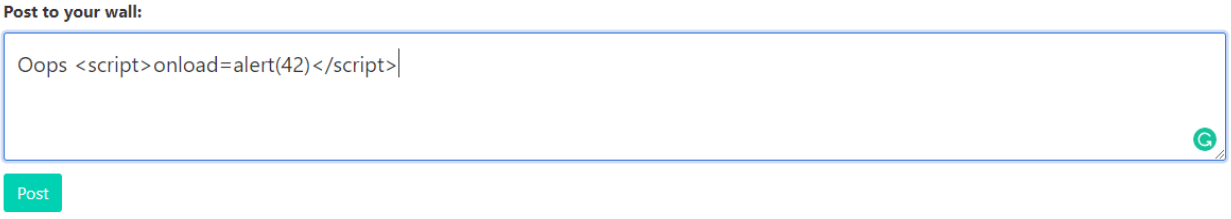


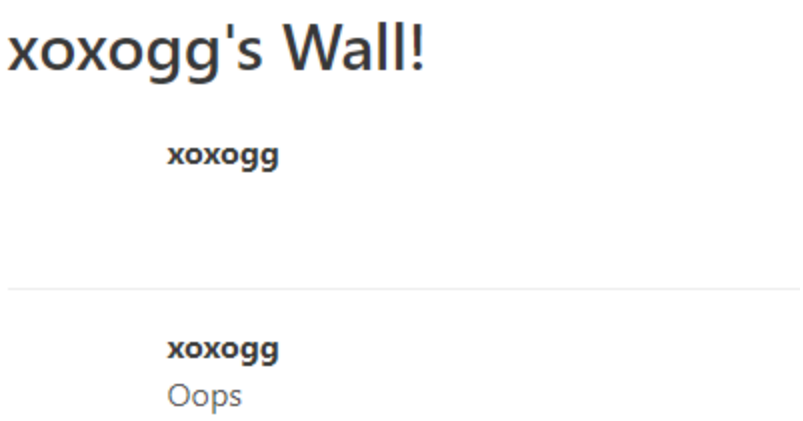
Due to these oversights, we can post a script directly to our wall. Now, when anybody loads our wall, the script will execute.



A simple and relatively harmless demonstration of this is simply displaying an alert when the wall is loaded, as shown in the post above and the screenshot below, but a much more malicious script could easily take its place. In fact, the attacker can run any Javascript that they please. This is extremely dangerous.

Further, if we simply add text to our post, the user will likely be none the wiser. As we can see in the screenshots below, our first post (without additional text) looks a little suspect. However, our second post looks entirely regular. Oops, indeed…





There are several potential ways that we can go about fixing this vulnerability. Here, we will draw on two excellent examples from the OWASP XSS Prevention Cheat Sheet. One idea is to escape untrusted data and sanitize our input before inserting it into our database or HTML. Our attack can be prevented easily with the suggested sanitation from the cheat sheet, shown below.

```
 & --> &amp;
 < --> &lt;
 > --> &gt;
 " --> &quot;
 ' --> &#x27;     &apos; not recommended because its not in the HTML spec (See: section 24.4.1) &apos; is in the XML and
XHTML specs.
 / --> &#x2F;     forward slash is included as it helps end an HTML entity
```

This turns our input into a harmless ASCII string. Now, if we were to run this recursively on our input string, we can prevent many forms of stored XSS and have effectively bandaged the wound.

Another would be to implement and use the proper security methods to our HTTP. By taking advantage of X-XSS-Protection Response Headers that are built into modern web browsers, we can prevent Javascript from being stored and executed on our website.

2. <u>SQL Injection via Cookies:</u>
   Another vulnerability in the server is the ability to inject SQL commands. One example of this was shown in the “weaponized” vulnerability earlier, which used profile updates to inject and execute commands. However, there is another vulnerable function that can be found in the code that loads a user’s own wall.
   For example, when we login as a user, we are automatically redirected to our own wall.

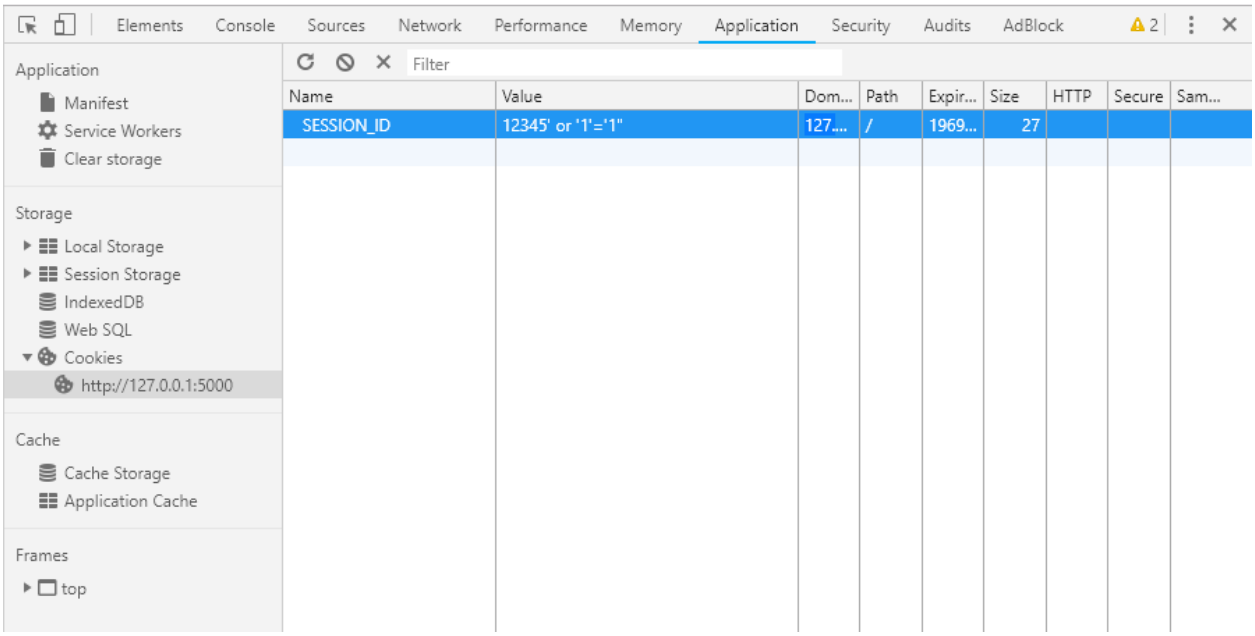
When we do so, the following code is executed. Pay special attention to the highlighted function, get_username_from_session().

```
@app.route('/wall')
@app.route('/wall/<other_username>')
@auth_helper.get_username
@auth_helper.csrf_protect
def wall(username, other_username=None):
    other_username = other_username or auth_helper.get_username_from_session()
    if not other_username:
        return redirect(url_for('index'))

    other_username = escape_sql(other_username)
    if not auth_helper.is_valid_username(other_username):
        return render_template('no_wall.html', username=other_username)

    db_posts = database.fetchall("SELECT post FROM posts WHERE username='{}';".format
        (other_username))
    posts = [ post[0] for post in db_posts ]
    avatar, age = get_user_info(other_username)

    return render_template('wall.html', username=username, other_username=other_username,
        posts=posts, avatar=avatar, age=age)
```

When the user navigates to their own wall, this function is called. In it, cookies are used to determine which user is navigating to the page and the server fetches from the current user by querying the “sessions” table.

```
def get_username_from_session():
    session = request.cookies.get('SESSION_ID', '')
    found_session = database.fetchone("SELECT username FROM sessions WHERE id='{}';".format(session)
        )
    username = found_session[0] if found_session else None
    return username
```

A simple, manual edit to our own cookie causes this unsanitized user input to be executed directly in our query. Here, we are using the example from lecture, but we can do whatever we please. As shown, the server includes our unsanitized string directly in its SQL command!



While we are currently only place in a relatively harmless and simple string, anything more malicious could easily take its place should the wrong user stumble across this exploit. For example, they could drop tables with the command “0; DROP TABLE users; --" or fetch another user’s cookie from the sessions table. The possibilities are almost endless, and frightening, too!

It would be hard to sanitize the input for something we have no control over (the user can change their cookie to whatever they please), so we need to take a different approach to correcting this mistake. A good idea is using bound parameters, which is also known as the “prepare” statement. This way, SQL statement strings are created with placeholders and compiled into an internal form. Afterwards, these prepared queries are executed using a list of parameters. This created a parse tree of sorts that parses the placeholder in place, which prevents attacks of this sort. Another good step to take would be limiting database permissions and segregating users. For example, allowing limited rights such as query-only access to the users table, and no access to any others, would help to mitigate these attacks. While it does require some reworking of the server code, it is well worth the hassle!

Other Issues

1. Passwords:
   The first thing that I sought to exploit was the login page. Immediately, after looking at the code, I noticed that the passwords are stored in the database with only a simple SHA2 56 hash. While infinitely more robust than simply storing the actual password string itself, this can easily be sniffed out and exploited by a nosy and experienced hacker. For example, a dictionary attack is a simple method of undermining the server’s security features. A simple fix for this is using a strong salt.

2. Character escaping:
   Another vulnerability that is readily apparent with a simple glance through the code is the character escaping (or rather, the lack thereof). While the server does take steps towards preventing SQL injections and escaping some HTML content, it falls short in a variety of ways. A much more robust tactic would be to use whitelists instead of blacklists.
3. Cookies:
   The server uses cookies to host sessions. In fact, it does many things right regarding managing cookies correctly and safety. For example, the server generates them anew when a URL is entered to prevent impersonation. This method of on-the-fly cookie generation is a great first step. However, there are two major flaws with the server's cookie implementation. First and foremost, session IDs do not expire. Further, when the user logs out, the ID is only cleared from the browser and not the database itself. In fact, the server stores cookies in the database with no timeout or expiration date. As a result, we can impersonate a user of we are able to guess or otherwise obtain any of their previous session IDs. If we simply give the cookies a short period of validity, after which the ID expires, that solves the first problem easily. Tackling the second is trivial as well: we simply need to remove the entry from the "sessions" table when the session is no longer valid or the user has logged out.

Feedback

I really enjoyed this project. It was like a more complex version of the problem on our most recent homework where we were seeking to find the vulnerabilities on a website. However, without any hints or directions, it was both more challenging and more fun as well! Ultimately, although I had trouble finding the last two vulnerabilities, it was engaging and rewarding.