Brandon Wang
UC Berkeley CS-161

For this project, I used Google Chrome under brantheman60@berkeley.edu

1. **Log in as user test**
   By doing View Page Source on the Login page, we see the comment <!— Demo Login/Password: Username = 'test', Password = 'takethisoutwhenpushingtomaster' —>, which tells us how to log in as user 'test'.

   To prevent this security issue, remove all of these kinds of HTML comments before publicly publishing the website.

2. **Change the text of ip.txt**
   The exploit here is that when a user is shared a file with the same name as an existing file, the existing file will be overwritten. So, first I created and uploaded a file called ip2.txt, which contained the content '161.161.161.161'. Then, I renamed it to ip.txt. Finally, I shared ip.txt to cs161, thus changing their content to my file content.

   To prevent this security issue, write a line of code that prevents the website from accepting files with the same filename, or changing the receiving filename (eg. "ip(1).txt").

3. **Obtain shomil's password hash**
   The Sign up page probably uses the following SQL code to search if the user is in the database:
   SELECT username FROM users WHERE username='%s'
   If I input:
   Username: '; SELECT md5_hash FROM users WHERE username = 'shomil
   Password: asdf (to make the input valid)
   then the new code will be SELECT username FROM users WHERE username=''; SELECT md5_hash FROM users WHERE username = 'shomil'
   and will only return the second request: 'Username 7f3af3a3ffd282bc516d4c45efa9112d already exists!'

   To prevent this security issue, sanitize the input first to stop the input from being read as SQL code. For instance, the code could add a backslash '\' in front of every single quote and double quote to inform the program that the user's input is a string and should not terminate the username string input. Additionally, the developer could use prepared statements to ensure only a specific value type can be accepted in the SELECT statement.

4. **Gain access to nicholas's account**
   When I log in as any user, I automatically create a session cookie which can be viewed and edited in Inspect -> Applications -> Cookies. The website runs SELECT username, expires FROM sessions WHERE token = '%s' whenever I send an HTTPS request by uploading/listing/sharing files, so my SQL injection attack had to be done by changing the session_token value in the cookie. Since I cannot DROP, INSERT, or UPDATE the sessions table, I simply set the session_token to ' UNION SELECT 'nicholas', '2585828363, which produces the new code SELECT username, expires FROM sessions WHERE token = '' UNION SELECT 'nicholas', '2585828363' that returns 'nicholas' as username and

'2585828363' (any valid expiration value) as expires.

Much like Flag 3, to prevent this security issue, sanitize the session_token value first to prevent it from read as SQL code. For instance, the code could add a backslash '\' in front of every single quote and double quote to inform the program that the cookie should not terminate the username string input.

5. **Leak cs161's session cookie**
   UnicornBox contains a vulnerability where if you rename any file to any type of HTML code (which can include Javascript script eg. <script>alert("Hello!");</script>), it will use this HTML code when opening the "List files" page. To share this exploit with cs161 and to fetch their document cookie, I uploaded a random file called hi.js, then renamed it to <script>fetch('/report?cookie=' + document.cookie)</script>, then shared this file with cs161. The next time cs161 opens the "List files" page, he will inadvertently run the Javascript script and send an HTTP request to the /report?cookie page.

   To prevent this security issue, the easiest solution would be to perform HTML sanitization on the file name when renaming a file, specifically for keywords such as <script>, <object>, <embed>, and <link> which are recognized as the insertion of new Javascript code. By filtering the inputs and preventing attacker-created malicious Javascript code from being run in this fashion, users will no longer to susceptible to an XSS Stored Attack when they enter the site.

6. **Create a link that deletes users' files**
   I noticed that List Files -> Search for a file is vulnerable to an XSS Reflected Attack, so it would run Javascript code that would delete the user's files if I entered <script>fetch('/site/deleteFiles', { method: 'POST' });</script> into the search bar and entered. To make any user encounter the same vulnerability, I created a URL that replicates the steps above; it enters the malicious Javascript code in 'Search for a file' and searches. This is the resulting URL:
   https://proj3.cs161.org/site/search?term=%3Cscript%3Efetch%28%27%2Fsite%2FdeleteFiles%27%2C+%7B+method%3A+%27POST%27+%7D%29%3B%3C%2Fscript%3E

   Much like Flag 6, to prevent this security issue, the easiest solution would be to perform HTML sanitization on the 'Search for a file' input, specifically for keywords such as <script>, <object>, <embed>, and <link> which are recognized as the insertion of new Javascript code. By preventing attacker-created malicious Javascript code from being run in this fashion, users will no longer to susceptible to an XSS Reflected Attack when they click on the URL. Another solution would be to add a web application firewall to prevent certain POST requests.

7. **Gain access to the admin panel**
   First, I found the admin's md5_hash password (82080600934821faf0bc59cba79964bc) in the users database using a similar SQL injection attack as in Flag 3. Then, I ran a dictionary attack using the known hash value and known hashing algorithm to identify a matching

password that hashes to 82080600934821faf0bc59cba79964bc (in reality, I plugged the hash into https://www.md5online.org/md5-decrypt.html, which runs the dictionary attack on a huge password database much faster than I could have done by myself on my own computer). I used the first password I found (P@ssword123, which looks like a very typical password) for the administrator account, which worked because admin had used the same password for his user account as for his administrator account.

To prevent this security issue, the developer could add a cryptographic salt to the password before md5 hashing it. This won't entirely prevent a dictionary attack on a single particular user, but salt hashing will make it much harder and take much longer to generate a hash table for common passwords. Of course, another protection against a dictionary attack (that's also much more secure) is having admin use different passwords for his user and administrator accounts, or at least using a very strong password that would be difficult to find with a dictionary attack.

Notes:

users

id	username	md5_hash	(password, determined by reversing hash or otherwise)
1	test	024c261beea4fdc2640b477e5c2a8dcb	takethisoutwhenpushingtomaster
2	admin	82080600934821faf0bc59cba79964bc	P@ssword123
3	shomil	7f3af3a3ffd282bc516d4c45efa9112d	
4	nicholas	5a1c83ebdadfde0d5779d0a3a0337620	
5	cs161	198a529cef118b643014de9ec753ca7c	

sessions

id	username	token	expires
1	shomil	2uzW9Ka8imGHYMSfwiyyvA==	2585828363
2	test	37g+Fi3TpYldRT09S009Yw==	1596691231

```
CREATE TABLE IF NOT EXISTS users (
  id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
  username TEXT,
  md5_hash TEXT
);

CREATE TABLE IF NOT EXISTS sessions (
  id INTEGER NOT NULL PRIMARY KEY,
  username TEXT,
  token TEXT,
  expires INTEGER
);

INSERT INTO users VALUES(NULL, 'test', '024c261beea4fdc2640b477e5c2a8dcb');
INSERT INTO users VALUES(NULL, 'admin', '82080600934821faf0bc59cba79964bc');
INSERT INTO users VALUES(NULL, 'shomil', '7f3af3a3ffd282bc516d4c45efa9112d');
INSERT INTO users VALUES(NULL, 'nicholas', '5a1c83ebdadfde0d5779d0a3a0337620');
INSERT INTO users VALUES(NULL, 'cs161', '198a529cef118b643014de9ec753ca7c');
INSERT INTO sessions VALUES(NULL, 'shomil', '2uzW9Ka8imGHYMSfwiyyvA==',
2585828363);
INSERT INTO sessions VALUES(NULL, 'test', '37g+Fi3TpYldRT09S009Yw==', 1596691231);
```