**ACTS 3, 4, 5, 6, and 7 QAs**

Jonah Chrzanowski, Smayan Daruka, Tyler Olexa

GITHUB REPO: <https://github.com/tjo1858/CSEC380-Project>

**ACT3:**

* Provide a link to the test cases you generated for this activity.
* How do you ensure that users that navigate to the protected pages cannot bypass authentication requirements?

We ensure that users cannot navigate to protected pages by creating a new session everytime a user logins in. This limits other people from accessing pages that are locked behind our authentication. These session IDs are long and difficult to guess, making bypassing the authentication requirements much more difficult. We also redirect the user back to the login page if there is not an active session or if the GET method is being used for requests when logging in. All this ensures that it is extremely difficult to bypass authentication.

* How do you protect against session fixation?

Session fixation can be protected against by generating a unique unpredictable session ID for each user when they log in. We also make these session IDs expire as soon as they close the page and/orlog out, thus ensuring that it is not possible to hijack an active session since a single session ID cannot be used twice. In other words, even if an attacker were to determine the session ID being used, it would be of no use since the session ID when used again would be considered invalid.

* How do you ensure that if your database gets stolen passwords aren’t exposed?

We take a salted hash of the passwords using the Werkzeug library in Python in our database. We then hash the inputted password from the user and check the hash against the salted hash stored in the database. This prevents a user from using a hash as a password and prevents a user from having a plain text password if the SQL database was ever leaked.

* How do you prevent password brute force?

We implemented rate-limiting on our website to ensure that passwords cannot be brute forced. The way this is set up is that we are only allowing 10 requests a minute to our login page from a specific source. This prevents an attacker from implementing a brute force attack.

* How do you prevent username enumeration?

Username enumeration can be prevented by not serving a specific message when a client enters an incorrect username and password. When a username is entered incorrectly, instead of responding with something like "incorrect username", respond with something like, "incorrect username and/or password." This will prevent enumeration of any usernames.

* What happens if your sessionID is predictable, how do you prevent that?

If the session ID was predictable, which ours is not as stated in a previous question, you could manually set the session ID to be a series of cryptographically secure random bytes. This would make the session ID less predictable and more secure, stopping session fixation.

**ACT4:**

* How do you prevent XSS is this step when displaying the username of the user who uploaded the video?

We do a check of the session ID of the user logged in and if the user ID is not the same as the ID that uploaded the video, then the database pulls the name of the user that did upload the video and displays it. There's no opportunity to use XSS to change the name, since it's already in the database.

* How do you ensure that users can’t delete videos that aren’t their own?

Users cannot delete videos uploaded by other people since we check for the current session username and compare it against the username of the person that uploaded the original video. By ensuring that videos can only be deleted when the two usernames match, we can successfully mitigate the problem of users deleting other peoples' videos.

**ACT5:**

* How would you fix your code so that these issues were no longer present?

Go back to using the pymysql module instead of the MySQLdb module. The pymysql module doesn't allow the user to execute multiple lines of sql in the cursor.execute() call, thereby mitigating the ability to inject multiple lines of sql code.

* What are the limitations, if any that, of the SQL Injection issues you’ve included?

Blind sql injection was tough to exploit because none of our sql select statements show any output to the user.

**ACT6:**

* How would you fix your code so that this issue is no longer present?

One possible way to fix this vulnerability is to check the type of file being uploaded. Since this is a video sharing platform, we could parse the URL to see whether it ends with .mp4 or other video file formats and disregard other links. This way, we can possibly ensure that an attacker will not be able to perform SSRF.

* How does your test demonstrate SSRF as opposed to just accessing any old endpoint.

The demonstration video shows that the URL we entered into our link upload results with a 200 code. This proves that the request was made by the server itself and not any other endpoint.

**ACT7:**

* How would you fix your code so that this issue is no longer present?

Don't use os.system calls liberally, only use specific os calls like os.remove for file deletion. Also include some kind of input sanitizaiton process to prevent users from escaping input fields and modifying/injecting code.