Assignment: *Mini-Project 1 - Salt & Pepper Hash and Web Exploit Prevention*
Team Name: *Secure the Flag*
Team Members: *Amber Fisher, Giovanni Gaspard, Jesse Boakye-Donkor, Daniel Hamilton*
GitHub Link: *https://github.com/kwaku97/secure-the-flag*

SQL Injection

After a user creates an account, the user has the ability to post data to their account. After logging in, the user has the ability to retrieve data only if they know the name of the account they are trying to retrieve the data from. Initially the user was able to post any data for the search input. Without cleaning the value, the input was vulnerable to an SQL injection attack. If the user were to put the line a' or 'a' = 'a. Essentially, this statement is always true, so the whole database would be queried. This will cause all posts in the database to be exposed. So if there were any private accounts, these accounts would be exposed. This is considered a reflected attack due to it running as soon as the data is received. How I solved this was to remove the ability for the user to use certain characters that would allow for a query and convert the data on the backend.

Cross Site Scripting (XSS)

After logging into an account, a user can create POSTs to the server and view POSTs that other users have made. Due to this feature, our unprotected site contains a Stored XSS vulnerability where a user can embed JavaScript code into a POST using HTML tags described in Security Shepherd's XSS lesson (<img>, <script>, etc). Once another user retrieves POSTs from the attacker's username, the victim's browser will interpret the attacker's carefully constructed POST as valid HTML, accidentally running the malicious JavaScript code. This is considered a Stored XSS attack because the attack string is stored on the server and triggered by a victim's action to open the stored message on the client's browser.

To mitigate this attack, I replaced key characters (<>"'/[]&) with their safe HTML entity alternatives (&lt, &gt, etc). This way, the malicious string cannot be interpreted as HTML code to be rendered by the browser. Additionally, my sanitization functions will *recursively* remove event handler names (regardless of capitalization) until they do not exist in the string anymore. For example, a clever attacker may make a POST like "onerONMOUSEOVERror," expecting ONMOUSEOVER to be detected and removed, leaving only "onerror." Potentially, this could be used as an event handler too; however, my recursive sanitization function will catch the nested event handlers and remove both. Note, this mitigation only filters *new entries* into the database. If attacks already are stored in the database, this *will not* protect users against them.

Insecure Direct Object Reference (IDOR) + URL Manipulation

After creating an account and logging in, only a user can add posts and view their posts on their account. The requests made to the server for adding and viewing posts uses the username as a parameter. On the unprotected version of our site, the username is encoded with a base64 encoding. This is insecure because a malicious attacker can encode any username and place it into the request parameters to add or retrieve the posts of any user.

To close up this vulnerability on the secure version of our site, we used a JSON web token to encrypt the usernames. This JSON web token signs the username with a secret key and sets an expiration time for the token. This process makes it difficult for an attacker to set their own token, since they do not know the secret key. Also, the token expires and is always recreated upon logging in.

Cross-Site Request Forgery (CSRF) + Session Hijacking

The encryption of the usernames as described above in addition to an encrypted CSRF token in each POST request means only the verified user can add a post to the website. Due to the same-origin policy, GET requests do not need to have a CSRF token because the attacker's site cannot see what a request returns nor a user's cookies. In the feeble version of the website, an attacker could easily get the encoded username if they knew the plain-text username since they only needed to base64-encode it. They could then forge a POST request with the encoded username and their malicious post. In the hardened version, these two issues are mitigated using the encrypted username and CSRF token, which both change when a user logs out and logs back in. Again, due to SOP, the attacker cannot see the user's cookies. As a result, an attacker cannot just guess the username or CSRF token since they would have to guess the correct hash scheme and number of salt rounds. The hardened POST request will fail if 1) the username is encrypted incorrectly, 2) the CSRF token is not present, or 3) the CSRF token is incorrect when compared against the encrypted username. Hence, the website will be secure against CSRF attacks since the attacker cannot forge a POST request without using brute force.

Password Hashing vs Password Encryption

The difference between hashing a password and encrypting a password is simple. When a password is encrypted, it means the password can be decrypted with a key. It is a two way function,essentially a lock that will open with the proper key. Hashing is a one way function which takes the password and scrambles the password to a unique test/message. Hashing a password is irreversible. Hashing is safer for the reason that it's reversible. Usually a key is lingering somewhere for situations where you may need to decrypt for comparisons. It's not impossible to get a password from a hashed password, it just depends on what type of hash you use. Some algorithms are safer than others.

Password Salting & Peppering

Salt is randomly-generated data added to a password prior to the hash. Pepper is similar but the data is not stored with the hashed password but instead hidden away.The problem is that the pepper has to be stored somewhere, which requires extra resources that successful password hashing will always be reliant on.