**Project Report**

**CSD102 Hackathon 2**

**23 April, 2023**

**Problem no.2:**

Build a login system for a web application and ensure that user passwords are stored securely.

Describe a system that:

Stores user passwords in a way that makes it difficult for an attacker who gains access to the database to determine users' actual passwords.

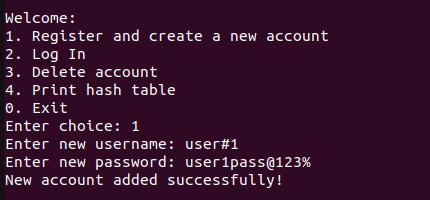
Allows users to log in by verifying their password without storing the password in plain text.

Provides an additional layer of security to prevent brute-force attack

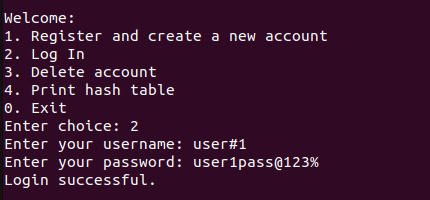
**Program written by**:

* Aditya Raghuram (2210110126)
* Arnav Aditya (2210110189)
* Arnav Jalan (2210110192)
* Rachit Sen (2210110489)

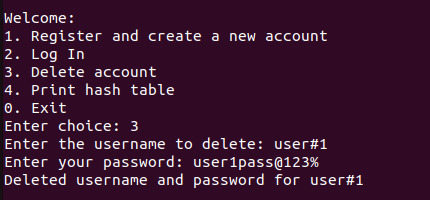
**OUTPUT IMAGES:**

****

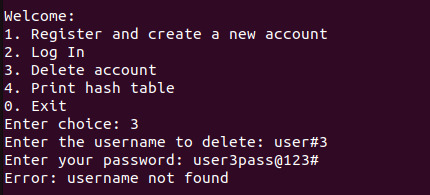
Registering a New Account

****

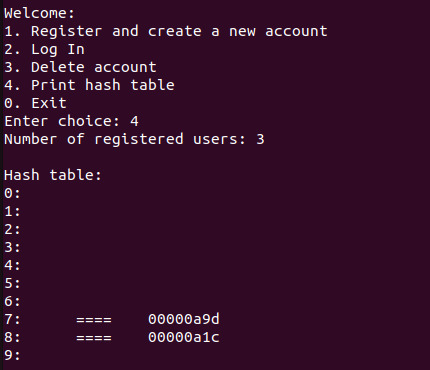
Successful Login

****

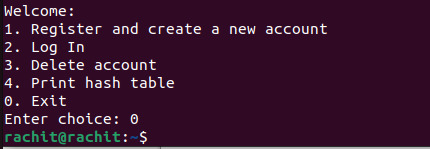
Deleting an Account

****

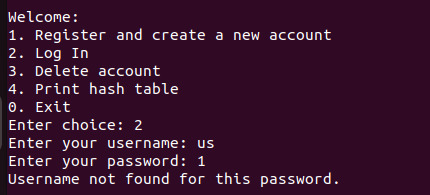
Deleting an account which does not exist



Printing the Hash Table

****

Exiting the Program



Unsuccessful Login

**TIME COMPLEXITY:**

* The time complexity of the hash function is O(n), where n is the length of the string.
* The time complexity of the generate\_salt function is O(n), where n is the length of the username and password strings.
* The time complexity of adding a node to the hash table is O(1) in the best case and O(n) in the worst case, where n is the number of nodes already in the linked list for the given index in the hash table.

* The time complexity of finding a node in the hash table is O(1) in the best case and O(n) in the worst case, where n is the number of nodes in the linked list for the given index in the hash table.

* The time complexity of printing the hash table is O(n), where n is the number of nodes in the hash table.

Overall, the time complexity of the program depends on the number of users registered and the number of nodes in the hash table, and it can be considered O(n) in the worst case.

**SPACE COMPLEXITY:**

The program uses a fixed amount of memory for the hash table, salt, username, password, and confirm\_password variables, which are all stored on the stack.

* The memory required for the hash table is O(m), where m is the size of the hash table.
* The memory required for the salt is O(s), where s is the size of the salt.
* The memory required for the username and password variables is O(1), assuming a fixed maximum length for these strings.

The memory used by the program also includes the dynamically allocated memory for the linked list nodes. Each node requires space for a pointer to the next node, the username string, and the hash value. The memory required for each node is O(n), where n is the length of the username string. The total memory required for all the nodes in the hash table is O(u), where u is the number of registered users.

Overall, the space complexity of the program depends on the size of the hash table and the number of registered users, and it can be considered O(u + m) in the worst case.

**ALGORITHM:**

1. **Hashing algorithm:**

The program defines a hash function that adds up the ASCII values of the characters in a string. The program is then salted and hashed and hence stored in the created hash table.

1. **Salt generation:**

This program also defines a function that generates a random salt string. The salt is mixed with the password which is entered by the user and hence hashed into the hash table.

1. **Case 1 Addition:**

When the user chooses to register, the program prompts the user to enter a username and password. The program generates a random salt string using the username and password as seeds, mixes the password and salt together, hashes the result, and stores the username and hash value in a new node. The program then adds the new node to the hash table.

1. **Case 2 Verification:**

When the user chooses to log in, the program prompts the user to enter a username and password. The program generates a salt string using the username and password as seeds, mixes the password and salt together, hashes the result, and uses the hash value to find the corresponding node in the hash table. If the username and hash value match, the user is logged in.

1. **Case 3 Deletion:**

When the user chooses to delete their account, the program prompts the user to enter a username and password. The program generates a salt string using the username and password as seeds, mixes the password and salt together, hashes the result, and uses the hash value to find the corresponding node in the hash table. If the username and hash value match, the node is deleted from the hash table.

1. **Print and exit:**

When the user chooses to print the hash table, the program displays the hash table along with the number of registered users.

**CHALLENGES FACED:**

1. Finding an appropriate way to secure the hashing of the password (salt, SHA, or other means).
2. Difficulty in making the salt deterministic without making it very easy to brute force and crack.
3. Decision to use username in the hashing and salting process. Just using the password was deemed not enough.
4. Storing the password appropriately so that it can be accessed later. Random salt generation prevented that.

**FUTURE IMPROVEMENTS:**

**Minor:**

1. Beautify the program’s working so that user experience is better and easier.
2. Add a Master Username and Password to reset any user’s password.

**Major:**

1. Setting a limit during logging in to avoid vulnerability to Brute Force attacks
2. Make the hashing more secure by using SHA or bcrypt library.
3. Use of pointers in all functions to make it easier to call variables and their respective data.
4. Add the means of storing spaces in the username and password system.
5. Obfuscate the display and storage of the password further to prevent plain text leaks.