21AIE431

Applied Cryptography

Batch - B _ 17

User Authentication System using Double Hashing

Report

Team Members:

Parthvi Manoj - CB.EN.U4AIE21143

Sakthi Swaroopan - CB.EN.U4AIE21159

Sanjay Chidambaram - CB.EN.U4AIE21160

Taruneshwaran T - CB.EN.U4AIE21170

Double Hashing

Double hashing is a collision resolution technique used in hash tables. It works by using two hash functions to compute two different hash values for a given key. The first hash function is used to compute the initial hash value, and the second hash function is used to compute the step size for the probing sequence.

Double hashing has the ability to have a low collision rate, as it uses two hash functions to compute the hash value and the step size. This means that the probability of a collision occurring is lower than in other collision resolution techniques such as linear probing or quadratic probing.

However, double hashing has a few drawbacks. First, it requires the use of two hash functions, which can increase the computational complexity of the insertion and search operations. Second, it requires a good choice of hash functions to achieve good performance. If the hash functions are not well-designed, the collision rate may still be high.

Advantages of Double hashing

- The advantage of Double hashing is that it is one of the best forms of probing, producing a uniform distribution of records throughout a hash table.
- This technique does not yield any clusters.
- It is one of the effective methods for resolving collisions.

Hash Salting

Hash salting is a technique used to enhance the security of password storage in computer systems, especially in the context of user authentication. It involves adding a random or unique value, known as a "salt," to a password before hashing it. The salt is typically a random string of characters generated for each user or password and stored alongside the hashed password.

- Random Data Addition: A salt is typically a random or unique piece of data. It can be a random string of characters or a value generated for each user or piece of data.
- Combined with Input Data: The salt is combined with the input data (e.g., a password) before hashing. This means that the salted input is hashed, making it different from the original input.
- Stored Securely: The salt is usually stored alongside the hash in a secure database. It is not considered a secret and does not need to be kept confidential.

Advantages of Hash Salting

- Password Security: Hash salting significantly improves password security:
  Without salting, identical passwords would produce the same hash, revealing patterns in the data.
  With salting, even identical passwords will produce different hashes due to the unique salt for each user.
- Protection Against Precomputed Tables: Salting prevents the use of precomputed tables (rainbow tables) for password cracking. Rainbow tables are ineffective because the attacker would need to compute a new table for each salt value.
- Security Against Collision Attacks: Salting mitigates collision attacks, where different inputs produce the same hash. Even if two users have the same password, their hashes will be different because of their unique salts.
- Enhanced Security During Data Breaches: In the event of a data breach, an attacker would need to reverse-engineer the salted hashes for each user separately, which is much more computationally intensive and time-consuming.
- Randomness Enhances Security: Using a random or unique salt for each user makes it harder for attackers to exploit patterns in the data or predict the salt value.
- Maintains Privacy: Salted hashes do not reveal similarities in passwords (e.g., common passwords) among users, enhancing privacy.

Implementation:

User Registration:

- When a user registers, their username and password should be collected.
- Generate a unique salt for the user.
- Combine the salt with the password and hash it using a strong cryptographic hash function (e.g., SHA-256). This is the first level of hashing.
- Hash the result again using the same or a different cryptographic hash function. This is the second level of hashing.
- Store the double-hashed password and the salt in a secure database.

User Authentication:

- When a user attempts to log in, retrieve their stored salt and double-hashed password from the database.
- Collect the password provided during login.

- Apply the same double hashing process (using the stored salt) to the provided password.
- Compare the resulting double-hashed password with the stored double-hashed password.
- If they match, the login is successful.

Encryption of Sensitive Data:

- For sensitive user data (e.g., personal information, financial data), use strong encryption algorithms (e.g., AES) to encrypt the data.
- Store the encrypted data in the database.

Additional Security Measures:

- Implement secure password policies to encourage strong passwords (length, complexity).
- Use a secure random number generator to generate salts.
- Consider implementing two-factor authentication (2FA) for an extra layer of security.
- Protect against brute force attacks by implementing account lockout mechanisms or rate limiting.

Secure Data Storage:

- Store user data and credentials in a secure and well-protected database.
- Use database encryption to protect data at rest.

Secure Communication:

- Encrypt data transmission between the client and server using secure communication protocols (e.g., HTTPS).

Regular Security Audits and Updates:

- Periodically review and update the system to address security vulnerabilities.
- Conduct security audits and penetration testing to identify and fix weaknesses.

User Education:

- Educate users about the importance of strong passwords and security practices.
- Encourage users to update their passwords regularly.

Code:

GUI.py

It imports the required modules, including tkinter for creating the GUI, and imports functions from a module named DH (presumably for double hashing), and initializes user data using dh.init_data().

It defines several functions for handling various actions and operations within the GUI, such as depositing and withdrawing funds, logging in, and signing up. The key functions include:

- update_credit_label(): Updates and displays the user's credit.
- deposit_button_command(): Handles depositing funds.
- withdraw_button_command(): Handles withdrawing funds.
- log_out_button_command(): Logs the user out of the bank system.
- bank_system_window(): Creates a sub-window for the bank system.
- username_string_prune(username): Cleans up the username by stripping whitespace and replacing spaces with underscores.
- sign_up_button_command(): Handles the sign-up process.
- log_in_button_command(): Handles the log-in process.

It creates the main GUI window using tkinter and defines widgets such as labels, entry fields, and buttons for username, password, sign-up, and log-in.

The mainloop() function keeps the GUI application running, waiting for user interactions.

Overall, this code provides a basic framework for a bank login system GUI. Users can sign up, log in, deposit, and withdraw funds. It also uses the DH module to manage user data, and it utilizes double hashing for password security. However, it's important to note that this code snippet relies on external functions and data management provided by the DH module, which is not included in the code snippet.

The DH module is expected to handle data storage, retrieval, and password hashing. To complete the application, you would need to ensure that the DH module and related functionalities are properly implemented.

DH.py

- init_data(): Initializes the JSON file to store user data if it doesn't exist.
- load_data(): Loads user data from the 'users.json' file and returns it as a Python dictionary.
- save_data(data): Saves user data (provided as a dictionary) to the 'users.json' file.

- force_save_data(): Forces a save of the user data that is currently loaded.
- double_hash(password): Double hashes the given password using SHA-256.
- username_check(username): Checks if a username already exists in the user data.
- password_check(username, password): Checks if the provided password is correct for a given username by comparing double-hashed passwords.
- get_credit(username): Retrieves the credit value associated with a username.
- set_credit(username, password, credit): Updates the credit value for a user if the provided username and password are correct.
- log_in(username, password): Attempts to log in a user by checking their username and password.
- sign_up(username, password): Registers a new user by double hashing their password and storing it in the user data.

The code is designed for user authentication and credit management, with passwords being double-hashed for added security. Note that the 'salt' variable is defined but not used in the provided code. It might have been intended for salting the passwords during hashing, but the salting logic is currently commented out.

Conclusion:

In conclusion, implementing a user authentication system that incorporates double hashing, hash salting, and encryption is a crucial step toward enhancing the security and privacy of user data in various applications and services. This combination of security measures offers multiple layers of protection against common threats, such as password cracking, data breaches, and unauthorized access.