CY-211 Project Report

Title : Hashing for Security : A Comprehensive Approach to Data Integrity and Authentication

Section : E-CYS

Members :

2023093 , Ali Ahmed Chaudhry

2023278 , Khawaja Moawiz Ur Rehman

2023403 , Muhammad Bin Waseem

2023416 , Muhammad Haider Iqbal

Overview

This account management system prioritizes **data security** by incorporating a **complex hashing function** that mimics the behavior of the widely used **SHA-256** cryptographic algorithm. The system implements essential user features such as **signup**, **signin**, **password recovery**, and **account management**, all while ensuring that sensitive user data, especially passwords, is handled securely using the custom hashing function.

Detailed Breakdown

Core Components :

- **Account Class**
  - The main class handling all operations, including user authentication, data storage, and complex hashing.
- **Linked List for User Data**
  - Each user is represented as a node (Node structure), and all user data is stored in a linked list (head pointer to the first node).
- **Hashing Mechanism**
  - A custom hash function based on **SHA-256 principles** generates secure passwords using salts and peppers.

Functionalities :

➢ **SignUp :-**

- **Input Handling**
  - Takes a username and password from the user.
  - Verifies if the username already exists using userexists().

```
                    Account SignUp

Username : Muhammad

Password requirements:
- At least one uppercase letter
- At least one lowercase letter
- At least one special symbol (e.g., @, #, $, etc.)
- At least one numeric digit (0-9)
- At least 8 characters long

Enter a password : ABC@123abc
Strong password accepted.
```

- **Password Strength Checker**

  - The system includes a password checker that ensures users create strong passwords, enhancing the effectiveness of the hashing process by ensuring only strong, complex passwords are used.

  - The password must meet the following criteria:

```
Password requirements:
- At least one uppercase letter
- At least one lowercase letter
- At least one special symbol (e.g., @, #, $, etc.)
- At least one numeric digit (0-9)
- At least 8 characters long

Enter a password :
```

- **Password Hashing**:

  - The password, along with the salt and pepper, is hashed using the complex_hash() function.
  - The complex_hash uses custom SHA-256-like operations to produce a secure hashed password.

- **Optional Security Questions**:

  - Users can provide answers to questions (date of birth, favorite place, and favorite food) to enable two-factor authentication for recovery.

```
                    Account SignUp
Enter your date of birth(D/M/Y) : 29/09/2003

Enter your favorite place : UAE

Enter your favorite food : Shawarma
```

- **Data Storage**:
  - User data (username, hashed password, hashed security answers) is stored in a linked list node and linked to the list.

➢ **SignIn**

- **Username Validation**:
  - Validates the existence of a username using searchuser().

```
int searchuser()                                    //searching for user
{
    Node *temp = head;
    while (temp != nullptr)
    {

        if (username == temp->Username)
        {
            K=temp->key;
            pepper = temp->Pepper;
            salt = temp->Salt;
            hashedpassword = temp->password;
            Enabled=temp->Enabled;
            hasheddate=temp->date;
            hashedfood=temp->food;
            hashedplace=temp->place;
            return 1;
        }
        temp = temp->next;
    }
    return 0;
}
```

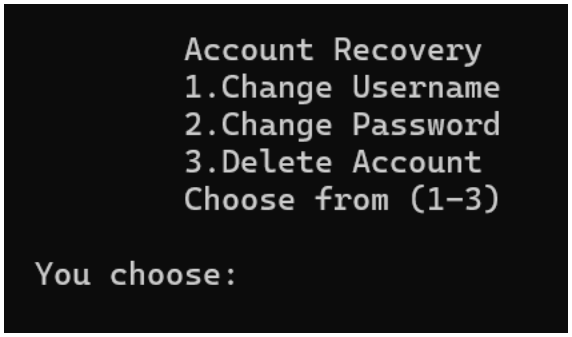
- **Password Validation**:
  - Compares the user-inputted password (hashed with stored salt and pepper) against the stored hash.

```
cout << "\nPassword : ";
getline(cin,input);
if (complex_hash(input,salt,pepper,K) != hashedpassword)
{
    PlaySound(TEXT("C:/Users/PC/Downloads/Denied.wav"), NULL, SND_FILENAME | SND_SYNC);
    cout << "\nWrong Password.";
    count++;
    if (count >= 3)
    {
        cout << "\n\t\t\tForgot Password?(y/n)";
        cin >> check;
        cin.ignore();  // To clear the buffer after reading a char input
    }
```

- **Security Measures**:
  - After three failed attempts, the user is prompted for password recovery.

➢ **Password Recovery**

- **Two-Factor Authentication**:
  - Users with enabled security questions must provide correct answers to reset their password.
- **Password Change**:
  - If authenticated, the user can set a new password, hashed and securely stored.
- **Disabled Two-Factor**:
  - If two-factor authentication is not enabled, the system denies recovery, emphasizing the importance of additional security measures.
- **Account Recovery**



- **Change Username**:
  - Checks if the new username is available and updates it in the linked list.
- **Change Password**:
  - Prompts the user for a new password, hashes it, and updates the linked list node.
- **Delete Account**:
  - Removes the user's node from the linked list and securely wipes sensitive data from memory.

➢ **Complex Hashing Function**

- **Padding**:
  - Implements a padding mechanism akin to SHA-256 to ensure input length is a multiple of 512 bits.
- **Custom SHA-256 Operations**:
  - Uses predefined constants (k) and bitwise operations (e.g., rotations) to process message blocks.
- **Salt and Pepper**:
  - Adds these random values one to the input & one to hash result to strengthen the hash against precomputed attacks (rainbow tables).
  - Length of Salt and Pepper = **32 characters**
  - Total possible characters of Salt and Pepper **= 95 characters**

KeySpace of Salt = $\mathbf{95^{32}}$

KeySpace of Pepper = $\mathbf{95^{32}}$

<u>Salt and pepper are generated using the following algorithm :</u>

```
// Function to generate a cryptographically secure random character
char generateSecureRandomCharacter()
{

    const string allowedChars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%^&*()_+-=[]{}|;:,.<>?`/\"' ";


    // Use random_device to get a cryptographically secure random number

    random_device rd;
    mt19937 gen(rd());              // Mersenne Twister generator
    uniform_int_distribution<> dis(0, allowedChars.size() - 1);

    // Randomly pick a character from the allowed list
    return allowedChars[dis(gen)];
}
```

- **Final Hash**:
  - Outputs a unique and secure hash, converted into a custom base using CHARSET.
  - KeySpace of Final Hash =(*KeySpace of Salt*) ⊗ (*KeySpace of HashResult*)

$$= 95^{32} \otimes 95^{32} = 95^{64}$$

➢ **Memory Security**

- **SecureZeroMemory**:
  - Ensures sensitive data like passwords are erased from memory after use to prevent unauthorized access.

➢ **Linked List Management**

  - Functions like insertNode, delnode, and printlist handle linked list operations for account management.
  - Each node contains a user's credentials and security-related information.

Strengths

- **Security-Oriented Design**:
  - Incorporates salts, peppers, and a custom hashing function.
  - Implements secure memory handling practices.
- **User Management**:
  - Provides robust functionality for creating, updating, and deleting accounts.
- **Two-Factor Authentication**:
  - Adds an optional layer of security.

Weaknesses and Recommendations

- **File I/O for Persistence**:
  - The system does not persist user data to disk. Consider integrating file-based or database storage to retain data across sessions.
- **Platform Dependence**:
  - The use of Windows-specific headers (windows.h, PlaySound) limits portability. Consider abstracting these dependencies.
- **Code Structure**:
  - The code can be modularized further, separating concerns (e.g., hashing, I/O, user management) into distinct classes or files.

Main code explained:

```cpp
// Process the padded message in 512-bit blocks
for (size_t i = 0; i < padded.size(); i += 64)
{
    vector<unsigned int> w(64);
    for (int t = 0; t < 16; ++t)
    {
        w[t] = (padded[i + t * 4] << 24) | (padded[i + t * 4 + 1] << 16) |
               (padded[i + t * 4 + 2] << 8) | padded[i + t * 4 + 3];
    }

    for (int t = 16; t < 64; ++t)
    {
        unsigned int s0 = rightRotate(w[t - 15], 7) ^ rightRotate(w[t - 15], 18) ^ (w[t - 15] >> 3);
        unsigned int s1 = rightRotate(w[t - 2], 17) ^ rightRotate(w[t - 2], 19) ^ (w[t - 2] >> 10);
        w[t] = w[t - 16] + s0 + w[t - 7] + s1;
    }

    unsigned int a = h[0], b = h[1], c = h[2], d = h[3], e = h[4], f = h[5], g = h[6], h_val = h[7];

    for (int t = 0; t < 64; ++t)
    {
        unsigned int s1 = rightRotate(e, 6) ^ rightRotate(e, 11) ^ rightRotate(e, 25);
        unsigned int ch = (e & f) ^ ((~e) & g);
        unsigned int temp1 = h_val + s1 + ch + k[t] + w[t];
        unsigned int s0 = rightRotate(a, 2) ^ rightRotate(a, 13) ^ rightRotate(a, 22);
        unsigned int maj = (a & b) ^ (a & c) ^ (b & c);
        unsigned int temp2 = s0 + maj;
```

1. Padding the Message:

The input message is padded to ensure its length is a multiple of 512 bits (64 bytes).

Padding Process:

First, a byte 0x80 (binary 10000000) is appended to the message.

Then, additional zero bytes (0x00) are added to ensure that the length of the message modulo 512 equals 448. This ensures the final block has enough space to append the message length.

Finally, the original message length (in bits) is appended as a 64-bit integer in big-endian format.

2. Message Block Processing:

After padding, the message is divided into 512-bit blocks, which are processed one at a time.

Message Expansion:

The first 16 words (32 bits each) are extracted from the message block and used directly.

The remaining 48 words are generated using a recursive process, where each new word is a combination of the previous words with bitwise rotations and shifts.

SHA-256 Core Processing:

Each block is processed using 64 rounds (one for each word).

Two main operations are involved in each round:

Ch (Choice): Determines which value to select between the current hash and the words.

Maj (Majority): Takes the majority of three input values, contributing to the final result.

The hash values h[0] to h[7] are updated after each round based on bitwise operations (e.g., right rotations, shifts) and a set of constants k[t] that provide fixed values to ensure different cryptographic behavior in each round.

3. Final Hash Calculation:

After all message blocks are processed, the hash values (h[0] to h[7]) are combined to form the final hash.

The final 256-bit hash is then divided into 8 chunks (32 bits each), and each chunk is converted into a character using a custom character set. This provides a human-readable and flexible representation of the final hash.

4. Salt Insertion:

A salt is added to the hash result to prevent attacks such as rainbow table or dictionary attacks. The salt is inserted at a particular index (num), which is order dependent .

This step makes the final Hash more random to the same input with different salts the final output will always be different.

5. Character Set for Final Hash:

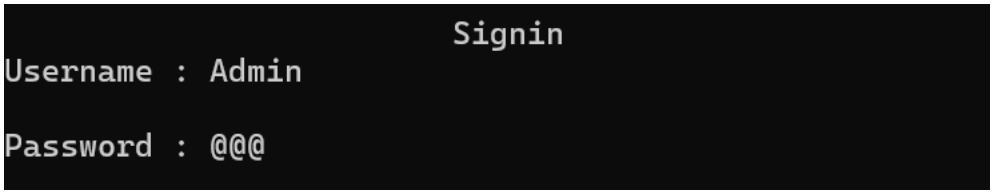
Our Character Set (CHARSET) consists of printable ASCII characters – letters, numbers, and signs. This set is utilised to transform the final hash value to a string which might enhance the output for useful storage or practical display.

- Total possible values of Char Set = **95 characters**

6. Output:

After all the transformations the modified hash is concatenated with the salt at the position defined and is returned as string. The output is a string value calculated based on the input message, salt, and the pepper .

Role-Based Authentication for Admin



We have a basic role-based authentication system put in practice. When the user sends his/her username and password, the program checks whether the username entered is 'Admin'. If so the system gives a go ahead with the password. If the entered password is '@@@', the admin receives the ability to work with additional administrative options, for example, viewing a list of users.

```
                List Of Accounts :

1.      Username :  Muhammad

        Salt : IwIM&UCbR $Zlayh%!oaQ|^cB6loI4Xi

        Pepper : 5104*>8"]-G5!"$/`god^74^/p.3BrLg

        Password : hIwIM&UCbR $Zlayh%!oaQ|^cB6loI4XijY>$RWX0HcdZ9PH9A^K9tALU-tE6Kpc
```

This is a good example of how one works with different users in this system – while all of them would be limited, the admins would have more access than the others. In a real-world application, hashed passwords and salts can be applied to safe store and authenticate users and passwords more securely.

Key Properties & Security Features of the Hashing Algorithm:

1. **Avalanche Effect**:
    - A small change in the input results in a drastically different hash output.
    - This ensures that similar inputs don't produce similar hashes, making it harder for attackers to predict or reverse-engineer the hash.
2. **Deterministic**:
    - The algorithm always generates the same hash for the same input.
    - This property ensures that hashes are consistent, making it easy to verify the integrity of data.
3. **Fixed-Length Output**:
    - The resulting hash is always the same size, regardless of the input size.
    - This makes the hash manageable and efficient for storage and comparison.
4. **Salted Hashing**:
    - A unique **salt** is added to the input before hashing to prevent attackers from using precomputed hash tables (like rainbow tables).
    - By inserting the salt at a specific position within the hash, it ensures uniqueness for every hash, even for identical inputs.

5. **Pepper**:
    - An additional secret **pepper** is used to modify the input before hashing.
    - The pepper increases complexity and randomness, further protecting against brute-force attacks and making reverse-engineering more difficult.
6. **Collision Resistance**:
    - It is computationally infeasible to find two different inputs that produce the same hash value.
    - This ensures that the algorithm maintains data integrity and prevents accidental or intentional collisions.
7. **Preimage Resistance**:
    - For any given hash value it is computationally very difficult to decode and work out the original inputs.
    - This makes it secure from unauthorized access hence protecting the original data.
8. **Keyspace**:
    - With the salt, the keyspace expands to 95^{64}, making brute-force attacks significantly more difficult.
    - A larger keyspace enhances the security of the hash by making it harder for attackers to exhaustively search for the original input.
9. **Custom Charset**:
    - The hash output is encoded using a custom charset (CHARSET), making the result human-readable and flexible.
    - While this doesn't directly affect cryptographic security, it makes the hash more usable in applications requiring printable characters.

Conclusion

Therefore, this account management system adopts a highly secure approach for the storage of data by incorporating a customized hashing algorithm which is based on SHA-256 but which in addition boasts the use of salt, pepper and charset. These security features like **avalanche effect**, **collision resistant** and **preimage resistant** help to protect the password of users.

The system includes basic account requirements as signing up into an account, signing into an account, password reset, and admin privileges. Also never ceases to implement a secure environment for the users . For instance, two-factor authentication makes the tool secure enough to protect all user data