Password Storage and Validation Using Hashing

RYAN NASH

CSC-2710-01

APRIL 23RD, 2025

Project Overview

Main Goals:

- Build a secure password registration and login system.
- Use cryptographic hashing (SHA-256) and salting to protect credentials.
- Implement brute-force protection through lockout mechanism.
- Store user data using hash tables for efficiency

Problem Space

01

Password breaches and data leaks remain a major threat in modern systems

02

Storing passwords in plaintext or using weak hashing is still common

03

My project aimed to fix this by securely storing and verifying passwords using hashing and salting.

How Hashing Was Used

- SHA-256 used to convert (salt + password) into a secure hash.
  - Salt ensures each hash is unique, even for identical passwords.
- On login, password + salt are re-hashed and compared to the stored hash.
- All operations run in constant time using unordered_map

Key Features

User Registration:

- Generates a unique salt
- Hashed the password with SHA-256
- Stores (salt, hash) pair securely

Login Verification:

- Uses the stored salt to re-hash and validate input
- Limits attempts to prevent brute-force attacks

Code Walkthrough

- generate_salt(): Random 16-character alphanumeric string
- sha256(): Computes a SHA-256 hash using OpenSSL
- register_user(): Stores username, salt, and hashed password
- login_user(): Validates login with 3-try lockout policy
- unordered_map(): Efficient key-value store for fast retrieval

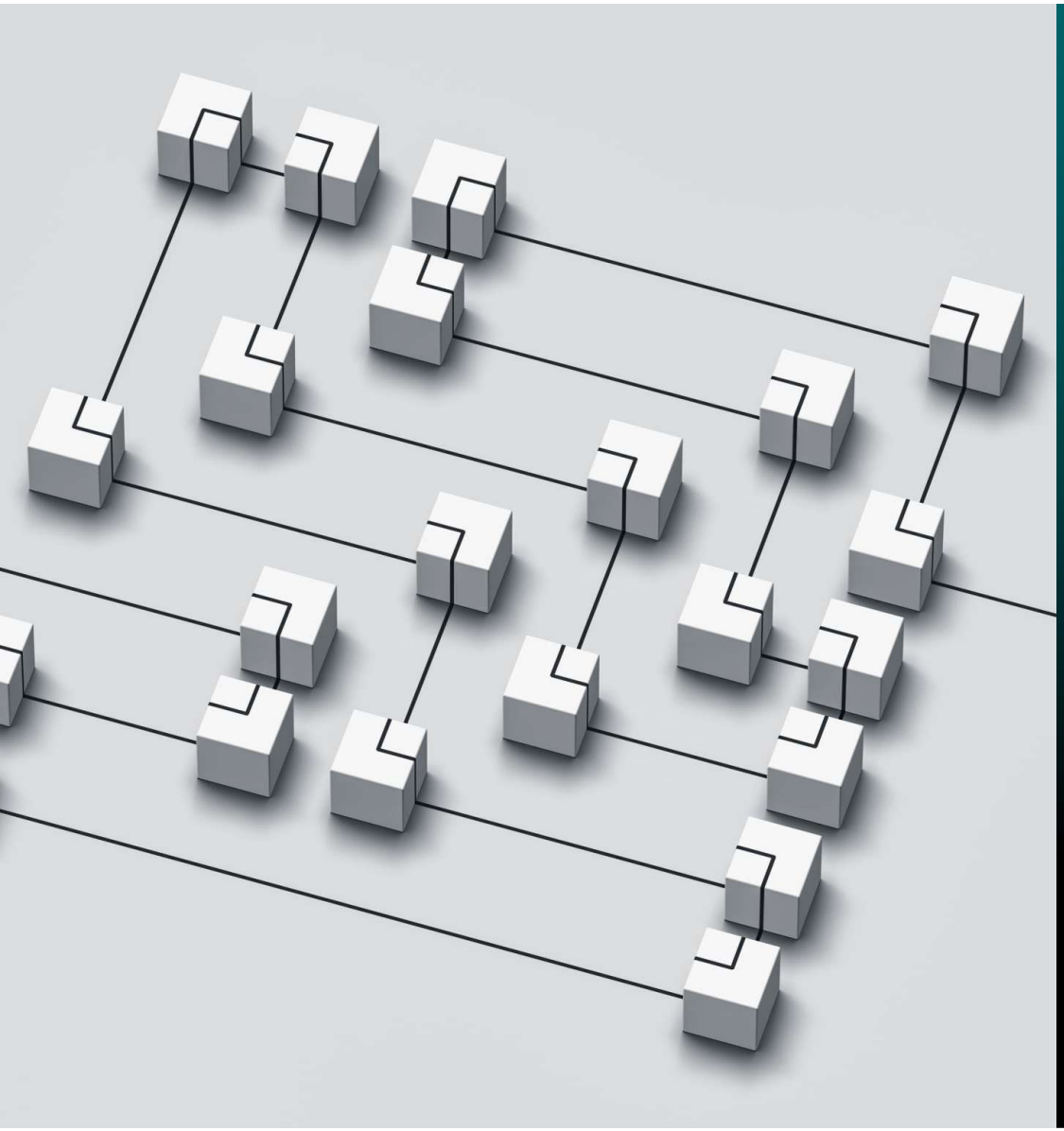
C++ Code Snippets

```cpp
// Register a new user
void register_user() {
    string username, password;
    cout << "Enter username: ";
    cin >> username;

    if (user_db.find(username) != user_db.end()) {
        cout << "Username already exists.\n";
        return;
    }

    cout << "Enter password: ";
    cin >> password;

    string salt = generate_salt();
    string hashed_password = sha256(salt + password);
    user_db[username] = {salt, hashed_password};

    cout << "User registered successfully!\n";
}
```

```cpp
// Compute SHA-256 hash from input string
string sha256(const string& input) {
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256((unsigned char*)input.c_str(), input.size(), hash);

    stringstream ss;
    for (int i = 0; i < SHA256_DIGEST_LENGTH; ++i) {
        ss << hex << setw(2) << setfill('0') << (int)hash[i];
    }
    return ss.str();
}
```

```cpp
// Generate a random salt of given length
string generate_salt(int length = 16) {
    string chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    string salt;
    for (int i = 0; i < length; ++i) {
        salt += chars[rand() % chars.size()];
    }
    return salt;
}
```

Demo Output

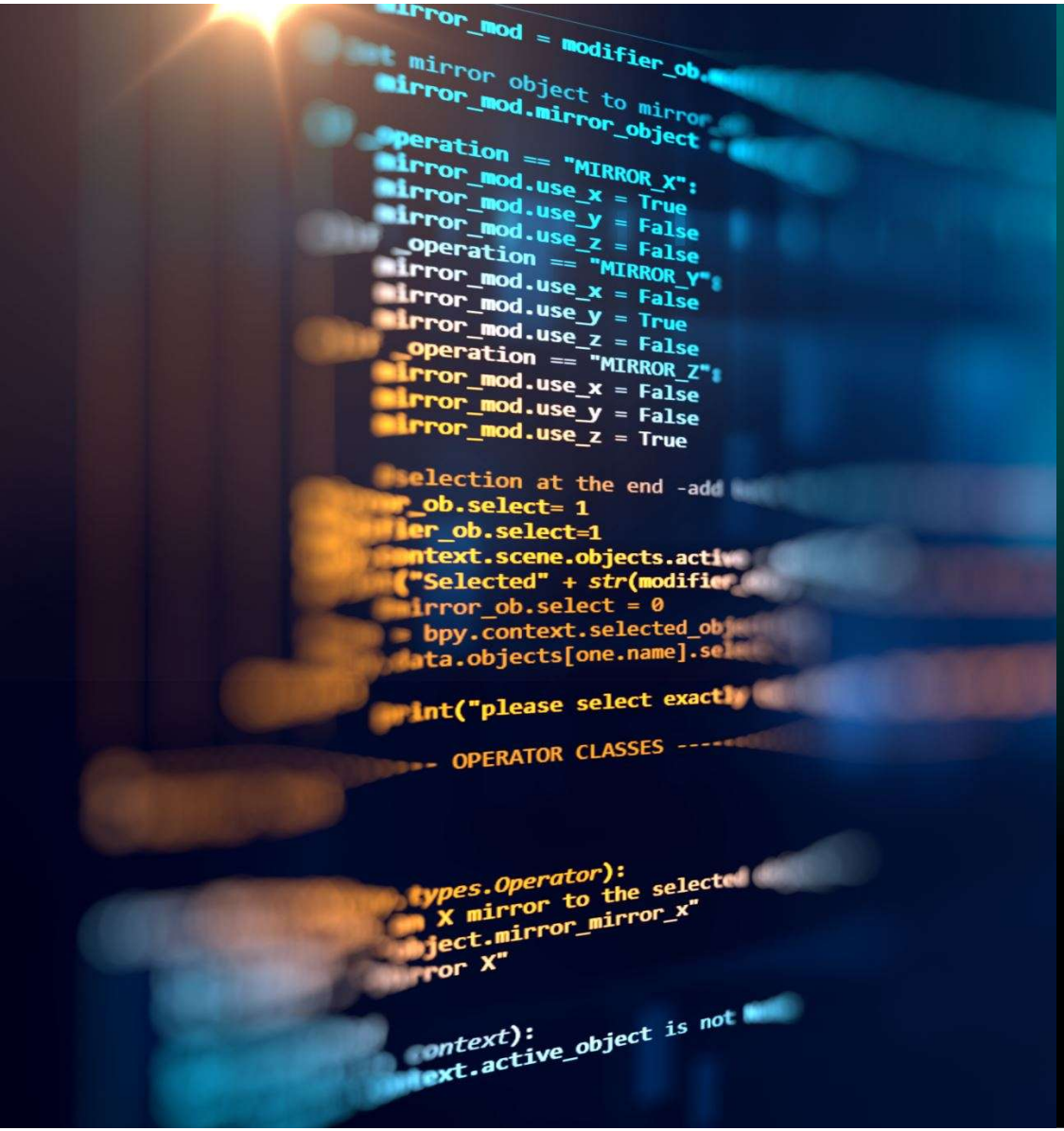


Challenges and Solutions

- OpenSSL Integration: Required low-level data handling (byte arrays, hex formatting)
- Salt Design: Ensured randomness and uniqueness to prevent hash collisions.
- Hash Comparison: Carefully handled string matching and lockout conditions.
- Testing: Ran multiple rounds of registration and login scenarios to ensure stability.

Reflection

- Practical use of hashing and salting in cybersecurity
- How to structure modular C++ code for authentication systems
- Challenges of real-world cryptography in low-level languages
- Importance of brute-force protection and user experience in security design



Final Thoughts & Future Improvements

- Final Result:
  - Fully functional password system using hashing best practices.
  - Strong foundation for real-world security features.
- Future Additions:
  - Password strength validation
  - Account recovery or multi-factor authentication
  - Web-based front end


Thank You!