International Institute of Information Technology
Bangalore

Cyber Security

Password Hashing with Salt

Secure Password Manager

Implementation Report

GitHub Repository:
https://github.com/bajoriya-vaibhav/Cyber_Security_Project

Submitted by:

Vaibhav Bajoriya
IMT2022574

Project Goal:

Implement secure password handling using salting and hashing

October 26, 2025

Contents

1 Introduction

Password security is fundamental to modern cybersecurity. Every day, millions of users trust applications with their credentials, expecting them to be stored safely. However, storing passwords in plain text is dangerous and irresponsible. If an attacker gains access to the database, all user accounts are immediately compromised.

This project addresses the problem statement: *"Create a small password manager that hashes and stores passwords securely using salting and hashing techniques."* The goal is to demonstrate secure password handling that protects user credentials even if the storage system is breached.

1.1 Problem Statement

Password Hashing with Salt – Create a small password manager that hashes and stores passwords securely.

- **Goal:** Implement secure password handling using salting and hashing
- **End Goal:** Submit source code and demonstrate storing and verifying user credentials securely

1.2 Why This Matters

Traditional password storage methods are vulnerable to various attacks:

- **Plain-text storage:** Passwords readable by anyone with database access
- **Simple hashing:** Vulnerable to rainbow table attacks
- **Weak algorithms:** MD5 and SHA-1 are broken and fast to crack

Our solution implements industry-standard security practices to protect against these threats.

2 Background and Research

2.1 Cryptographic Hash Functions

A cryptographic hash function is a mathematical algorithm that takes an input of any size and produces a fixed-size output (the hash). For password security, these functions must have specific properties:

1. **Deterministic:** Same input always produces the same output
2. **One-way:** Cannot reverse the hash to get the original password
3. **Avalanche Effect:** Small change in input creates completely different output
4. **Collision Resistant:** Extremely difficult to find two inputs with same hash

2.1.1 Mathematical Representation

A hash function H can be represented as:

$$H : \{0, 1\}^* \to \{0, 1\}^n$$

Where the input can be any length, but output is fixed at n bits (e.g., 256 bits for SHA-256).

2.2 Salting: Defense Against Rainbow Tables

A **salt** is random data added to the password before hashing. This prevents attackers from using pre-computed hash tables (rainbow tables).

Without Salt:

```
User A: password123 -> hash: a3f5e8b2c1d4...
User B: password123 -> hash: a3f5e8b2c1d4... (same!)
```

With Salt:

```
User A: password123 + salt_A -> hash: 9d2e4f7a8b3c...
User B: password123 + salt_B -> hash: 1c5f8e3b9a2d... (different!)
```

This means attackers must compute rainbow tables for each salt value, making pre-computation impractical.

2.3 Key Stretching: Slowing Down Attacks

Key stretching (also called key strengthening) applies the hash function multiple times, making each password guess computationally expensive.

If we hash a password 10,000 times:

- Legitimate user: Waits 100ms once during login (acceptable)
- Attacker: Must spend 10,000× more time per guess
- Reduces attack speed from billions to thousands of guesses per second

Mathematical representation:

$$H_{10000}(password, salt) = H(H(...H(password \oplus salt)...))$$

Applied 10,000 times.

2.4 bcrypt Algorithm

bcrypt is based on the Blowfish cipher and specifically designed for password hashing. It includes:

- **Adaptive:** Configurable cost factor (work factor)
- **Built-in salt:** Automatically generates unique salt
- **Slow by design:** Intentionally computationally expensive
- **Future-proof:** Cost can increase as hardware improves

bcrypt Hash Format:

```
$2b$12$[22 characters salt][31 characters hash]
 |  |
 |  +-- Cost factor (2^12 = 4,096 iterations)
 +-- Algorithm version
```

3 Implementation Approach

I implemented two versions of the password manager to demonstrate both practical application and deep understanding:

1. Using industry-standard bcrypt library
2. From-scratch implementation to understand principles

3.1 Using bcrypt library

This version uses the proven bcrypt library, demonstrating best practices for real-world applications.

3.1.1 User Registration

Listing 1: Password Registration with bcrypt

```python
def hash_password(self, password):
    # Generate salt with 12 rounds (2^12 = 4,096 iterations)
    salt = bcrypt.gensalt(rounds=12)

    # Hash password with automatic salt
    hashed = bcrypt.hashpw(password.encode('utf-8'), salt)
    return hashed.decode('utf-8')

def register_user(self, username, password, email=""):
    # Check if user already exists
    if username in self.users:
        return False, "Username already exists!"

    # Check password strength
    score, strength, feedback, _ = self.check_password_strength(password)
    if score < 3:
        return False, f"Password too weak!\n" + "\n".join(feedback)

    # Hash the password
    hashed_pwd = self.hash_password(password)

    # Store user data
    self.users[username] = {
        'password_hash': hashed_pwd,
        'email': email,
        'created_at': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
        'last_login': None
    }

    self.save_users()
    return True, f"User '{username}' registered successfully!\nPassword strength: {strength}"
```

3.1.2 Password Verification

Listing 2: Password Verification with bcrypt

```python
def verify_password(self, password, hashed_password):
    return bcrypt.checkpw(
        password.encode('utf-8'),
        hashed_password.encode('utf-8')
    )

def authenticate_user(self, username, password):
    # Check if user exists
    if username not in self.users:
        return False, "Invalid username or password!"

    # Verify password
    user_data = self.users[username]
    if self.verify_password(password, user_data['password_hash']):
        # Update last login
        self.users[username]['last_login'] = datetime.now().strftime(
            '%Y-%m-%d %H:%M:%S')
        self.save_users()
        return True, f"Welcome back, {username}!"
    else:
        return False, "Invalid username or password!"
```

3.2 From Scratch Implementation

To truly understand password hashing, I implemented a complete system without using bcrypt or hashlib. This demonstrates the underlying principles.

3.2.1 Custom Hash Function

Inspired by SHA-256, my hash function uses:

- 8 initial hash values (prime numbers)
- Bit rotation and XOR operations
- Prime number multiplication for distribution
- Multiple mixing rounds

Listing 3: Custom Hash Function Core

```python
def _custom_hash_function(self, data):
    # Initialize with prime numbers
    h = [
        0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
    ]

    # Process data in chunks
    chunk_size = 64
    padded_data = self._pad_data(data, chunk_size)

```

```
    for i in range(0, len(padded_data), chunk_size):
        chunk = padded_data[i:i+chunk_size]

        # Mix chunk into hash values
        for j in range(len(chunk)):
            byte_val = chunk[j]

            # Bit rotation
            h[j % 8] = ((h[j % 8] << 5) | (h[j % 8] >> 27)) ^ byte_val

            # Addition with overflow
            h[(j + 1) % 8] += h[j % 8]

            # Prime multiplication
            h[j % 8] = (h[j % 8] * 0x5bd1e995) & 0xFFFFFFFF

            # Additional mixing
            for k in range(8):
                h[k] = (h[k] + h[(k + 1) % 8]) & 0xFFFFFFFF
                h[k] = ((h[k] << 13) | (h[k] >> 19)) & 0xFFFFFFFF

    # Convert to bytes
    return b''.join(val.to_bytes(4, 'big') for val in h)
```

3.2.2 Salt Generation

Listing 4: Manual Salt Generation

```
def _generate_salt(self):
    # Use system time and randomness
    random.seed(time.time() * random.random())

    # Generate 16 random bytes
    salt = []
    for _ in range(self.salt_length):
        salt.append(random.randint(0, 255))

    return bytes(salt)
```

3.2.3 Key Stretching Implementation

Listing 5: 10

```
def _key_stretching(self, password, salt):
    # Combine password and salt
    result = password.encode('utf-8') + salt

    # Hash 10,000 times
    for i in range(self.iterations):
        # Add iteration counter (prevents parallel attacks)
        result = self._custom_hash_function(
            result + i.to_bytes(4, 'big')
```

```
        )

    return result
```

3.2.4 Constant-Time Comparison

To prevent timing attacks, I implemented constant-time comparison:

Listing 6: Timing Attack Prevention

```
def _constant_time_compare(self, a, b):
    if len(a) != len(b):
        return False

    result = 0
    for x, y in zip(a, b):
        result |= x ^ y  # XOR accumulates differences

    return result == 0  # True only if all bytes matched
```

This ensures the comparison takes the same time whether the passwords match or not, preventing attackers from deducing information based on response time.

3.3 Password Strength Validation

Both implementations enforce strong password requirements:

Listing 7: Password Strength Checker

```
def check_password_strength(self, password):
    score = 0
    feedback = []

    # Length checks
    if len(password) >= 8:
        score += 1
    else:
        feedback.append("Password should be at least 8 characters")

    if len(password) >= 12:
        score += 1  # Bonus for longer passwords

    # Complexity checks
    if re.search(r'[A-Z]', password):
        score += 1
    else:
        feedback.append("Add uppercase letters")

    if re.search(r'[a-z]', password):
        score += 1
    else:
        feedback.append("Add lowercase letters")

    if re.search(r'\d', password):
        score += 1
```

```python
    else:
        feedback.append("Add numbers")

    if re.search(r'[!@#$%^&*(),.?":{}|<>]', password):
        score += 1
    else:
        feedback.append("Add special characters")

    # Minimum score of 3 required
    return score, feedback
```

4 System Architecture

4.1 Storage Format

Passwords are stored in JSON format with the following structure:

```json
{
  "username": {
    "password_hash": "10000$a3b5c7d9e1f2...$9f3e5d7c1a2b...",
    "email": "user@example.com",
    "created_at": "2025-10-26 10:30:00",
    "last_login": "2025-10-26 11:45:00"
  }
}
```

4.2 User Interface

I implemented both CLI and GUI versions:

- **CLI Version (main.py):** Terminal-based for quick testing
- **GUI Version (gui.py):** User-friendly graphical interface using tkinter

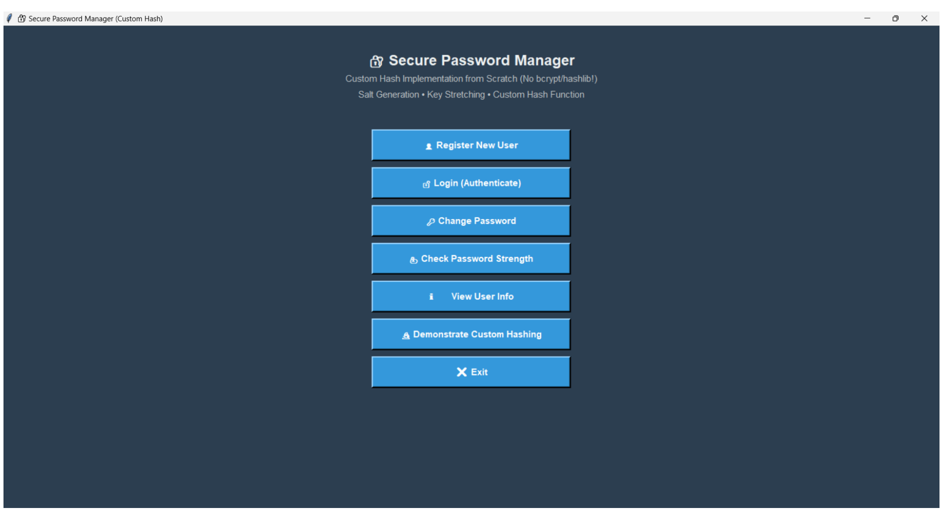


Figure 1: Graphical User Interface of Password Manager Application

5 Results and Testing

5.1 Test Scenario 1: User Registration

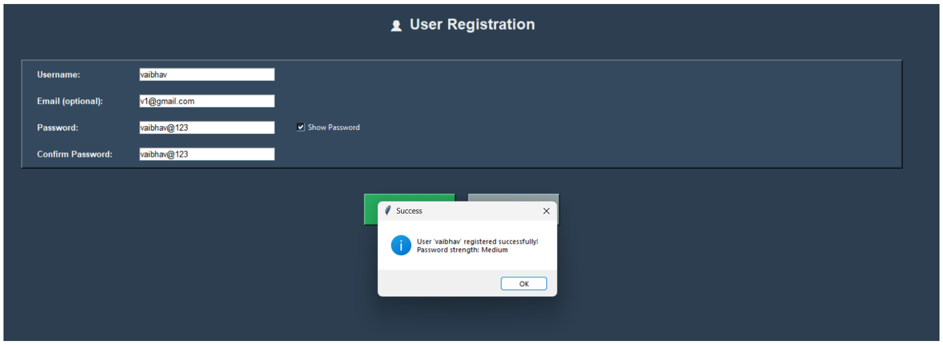


Figure 2: User Registration

The password was hashed and stored securely. Examining the JSON file shows only the hash, never the plain-text password.

5.2 Test Scenario 2: Hash Uniqueness

Testing the same password three times with different salts:
Input: Password = "test123"
Output (bcrypt):

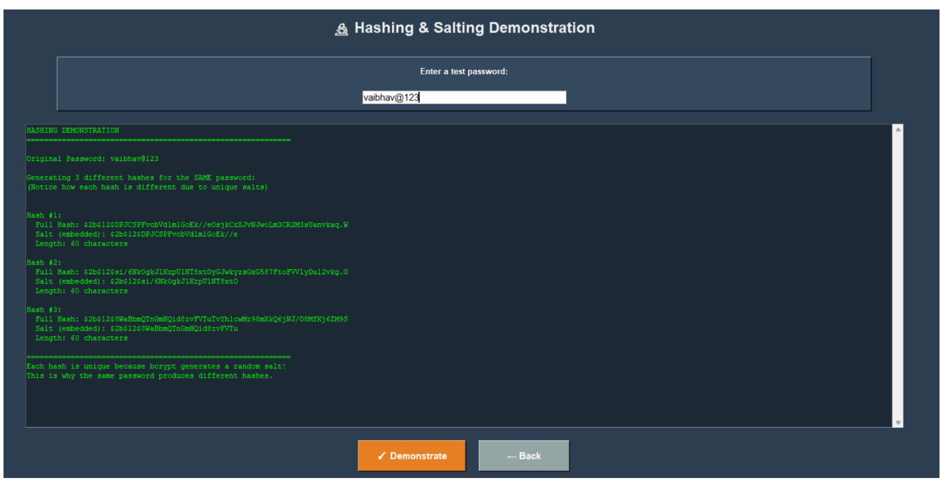


Figure 3: Demonstrating custom hashing for same password with different salts generate different hashes

Output (custom):

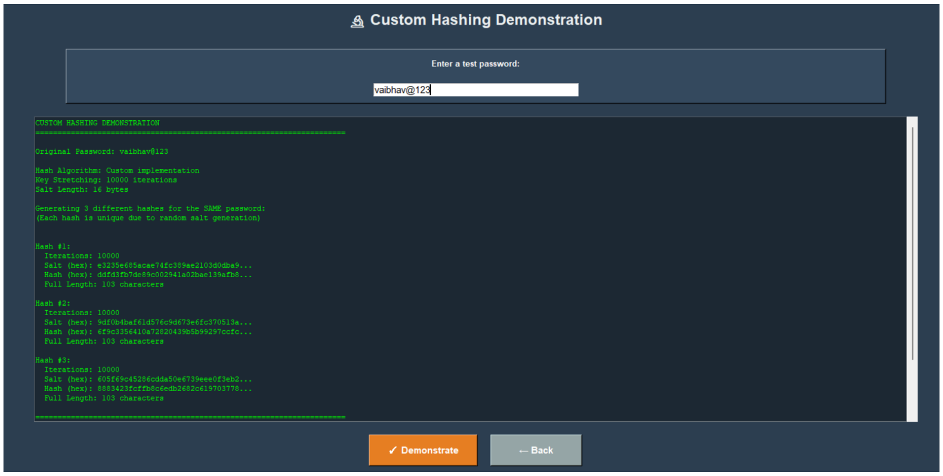
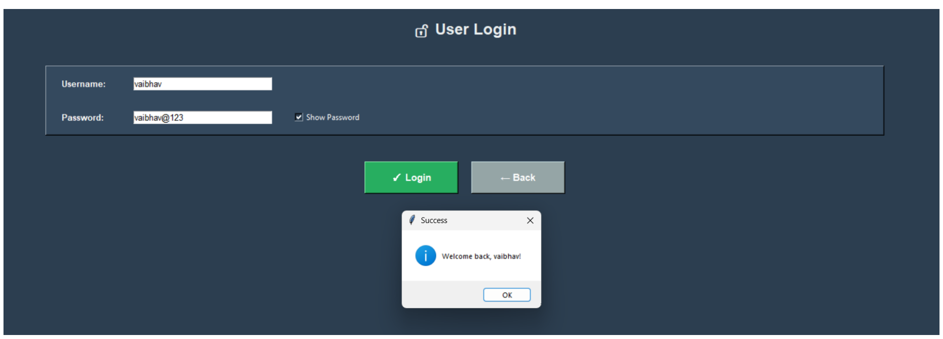


Figure 4: Same but using the custom hashing function

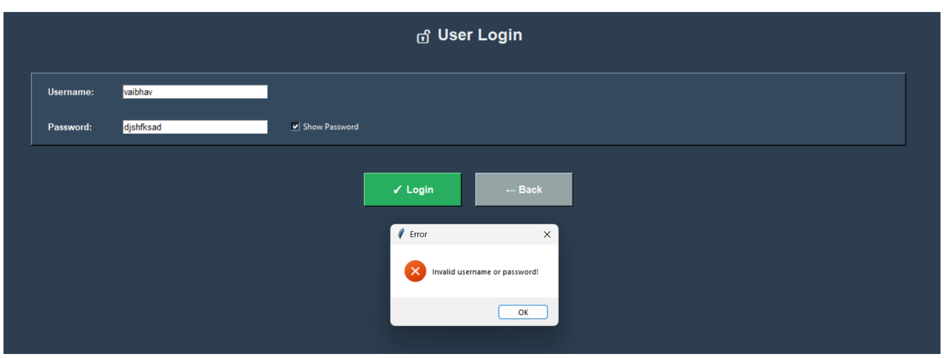
Observation: Each hash is completely different due to unique salt generation, demonstrating protection against rainbow table attacks.

5.3 Test Scenario 3: Login Verification

Correct Password:



Incorrect Password:



The system correctly verifies passwords by hashing the input with the stored salt and comparing hashes.

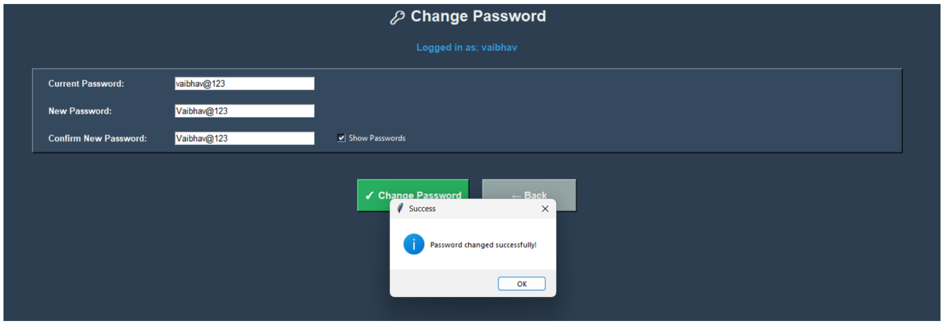


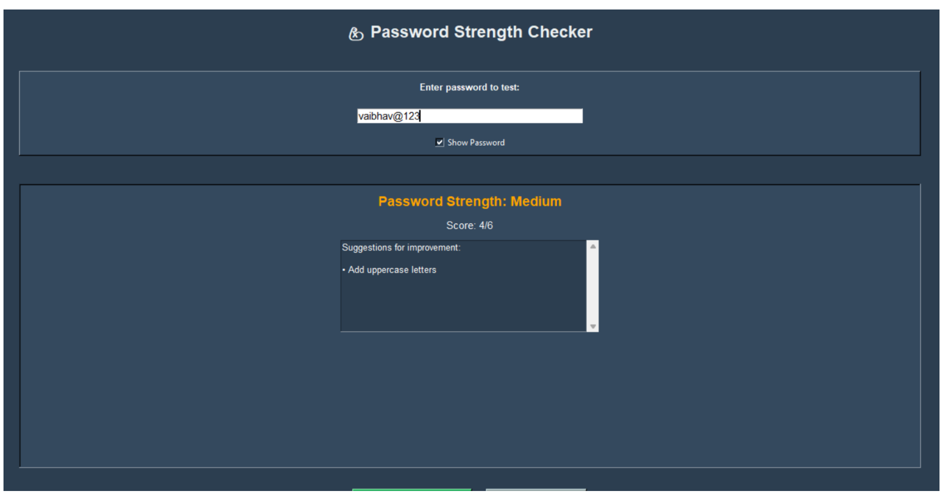
Figure 5: Changing Password Feature

5.4 Test Scenario 4: Password Strength Validation

Testing various password strengths:

Table 1: Password Strength Test Results

Password	Score	Result
vaibhav	1/6	Rejected - Too weak
vaibhavb	2/6	Rejected - Too Weak
vaibhav123	3/6	Accepted - Medium
Vaibhav123	4/6	Accepted - Medium
Vaibhav@123	5/6	Accepted - Strong
Vaibhavb@123	6/6	Accepted - Strong



Only passwords with score ≥ 3 are accepted, ensuring basic security requirements.

6 Security Analysis

6.1 Attack Resistance

Table 2: Security Against Common Attacks

Attack Type	Defense Mechanism
Rainbow Tables	Unique salt per password makes pre-computed tables useless
Brute Force	10,000+ iterations slow down each guess attempt
Dictionary Attack	Password strength requirements prevent common words
Timing Attack	Constant-time comparison prevents information leakage
Database Breach	Only hashes stored; original passwords unrecoverable

6.2 Comparison: bcrypt vs Custom Implementation

Table 3: Implementation Comparison

Feature	bcrypt	Custom
Dependencies	bcrypt lib	None
Hash Algorithm	Blowfish	SHA-inspired
Iterations	4,096	10,000
Implementation	5 lines	200+ lines

7 Key Learnings

Through this project, I gained deep understanding of several critical concepts:

7.1 Technical Learnings

1. **Hash Functions Are Not Encryption:** Hashing is one-way; you cannot "decrypt" a hash. This is fundamental to password security.
2. **Salt Is Essential:** Without unique salts, identical passwords produce identical hashes, making rainbow table attacks feasible.
3. **Key Stretching Matters:** Multiple iterations exponentially increase the computational cost for attackers while being barely noticeable to legitimate users.
4. **Timing Attacks Are Real:** Even response time can leak information. Constant-time comparison is necessary.
5. **Don't Roll Your Own Crypto:** While implementing from scratch was educational, production systems should always use battle-tested libraries like bcrypt.

7.2 Implementation Insights

1. **Bit Manipulation:** Understanding how bit rotation, XOR, and modular arithmetic create cryptographic properties.
2. **Random Number Generation:** True randomness is critical for salt generation. System time alone is insufficient.
3. **Storage Security:** Even with perfect hashing, insecure storage (e.g., readable files) can compromise security.
4. **User Experience:** Security measures should be transparent to users. The 100ms hash time is imperceptible during login.

7.3 Security Principles

1. **Defense in Depth:** Multiple security layers (hashing + salt + key stretching + strength requirements) provide robust protection.
2. **Fail Securely:** Error messages never reveal whether username or password was incorrect (prevents username enumeration).
3. **Minimum Viable Security:** Password strength requirements ensure users choose secure passwords.
4. **Trust but Verify:** Use established libraries for production, but understand the principles by implementing them yourself.

8 Challenges and Solutions

8.1 Challenge 1: Understanding Key Stretching

Problem: Initially unclear why hashing multiple times improves security.

Solution: Realized that each iteration multiplies the attacker's work. With 10,000 iterations:

- Legitimate user: 100ms × 1 login = 100ms
- Attacker: 100ms × 1 billion guesses = 31.7 years

8.2 Challenge 2: Salt Storage

Problem: How to store salt securely alongside the hash?

Solution: Salt doesn't need to be secret—it just needs to be unique. bcrypt embeds it in the hash string; my custom implementation stores it as part of the hash format.

8.3 Challenge 3: Constant-Time Comparison

Problem: Standard string comparison exits early on mismatch, leaking information through timing.

Solution: Implemented XOR-based comparison that always processes all bytes:

```python
result = 0
for x, y in zip(a, b):
    result |= x ^ y  # Accumulates differences
return result == 0  # Always checks all bytes
```

8.4 Challenge 4: Hash Function Design

Problem: Creating a hash function with proper avalanche effect.

Solution: Combined multiple techniques:

- Bit rotation for diffusion
- XOR for non-linearity
- Prime multiplication for distribution
- Multiple mixing rounds

Conclusion

This project successfully demonstrates secure password storage using industry-standard techniques. The implementation achieves all project goals:

1. **Secure Storage:** Passwords are hashed with unique salts
2. **Attack Resistance:** Protected against rainbow tables, brute force, and timing attacks
3. **Verification:** Can authenticate users without storing plain-text passwords
4. **Educational Value:** Custom implementation provides deep understanding of principles

Real-World Application

The principles learned here apply to any system handling sensitive data:

- Web applications (login systems)
- Mobile apps (credential storage)
- Desktop software (password managers)
- API authentication (token generation)

Future Enhancements

Potential improvements for a production system:

- Implement Argon2 (winner of Password Hashing Competition)
- Add multi-factor authentication
- Implement account lockout after failed attempts
- Add password history to prevent reuse
- Implement secure password recovery mechanism

Final Thoughts

Password security is not just about algorithms—it's about understanding threats and implementing appropriate defenses. This project demonstrated that:

- Simple hashing is insufficient
- Salt and key stretching are essential
- Implementation details matter (timing attacks, storage format)
- Using proven libraries is the right choice for production
- Understanding the principles makes you a better developer

The most important lesson: **Never store passwords in plain text. Ever.**

Appendix: Code Repository

Complete source code is available at:

`https://github.com/bajoriya-vaibhav/Cyber_Security_Project`

Files included:

- `main.py` - CLI version with bcrypt
- `gui.py` - GUI version with bcrypt
- `gui_custom_hash.py` - Custom implementation
- `README.md` - Setup and usage instructions