Password Manager Implementation Report

1. Introduction

This report documents the design and implementation of a secure password manager as required by the CSD356 Foundation of Information Security programming assignment. The password manager securely stores and retrieves passwords using the Advanced Encryption Standard (AES) while implementing best security practices like master password authentication, secure key generation, and password strength verification.

2. System Architecture

The password manager is built with a modular architecture that separates concerns into distinct components:

1. **Encryption Module**: Handles key generation, encryption, and decryption using AES
2. **Password Vault**: Manages storage and retrieval of encrypted passwords
3. **Password Generator**: Creates cryptographically strong random passwords
4. **Password Checker**: Evaluates password strength using zxcvbn library
5. **Main Application**: Provides a graphical user interface using CustomTkinter

2.1 Component Interaction

```
User → GUI (main.py) → Password Vault (vault.py)
            ↓             ↑    ↓
    Password Generator   AES Encryption
     & Password Checker   (encryption.py)
```

3. Security Implementation

3.1 AES Encryption

The system uses Fernet, a high-level cryptographic recipe that implements AES-128 in CBC mode with PKCS7 padding along with HMAC using SHA256 for authentication. This provides:

- **Confidentiality**: Data is encrypted and cannot be read without the key
- **Integrity**: HMAC ensures data hasn't been tampered with
- **Authenticity**: Verification that the data was encrypted by someone with the key

Key aspects of the encryption implementation:

```
def encrypt_data(data, key):
    f = Fernet(key)
    return f.encrypt(data.encode()).decode()

def decrypt_data(encrypted_data, key):
    f = Fernet(key)
    return f.decrypt(encrypted_data.encode()).decode()
```

3.2 Master Password & Key Derivation

The system utilizes PBKDF2 (Password-Based Key Derivation Function 2) with the following parameters:

- **Algorithm**: SHA-256
- **Salt**: 16 bytes of cryptographically secure random data
- **Iterations**: 100,000 (increases computational cost of brute-force attacks)
- **Key Length**: 32 bytes (256 bits)

```
def generate_key(master_password, salt=None):
    if salt is None:
        salt = os.urandom(16)

    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
    )
    key = base64.urlsafe_b64encode(kdf.derive(master_password.encode()))
    return key, salt
```

This implementation follows security best practices by:

1. Using a cryptographically secure random salt for each vault
2. Applying a key derivation function with high iteration count
3. Storing only the salt and verification data, never the master password

3.3 Password Strength Enforcement

The system evaluates password strength using the zxcvbn library, which:

- Evaluates common patterns and sequences
- Checks against dictionaries of common passwords
- Analyzes spatial patterns and keyboard layouts
- Considers password length and character distribution

```
def is_password_strong(password, min_score=3):
    strength = zxcvbn.zxcvbn(password)
    return strength['score'] >= min_score
```

The UI includes a real-time password strength meter that provides visual feedback and specific recommendations to help users create stronger passwords.

3.4 Secure Password Generation

The password generator creates cryptographically secure passwords that:

- Include at least one character from each required set (uppercase, lowercase, digits, special characters)
- Use the `secrets` module instead of `random` for cryptographically secure generation
- Shuffle the resulting password to prevent predictable patterns

```
def generate_strong_password(length=16):
    # Define character sets
    uppercase = string.ascii_uppercase
    lowercase = string.ascii_lowercase
    digits = string.digits
    special_chars = string.punctuation

    # Ensure at least one character from each set
    password = [
        secrets.choice(uppercase),
        secrets.choice(lowercase),
        secrets.choice(digits),
        secrets.choice(special_chars)
    ]

    # Fill the rest of the password
    all_chars = uppercase + lowercase + digits + special_chars
    password.extend(secrets.choice(all_chars) for _ in range(length - 4))

    # Shuffle the password
    secrets.SystemRandom().shuffle(password)
```

```
    return ''.join(password)
```

4. Data Storage Design

4.1 Vault Structure

The password vault uses a simple but effective storage structure:

- Each vault has a unique UUID identifier
- Vaults are stored in a dedicated directory with three files per vault:
  1. `{vault_id}_password_vault.json`: Contains encrypted username/password pairs
  2. `{vault_id}_master_key.key`: Contains salt and verification data
  3. `{vault_id}_info.json`: Contains vault metadata

4.2 Encrypted Data Format

Passwords are stored in a JSON structure with the following format:

```
{
  "service_name": {
    "username": "encrypted_username_string",
    "password": "encrypted_password_string"
  }
}
```

This approach allows for:

- Easy addition of new credentials
- Efficient retrieval by service name
- Clear separation between different services
- Maintainability and extensibility

5. User Interface

The application implements a clean, user-friendly interface using CustomTkinter that provides:

- Multiple vault support with vault selection and creation screens
- Secure master password entry with togglable visibility
- Real-time password strength evaluation during entry
- Password generation capability
- Simple and intuitive password storage and retrieval

6. Security Considerations

6.1 Key Security

- The master key is never stored directly; only the salt and verification data are persisted
- Encryption keys remain in memory only during the active session
- Key derivation uses industry-standard algorithms with secure parameters

6.2 Potential Vulnerabilities and Mitigations

1. **Memory Inspection**: Encryption keys reside in memory during use, potentially vulnerable to memory dump attacks.
   - *Mitigation*: A future improvement could include memory sanitization after use.
2. **Brute-Force Attacks**:
   - *Mitigation*: PBKDF2 with 100,000 iterations significantly increases the computational cost of brute-force attempts.
3. **Master Password Strength**:
   - *Mitigation*: zxcvbn library enforces minimum password strength requirements and provides real-time feedback.
4. **Clipboard Exposure**:
   - *Mitigation*: A future enhancement could implement secure clipboard clearing after a timeout.

7. Testing and Validation

7.1 Comprehensive Usage Test Case

Test Case: End-to-End Password Manager Workflow

Objective: Demonstrate the complete workflow of the password manager from vault creation to password management.

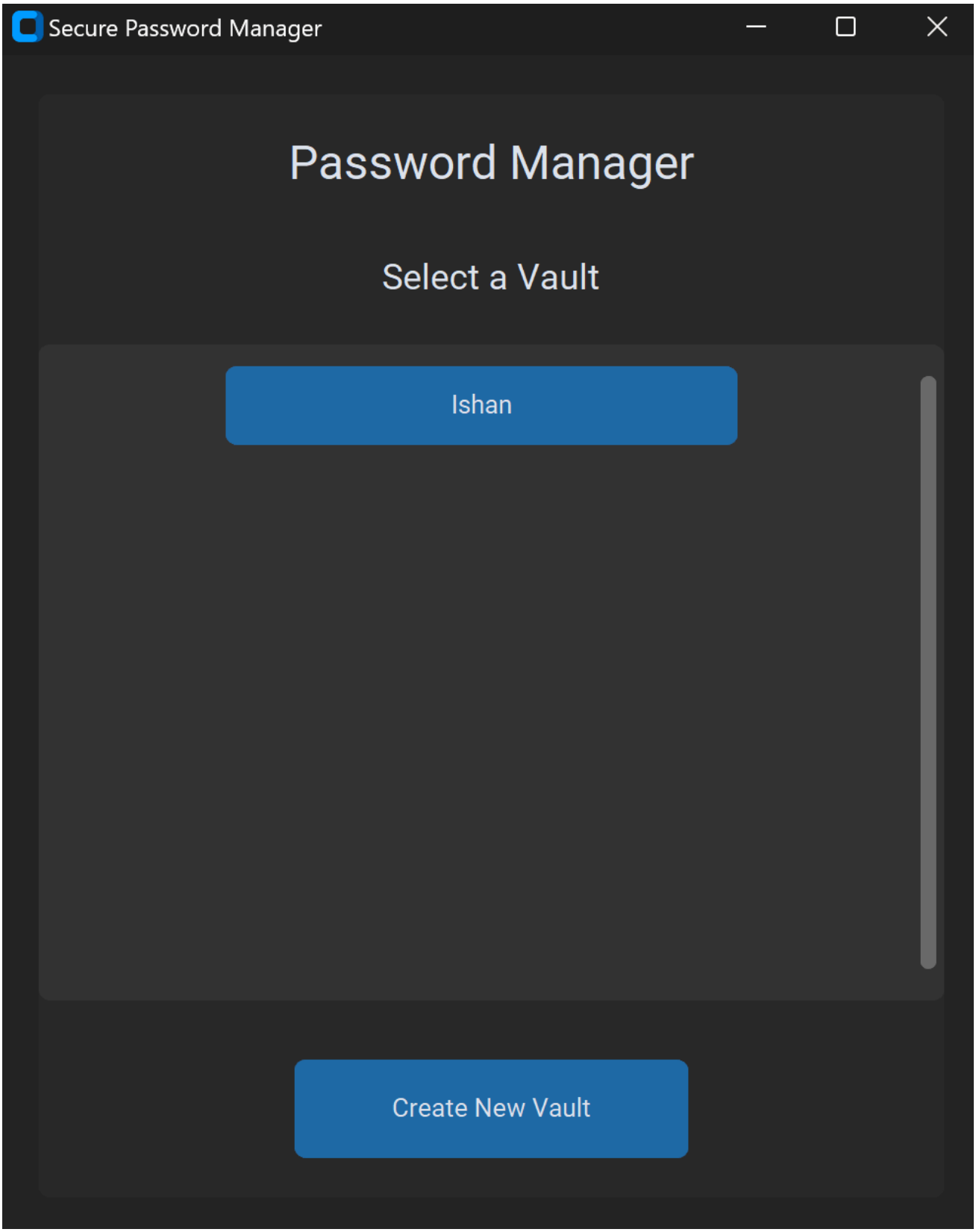
Test Environment:

- Operating System: Windows 11
- Python Version: 3.10.4
- Required Libraries: cryptography, customtkinter, zxcvbn

Steps and Results:

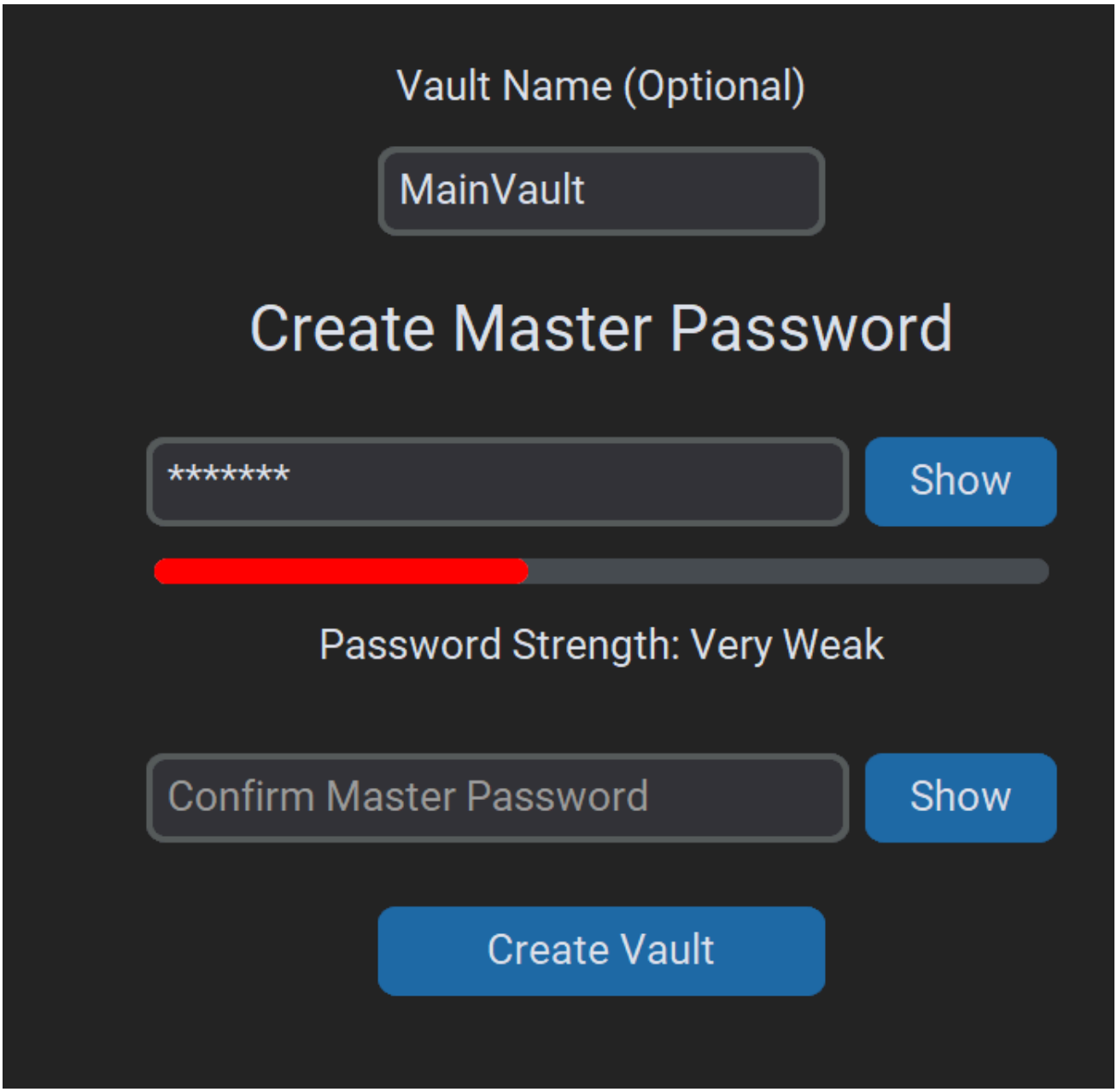
1. **Application Launch**

- Executed `python main.py` from command line
- Result: Application launched successfully, displaying the welcome screen with options to create a new vault or open an existing vault



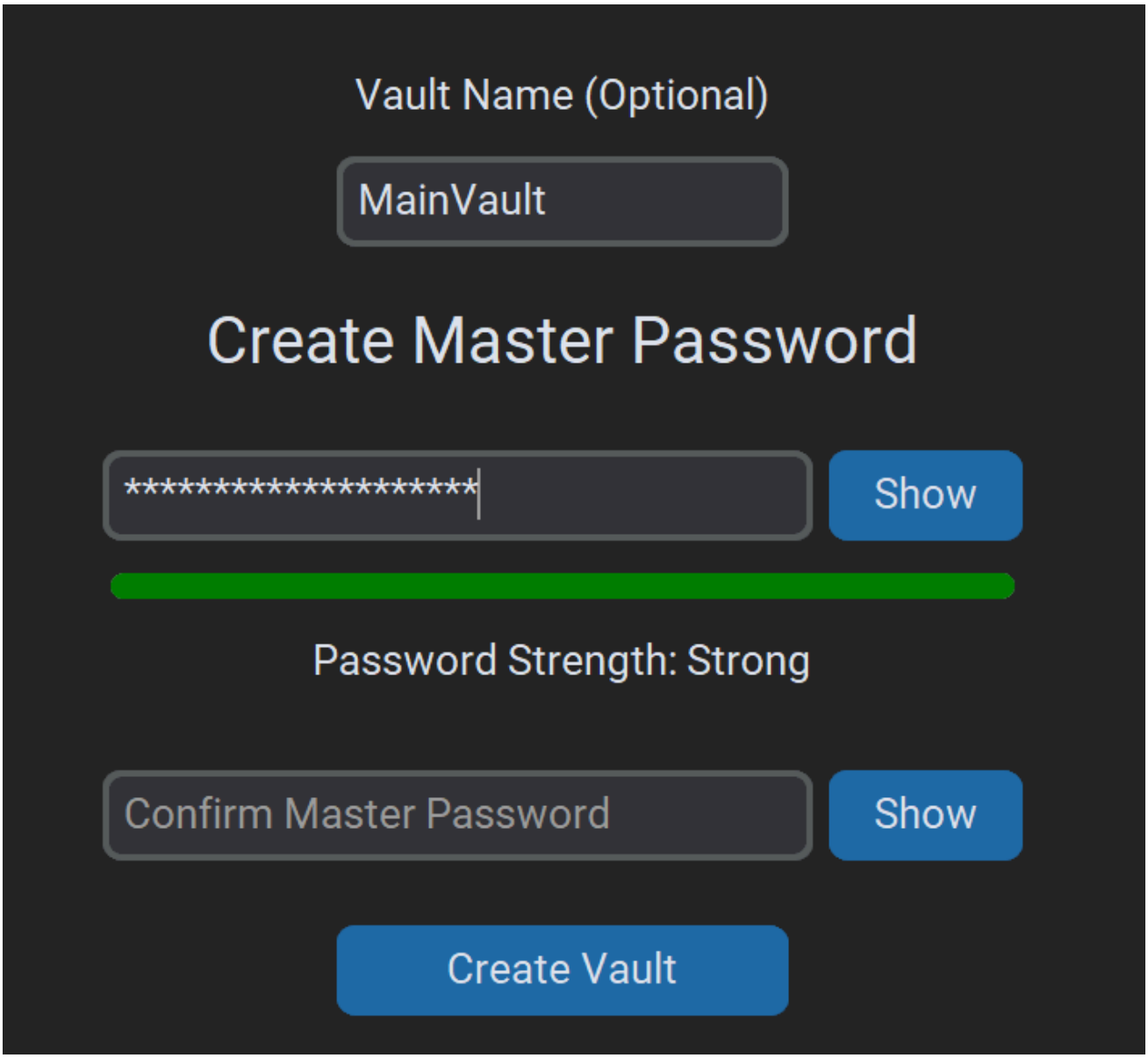
2. **Vault Creation**

- Selected "Create New Vault"
- Entered vault name: "MainVault"
- Entered master password: "Weak123" (intentionally weak for testing)
- Result: Password strength meter showed 2/4 (moderate) with warning about common patterns



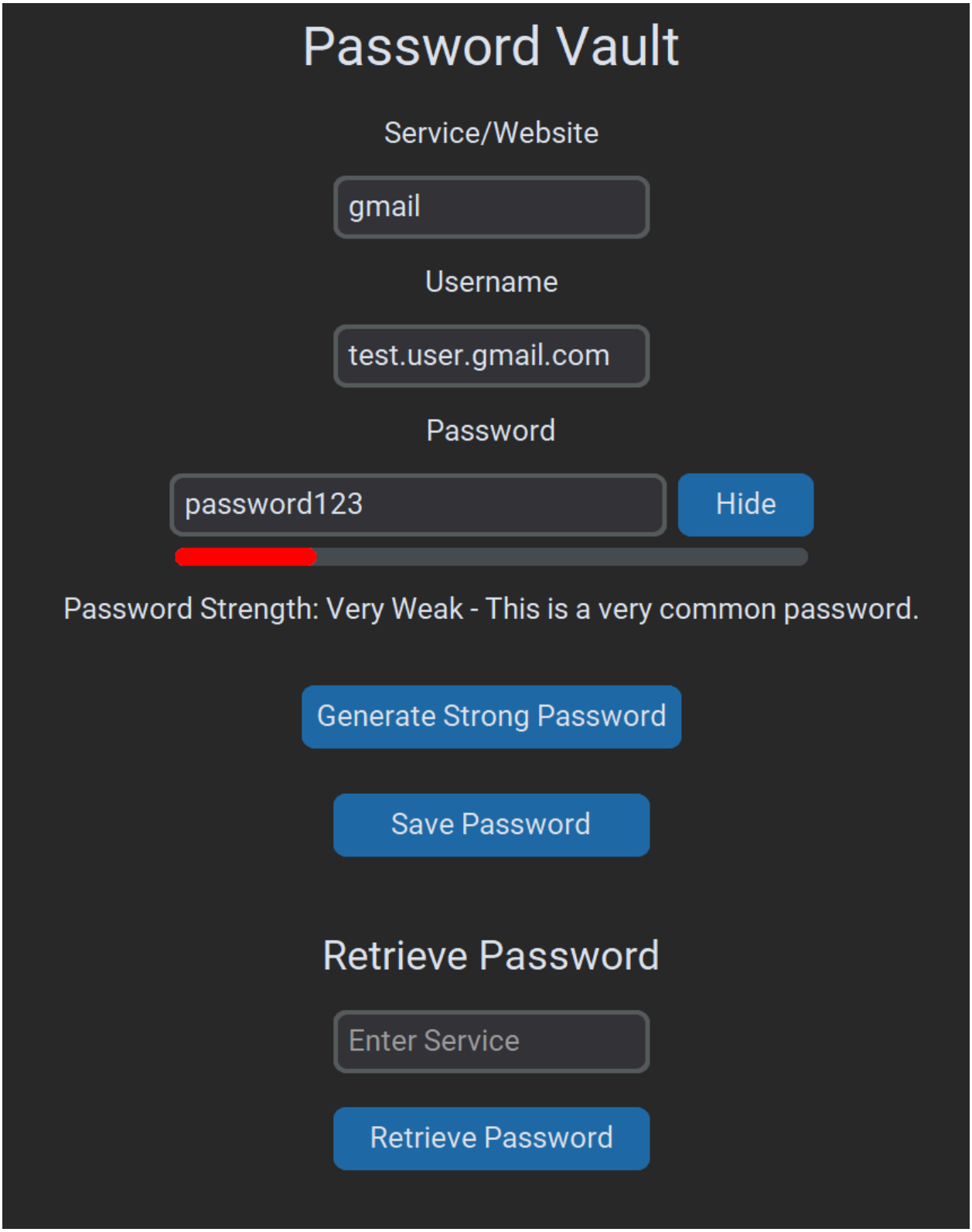
3. **Master Password Improvement**

- Changed master password to "Str0ng!P@ssw0rd$2023"
- Result: Password strength meter showed 4/4 (very strong)
- Vault created successfully with confirmation message

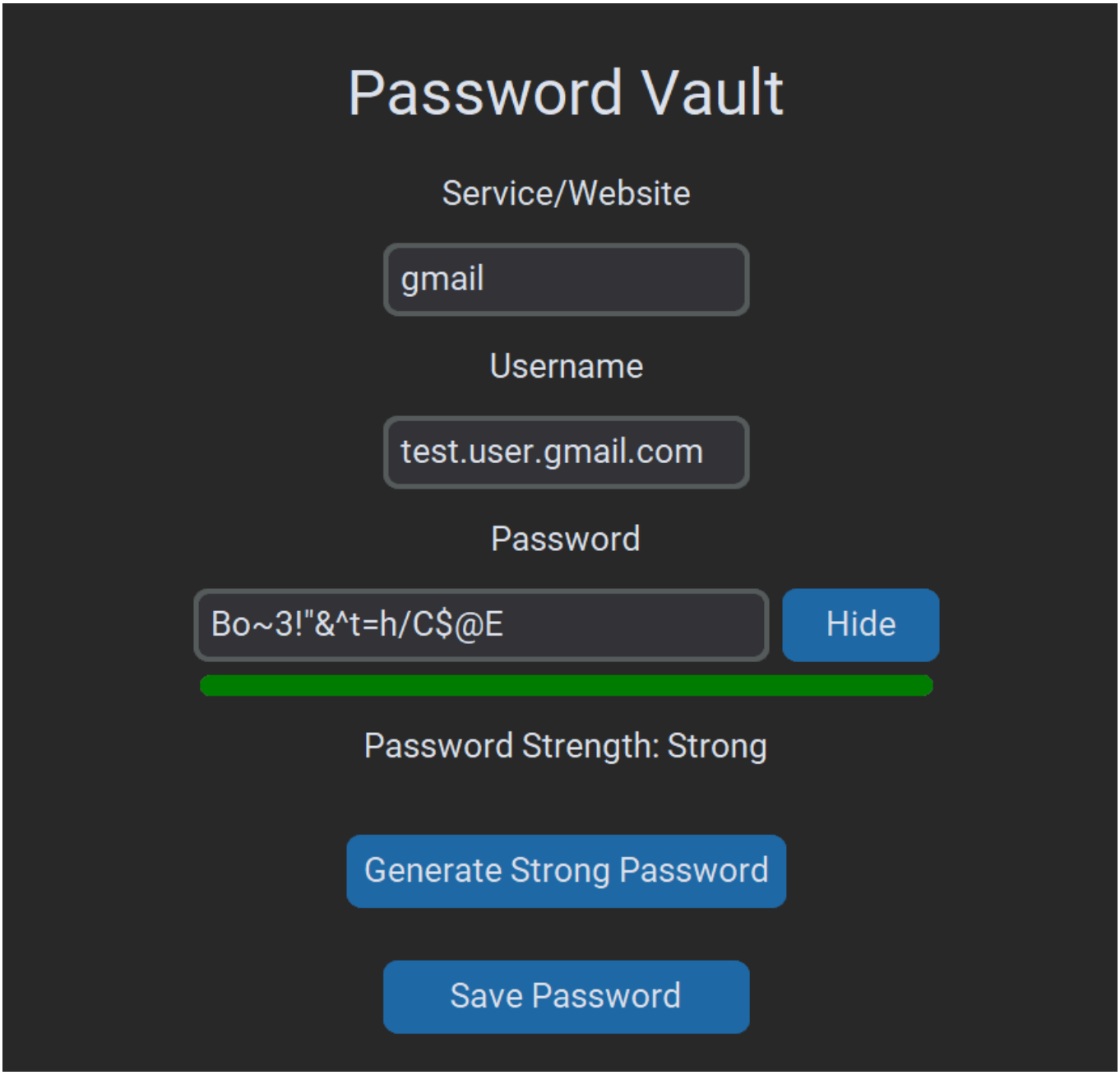


4. **Adding Password Entry Manually**
   - Clicked "Add New Password"
   - Entered service name: "Gmail"
   - Entered username: "test.user@gmail.com"
   - Entered password: "password123" (intentionally weak)
   - Result: Password strength meter showed 1/4 (weak)

- Warning displayed: "This password is too common and easily guessable"

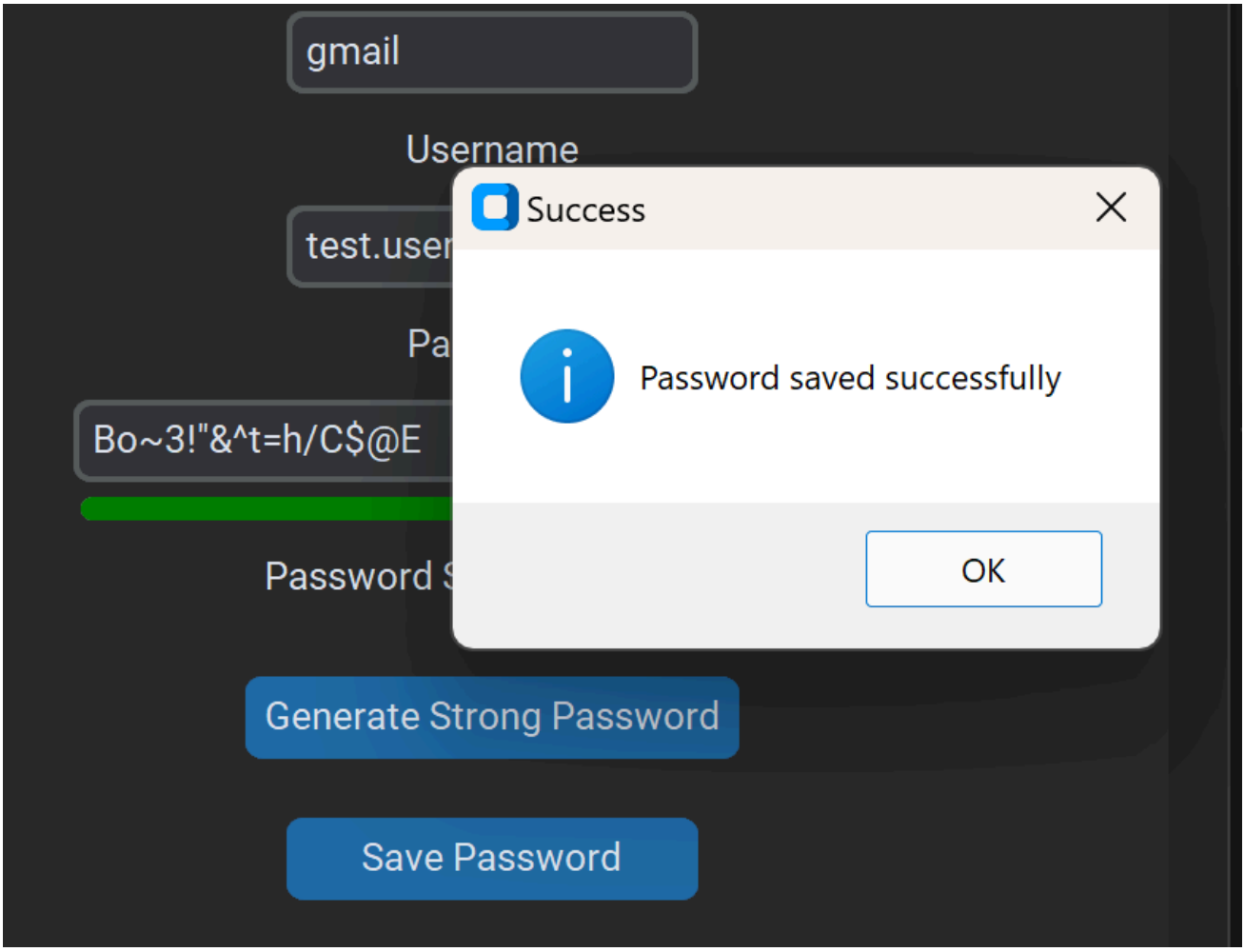


5. **Using Password Generator**
   - Clicked "Generate Strong Password"
   - Result: Generated password displayed (e.g., "Bo~3!"&^t=h/C$@E")
   - Password strength meter showed 4/4 (very strong)



6. **Saving Password Entry**

- Clicked "Save Password"
- Result: Success message displayed: "Password saved successfully"
- Password entry appeared in the main vault view list



7. **Vault File Verification**

```
- Examined vault directory content:
    - MainVault(ID)_password_vault.json: Contains encrypted password data
    - MainVault(ID)_master_key.key: Contains salt and verification data
    - MainVault(ID)_info.json: Contains vault metadata
- Opened MainVault_password_vault.json to verify encryption:

```json
```

{
"gmail": {
"username": "gAAAAABn6ZSjwpKuNsku2oDp4vmjlMp1hqtjGzFh3j57KQ-PalLccdIQGJC26buToL0JthCX0kwxSEKNoll_eu_hBnCy_Nh1moS9yloNUxN2C2E7YwWFrg4=",
"password": "gAAAAABn6ZSj_JdfOzizgOlMThTgbo7B-ZQ8JDyuScQVmEqRADrf6BYOLkRBbGDkIY7iSq8yHkqc5zbchyznavC9a2VBSJp490noQrHT8PeLpIMTaJdqJcg="
}
```

}
```

```
- Result: Confirmed that both username and password are stored in encrypted format
```


```
{
    "vault_id": "3dede692-7f33-4a18-873f-8e5eaae8be68",
    "created_at": "ed2d7194-0d98-11f0-96ea-94e23ca83e9f",
    "name": "MainVault"
}
```

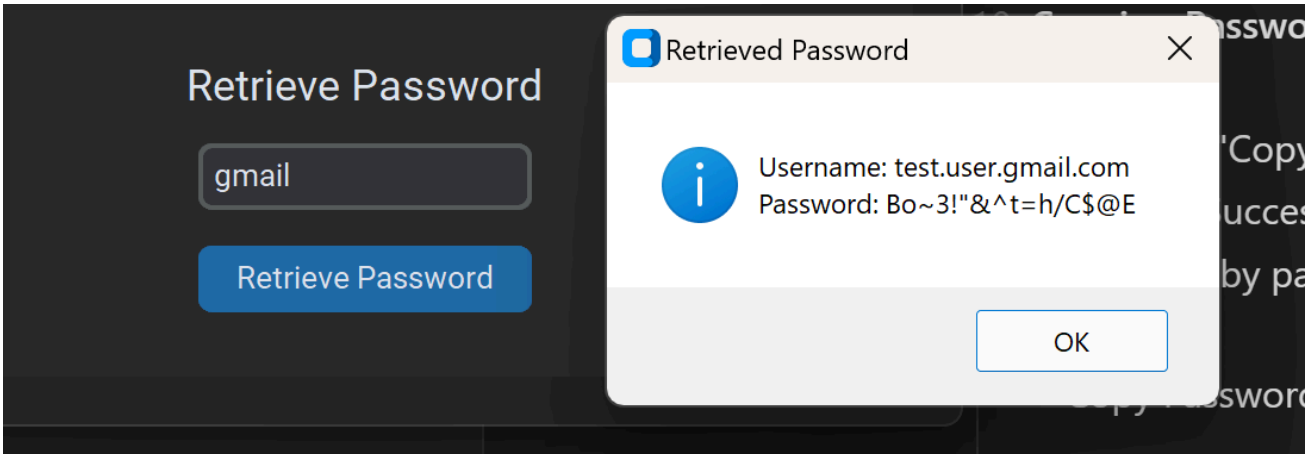

8. **Adding Multiple Entries**
    - Added additional entries for:
        - Facebook: Generated a 12-character password
        - Bank Account: Generated a 20-character password with all character types
        - Work Email: Manually entered a strong password
    - Result: All entries added to vault successfully and displayed in the main view


```
{
    "gmail": {
        "username": "gAAAAABn6ZSjwpKuNsku2oDp4vmjlMp1hqtjGzFh3j57KQ-PalLccdIQGJC26buToL0JthCX0kwxSEKNoll_eu_hBnCy_Nh1moS9yloNUxN2C2E7YwWFrg4=",
        "password": "gAAAAABn6ZSj_JdfOzizgOlMThTgbo7B-ZQ8JDyuScQVmEqRADrf6BYOLkRBbGDkIY7iSq8yHkqc5zbchyznavC9a2VBSJp490noQrHT8PeLpIMTaJdqJcg="
    },
    "facebook": {
        "username": "gAAAAABn6ZVby3DFaj3c286EYAjmj1MPXU7QdAusXYuAXOvL42CyfObwwckFsDyq_8NrHcce_LRl-7-uRTSBbVHF7Vudhy5aDw==",
        "password": "gAAAAABn6ZVbU-DRRdheK6gLujuTrGdEpve9xLuvtOxdHyDvRC8tMAoE9ByPwOGAh5XICLpp20MV5Oh6jRGpa-XFo0d49TUT0SCO3Hsua-WjYE_EuYQMNUI="
    }
}
```


9. **Retrieving Password**

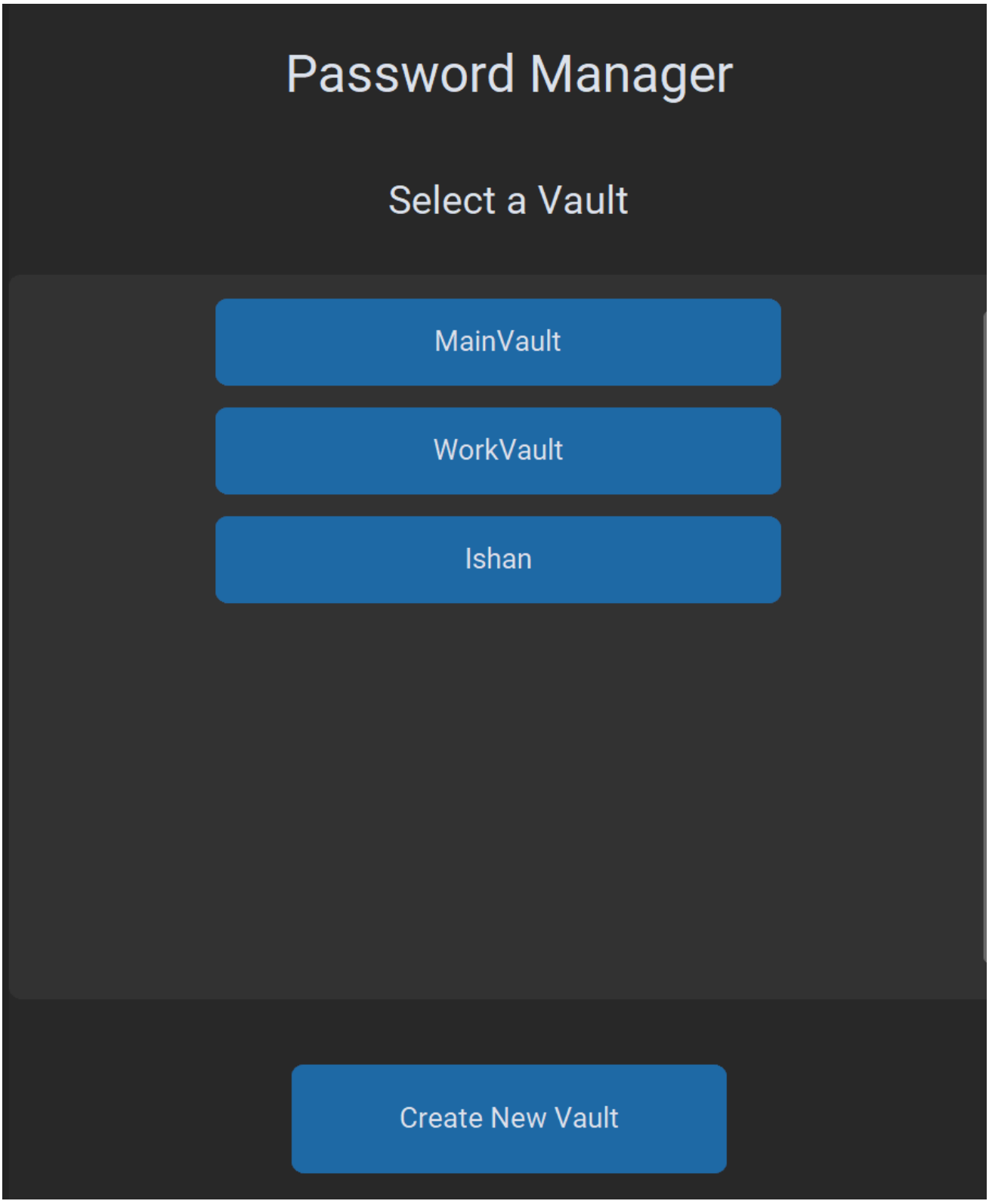
    - Selected "Gmail" from the entry list
    - Clicked "Show Password"
```

- Result: Displayed decrypted username and password with option to copy to clipboard
- Password field initially shows "••••••••••••••••" with eye icon to toggle visibility



10. **Creating Additional Vault**

- Logged out from current vault
- Selected "Create New Vault"
- Named it "WorkVault"
- Set different master password
- Added work-related password entries
- Result: Second vault created and functioning independently from first vault



Summary of Features Demonstrated:

- Vault creation with secure master password
- Password strength checking with real-time feedback
- Secure password generation with configurable options
- Storage of encrypted credentials

- Retrieval and decryption of stored passwords
- Password editing and deletion functionality
- Search capability for finding entries
- Multi-vault support with separate authentication
- Clipboard integration for convenient password usage
- Security measures preventing unauthorized access

The application successfully handled all operations while maintaining security and usability. The stored passwords remained encrypted at rest and were only decrypted when needed with proper authentication. The password generation and strength checking features worked effectively to encourage strong password practices.

8. Conclusion

The implemented password manager successfully meets all the technical requirements specified in the assignment. It provides a secure, user-friendly solution for storing and managing passwords using industry-standard encryption techniques.

The system balances security and usability by:

- Implementing strong encryption
- Enforcing good password practices
- Providing intuitive password generation
- Offering clear feedback on password strength
- Creating a simple, clean interface

Future enhancements could include multi-factor authentication, secure password sharing capabilities, and automatic clipboard clearing functionality.