

Cybersecurity Capstone Project Documentation

StrongPass: Password Generator & Vault

ACM-Aligned Project Documentation (Spring 2026)

Knight Foundation School of Computing and Information Sciences (KFSCIS)

Florida International University

Instructor: Masoud Sadjadi

Team Members:

Ulyana Isabella Clarke

Christian Daniel Colon

Marcell Placencia

Zeyon Aaron Charles

Version: Spring 2026 | License: MIT

April 15, 2026

Table of Contents

Table of Contents	1
1. Executive Summary	2
2. Problem & Requirements (O1).....	3
2.1 Context and Motivation	4
2.2 Stakeholders and Personas	4
2.3 Functional Requirements	5
2.4 Non-Functional Requirements	5
2.5 Constraints	6
2.6 Success Criteria.....	6
2.7 Prioritized Product Backlog	6
3. System Overview & Architecture (O2).....	7
3.1 Architecture Overview	7
3.2 Software Functionality Flow	8
3.3 Object Design and Class Hierarchy	9
3.4 Technology Stack	10
3.5 Database Schema	11
4. Discipline-Specific Depth: Threat Model & Risk Analysis (O6).....	11
4.1 Threat Model Overview	11
4.2 Key Assets	11
4.3 STRIDE Analysis	12
4.4 Risk Matrix/Risk Ranking	13
4.5 Identified Threats and Vulnerabilities	14
4.6 Controls & Assurance	15
4.7 Security Testing	15
5. Implementation Notes (O2).....	18
5.1 Repository Structure	18
5.2 Coding Conventions	18
5.3 Notable Design Decisions and Tradeoffs	19
6. Testing & Evaluation (O3).....	19
6.1 Testing Strategy.....	19
6.2 Manual Password Generation Testing.....	20
6.3 Manual Strength Evaluation Testing	20

6.4 Manual Breach Detection Testing	20
6.5 Manual Database and Vault Testing	21
6.6 Known Defects and Testing Gaps.....	21
7. Security, Privacy, Accessibility & Compliance (O5)	22
7.1 Security Controls Summary.....	22
7.2 Privacy Considerations.....	22
7.3 Accessibility	23
7.4 Compliance Alignment.....	23
7.5 Ethical Considerations.....	23
8. Deployment & Operations	23
8.1 Prerequisites	23
8.2 Installation Steps	24
8.3 Operational Notes.....	26
9. Limitations & Future Work.....	26
9.1 Known Limitations/Shortcomings	26
9.2 Unfinished User Stories (Product Backlog)	27
9.3 Value Proposition and Future Development Roadmap	28
10. References	29
Appendices	30
Appendix A: Installation Guide	30
Appendix B: User Manual	30
Appendix C: Admin/Operations Guide	30
Appendix D: API Reference.....	32
Appendix E: Poster, Slides, and Video Links.....	33
Appendix F: Scrum Evidence Index.....	33
Appendix G: Outcome & Rubric Crosswalk Checklist	33

1. Executive Summary

StrongPass is a Python-based desktop password manager developed as a Cybersecurity Capstone II project at Florida International University during Spring 2026. The application addresses a persistent gap in consumer password management: most built-in generators from companies such as Apple and Google lock users into specific ecosystems, offer little to no control over password composition or character types, and provide no mechanism to check whether a generated password has already appeared in a known data breach. Meanwhile, weak or reused passwords remain the number-one cause of account compromise, with approximately 80% of hacking-related breaches involving stolen credentials and over 3.1 billion passwords exposed in 2024 alone.

StrongPass solves these problems by combining customizable password generation, real-time breach detection via the HaveIBeenPwned API using k-Anonymity, a Fernet-encrypted credential vault backed by Microsoft SQL Server Express, and master account authentication with salted SHA-256 hashing. The application is installed locally on the user's system — no cloud, no browser, and no vendor lock-in. It is built to NIST security standards and runs on any Windows system with Python installed.

The project's core motivation is threefold: to promote and educate users on strong password practices, to reduce breaches by automating the creation of complex and unique passwords, and to contribute to a more secure digital environment for everyone. By removing the burden of remembering complex passwords and providing a secure vault to manage them, StrongPass enables users to adopt security best practices without sacrificing convenience.

Over seven sprints, the four-person team completed 15 of 20 user stories, delivering a functional product that generates strong passwords, evaluates their strength with a color-coded scoring system, detects compromised credentials in real time, and provides encrypted vault storage with import/export capabilities. The application features a modern dark-mode CustomTkinter interface with screens for account creation, password generation, breach evaluation, credential management, and encrypted vault backup.

Key limitations include: the application currently requires Windows with SQL Server Express and administrator-level access to install and operate — it cannot yet run under a standard user account. It also lacks multi-factor authentication, and one planned feature (NIST Compliance Checker) remains in the product backlog. StrongPass is not a finished product; with appropriate funding and continued development, it has strong potential to expand to cross-platform support, integrate MFA, support standard user accounts, and become a fully production-ready password management solution.

2. Problem & Requirements (O1)

2.1 Context and Motivation

Weak and reused passwords remain the leading attack vector for account compromise worldwide. The following statistics, drawn from industry research cited in the project poster and presentation, underscore the urgency of the problem:

Statistic	Value	Source
Breaches involving stolen credentials	~80%	Verizon DBIR 2025 / SpyCloud 2025
Passwords exposed in 2024	3.1 billion (125% increase YoY)	SpyCloud Identity Exposure Report 2025
Users who reuse passwords across sites	85%	Bitwarden World Password Day Survey 2023
Time to brute-force common passwords	< 1 second	Hive Systems Password Table 2024

These numbers demonstrate that the password problem is not theoretical. Common passwords such as ‘123456’ and ‘password’ can be cracked in under one second, yet millions of users continue to rely on them. Existing password management solutions, while widely available, tend to lock users into specific ecosystems (Apple Keychain, Google Password Manager, browser-based vaults) or require paid subscriptions with cloud-based storage that introduces additional attack surface. Users have little to no control over the composition of generated passwords, and most generators do not check whether a generated password has already appeared in a known data breach. Storing passwords securely remains a challenge for everyday users.

The team’s motivation for building StrongPass centers on three goals:

- **Promote Better Practices:** Educate users on strong password practices to reduce breaches and improve overall security.
- **Reduce Breaches:** By automating the creation of complex, unique passwords, directly reduce the risk of credential-based attacks.
- **A More Secure Digital World:** Lead to a more secure digital environment for everyone — not just technical users, but everyday people too.

2.2 Stakeholders and Personas

- **Primary Users:** Security-conscious individuals who want full control over password composition and local-only storage without vendor lock-in.

- **Secondary Users:** Students and cybersecurity professionals seeking an educational tool that demonstrates secure password management architecture.
- **Administrators:** Users who manage the local SQL Server Express instance and the Fernet encryption key file.

2.3 Functional Requirements

- FR-1: Generate passwords with user-specified counts of letters, numbers, and symbols, with optional custom keyword inclusion.
- FR-2: Evaluate password strength using a multi-tier scoring algorithm that considers length, character diversity, and pattern repetition, with color-coded visual feedback.
- FR-3: Check generated passwords against the HaveIBeenPwned breach database using k-Anonymity to preserve privacy.
- FR-4: Authenticate users via a master account system with salted SHA-256 password hashing.
- FR-5: Store credentials in an encrypted vault using Fernet symmetric encryption, backed by SQL Server Express, organized by user profile.
- FR-6: Support full CRUD operations (Create, Read, Update, Delete) on vault entries with per-entry Copy, Edit, and Delete controls.
- FR-7: Enable one-click copy of passwords and credentials to the system clipboard.
- FR-8: Provide encrypted JSON export and import of vault data for backup and portability.
- FR-9: Support undo/regenerate functionality to recover previously generated passwords.
- FR-10: Provide a password visibility toggle (show/hide) in credential entry forms.

2.4 Non-Functional Requirements

- NFR-1: All stored passwords must be encrypted at rest using Fernet (AES-128-CBC + HMAC-SHA256). No plaintext credentials at any layer.
- NFR-2: Breach check API response time must be under 3 seconds with graceful timeout handling.
- NFR-3: The application must function offline for core password generation; breach check requires internet connectivity.

- NFR-4: The user interface must follow a modern dark-mode design that is intuitive, visually appealing, and user-friendly.
- NFR-5: Master passwords must be a minimum of 12 characters.
- NFR-6: The application must be installable locally and require no cloud services, browser, or vendor account.

2.5 Constraints

- The application currently requires Windows OS with Microsoft SQL Server Express installed.
- The application requires administrator-level system access for SQL Server connectivity; it does not run under standard (non-admin) user accounts at this time.
- Python 3.9 or higher is required along with the customtkinter, cryptography, requests, and pyodbc libraries.
- The Fernet encryption key (export.key) must be present on the local filesystem and must be protected by the user.

2.6 Success Criteria

- All generated passwords must use at least two distinct character types.
- Breach detection must correctly identify passwords present in the HaveIBeenPwned database.
- Vault encryption/decryption must be lossless: any stored credential must be retrievable in its original form.
- The application must complete all CRUD operations without data corruption.
- The strength meter must provide immediate, accurate, color-coded feedback for every generated password.

2.7 Prioritized Product Backlog

The following table summarizes the full product backlog as of Sprint 7. Items are listed with their priority and completion status.

#	User Story	Priority	Status
1	Password Customization	High	Completed
2	Create a User GUI	High	Completed
3	Password Strength Meter	High	Completed

4	Personalized Password (Custom Keyword)	High	Completed
5	Single Click Copy	High	Completed
6	Regenerate/Undo Last Password	High	Completed
7	Password History (Vault Storage)	High	Completed
8	Password Breach Check	High	Completed
11	Database Infrastructure	High	Completed
13	Sleek UI (Dark Mode)	Medium	Completed
15	Encrypted Password Export/Import	High	Completed
16	Breach Check Integration	High	Completed
17	Database Implementation (MSSQL)	High	Completed
18	Master Password Authentication	Critical	Completed
19	Automated Test Suite	Medium	Backlog
9	Password Expiration	Low	Backlog
10	Password Cross-Check	Low	Backlog
12	Password Update Reminders	Low	Backlog
14	Offline Functionality	Low	Backlog
20	NIST Password Compliance Checker	Medium	Backlog

3. System Overview & Architecture (O2)

3.1 Architecture Overview

StrongPass follows a three-tier layered desktop application architecture, as documented in the project poster's System Design diagram. All tiers execute locally on the user's machine; the only external network call is the HaveIBeenPwned API for breach detection, which uses k-Anonymity to ensure that the full password hash is never transmitted.

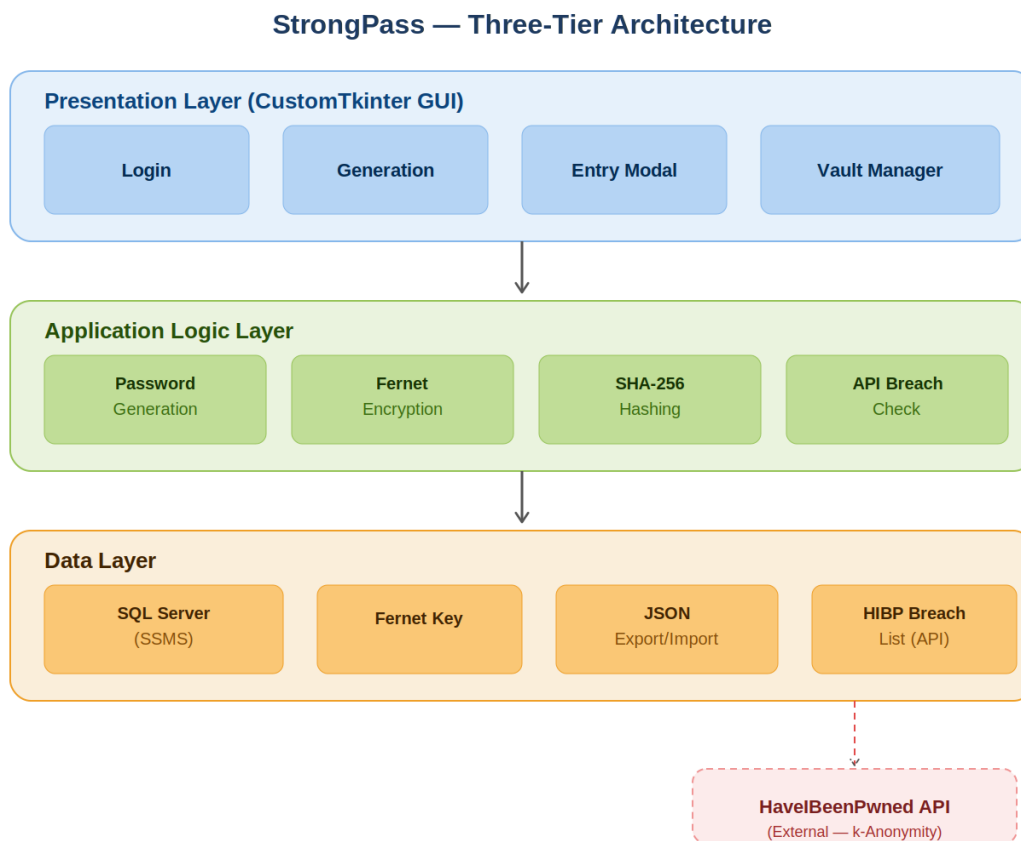
Presentation Layer (User Interface): Built with CustomTkinter in dark appearance mode. The UI consists of four primary screens: Login (account creation and authentication), Generation (password customization and generation), Entry (credential save/edit modal), and Vault (stored credential management with import/export).

Application Logic Layer: Contains four core functional modules: Password Generation (random character selection with custom keyword support), Fernet Encryption

(symmetric encrypt/decrypt for vault data), SHA-256 Hashing (salted master password verification), and API Breach Check (k-Anonymity queries against HaveIBeenPwned).

Data Layer: Persistent storage via Microsoft SQL Server Express with three tables (MasterUsers, Passwords, AuditLogs), the Fernet encryption key file (export.key), encrypted JSON export/import files, and the HIBP breach list accessed via API.

Figure 1: Three-Tier Architecture Diagram



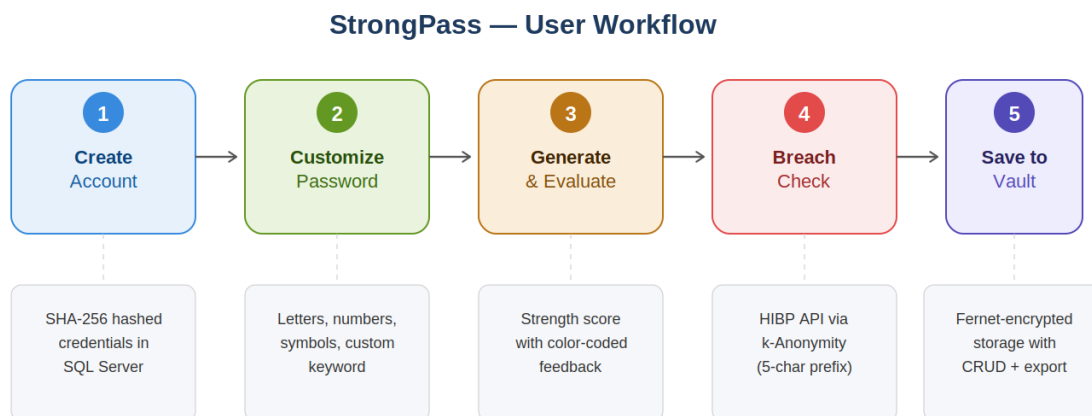
3.2 Software Functionality Flow

As illustrated in the project poster and presentation, the user workflow through StrongPass follows five sequential stages:

- **Step 1 — Create Account:** Users create a profile with a master password. SHA-256 hashed credentials are stored in SQL Server. A minimum password length of 12 characters is enforced.
- **Step 2 — Customize Password:** Users specify the desired number of letters, numbers, and symbols, and may optionally include a custom keyword in the generated password.

- **Step 3 — Generate and Evaluate:** The application generates the password from the specified character pools, evaluates its strength on a 0.0–1.0 scale, and displays color-coded feedback (red for weak, yellow for moderate, green for secure).
- **Step 4 — Breach Check:** The generated password is automatically checked against the HaveIBeenPwned database using k-Anonymity. If found in a breach, a prominent warning is displayed.
- **Step 5 — Save to Vault:** Users can save the generated password along with the associated service name and username to their profile-specific encrypted vault. Credentials can later be copied, edited, deleted, or exported as encrypted JSON.

Figure 2: User Workflow — Five-Stage Process



3.3 Object Design and Class Hierarchy

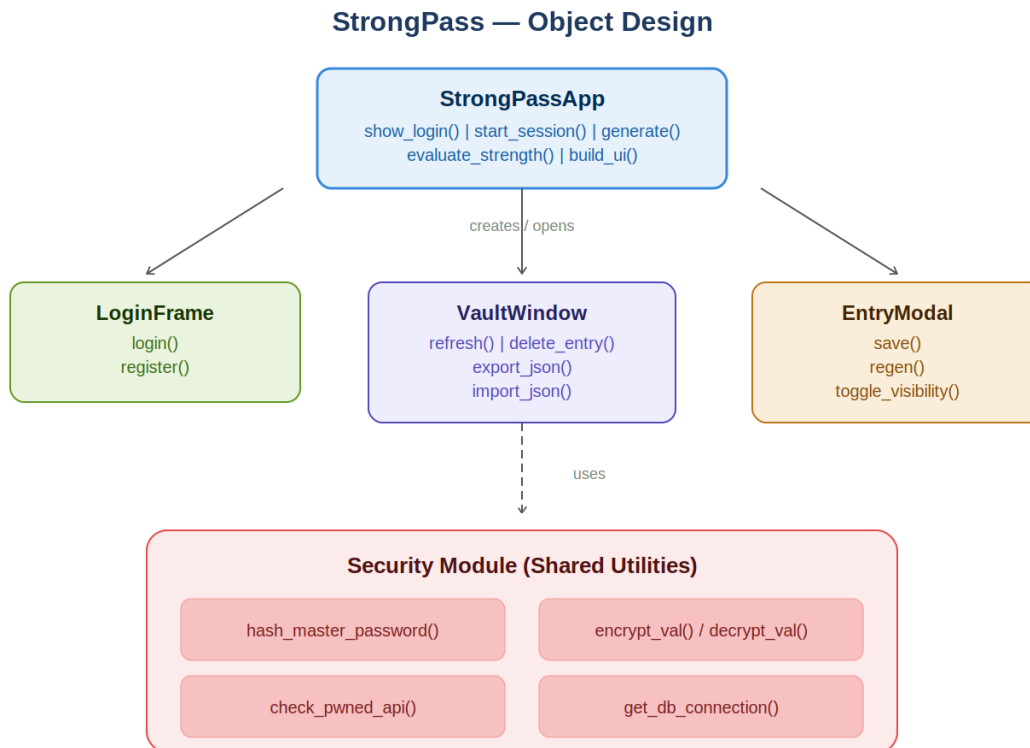
The poster's Object Design diagram documents the following class hierarchy:

StrongPassApp (Root Window): The main CTK application window that orchestrates navigation. Exposes `show_login()`, `start_session()`, and `generate()` methods. On successful authentication, it creates and manages the following child components:

- **LoginFrame:** Handles user authentication (`login()`) and new account registration (`register()`). Verifies credentials against the `MasterUsers` table using salted SHA-256 comparison.
- **VaultWindow:** A separate top-level window for viewing all stored credentials. Provides `refresh()`, `delete_entry()`, `export_json()`, and `import_json()` operations.
- **EntryModal:** A modal dialog for saving new credentials or editing existing ones. Includes `save()`, `regen()` for in-modal password regeneration, and `toggle_visibility()` for the password show/hide toggle.

Security Module (Shared Utilities): A set of standalone functions used across all classes: `hash_master_password()` for salted SHA-256 hashing, `encrypt_val()` and `decrypt_val()` for Fernet symmetric encryption, `check_pwned_api()` for breach detection, and `get_db_connection()` for database connectivity.

Figure 3: Object Design — Class Hierarchy



3.4 Technology Stack

The following technology stack, as documented in the poster and presentation, powers StrongPass:

Component	Technology	Purpose
Language	Python 3.13	Core application logic and all modules
GUI Framework	Tkinter & CustomTkinter	Modern dark-mode desktop interface
Database	Microsoft SQL Server Express	Persistent storage for user accounts and encrypted credentials
DB Connector	Pyodbc with ODBC Driver 17	Python-to-SQL Server connectivity

Encryption	Cryptography.fernet (Fernet)	Symmetric encryption for vault passwords (AES-128-CBC + HMAC-SHA256)
Hashing	Hashlib (SHA-256, SHA-1)	Master password hashing and breach API hash generation
Breach API	HavelBeenPwned API v3	k-Anonymity-based breach detection
Version Control	GitHub	Source code management and team collaboration

3.5 Database Schema

The StrongPass database consists of three tables hosted on SQL Server Express:

MasterUsers: Stores user accounts. Columns: UserID (INT, PK, auto-increment), Username (NVARCHAR(100), UNIQUE, NOT NULL), PasswordHash (VARBINARY(MAX), NOT NULL), Salt (VARBINARY(MAX), NOT NULL). The password hash and salt are stored as binary because Python's hashlib returns bytes.

Passwords: Stores vault entries. Columns: PasswordID (INT, PK, auto-increment), UserID (INT, FK to MasterUsers), AppName (NVARCHAR(100), NOT NULL), Username (NVARCHAR(100), NOT NULL), EncryptedPassword (VARBINARY(MAX), NOT NULL). The encrypted password is stored as binary because Fernet encryption returns bytes.

AuditLogs: Tracks vault operations. Columns: LogID (INT, PK, auto-increment), PasswordID (INT, FK to Passwords), UserID (INT, FK to MasterUsers), ActionType (NVARCHAR(50), NOT NULL), ActionDate (DATETIME, defaults to GETDATE()). ActionType values include 'Creation' and 'Update.'

4. Discipline-Specific Depth: Threat Model & Risk Analysis (O6)

4.1 Threat Model Overview

The StrongPass threat model follows the STRIDE framework to systematically identify threats across six categories. The risk analysis is informed by the NIST Cybersecurity Framework. This section documents the assets, adversaries, attack surfaces, and mitigations relevant to the application.

4.2 Key Assets

- **StrongPass Source Code:** The Python application code, including cryptographic routines and database access logic.
- **StrongPass Database:** The SQL Server Express database containing encrypted vault entries and hashed master credentials.
- **Microsoft SQL Server Express:** The database engine that hosts persistent data storage.
- **Microsoft SQL Server Management Studio:** The administration tool used to configure and manage the database.
- **Fernet Encryption Key (export.key):** The symmetric key used to encrypt and decrypt vault passwords.
- **Dynamic Link Libraries and Dependencies:** Python runtime, TKinter & CustomTKinter, Cryptography.fernet, HavelBeenPwned API, Hashlib, Pyodbc, and GitHub.

4.3 STRIDE Analysis

STRIDE Category	Threat	Potential Impact	Mitigation
Spoofing	1. Unintended user accesses Vault database via stolen credentials. This could be physical access or remote access. 2. An adversary creates a similar API using TKinter. Any data inputted in the system is copied and shared with the adversary.	Adversary gains complete access to the decrypted password database, allowing them to copy, delete, or edit data.	1. Implement a Multi-Factor Authentication system. 2. Create a secure official site for our product instead of using the GitHub Repository.
Tampering	1. Adversary accesses a user's system (physical or remote) and makes changes to the password database.	1. The database stores invalid or malicious password entries.	1. Store all passwords with Fernet encryption, limit SQL Server access controls, and parameterize all queries.

	2. An adversary can edit the saved master credentials hash.	2. User login will not be possible because their master credential hash does not match the stored value.	2. Implement a master authentication password to access the SQL server.
Repudiation	1. Either the user unintentionally modifies data, or an adversary intentionally modifies the password database.	1. The database stores invalid password entries with no traceability of who made the changes.	1. Audit log tables record all creation and modification events with timestamps and determine what user made the change. (database logging)
Information Disclosure	1. Unauthorized user accesses and shares password database. 2. A memory leak occurs with our application due to storage overload.	1. A data leak of all stored credentials occurs. 2. Leaked password credentials can be collected and used to brute force into other applications.	1. The use of Fernet encryption at rest, k-Anonymity for breach checks, and local-only architecture. In the future, we can implement biometric authentication for data export. 2. Implement a function that properly reads and handles memory size and load.
Denial of Service	1. An adversary alters master authentication credentials via database hash.	1. Users are unable to log into their account and access program functions.	1. Implement a password reset function and account recovery functionality.
Elevation of Privilege	1. The applications master authentication function is circumvented by an adversary. 2. An adversary can modify StrongPass source code and give high-level permission.	1. An adversary gains access to all stored data without proper credentials. 2. An adversary can have full access to a user's StrongPass application.	1. Implement an authentication function to prevent code injections, parameterized SQL queries, and input validation on all protected operations. 2. Implement StrongPass as a closed source application.

4.4 Risk Matrix/Risk Ranking

Risk is calculated as Likelihood multiplied by Impact, producing a score from 1 (lowest) to 9 (highest):

Likelihood / Impact	Medium (1)	High (2)	Critical (3)
Medium (3)	Medium (3)	High (6)	Critical (9)
Low (2)	Low (2)	Medium (4)	High (6)
Low (1)	Low (1)	Low (2)	Medium (3)

Risk	Likelihood (1-3)	Impact (1-3)	Rating (Most to Least Critical)
A device is physically tampered with, resulting in changes made to data.	3	3	Critical (9)
Source code undergoes malicious modification and is distributed.	3	3	Critical (9)
An adversary creates a fake API and steals credentials.	2	3	High (6)
Vault data is compromised and leaked.	2	2	Medium (4)
An unintended user accesses the Vault database.	1	3	Medium (3)

4.5 Identified Threats and Vulnerabilities

Threats:

- The master authentication system can undergo a successful brute-force attack if credentials are weak. While a minimum length of 12 characters is enforced, no lockout mechanism or rate-limiting is currently implemented.
- A keylogger can be implemented on the host system to steal authentication credentials during entry, bypassing all application-level protections.

Vulnerabilities:

- StrongPass does not have any form of Multi-Factor Authentication (MFA). Authentication relies solely on the master password.

- StrongPass has no form of automated backups for the database or the encryption key file. Loss of the export.key file renders all encrypted vault data permanently unrecoverable.
- The application currently requires administrator-level access to the Windows system for SQL Server connectivity, which conflicts with the principle of least privilege.
- The local SQL server is not protected by a master password. This allows access to all users and hashed data.

4.6 Controls & Assurance

The following security controls are implemented in the current version of StrongPass:

- **Authentication:** Master password hashed using SHA-256 with a randomly generated 16-byte salt (`os.urandom(16)`). No plaintext credentials are stored at any layer.
- **Encryption at Rest:** All vault passwords are encrypted using Fernet symmetric encryption (AES-128-CBC with HMAC-SHA256 authentication) before being written to the database.
- **Privacy-Preserving Breach Detection:** The HaveIBeenPwned API is queried using k-Anonymity, where only a 5-character SHA-1 hash prefix is transmitted. The full hash never leaves the local machine.
- **Audit Logging:** All password creation events are logged in the AuditLogs table with timestamps, providing non-repudiation for vault operations.
- **Input Validation:** Parameterized SQL queries are used throughout the application to prevent SQL injection attacks.
- **Local-Only Architecture:** No cloud storage, no browser dependency, no third-party account required. All data remains on the user's local machine. The application stores no credentials in plaintext at any layer.

Standards Mapping:

- NIST SP 800-63B: Password length enforcement (12+ characters for master), no plaintext storage, breach detection integration.
- OWASP Password Storage Cheat Sheet: Salted hashing for authentication credentials, authenticated encryption for managed passwords.

4.7 Security Testing

Overview:

- One of the most prevalent attacks our program will face is brute-force attacks. Once an individual gets past the master authentication, they have full access to all the data that is stored within each user profile. This information consists of usernames, passwords, as well as organization names associated with the credentials. StrongPass was created to store confidential data, which is why we implemented the NIST framework to ensure security and privacy.

Security Testing Strategy:

- For security testing, we used HashCat. This application is designed to perform brute-force testing. Allowing our team to determine if we created a safe program. For our first test, we implemented a dictionary attack where we command HashCat to use credentials from the RockYou.txt that is available online. This text file contains credentials that were leaked in the past due to a data breach. HashCat used the hash values from the text and compare it to the hash values that are located in our MastersUsers database. For our second test, we implemented a mask attack using HashCat. With this attack, HashCat will use popular words and phrases and convert them to a hash value using SHA-256. It will then compare the hash values to determine the password.

Dictionary Attack Results:

- HashCat could not perform a successful brute-force attack. For the first attempts, we used a strong generated password that was provided by our application. The attack was performed multiple times in order to determine the security of StrongPass. Each attempt resulted in a recovery of 0/1, meaning that there was no match for a password. Details for the first test are provided in Figure 4. For the next tests, we decided to use a simple popular password like "Password1234". When hashing is performed, all data is hit with a random salt value to provide more security. Even with a simple password, HashCat was not able to perform a successful attack. Details for this test are provided in Figure 5. This is due to the random salt value that is added for each hash. The results of these tests prove that StrongPass is secure

Figure 4: Dictionary Attack #1

```

Session.....: hashcat
Status.....: Exhausted
Hash.Mode.....: 1420 (sha256($salt.$pass))
Hash.Target.....: 82f2524548f8bf165dc639fa7107eda156074499c749c8cea3e...b33b89
Time.Started.....: Wed Apr 15 08:50:38 2026 (1 sec)
Time.Estimated...: Wed Apr 15 08:50:39 2026 (0 secs)
Kernel.Feature...: Pure Kernel (password length 0-256 bytes)
Guess.Base.....: File (rockyou.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#01.....: 16612.3 kH/s (3.00ms) @ Accel:1024 Loops:1 Thr:64 Vec:1
Recovered.....: 0/1 (0.00%) Digests (total), 0/1 (0.00%) Digests (new)
Progress.....: 14344384/14344384 (100.00%)
Rejected.....: 0/14344384 (0.00%)
Restore.Point....: 14344384/14344384 (100.00%)
Restore.Sub.#01..: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#01...: 232242 -> $HEX[042a0337c2a156616d6f732103]
Hardware.Mon.#01.: Temp: 49c Fan: 33% Util: 34% Core:1807MHz Mem:7301MHz Bus:4

```

Figure 5: Dictionary Attack #2

```

Session.....: hashcat
Status.....: Exhausted
Hash.Mode.....: 1410 (sha256($pass.$salt))
Hash.Target.....: 82f2524548f8bf165dc639fa7107eda156074499c749c8cea3e...b33b89
Time.Started.....: Wed Apr 15 08:53:25 2026 (1 sec)
Time.Estimated...: Wed Apr 15 08:53:26 2026 (0 secs)
Kernel.Feature...: Pure Kernel (password length 0-256 bytes)
Guess.Base.....: File (rockyou.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#01.....: 16161.5 kH/s (4.82ms) @ Accel:1024 Loops:1 Thr:64 Vec:1
Recovered.....: 0/1 (0.00%) Digests (total), 0/1 (0.00%) Digests (new)
Progress.....: 14344384/14344384 (100.00%)
Rejected.....: 0/14344384 (0.00%)
Restore.Point....: 14344384/14344384 (100.00%)
Restore.Sub.#01..: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#01...: 232242 -> $HEX[042a0337c2a156616d6f732103]
Hardware.Mon.#01.: Temp: 48c Fan: 33% Util: 31% Core:1807MHz Mem:7301MHz Bus:4

```

Mask Attack Results:

- The mask attack results came back inconclusive. The downside of using this form is that it requires a lot of processing power. As the attack was being performed, the estimated time for the attack kept increasing to an absurd amount. Our systems were not capable of performing this attack due to lack of resources. Details for the first test are provided in Figure 6.

Figure 6: Mask Attack

```
Session.....: hashcat
Status.....: Running
Hash.Mode.....: 1420 (sha256($salt.$pass))
Hash.Target.....: f4c062267ae84f902e3674d125384ce56d1d7e07e470ed9d4ae...D0F671
Time.Started.....: Wed Apr 15 11:06:16 2026 (30 mins, 28 secs)
Time.Estimated...: Wed Apr 15 15:14:11 2026 (3 hours, 37 mins)
Kernel.Feature...: Pure Kernel (password length 0-256 bytes)
Guess.Mask.....: ?l?l?l?l?l?l?l [8]
Guess.Queue.....: 1/1 (100.00%)
Speed.#01.....: 12921.9 kH/s (14.41ms) @ Accel:3 Loops:64 Thr:256 Vec:1
Recovered.....: 0/1 (0.00%) Digests (total), 0/1 (0.00%) Digests (new)
Progress.....: 40225800192/208827064576 (19.26%)
Rejected.....: 0/40225800192 (0.00%)
Restore.Point....: 2285568/11881376 (19.24%)
Restore.Sub.#01..: Salt:0 Amplifier:8832-8896 Iteration:0-64
Candidate.Engine.: Device Generator
Candidates.#01...: waumaric -> bueuecam
```

5. Implementation Notes (O2)

5.1 Repository Structure

The project is organized as a single-file application with supporting SQL schema files:

- **strongpass.py** — Main application file (~350 lines). Contains all UI classes (StrongPassApp, LoginFrame, VaultWindow, EntryModal, BrandLogo), all security functions (hashing, encryption, breach check), and all database operations.
- **export.key** — Fernet encryption key file. Auto-generated on first run if not present. Must be protected by the user.
- **masterusers_table.sql** — SQL DDL for the MasterUsers table.
- **passwords_table.sql** — SQL DDL for the Passwords table.
- **auditlogs_table.sql** — SQL DDL for the AuditLogs table.
- **README.md** — Project overview, setup instructions, dependency list, and security features summary.

5.2 Coding Conventions

- Python 3.13 with PEP 8 style guidelines.
- CustomTkinter for all UI components, with `ctk.set_appearance_mode("dark")` for consistent dark-mode rendering.
- Constants (COLORS, FONTS, DB_CONN_STR) defined at module level for centralized configuration and easy theming.

- Database connections are opened per-operation and closed immediately after use to prevent resource leaks.
- All database queries use parameterized statements (cursor.execute with ? placeholders) to prevent SQL injection.
- Inline comments throughout the codebase explain design decisions and security rationale.

5.3 Notable Design Decisions and Tradeoffs

Single-file architecture: The entire application resides in a single Python file (strongpass.py). This simplifies distribution and deployment — users only need to run one file — but limits modularity and testability. For a production release, the codebase should be refactored into separate modules for security, UI, and data access.

SQL Server Express vs. SQLite: The team chose SQL Server Express for its enterprise-grade features and alignment with the cybersecurity curriculum. However, this introduces a significant deployment burden: users must install SQL Server Express, configure ODBC drivers, and have administrator access. SQLite would have dramatically simplified deployment and enabled cross-platform support, but was deemed less representative of real-world enterprise environments.

SHA-256 vs. PBKDF2/Argon2: The current implementation uses SHA-256 with a random 16-byte salt for master password hashing. While this prevents plaintext storage, industry best practice (NIST SP 800-63B, OWASP) recommends PBKDF2 or Argon2 with a high iteration count (100,000+) to resist brute-force attacks. Upgrading to PBKDF2/Argon2 is a priority for future development.

Fernet for vault encryption: Fernet provides authenticated encryption (AES-128-CBC + HMAC-SHA256), ensuring both confidentiality and integrity. The tradeoff is that the encryption key (export.key) must be stored on disk as a separate file. If this file is compromised, all vault data is exposed. A key-derivation approach tied to the master password would eliminate this single point of failure.

BrandLogo canvas widget: A custom CTKCanvas class draws the lock/shield logo programmatically using basic canvas shapes (arcs, rounded rectangles, ovals), avoiding any dependency on external image assets and keeping the application fully self-contained.

6. Testing & Evaluation (O3)

6.1 Testing Strategy

Testing for StrongPass relied on manual functional testing performed by team members during each sprint. User Story #19 (Automated Test Suite) was planned to deliver a pytest-based framework with unit tests, integration tests, and coverage reporting. However, upon review of the codebase, no automated test files, test functions, or pytest configuration were implemented. The repository contains no `test_*.py` files, no `unittest` imports, and no CI/CD pipeline configuration. All testing documented in this section was conducted manually by team members during development and sprint review sessions.

This is a significant gap. The absence of automated tests means that regression detection depends entirely on manual re-testing, which is error-prone and does not scale. Implementing a comprehensive automated test suite remains a high-priority recommendation for future development.

6.2 Manual Password Generation Testing

The following password generation behaviors were verified through manual testing during sprint reviews:

- Generated passwords were visually confirmed to contain the requested counts of letters, digits, and symbols across multiple generations.
- The fallback logic was tested by entering zero for all character types; the application correctly defaults to adding letters rather than crashing.
- Custom keyword inclusion was verified by entering keywords and confirming they appear within the generated output.
- Only safe, website-compatible special characters (from the defined set `!@#$%^&*()-_+=[]{}|;:,<.>?`) were observed in generated passwords.

6.3 Manual Strength Evaluation Testing

Strength scoring was tested by generating passwords of varying complexity and observing the UI feedback:

- Short, simple passwords (e.g., fewer than 8 characters, single character type) displayed red indicators and low scores.
- Longer passwords with mixed character types (12+ characters with uppercase, lowercase, digits, and symbols) displayed green indicators and high scores.
- The color-coded progress bar was observed to transition correctly between red, yellow, and green based on the `evaluate_strength()` score thresholds (below 0.5, 0.5–0.8, above 0.8).

6.4 Manual Breach Detection Testing

Breach detection was tested manually by generating or entering known-breached and known-safe passwords:

- Known breached passwords (e.g., 'password', '123456') correctly triggered the "WARNING: BREACH DETECTED" status with a red progress bar.
- Randomly generated long passwords consistently returned the "SECURE" status with a green indicator.
- When the internet connection was unavailable, the breach check failed silently (returning False) and the application continued to function without crashing, confirming graceful timeout handling.

6.5 Manual Database and Vault Testing

Database and vault operations were tested manually through the application's UI:

- Creating a new master account and logging in with the correct credentials succeeded; incorrect passwords were rejected with an error dialog.
- The 12-character minimum master password requirement was enforced during registration.
- Saving a credential to the vault, then viewing it in the Vault Manager, confirmed that the entry appeared with the correct application name and username.
- Editing a vault entry and re-opening the vault confirmed that changes persisted correctly.
- Deleting a vault entry removed it from both the Passwords table and the associated AuditLogs records.
- Exporting the vault as encrypted JSON and re-importing it into a fresh session confirmed that all entries were restored without data loss.
- The password visibility toggle in the Entry Modal correctly switched between masked (dots) and plaintext display.

6.6 Known Defects and Testing Gaps

- **No automated test suite:** User Story #19 was planned but not implemented. The repository contains no test files, no pytest framework, no coverage reporting, and no CI/CD pipeline. All quality assurance was performed manually.
- **No regression testing capability:** Without automated tests, there is no mechanism to detect regressions when code is modified. This is a critical gap for ongoing development.

- The application does not validate that the SQL Server Express instance is running before attempting a connection. If the server is unavailable, the user sees a generic error rather than a helpful diagnostic message.
- The strength evaluator does not check for dictionary words or common password patterns (this was planned as User Story #20 — NIST Compliance Checker).
- No session timeout is implemented; once authenticated, the session remains active until manual sign-out or application close.
- Error handling in the breach check uses a bare except clause, which silently catches all exceptions rather than handling specific failure modes.

7. Security, Privacy, Accessibility & Compliance (O5)

7.1 Security Controls Summary

StrongPass implements defense-in-depth with multiple layers of security, as detailed in the poster's implementation section and the presentation's security overview:

- **Master Account Authentication:** Salted SHA-256 hashing with a 16-byte random salt per account prevents plaintext credential storage. Users create a profile with a master password that meets a 12-character minimum requirement.
- **Encryption at Rest:** Fernet (AES-128-CBC + HMAC-SHA256) encrypts all vault passwords before database storage. The application stores no credentials in plaintext at any layer.
- **Privacy-Preserving Breach Detection:** The HaveIBeenPwned breach check uses k-Anonymity, transmitting only a 5-character SHA-1 hash prefix. The API never receives enough information to determine the actual password being checked.
- **No Cloud Dependencies:** All data is stored locally on the user's machine. No user data is transmitted to any cloud service. The application is installed locally for easy, secure access.
- **Parameterized Queries:** All SQL statements use parameterized inputs (? placeholders) to prevent injection attacks.
- **Encrypted Export/Import:** Vault data can be exported as encrypted JSON files and re-imported, with passwords remaining encrypted throughout the process.

7.2 Privacy Considerations

StrongPass is designed with a privacy-first, local-only architecture. All credential data remains on the local machine. The only outbound network request is the breach check, which uses k-Anonymity to ensure that the HaveIBeenPwned service cannot determine which password is being checked. No telemetry, analytics, or usage data is collected or transmitted. Profile-specific vaults ensure that each user's data is isolated from other users on the same machine.

7.3 Accessibility

The application uses a high-contrast dark-mode color scheme providing clear visual hierarchy. Password strength is communicated through both color coding (red for weak, yellow for moderate, green for secure) and text labels (WEAK PASSWORD, MODERATE PASSWORD, SECURE, WARNING: BREACH DETECTED). The UI features large, clearly labeled buttons and input fields with descriptive placeholders. The password visibility toggle allows users to verify entered credentials.

7.4 Compliance Alignment

- **NIST SP 800-63B:** Minimum password length enforcement, no plaintext storage, breach detection integration, use of approved cryptographic algorithms.
- **OWASP Password Storage Cheat Sheet:** Salted hashing for master credentials, authenticated encryption for stored passwords, secure key management.

7.5 Ethical Considerations

StrongPass is intended for educational and personal use. The application promotes responsible password hygiene by generating strong, unique passwords and discouraging password reuse through its vault system. The breach detection feature educates users about the real-world consequences of compromised credentials without storing or transmitting sensitive data. The project is licensed under MIT for open educational use.

8. Deployment & Operations

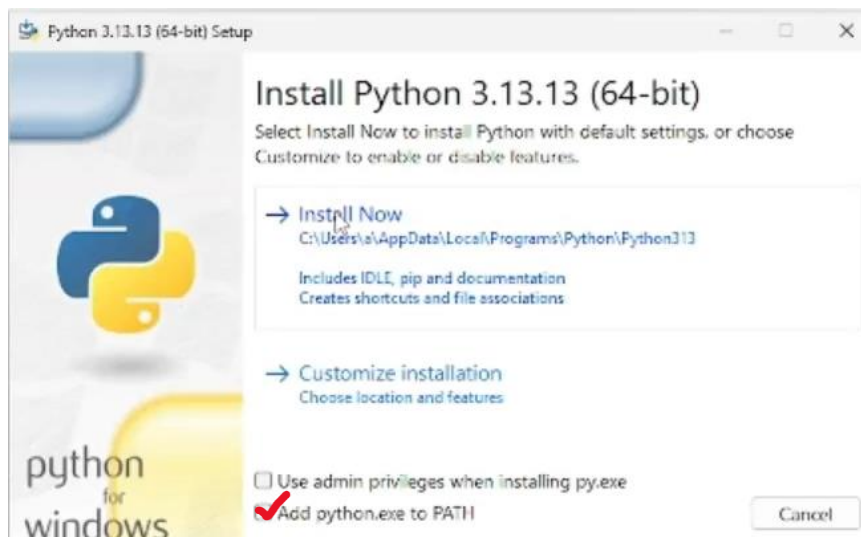
8.1 Prerequisites

- Windows operating system (Windows 10 or later).
- Microsoft SQL Server Express installed and configured with a local instance.
- Microsoft SQL Server Management Studio (SSMS) for database setup.
- ODBC Driver 17 for SQL Server installed.

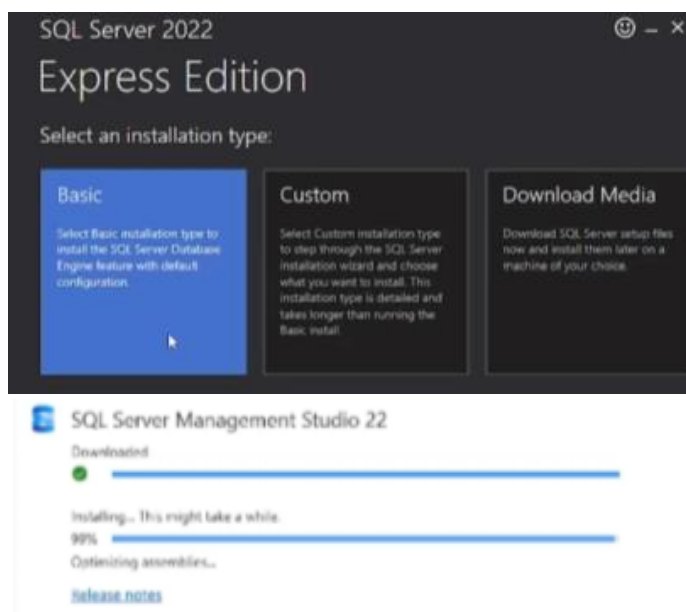
- Python 3.9 or higher (developed and tested on Python 3.13).
- Administrator-level system access (required for SQL Server connectivity in the current version).

8.2 Installation Steps

- **Step 1:** Install Python 3.13 from python.org and ensure it is added to the system PATH.



- **Step 2:** Install Microsoft SQL Server Express and SQL Server Management Studio (SSMS).



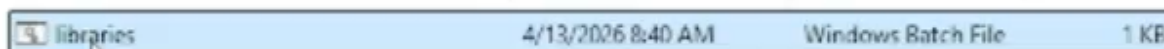
- **Step 3:** Using SSMS, run the StrongPass DB Creation.sql script (this will create the database and all the required tables), connect to SQL express service on PC

(.\ refers to the current PC), Authenticate with Windows Credentials, Optional Encryption, Trust Server Certificate, Connect

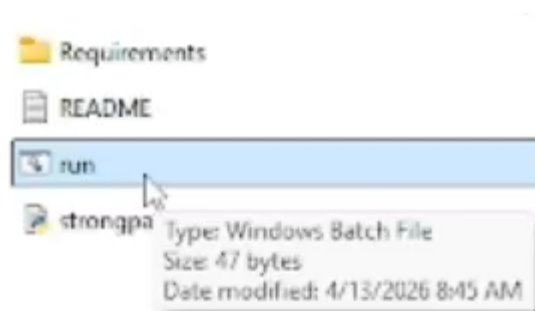
Open the SQL query saved in the requirements folder and execute

```
StrongPass D...KLD8\sa (73) X
1  -- 1. Create the database container
2  CREATE DATABASE StrongPass;
3  GO
4
5  -- 2. Tell the system to use the new database for the following commands
6  USE StrongPass;
7  GO
8
9  -- 3. Create the MasterUsers table
10 CREATE TABLE MasterUsers (
11     UserID INT PRIMARY KEY IDENTITY(1,1),
12     Username NVARCHAR(100) UNIQUE NOT NULL,
13     PasswordHash VARBINARY(MAX) NOT NULL,
14     Salt VARBINARY(MAX) NOT NULL
15 );
16
17 -- 4. Create the Passwords table
18 CREATE TABLE Passwords (
19     PasswordID INT PRIMARY KEY IDENTITY(1,1),
20     UserID INT FOREIGN KEY REFERENCES MasterUsers(UserID),
21     AppName NVARCHAR(100) NOT NULL,
22     Username NVARCHAR(100) NOT NULL,
23     EncryptedPassword VARBINARY(MAX) NOT NULL
24 );
25
26 -- 5. Create the AuditLogs table
27 CREATE TABLE AuditLogs (
28     LogID INT PRIMARY KEY IDENTITY(1,1),
29     PasswordID INT FOREIGN KEY REFERENCES Passwords(PasswordID),
30     UserID INT FOREIGN KEY REFERENCES MasterUsers(UserID),
31     ActionType NVARCHAR(50) NOT NULL,
32     ActionDate DATETIME DEFAULT GETDATE()
33 );
```

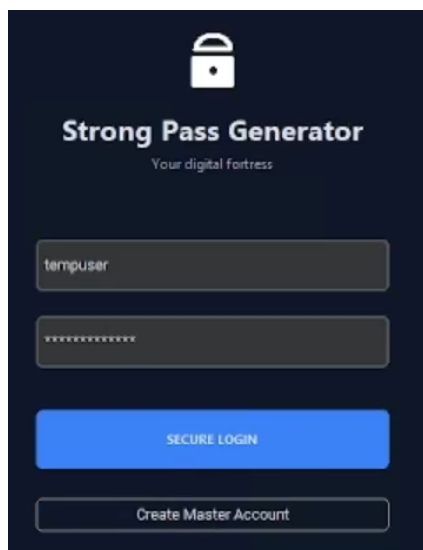
- **Step 4:** Install Python dependencies: pip install customtkinter cryptography requests pyodbc (this is facilitated by that .bat file in the requirements folder)



- **Step 5:** Run the application: py strongpass.py or open run.batch



- **Step 6:** On first launch, use the 'Create Master Account' button to register a master account with a password of at least 12 characters.



8.3 Operational Notes

- The Fernet encryption key (export.key) is auto-generated on first run. Back up this file securely; loss of the key renders all encrypted vault data permanently unrecoverable.
- The SQL Server Express connection string is hardcoded to a local instance (.\\sqlexpress) with Trusted_Connection. Modify DB_CONN_STR in strongpass.py if using a different server configuration.
- The application does not currently support standard (non-administrator) Windows user accounts due to SQL Server permission requirements. This is a known limitation targeted for future resolution.
- Core password generation works without internet connectivity. Breach detection requires an active internet connection to reach the HaveIBeenPwned API.

9. Limitations & Future Work

9.1 Known Limitations/Shortcomings

- **Windows-Only:** The application depends on Microsoft SQL Server Express and pyodbc with ODBC Driver 17, limiting it exclusively to Windows systems. Cross-platform deployment on macOS or Linux is not supported.
- **Administrator Access Required:** SQL Server connectivity currently requires administrator-level permissions. The application cannot run under a standard (non-admin) Windows user account, which is a significant barrier for general consumer adoption.
- **No Multi-Factor Authentication:** Authentication relies solely on the master password. There is no second factor (TOTP, biometric, hardware key).
- **SHA-256 Instead of PBKDF2/Argon2:** The master password hashing uses SHA-256 with a salt, which is computationally fast and therefore less resistant to brute-force attacks than PBKDF2 or Argon2 with high iteration counts.
- **No Automated Backup:** Loss of the export.key file or the SQL Server database results in permanent, unrecoverable data loss.
- **Single-File Architecture:** The entire application resides in one Python file (~350 lines), which limits testability and maintainability at scale.
- **No Session Timeout:** Once authenticated, the session remains active indefinitely until the user signs out or closes the application.
- **No Dictionary Word Detection:** The strength evaluator checks length and character diversity but does not detect common dictionary words or keyboard patterns.

9.2 Unfinished User Stories (Product Backlog)

The following user stories remain in the product backlog and represent the prioritized roadmap for continued development:

- **User Story #9 — Password Expiration:** Allow users to specify a time period for stored passwords, alert when the period expires, and show which passwords will expire soon.
- **User Story #10 — Password Cross-Check:** Compare newly generated passwords against existing vault entries to detect reuse, prompting the user to keep or discard duplicates.
- **User Story #12 — Password Update Reminders:** Notify users when passwords are older than the recommended period (90 days), with options to mark as updated or generate replacements.

- **User Story #14 — Offline Functionality:** Ensure all core features work without internet connectivity by removing API dependencies from the generation and evaluation paths.
- **User Story #19 — Automated Test Suite:** Implement a pytest-based testing framework with unit tests for password generation logic, strength calculation, and breach check (with mocked API responses), integration tests for database CRUD operations, coverage reporting ($\geq 90\%$ target), and CI/CD pipeline configuration via GitHub Actions. This story was planned during Sprints 4–6 but was not implemented; the repository currently contains no automated tests.
- **User Story #20 — NIST Password Compliance Checker:** Validate passwords against NIST SP 800-63B requirements including minimum length, dictionary word detection, sequential/keyboard pattern detection, and display a compliance score with specific violations.

9.3 Value Proposition and Future Development Roadmap/Wishlist

As articulated in the project presentation, StrongPass delivers value through four pillars: automated security (generating complex passwords with a higher degree of security than users achieve on their own), elimination of memorization burden (the vault handles storage so users don't need to remember complex passwords), platform freedom (runs anywhere Python is installed with no browser, ecosystem, or cloud dependency), and a foundation built to grow (password expiration, cross-check, and multi-user features are already in the roadmap).

StrongPass is not a finished product. With appropriate funding and continued development, it has strong potential to evolve into a fully production-ready password management solution. The following priorities are recommended:

- **1. Cross-Platform Database Migration:** Replace SQL Server Express with SQLite or an embedded database to enable deployment on macOS and Linux systems and eliminate the administrator access requirement.
- **2. Standard User Account Support:** Refactor database connectivity to work under non-administrator Windows accounts, removing a critical barrier to consumer adoption.
- **3. Multi-Factor Authentication:** Integrate TOTP-based MFA to provide a second authentication factor, addressing the Spoofing threat identified in the STRIDE analysis.
- **4. Key Derivation Upgrade:** Replace SHA-256 with PBKDF2 or Argon2id for master password hashing with 100,000+ iterations, as recommended by NIST SP 800-63B.

- **5. Master-Password-Derived Encryption:** Derive the Fernet encryption key from the master password rather than storing it as a separate file, eliminating the export.key single point of failure.
- **6. Session Management:** Implement configurable session timeouts (e.g., 15 minutes of inactivity) and screen lock functionality.
- **7. Automated Backup & Recovery:** Scheduled encrypted backups of the vault database with recovery options, addressing the Denial of Service threat.
- **8. NIST Compliance Checker:** Complete User Story #20 to validate passwords against NIST SP 800-63B requirements with dictionary word and pattern detection.

10. References

- [1] National Institute of Standards and Technology. 2017. Digital Identity Guidelines: Authentication and Lifecycle Management. NIST Special Publication 800-63B. U.S. Department of Commerce. <https://csrc.nist.gov/pubs/sp/800-63b/upd2/final>
- [2] OWASP Foundation. 2024. Password Storage Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
- [3] Verizon. 2025. Data Breach Investigations Report. <https://www.verizon.com/business/resources/reports/dbir/>
- [4] SpyCloud. 2025. Identity Exposure Report 2025. <https://spycloud.com/resource/identity-exposure-report/>
- [5] Bitwarden. 2023. World Password Day Survey. <https://bitwarden.com/resources/world-password-day/>
- [6] Hive Systems. 2024. Password Table. <https://www.hivesystems.com/password-table>
- [7] Troy Hunt. 2023. Have I Been Pwned API v3 — k-Anonymity Model. <https://haveibeenpwned.com/API/v3>
- [8] Python Cryptography Authority. 2024. Fernet Symmetric Encryption. <https://cryptography.io/en/latest/fernet/>
- [9] Microsoft. 2024. SQL Server Express Documentation. <https://learn.microsoft.com/en-us/sql/sql-server/>
- [10] Python Software Foundation. 2024. hashlib — Secure Hashes and Message Digests. <https://docs.python.org/3/library/hashlib.html>

[11] CustomTkinter. 2024. Modern and Customizable Python UI Library.
<https://customtkinter.tomschimansky.com/>

[12] Microsoft. 2024. ODBC Driver for SQL Server. <https://learn.microsoft.com/en-us/sql/connect/odbc/>

Appendices

Appendix A: Installation Guide

See Section 8.2 for step-by-step installation instructions with all prerequisites and dependencies.

Appendix B: User Manual

Creating an Account: Launch strongpass.py. On the login screen, enter a username and a master password of at least 12 characters. Click 'Create Master Account.' A success message confirms account creation.

Logging In: Enter your username and master password, then click 'Secure Login.' On success, the main generator screen loads with the StrongPass branding header and Sign Out button.

Generating a Password: On the main screen, set the desired counts for Letters (default 12), Numbers (default 4), and Symbols (default 4). Optionally enter a Custom Keyword. Click 'Generate Secure Password.' The password appears in the display card with a color-coded strength progress bar and breach status text.

Copying a Password: Click directly on the generated password text to copy it to your clipboard. A 'COPIED TO CLIPBOARD' confirmation appears briefly.

Saving to Vault: After generating a password, click 'Save.' In the Credential Management modal, enter the Service/Application Name and Username. The generated password is pre-filled. Use the eye icon to toggle password visibility. Click 'Save Credentials.'

Managing Vault Entries: Click 'Vault' to open the Secured Vault window. Each entry displays the application name and username with Copy, Edit, and Delete action buttons. Use 'Export JSON' to save an encrypted backup and 'Import JSON' to restore from a backup file.

Regenerating in Modal: Within the Credential Management modal, click 'Regenerate Password' to replace the current password with a new randomly generated 16-character password. A confirmation prompt prevents accidental overwrites.

Appendix C: Admin/Operations Guide

Startup Procedure:

Ensure SQL Server is running (this is a service on your computer called SQLEXPRESS)

Launch the Application:

This can be done with “py strongpass.py”, or by clicking the “run.bat” file in the Strong Pass application folder

Verification Process:

Verify that the login screen loads, the database connection works (it will freeze if not), and password generation and vault work properly

Shutdown Procedure:

Close the application window manually or press “CTRL + C” while in the terminal

Incident Priority Levels

- **P1 (Critical):** Application not launching / database failure
- **P2 (High):** Login or encryption malfunction
- **P3 (Medium):** Vault or UI issues
- **P4 (Low):** Minor bugs or display issues

Incident Response Steps

1. Identify the issue (error message or failure point)
2. Verify database connectivity
3. Restart application
4. Check system dependencies (Python, SQL Server)
5. Apply fix or escalate

Backup Procedures

- Detach & Copy SQL Server database (StrongPass)
- Copy Encryption key file (export.key)
- Copy these to a secure location

Restore Procedures

- Attach SQL Server database (StrongPass)
- Add Encryption key file into project folder (export.key)
- Run program and test

For just credential backups:

- Export as .JSON from Vault
- Import as .JSON to Vault to restore

Appendix D: API Reference

- **Breach Check**
 - Integrates with the *Have I Been Pwned* API using k-anonymity
 - Only partial SHA-1 hashes are transmitted for security
- **Password Generation**
 - Generates secure passwords based on user-defined criteria
 - Includes optional breach checking and strength evaluation
- **Authentication**
 - Handles master user registration and login
 - Uses hashed passwords with salt (SHA-256)
- **Vault Management**
 - Stores encrypted credentials
 - Supports create, read, update, delete, import, and export operations
 -

Example

POST /api/v1/password/generate

```
{
  "letters": 12,
  "numbers": 4,
  "symbols": 4,
  "keyword": "Marcell",
  "check_hibp": true
}
```

Returns

```
{
  "password": "kRpMarcell!xQn7@3j#2",
  "strength_score": 0.95,
  "strength_label": "secure",
  "is_pwned": false,
}
```

```
"breach_count": 0  
}
```

Appendix E: Poster, Slides, and Video Links

Poster: [\[Link\]](#)

36"×48" horizontal poster submitted April 13, 2026 (CAP2_Poster_StrongPass.pdf). Includes Problem, Solution, Summary, System Design diagram, Object Design diagram, Current System status, Requirements, Implementation code snippets, and References.

Presentation Slides: [\[Link\]](#)

Introduction presentation (CAP2_StrongPass_IntroPresentation.pdf). Covers The Challenge (problem and statistics), Our Motivation, What Is Our Solution (features and tech stack), What Is Its Value (four value pillars), Software Functionality (5-step user flow), Examples (UI screenshots of login, generator, vault, and credential modal), Code Snippets (strength evaluator and save function), and closing.

Introduction Video: [\[Link\]](#)

Demo Video: [\[Link\]](#)

Appendix F: Scrum Evidence Index

All Scrum ceremonies are documented across seven sprints. Meeting minutes are available as .docx files in the project repository.

Sprint	Planning Meeting	Review Meeting	Retrospective
Sprint 1	01/12/2026	01/25/2026	01/25/2026
Sprint 2	01/26/2026	02/08/2026	02/08/2026
Sprint 3	02/09/2026	02/22/2026	02/22/2026
Sprint 4	03/02/2026	03/15/2026	03/15/2026
Sprint 5	03/16/2026	03/29/2026	03/29/2026
Sprint 6	03/30/2026	04/12/2026	04/12/2026
Sprint 7	04/13/2026	04/26/2026 (scheduled)	04/26/2026 (scheduled)

Backlog Grooming Sessions: Six backlog meeting minutes documented on the following dates: 01/19/2026, 02/02/2026, 02/16/2026, 03/09/2026, 03/23/2026, and 04/06/2026.

Note: Sprint 1 began with five team members. Ajani Cole departed the group during Sprint 1 due to a change in academic discipline, and the team continued with four members for the remainder of the project.

Appendix G: Outcome & Rubric Crosswalk Checklist

The following checklist maps each ACM Student Outcome to its location in this document and the corresponding evidence artifacts:

Outcome	Document Section	External Evidence	Status
O1: Problem Analysis & Requirements	§2 Problem & Requirements	Product backlog, User stories, Sprint planning minutes	☑ Complete
O2: System Design & Implementation	§3 System Overview & Architecture; §5 Implementation Notes	Source code (strongpass.py), Architecture diagrams (poster), README	☑ Complete
O3: Testing & Evaluation	§6 Testing & Evaluation	Manual test descriptions, Defect summary (no automated suite)	☑ Complete
O4: Communication & Collaboration	§1 Executive Summary; §9 Limitations & Future Work	Poster, Presentation slides, Videos, Scrum evidence index	☑ Complete
O5: Professional, Ethical, Legal & Societal	§7 Security, Privacy, Accessibility & Compliance	Encryption controls, Privacy architecture, NIST/OWASP mapping	☑ Complete
O6: Threat Modeling, Risk & Assurance	§4 Discipline-Specific Depth	STRIDE analysis, Risk matrix, Controls & standards mapping	☑ Complete