ENSE 375 – Software Testing and Validation

VaultGuard

Brydon Herauf (200454546)

Ansar Ahmed (200470692)

Gursharan Singh Rehal (200480626)

Table of Contents

List of Figures

1   Introduction

VaultGuard is a command-line password management tool designed to help users securely manage their credentials. The application encrypts and stores user-defined keys in JSON, supports multiple accounts, and includes clipboard functionality for quick access. It was built with an emphasis on security, reliability, and scalability. The application was thoroughly tested with a suite of structure-based and specification-based tests.

2   Design Problem

2.1   Problem Definition

Cyberattacks and credential-based breaches are becoming increasingly common in our digital world. Managing passwords securely is more challenging than ever for individuals and organizations. As a result, people often resort to weak or reused passwords. According to [1], 65% of individuals recycle passwords across different sites. This practice increases the risk of being affected by a breach. Our project addresses these issues by developing a secure and user-friendly password management software. It will allow users to generate, store, organize, and retrieve keys conveniently and safely.

2.2   Design Requirements

2.2.1   Functions

VaultGuard must contain the following functionality:

- Encrypts and stores user keys securely
- Generates strong passwords
- Registers, authenticates, and logs out users
- Stores keys on a per-user basis
- Allows users to view, copy, add, edit, and delete keys
- Organizes keys into user-defined categories

If time permits, the application may also do the following:

- Implements multi-factor authentication
- Tracks and displays logs of user activity

2.2.2   Objectives

VaultGuard must possess the following qualities:

- Secure - The system must use encryption for data in transit (if using an API) and at rest. Unauthorized access to keys must be forbidden.
- User-Friendly - VaultGuard must be intuitive and follow people-centered design principles.

- Scalable - Code shall be written in a way that supports multiple users and growing password lists without requiring major refactoring.

- Maintainable - Code shall be well- structured, and easy to add features like multi-factor authentication.

- Testable - The system shall be built using test-driven development processes. Code must support intuitive and comprehensive testing.

- Reliable - The system must function correctly at all times to the best of our abilities, and gracefully handle errors.

- Efficient - The system shall be light-weight, running quickly with limited resources.

2.2.3 Constraints

VaultGuard must conform to a number of constraints, including:

Economic Factors:

- Must be built using custom software and free libraries so the development cost is $0.

- The system must be usable on systems without a dedicated GPU so users do not have to purchase powerful hardware.

Regulatory Compliance:

- Data at rest shall be encrypted using AES-256.

- Passwords and cryptographic keys shall never be stored in plaintext.

- Access to decrypted credentials must be restricted to authenticated users only.

Reliability:

- System shall have a measured uptime of over 99%.

- Storage of unvalidated or improperly formatted passwords must be prevented.

- Credentials must be decrypted accurately in over 99% of test cases.

Sustainability and Environmental Factors:

- The application must avoid GPU requirements to support energy-efficient use on general-purpose hardware.

Ethics and Societal Impacts:

- User data must not be transmitted to external servers.

- The final product must be open-source and freely available on GitHub.

3   Solution

Our team explored multiple design approaches before settling on our final solution. While the first two approaches provided some of the desired functionality, they had serious flaws in terms of user experience, testability, and compliance with constraints. We improved upon this in solution three, and will be moving forward with development.

3.1   Solution 1

We began our brainstorming with the most basic possible implementation. In this version:

- A JSON file would contain a list of plaintext key-value pairs.
- A single script would provide a command-line interface for adding, deleting, copying, and viewing keys.

This solution follows a monolithic architecture, with no module separation of the user interface, storage logic, and input handling. While this approach would be easy to implement, it lacks critical features, fails on nearly every non-functional requirement, and violates a number of constraints. It is not secure as there is no encryption or authentication. It cannot support multiple users, and the use of a single file makes the application not scalable at all. Most importantly, it is extremely difficult to test:

- Path testing becomes impractical due to the large number of branches and tightly coupled logic.
- Integration testing becomes the only option as the lack of modularity prevents individual unit testing.

Despite its limitations, this version helped us establish a mental model for the application's core functionality, and acted as a good starting point for further discussion.

3.2   Solution 2

After ruling out the monolithic approach, we explored a more modular architecture. This improved solution contains:

- Support for multiple users through registration and login mechanisms.

- Enhanced security by encrypting each key individually before storage.

- Separate modules for managing users, encryption/decryption, key storage, and command-line interaction.

This solution meets most of our functional requirements, and significantly improves on our objectives. The addition of authentication and encryption makes it far more secure. Introducing basic modularity is a major step forward for scalability, maintainability, and testability. The most important improvement is the decoupling of components, which allows for unit testing. However, this solution still has many limitations:

- The command-line interface is not user-friendly and remains tightly coupled with encryption and storage logic.

- Coupling between modules makes path testing difficult and unit testing impractical in some areas.

- It is challenging to fully test all possible command sequences and usage scenarios.

Our second solution brought us much closer to satisfying the requirements, objectives, and constraints. However, it still falls short of our goals for robust testing and clean modular design. As a result, we proceeded with the design of a third and final solution.

3.3   Final Solution

Our final solution improves on our second solution by implementing better separation of concerns. Solution two uses highly coupled classes, where this solution is more modular. The functionality between the two is quite similar, but the reduced coupling will make testing far more practical. It will enable efficient and focussed unit testing, where each method is relatively short. This also allows for less complicated control flow graphs. Modules will communicate through well defined interfaces, enabling proper integration testing. The architecture is as
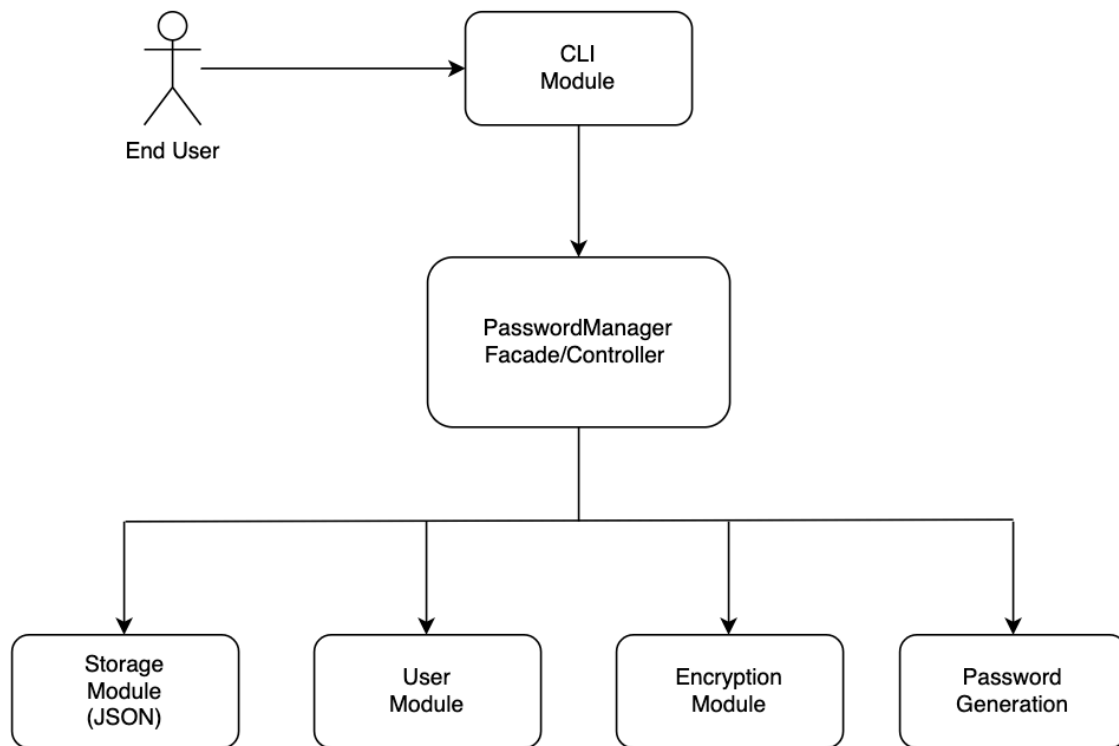
follows:

Figure 1: Application Architecture

The system consists of four lower level modules that abstract away complexities like storage, user validation, encryption, and password generation. These four services are pulled together in the PasswordManager controller, which exposes functions for logging in, registering, adding keys, etc. Finally, the user interacts with the system through a CLI, which is a thin input/output layer that uses PasswordManager as a facade for all features. The lower level modules know nothing of the modules beside or above them in the hierarchy.

3.3.1   Components

StorageService - This class is responsible for loading and saving user vaults as JSON files. This class was tested in a TDD style in StorageServiceTest.

UserService - This class is responsible for validating usernames and passwords, as well as generating salts. This class was tested in a TDD style in UserServiceTest. We also performed integration testing between this class and PasswordManager.

EncryptionService - This class is responsible for deriving encryption keys, encrypting, and decrypting. It was tested in a TDD style in EncryptionServiceTest.

PasswordGenerationService - This class is responsible for generating strong, random passwords. It was tested in a TDD style in PasswordGenerationServiceTest.

PasswordManager - This is the controller that pulls all lower level services into a fully functioning application. It handles login, registration, key retrieval, as well as adding, editing, and deleting keys. This class was tested in a TDD style in PasswordManagerTest. The register method was also verified with path and data flow testing. Integration between this class and the UserService was tested explicitly as well. Most TDD style tests in this class are informal integration tests since they do not utilize stubs for the lower level services.

CLI - Finally, this class is a thin layer that handles user input and output. It has a PasswordManager which abstracts away most of the system. To test the CLI, we used specification-based testing outlined in our testing document.

3.3.2   Environmental, Societal, Safety, and Economic Considerations

Our password manager was designed with low resource usage in mind. It operates fully offline, does not have a graphical interface, and avoids computationally expensive functions. We also thought about user privacy as we built this system . Some individuals may feel uncomfortable using closed-source, Internet-based password managers, where third parties have control over their data. Our solution is open-source, and allows users to run it on their own devices. We used powerful encryption algorithms to protect sensitive information, and tested our application thoroughly. We built this system using free tools so it can be shared openly at no cost.

3.3.3   Limitations

As with all systems, VaultGuard has limitations. Some notable ones are:

- No multi-factor authentication or session timers.

- Basic error handling through generic messages. A user only knows that something failed, not why.
- The CLI is efficient, but provides a less favourable user experience when compared with a GUI.
- We were not able to perform comprehensive structure-based testing on the entire application.
- Does not support user-defined categories as initially planned.

4 Team Work

4.1 Meeting 1

Date: June 5, 2025.

Agenda: Brainstorming of implementation strategies

Team Member	Previous Task	Completion State	Next Task
Brydon	N/A	N/A	N/A
Ansar	N/A	N/A	N/A
Gursharan	N/A	N/A	N/A

4.2 Meeting 2

Date: July 1, 2025.

Agenda: Assignment of lower level services

Team Member	Previous Task	Completion State	Next Task
Brydon	N/A	N/A	Encryption
Ansar	N/A	N/A	Password Generation
Gursharan	N/A	N/A	Users

4.3 Meeting 3

Date: July 8, 2025.

Agenda: Progress update on lower level services

Team Member	Previous Task	Completion State	Next Task
Brydon	Encryption	100%	Storage
Ansar	Password Generation	50%	Password Generation
Gursharan	Users	50%	Users

4.4   Meeting 4

Date: July 16, 2025.

Agenda: Planning the rest of the project

Team Member	Previous Task	Completion State	Next Task
Brydon	Storage	100%	PasswordManager, Structure-based testing
Ansar	Password Generation	100%	Skeleton of CLI, Black-box testing
Gursharan	Users	100%	Black-box testing

5 Project Management

When we began development, we started tracking tasks on a kanban board connected to a gantt chart. We started with the four lower level services, and built those out over two weeks. Then we spent approximately one week on the controller and CLI. Finally, we spent the last week of the project on testing and documentation. View our gantt chart here:

https://github.com/users/Brydon13/projects/6/views/4?sortedBy%5Bdirection%5D=asc&sortedBy%5BcolumnId%5D=193113542

6   Conclusion and Future Work

VaultGuard successfully meets most of the design functions and objectives that we set out to accomplish. The application allows users to register, login, generate passwords, and securely manage keys through a command-line interface. We originally hoped to give users the ability to group keys into categories, but decided to focus on core functionality instead. The system uses powerful encryption based on the user's password and a randomly generated salt. We implemented it using test-driven development, and validated it thoroughly. We satisfied our objectives by building a secure, scalable, reliable, testable, and efficient project, all while conforming to our original constraints.

Despite many successes, VaultGuard still has limitations. If we had more time for development, we would focus on improving user-experience through more detailed failure messages, and the ability to group keys into categories. We would also enhance security with multi-factor authentication and session timers. A graphical user interface would increase resource usage, but make for a more intuitive experience. Finally, further structure-based testing would improve our confidence in the application. Despite its limitations, VaultGuard is a secure, open-source, and useful password management tool with a strong foundation for future development.

7   References

[1] Enzoic, "8 Shocking Stats on Password Reuse," *Enzoic Blog*, May 11, 2022. [Online]. Available:

   https://www.enzoic.com/blog/8-stats-on-password-reuse/