TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Lauri Lunt 251664IVCM

Secure Programming: Secure Password Manager

Project Report

Supervisor: Ali Ghasempour

Tallinn 2025

Table of contents

1 Introduction 3

2 Implementation 4

3 Security Analysis 7

4 Conclusion 10

References 11

1 Introduction

Password management has long been the most important part of information security since password-based authentication was implemented. Since passwords are the primary mechanism for verifying identity across applications, password security may become crucial for protecting sensitive information. Organizations understand the risk of poor password hygiene, and more often, this understanding comes with security breaches that expose vulnerabilities in their current practices. Nowadays, authentication services are mostly outsourced to external providers. At the core, the issue remains the same. The worst scenario is reusing the same password across multiple applications. Although environments may change, the responsibility for secure password management remains the same. The result is the need for secure password management solutions.

The objectives of this project were to implement a secure password manager that ensures the confidentiality, integrity, and safe handling of stored credentials. While the project focuses specifically on creating a password manager, it also serves as a practical exploration of secure programming principles, which we learned through the whole semester in the Secure Programming class. The user interface was secondary. That's why I used only the command-line interface throughout this project.

The relevance of secure programming in this project was focused on secure programming concepts. The project prioritises error handling, input sanitization, key derivation, strong cryptographic algorithms for data encryption, data integrity validation, and secure I/O management, like checked file permissions and file status validation. Instead of trying to replicate production-grade password vaults, the project focuses on key points for how to properly implement essential security mechanisms. On this basis, the project both offers a functional password manager and demonstrates the application of secure programming principles.

The application requirements and how to use it were described in the README.md file in the application directory.

2 Implementation

The application was written in Python and follows a modular software architecture, where the main application logic is divided into logical components. These pieces include modules responsible for encryption, data storage, user interaction (dialogs), utilities, and event handling. The central module is the "*SecurePasswordManager"* class, which performs all cryptographic operations. These operations were encryption [1], decryption [1], key derivation based on passwords [2], as well as password entry management, so-called CRUD operations. The AES-GCM construction is used, consisting of the AES block cipher utilizing Galois Counter Mode (GCM) [1]. The design ensures that encryption is performed using AES-GCM with a 256-bit generated key, an authenticated encryption mode [1]. Every encryption task uses a 12-byte cryptographically secure nonce obtained from the operating system. For enhanced security, each file saved to disk uses a new nonce. By using these two security mechanisms in conjunction with data encryption, confidentiality, as well as integrity and authenticity, the data was ensured because it cannot be altered without detection, nor can any unauthorized party modify the ciphertext without being detected. To further ensure integrity, an integrity protection layer using HMAC-SHA256 from the Python "hmac" module is applied before encryption.

Master passwords were used to derive encryption keys using the PBKDF2-HMAC-SHA256 function with 300,000 iterations and a unique 32-byte salt [2]. This key-derivation process with that many iterations slows down brute-force attempts and increases the difficulty of offline password cracking, such as rainbow-table attacks. The salt was stored in the vault file, while the master password itself is never saved to a vault file or logged in a log file. The project also uses strong memory management practices to help prevent sensitive data from remaining in memory. Before the application exits, encryption keys, salts, stored credentials, and clipboard contents are wiped or overwritten in memory. This cleanup also occurs when the user interrupts the application (e.g., by pressing CTRL+C), as the signal handler triggers secure save and memory-clearing procedures.

Secure input validation and sanitation were essential parts of the application's defensive measures strategy. All user inputs were sanitized by a function that removes control characters, trims whitespace, shortens inputs that exceed the maximum allowed length, and ensures the text remains valid and printable [3]. Passwords used for vault creation must meet length and complexity requirements to reduce the risk of weak master passwords and to force users to think in a secure way.

Similar sanitation was applied to service names and usernames to prevent injection attempts and maintain consistent data formatting. If symbols or control characters were inserted, the output text formatting was messed up, and the characters printed were adversary from what was wanted to print on the screen.

The application also took precautions when interacting with the filesystem. Vault files were accepted only if they had the correct ".spm" extension. Also, the appropriate file permissions (0600) were required to protect the vault, meaning only read and write permissions for the user (not group and other) were set. Symbolic links must not be set to prevent symlink attacks and unauthorized file access. Atomic file operations add another layer of protection by ensuring that no partial or corrupted data is written to the vault, even in the event of unexpected interruptions. In the application, atomic file operations were implemented by creating a secure temporary file on the disk. After successful writing, the application removes the old vault file and moves the temporary file to the original name. A secure temporary filename was constructed from the original file, and random characters were added to the filename to prevent filename guessing and exploitation with a symlink attack. To ensure data integrity and unique encryption output each time the file is written, the entire file must be written to disk.

Logging was implemented with Python's native logging module. All necessary functions were implemented in this library, including log levels and output to the log file or the console. The logging was implemented to write a file to the working directory. The application logs all successful events with the INFO level and all errors to the ERROR level. Errors were displayed using generic messages that avoid revealing internal system details, thereby preventing information leakage. For debugging, DEBUG-level logging is implemented and can be enabled in utils.py.

The command-line interface provides a clear, structured user experience by offering dialogs for viewing, creating, updating, deleting, and searching entries, as well as a random password generator for generating temporary passwords. Passwords are handled with special care and are never printed in plain text on the console. Instead, they are copied to the clipboard, which is automatically cleared when the application terminates. The clipboard module (pyperclip [4]) must be installed in the preparation process and described in the requirements.txt file.

All these decisions reflect secure coding principles focused on reducing potential attack surfaces.

3 Security Analysis

The primary potential vulnerabilities were linked to memory leaks in low-level languages such as C and C++. The application is also unlikely to have memory leaks, since the garbage collector handles them [4]. That means Python handles memory automatically and removes unused objects to free space. Python's memory model cannot guarantee immediate deallocation of sensitive data. To mitigate this limitation, the application replaces sensitive data by clearing the clipboard and removing references to key materials and credentials before it exits. Signal handling further ensures that cleanup occurs even in unexpected termination scenarios. These measures align with best-effort security practices for high-level languages. That's why Python was selected as the programming language for this project.

Because the main source of vulnerabilities is mitigated by using a high-level language, the next main vulnerability in the application was brute-force attacks. While AES-GCM key derivation with PBKDF takes time, the adversary can copy the vault to their own computer and crack its password.

The third source of vulnerability is that an adversary can use phishing to get the password of the vault, or that the user reuses a previously leaked password. That’s why the password manager is important for all of us.

The next potential vulnerability was improper use of the programming language. The student had limited experience with Python and relied on class materials and the Internet to improve their basic skills.

But as mentioned earlier, the programmer followed the official documentation, relied on the course materials, and attempted to mitigate the next vulnerabilities described in the next sections.

The application uses multiple security mechanisms to protect against cryptographic vulnerabilities. The cryptographic architecture secures against unauthorized access using AES-GCM to encrypt all user data and against key disclosures using PBKDF2-generated

keys, which are vault-specific and unique to each vault. It is still secure even if the contents of the vault's file are compromised, because without entering the master password, the contents cannot be read. Integrity verification ensures that if malicious parties attempt to modify the encrypted files at decryption time, it can be detected using HMAC-SHA256 tags.

Input-related vulnerabilities can be overcome with the help of validation and sanitization processes. Since input like service names, usernames, or passwords can be tampered with by an attacker to enter malicious characters into input fields, the sanitization function only accepts printable characters while truncating any potentially malicious ones. The program checks for required fields that are not left blank. These steps effectively counter threats such as injection attacks, malformed input, and excessively long input strings intended to break application logic.

The vulnerabilities related to file handling can be remedied by the strict implementation of file permission rules and checks that vault files are not symbolic links. By these measures, malicious parties cannot utilize filesystem redirecting to access or damage files illegally. For secure atomicity related to files, there would be no incomplete writing of files or damage to vault contents while modifying files.

Error handling is also implemented securely - logs do not reveal stack traces or internal application details, reducing the likelihood that an attacker could extract useful information from application failures.

Static analysis used the SonarQube tool in the Cloud environment [6]. It helped identify and fix potential code vulnerabilities, including security, reliability, and maintainability issues. The most challenging part was maintainability, where methods must be split into smaller pieces to avoid complexity. For junior developers, it's hard to think about how to divide a method into reasonable parts that are not hard to understand for other readers. Because of using Visual Studio Code as an IDE, security issues were not a problem. When the Python project was first opened, VS Code offered Python extensions, such as Pylance and other native Python extensions for Visual Studio Code contributed by Microsoft.

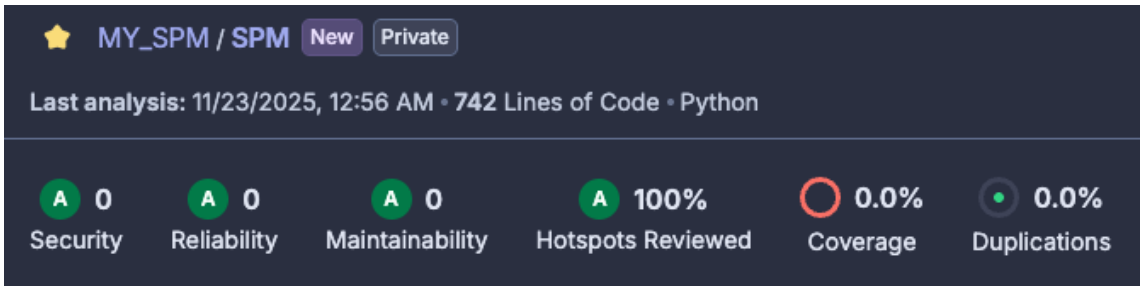

Figure 1. SonarQube result for this project. The code coverage for the test cases is out of scope due to time limitations. [6]

As seen, all topics were graded with an A, and code duplication is 0%. Code duplication indicates that all reusable code is placed in methods.

4 Conclusion

Python was new to the author from the beginning of the project. The author doesn't have much coding experience, except for scripting in bash, which happens almost every day. Coding in low-level languages like C or higher-level languages like Java or similar languages was a bit too challenging at this time.

All required parts were successfully implemented in the password manager project. The most helpful were the course materials and practice classes, where the author learned how to implement a secure cryptographic algorithm in Python. Input sanitization was the most challenging part of the project because Python behaved oddly with some characters, such as arrow keys.

In conclusion, the project and classes taught a mindset for achieving secure programming in real life, from a developer perspective and a tester/adversary perspective. Through the semester, the skills were improved, and new tactics, tricks, and nips were implemented in this project.

References

[1] Individual Contributors, „Welcome to pyca/cryptography,“ 28 November 2025. [Online]. Available: https://cryptography.io/en/latest/hazmat/primitives/aead/#cryptography.hazmat.primitives.ciphers.aead.AESGCM. [Accessed 28 November 2025]

[2] Individual Contributors, „Welcome to pyca/cryptography,“ 28 November 2025. [Online]. Available: https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions/#cryptography.hazmat.primitives.kdf.pbkdf2.PBKDF2HMAC. [Accessed 28 November 2025]

[3] David, „Removing control characters from a string in python,“ 2010. [Online]. Available:https://stackoverflow.com/questions/4324790/removing-control-characters-from-a-string-in-python/19016117#19016117. [Accessed 28 November 2025]

[4] AlSweigart, „A cross-platform clipboard module for Python,“ 26 September 2025. [Online]. Available: https://pypi.org/project/pyperclip/. [Accessed 28 November 2025]

[5] Coursera Staff, „What Is Python Memory Management?,“ 5 Jun 2025. [Online]. Available: https://www.coursera.org/articles/python-memory-management. [Accessed 28 November 2025]

[6] Lauri Lunt, “SonarQube Project: SPM,” 24 November 2025. [Online]. Available: https://sonarcloud.io/summary/overall?id=my-spm_spm&branch=master. [Accessed 28 November 2025]