COMP.SEC.300 Secure programming project
Password manager
Olivia Härkönen

Introduction

This report documents the design, implementation, and testing of a Java-based password manager developed as part of the COMP.SEC.300 Secure Programming course. The goal of the project was to create a secure application while mitigating common vulnerabilities. The report covers how to run the program, its internal structure, implemented security features, testing approach, and future development plans.

How to run

- Java 11+ must be installed
- Maven must be installed. Use Maven 3.6 or later
- Navigate to the root directory of the project (where `pom.xml` is located)
- Run: *"mvn clean javafx:run"*

General description

This Java-based password manager allows users to securely store, manage, and retrieve login credentials. Instead of remembering every password, users only need to remember one strong master password and use an authenticator app to log in with a Time-based One-Time Password (TOTP) for two-factor authentication. The master password is used to log into the application and derive a cryptographic key. The manager stores user passwords encrypted in a local SQLite database, protecting them from unauthorized access. When first registering, the user is shown a QR code with which they can register an authentication app.

After login, users can either view their saved credentials (service, username) or add new credentials. A built-in password generator creates secure, random passwords based on the user-specified length of the password when creating new credentials or editing existing ones. Users can also provide their own passwords, which must meet specific security requirements.

Structure of the program

The password manager application is organized into a modular Java package: fi.tuni.secprog.passwordmanager. Each class is responsible for a specific part of the program's functionality, promoting separation of concerns and maintainability.

```
fi.tuni.secprog.passwordmanager
├── AESUtil.java              // Handles encryption and decryption using AES
├── AESKeyHolder.java         // Holds the in-memory encryption key
├── App.java                  // Entry point, main GUI logic and layout
├── GUIElements.java          // Provides GUI components (buttons, labels, etc.)
├── DatabaseHelper.java       // Provides DB connection and setup
```

```
├── TOTPUtil.java                  // Handles 2FA TOTP key generation and verification
├── UserAuthentication.java        // Manages user registration, login, lockout logic
├── ManageCredentials.java         // Manages storing, updating, deleting site credentials
```

The application includes comprehensive unit and integration tests for core functionalities in following files:

- AESTest.java
- ManageCredentialsTest.java
- TOTPUtilTest.java
- UserAuthenticationTest.java

The application uses a lightweight SQL database with two main tables:

- users: Stores user account information, including usernames, hashed passwords, salt for encryption, failed login attempts, and lockout timestamps.
- credentials: Stores encrypted credentials (site name, username, password) associated with each user.

This structure ensures a secure, modular, and extensible foundation for a personal password manager application.

Secure programming solutions

The application has been developed with OWASP Top 10 and SANS Top 25 in mind. The specific security risks that have been considered are listed in the table 1.

Table 1: Specific application and software security risks taken into account

Code	Name	How the vulnerability was prevented
A01:2021	Broken Access Control	Implemented proper authorization to prevent insecure access via ID.
A02:2021	Cryptographic Failures	Ensured proper use of cryptographic modes and ensured IVs are securely generated and not reused. (Didn't use weak or deprecated cryptographic algorithms.) Stored passwords using strong adaptive and salted hashing functions bcrypt and PBKDF2.
A03:202 & CWE-89 & CWE-20	Injection	Prevented SQL injection by using prepared statements with parameterized queries and ensured input validation where it was needed.
A04:2021	Insecure Design	Implemented security-focused testing across tiers using both unit and integration tests.
A07:2021 & CWE-287& CWE-862	Identification and Authentication Failures	Prevented brute-force attacks using account lockouts, and uniform login responses. Implemented multi-factor authentication (2FA).
A08:2021	Software and Data Integrity Failures	Implemented dependency vulnerability scanning to ensure secure third-party components.

The application encrypts sensitive data before storing it to the SQL database. Credentials which the user stores (passwords and usernames of websites) are encrypted using AES-GCM to ensure

confidentiality and integrity. The master password, which user uses to log in, is hashed with bcrypt and random salt. PBKDF2 key derivation is used to derive the key from user's master password and saved salt.

The application enforces strong password policies, requiring users to set passwords with a minimum length of 8 characters, including upper and lowercase letters, numbers, and special characters. This includes the master password and the saved passwords. The master password security should have been done a little differently, by not forcing the user to fill these requirements, but rather to check passwords against known weak and commonly used password lists [3].

The application doesn't store any hardcoded credentials in the source code. Also input validation is done to make sure that all data received from users is safe and as expected. SQL injection is prevented by using prepared statements with parameterized queries. Brute-force attacks and credential stuffing is also prevented by uniform login and registering responses, 2FA, and by locking account after 5 failed login attempts. This project uses the open-source java-otp library by Jesse Chambers for Time-based One-Time Password (TOTP) generation and validation, using HMAC-SHA1 as specified in RFC 6238.

Testing and QA

This project includes a set of automated unit and integration tests to ensure the security, correctness, and reliability of the password manager application. The tests are written using JUnit 4 and most run against a temporary SQLite database file, which ensures a clean and isolated test environment.

Unit tests were written to validate self-contained components that do not depend on external systems. The primary focus was on cryptographic logic and TOTP management. The tests verify that core cryptographic operations and TOTP key generation and verification work as expected and that data encrypted by the system can be securely and correctly decrypted.

Integration tests validate that multiple components function correctly together, including interactions with the SQLite database, encryption layer, and user management logic. Both of the integration test classes (UserAuthenticationTest.java and ManageCredentials.java) initialize a temporary SQLite database for testing and clean it up after execution, ensuring isolation between tests and preventing side effects. The integration tests verify that the system works as a whole, covering the database schema, cryptographic layer, and application logic.

User Authentication Tests

- Tests login, registration, lockouts, SQL injection resistance.

Credential Management Tests

- Tests website credential management (store, update, delete, retrieve).
- Verifies that generated passwords meet security and complexity requirements.

Cryptographic Utilities:

- Validates encryption/decryption logic and key derivation.
- Tests secure random salt generation and uniqueness.
- Checks the integrity of static key storage and clearing logic.

TOTP Utilities:

- Validates secret key generation for correct format and uniqueness.

- Verifies correct TOTP URL generation with proper issuer, secret, and structure.
- Tests successful and failed TOTP verification, including invalid codes and expired time windows.

In addition to automated tests, manual testing was conducted to evaluate the system's behavior under various input conditions. This included testing with both valid inputs and invalid or edge-case inputs. The goal was to ensure that the application handles typical user interactions gracefully while also defending against unexpected or malicious input. These tests help verify input validation, error handling, and overall robustness of the system.

As part of the security analysis of the project's third-party dependencies, a thorough scan was performed using OWASP Dependency-Check (figure 1). This tool analyses the project's dependencies against known vulnerabilities in public databases like the NVD and CISA. The scan results indicate that the current versions of all dependencies are secure from known vulnerabilities.

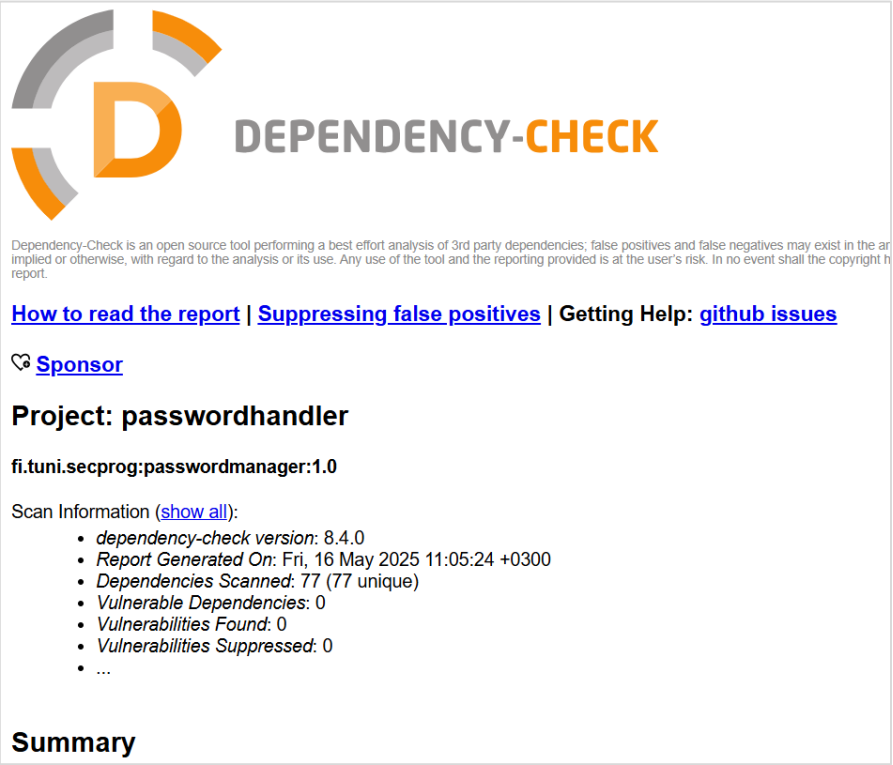

Figure 2: Screenshot of OWASP Dependency-Check scan report

To support consistent code quality and maintainability:

- Separation of concerns was followed, with clearly defined classes handling encryption, authentication, database access, and GUI logic.
- Tests are repeatable and isolated, using temporary databases that are cleaned up after execution.
- Code was tested against invalid and malicious input, such as attempts at SQL injection and weak passwords.
- Secure development guidelines were considered

Only minor adjustments were required to handle null values based on test results. No other changes were necessary. Together, the testing and QA practices contribute significantly to the reliability and

security of the application. While additional test coverage (such as GUI testing or boundary testing for all inputs) could further enhance confidence, the current test suite offers a solid foundation for ongoing development and future improvements.

Ideas for future development

During the project, many ideas for future development emerged. In addition to account locking, implementing audit logging would be beneficial for tracking access and user actions. Another useful feature could be an auto-lock mechanism that locks the application after a set period of inactivity (e.g., X minutes).

As previously mentioned, checking passwords against known blacklists—rather than enforcing complex composition rules like requiring uppercase letters, numbers, and special characters—would be a significant security improvement. This approach is also recommended by OWASP [1][3].

Other, less critical improvements could include master password management, such as allowing users to change or recover forgotten passwords. Integrating an authentication framework or library, such as the OWASP ESAPI Authentication module [1], could also enhance the application's security posture.

Use of AI

ChatGPT-4 was used during the initial phase of the project to brainstorm and refine feature ideas, as well as to check the grammar of this report. GitHub Copilot was utilized to assist in initializing the test database.

Sources

[1] OWASP Foundation, “OWASP Top Ten,” OWASP, 2021. [Online]. Available: https://owasp.org/www-project-top-ten/
[2] SANS Institute, “CWE/SANS Top 25 Most Dangerous Software Errors,” 2021. [Online]. Available: https://www.sans.org/top25-software-errors/
[3] National Institute of Standards and Technology, Digital Identity Guidelines: Authentication and Lifecycle Management, NIST Special Publication 800-63B, Jun. 2017. [Online]. Available: https://pages.nist.gov/800-63-3/sp800-63b.html