Secure Password Manager — Report & User Manual

Project: Multi-User Password Manager

Deliverables provided: - The application source code (password_manager_multiuser.py) — interactive CLI vault. - This comprehensive report describing design, implementation, security analysis, and tests. - A user manual explaining installation, usage and operational guidance.

---

1. Introduction

This project implements a multi-user password manager designed to store user credentials encrypted on disk while enforcing strong security, data isolation, and integrity checks. The goals were:

- Provide per-user vaults stored in a single SQLite database.
- Require a strong master password and enforce minimum password strength.
- Use modern KDFs and authenticated encryption (Argon2id + Fernet) to protect data at rest.
- Protect integrity of stored credential rows with an HMAC derived from the encryption key.
- Defend against common threats: brute-force, tampering, SQL injection, and accidental information leakage.
- Be practical and usable with an interactive CLI and optional clipboard support.

This report explains implementation details, the security rationale, limitations, and recommendations.

2. Implementation

Below is an overview of the implementation and the most important design choices.

2.1 High level architecture

- Single SQLite database (passwords.db) stores two tables:
  - users: per-user metadata including id, username, bcrypt_hash, argon2_salt, check_token, rate-limiting fields and created_at.
  - credentials: per-user encrypted credentials (password stored encrypted), hmac for integrity, service_key and display name, username, and optional url.
- Authentication flow:
  1. User enters username and master_password.
  2. bcrypt verifies password correctness from stored bcrypt_hash.
  3. Argon2id derives a symmetric key from the provided password and the stored argon2_salt.
  4. A check_token encrypted with the derived key is decrypted to ensure correct key derivation.
  5. On success, a Fernet instance is kept in memory for encryption/decryption of the user's credentials.
- Encryption/Integrity:
  - Fernet (AES-CBC + HMAC in the cryptography library) is used for authenticated encryption of each password value.
  - An additional HMAC key is derived from the Fernet key by HKDF and used to compute a row HMAC covering service_key, username, encrypted_password and url to detect tampering at row level.
- Rate limiting / lockout: per-user counters (failed_attempts, lockout_until) are stored and enforced to mitigate brute-force attempts.

2.2 Key derivation and storage

- **Argon2id** (via cryptography.hazmat.primitives.kdf.argon2.Argon2id) derives a KDF_LENGTH-byte key from the master password and per-user argon2_salt.
- bcrypt is used as a password verifier (bcrypt hash stored in users.bcrypt_hash) so that the system can quickly reject wrong passwords while still deriving the encryption key from Argon2id.
- The derived Argon2 key is URL-safe base64 encoded and passed to Fernet.

- The HMAC key is derived from the Fernet key using HKDF(SHA-256) with a static info label.

Rationale: Argon2id resists GPU/ASIC password cracking and is appropriate for deriving an encryption key. bcrypt provides compatibility and fast password checking without having to derive the encryption key for every authentication attempt (but we still derive in this implementation to verify check_token).

2.3 Database schema

- users(id TEXT PRIMARY KEY, username UNIQUE, bcrypt_hash BLOB, argon2_salt BLOB, check_token BLOB, failed_attempts INTEGER, lockout_until TEXT, created_at TEXT)
- credentials(id INTEGER PRIMARY KEY AUTOINCREMENT, user_id TEXT, service_key TEXT, service_display TEXT, username TEXT, password BLOB, url TEXT, hmac BLOB, FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE)

Prepared statements and parameterized queries are used throughout to prevent SQL injection.

2.4 Input validation

- Regex-based validation is applied to service names, usernames, credential usernames, URLs (optional), and passwords.
- Master passwords must meet a minimum complexity (12 chars, upper, lower, digit, special) to encourage strong keys.

2.5 Clipboard handling

- Optional pyperclip support — if installed, the vault can copy passwords to the clipboard for a short interval (default 30s) and then clears it in a background thread.

2.6 Logging

- JSON-formatted logs are written to vault.log with restricted permissions.
- Logs intentionally avoid storing raw secrets. For sensitive events (auth failures, registration), stack traces and raw error messages are sanitized.

2.7 Error handling & memory hygiene

- The code attempts to zero out sensitive derived key bytearrays when reasonable (bytearray overwrite and del) and deletes temporaries where practical.

- Exceptions are handled to avoid crashing and leaking information; failures are logged in sanitized form

3. Security Analysis

This section explains threats considered, mitigations implemented, and remaining risks.

3.1 Threat model (summary)

- **Local attacker with disk access** — attacker can read passwords.db and attempt offline cracking.
- **Remote attacker** — may attempt online brute force against the CLI (less likely because CLI is local), or attempt to trick users into revealing passwords.
- **Tampering** — an attacker with write access to the DB tries to alter stored password blobs or metadata.
- **Insider/other user** — ensure per-user isolation so that one user cannot read another's data.

3.2 Implemented mitigations

- **Strong encryption:** Per-user encryption with Argon2id-derived keys and Fernet ensures confidentiality of stored passwords at rest.
- **HMAC integrity:** Each credential row contains an HMAC computed with a key derived from the user's encryption key; tampering with password blobs or associated metadata is detected before decryption.
- **Rate limiting & lockout:** Failed attempt counters and lockout windows reduce online brute force feasibility.
- **Input validation & parameterized queries:** Regex validation and always using ? parameters prevents SQL injection and reduces accidental malformed input.
- **Least privilege filesystem permissions:** The program attempts to set chmod 600 on DB and log files to reduce accidental exposure.
- **No plaintext logging:** Logs avoid recording secrets, and when logging errors for security events, sensitive details are removed.
- **Per-user isolation:** user_id is enforced in credential queries so users can only operate on their own data.

3.3 Residual risks and limitations

1. **Argon2 parameters** — chosen parameters (memory and iterations) should match the deployment platform. Current values are a sensible default for desktop but must be increased for servers with high CPU power budgets or decreased for low-memory environments. Use a configuration mechanism to tune Argon2 parameters.

2. **Fernet/cryptography library constraints** — the project uses the cryptography library Fernet class which provides authenticated encryption. Ensure the library is kept up to date. If long-term key rotation, backup, or sharing is required, additional design is needed.

3. **In-memory secrets** — Python cannot guarantee complete zeroing of memory (due to copies, GC, interpreter internals). The code overwrites bytearray derived keys when practical, but full in-memory secrecy cannot be assured in Python.

4. **Database backups** — if passwords.db is backed up (cloud, snapshots), the encrypted data and salts are copied. Proper backup encryption and access controls are necessary.

5. **Tamper of code or runtime** — if an attacker can modify the running binary or Python interpreter, they can capture secrets at runtime. Use platform hardening and signed distributions for high assurance.

6. **Clipboard exposure** — copying passwords to clipboard is a convenience but creates risk (other apps or clipboard history may leak secrets). The program clears the clipboard after a short interval, but absolute safety is not guaranteed.

7. **Account enumeration** — some operations must avoid exposing whether a username exists. The code attempts to be constant time for certain flows, but perfect prevention of enumeration is hard in a local CLI environment

4. Conclusion

This project demonstrates practical secure programming techniques applied to a user-level password manager: strong KDFs, authenticated encryption, integrity checks, input validation, and rate limiting.