Secure Instant Point-to-Point Password Manager with TLS-Encrypted P2P Sharing

Final Project Report for Computer Security Course

Group Members:

Slade Robert

Nemo Najafian

Submitted on May 4, 2025

Table of Contents

1　Introduction

The Secure Instant Point-to-Point Password Manager is a decentralized application designed to address the critical need for secure password storage and sharing in today's digital landscape. Developed as a final project for our computer security course, the system enables users, referred to as Alice and Bob, to store encrypted password entries locally and share them securely over the Internet via peer-to-peer (P2P) communication. Built with Python 3.11, the application leverages AES-256-GCM encryption, PBKDF2-HMAC-SHA256 key derivation, and TLS-encrypted P2P communication to ensure confidentiality, integrity, and authenticity. A terminal-based graphical user interface (TUI), implemented with the Textual library, provides an intuitive platform for managing an encrypted vault and initiating P2P transfers. This project aligns with the course's requirements for a secure P2P messaging tool by supporting encrypted message exchange, robust key management, and a user-friendly interface. This report details the system's architecture, design, implementation, testing, security analysis, and compliance with project specifications, while exploring potential enhancements and extra credit opportunities.

2　System Architecture

The password manager is structured modularly to ensure maintainability and scalability. The project structure, as outlined in the `README (1).md`, includes:

- `crypto/`: Contains `crypto_utils.py`, implementing AES-256-GCM encryption and PBKDF2-HMAC-SHA256 key derivation.

- `p2p/`: Includes `p2p.py`, handling TLS-encrypted P2P communication and certificate generation.

- `ui/`: Contains `main_ui.py` and `style.tcss`, defining the Textual TUI and its styling.

- `data/`: Stores `vault.json`, the encrypted password vault.

- `main.py`: The application entry point.

- `Dockerfile` and `requirements.txt`: Support containerized deployment and dependency management.

The system operates as a standalone, decentralized application, eliminating reliance on external servers. Data flows from user input in the TUI to encryption for vault storage or TLS-encrypted P2P sharing. This modular design enhances code reusability and simplifies testing.

3 System Design and Implementation

3.1 Encryption and Key Derivation

The system's security relies on AES-256-GCM encryption, implemented in `crypto_utils.py`, using a 256-bit key derived from a master password. Key derivation employs PBKDF2-HMAC-SHA256 with a 16-byte random salt and 100,000 iterations, mitigating brute-force attacks. A 12-byte random nonce ensures unique ciphertexts for identical plaintexts. AES-GCM, a stream cipher mode, requires no padding and provides authenticated encryption for data integrity.

Listing 1: Key Derivation in crypto$_u$*tils.py*

```python
def derive_key(password: str, salt: bytes) -> bytes:
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100_000,
    )
    return kdf.derive(password.encode())
```

Encrypted entries are stored in `vault.json` as JSON objects with base64-encoded salt, nonce, and ciphertext.

3.2 P2P Communication with TLS

The P2P sharing functionality, in `p2p.py`, enables secure entry exchange using Python's `socket` library with TLS encryption via the `ssl` module. Self-signed RSA-2048 certificates are generated if absent, stored as `cert.pem` and `key.pem`. The `share_password` function sends a JSON payload, while `receive_password` listens for connections.
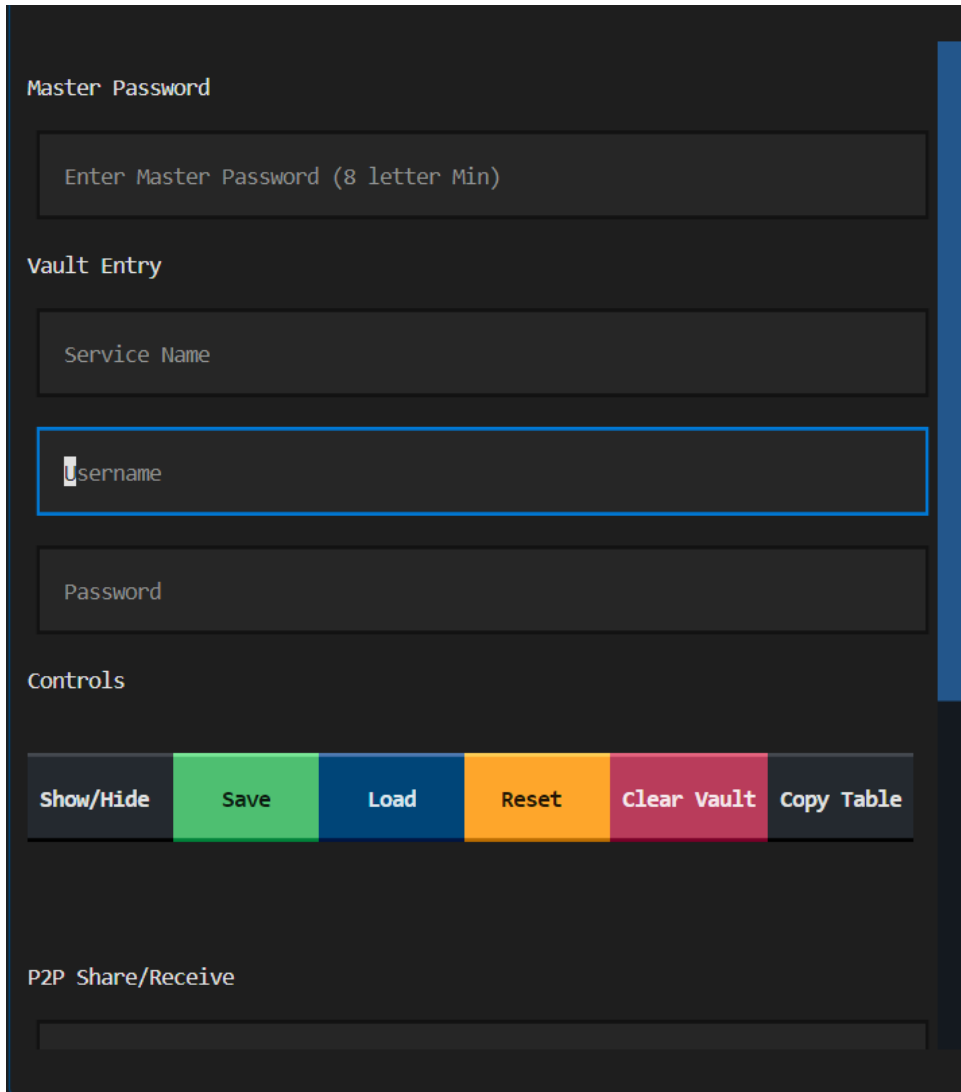
Listing 2: P2P Sharing in p2p.py

```python
def share_password(data: str, host="127.0.0.1", port=65432):
    create_self_signed_cert()
    context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
    context.check_hostname = False
    context.verify_mode = ssl.CERT_NONE
    with socket.create_connection((host, port)) as sock:
        with context.wrap_socket(sock, server_hostname=host) as ssock:
            ssock.sendall(data.encode())
```

TLS uses ciphers with $\geq$128-bit keys, and fresh session keys per connection enhance security.

3.3 User Interface

The TUI, implemented in `main_ui.py` using Textual, features a split-screen layout. The left panel includes input fields for master password, service name, username, password, peer IP, and port, with buttons for Save, Load, Share, Receive, and Clear Vault. The right panel displays a table of decrypted entries, with clickable rows for clipboard copying. A status widget provides feedback, styled via `style.tcss`.

figureTUI showing input fields, buttons, and vault table.

3.4 Key Management and Security Features

The master password-derived key is static, but P2P sharing uses TLS with fresh session keys. Random nonces in AES-GCM ensure unique ciphertexts. Security features include:

- Input validation (e.g., 8-character minimum master password).

- Error handling for decryption failures.

- Clipboard integration via `pyperclip`.

- Confirmation prompts for sensitive actions.

4   Testing and Validation

The system was tested in local Python and Docker environments to validate functionality, usability, and security. Test scenarios included:

- **Vault Operations**: Saved entries (e.g., "Google", "john.doe@gmail.com", "pass123"; "GitHub", "johndoe", "git456") with a master password. Loading decrypted all entries correctly; incorrect passwords failed, confirming key derivation security.

- **P2P Sharing**: Shared an entry between instances on `127.0.0.1:65432`. The receiver's TUI auto-filled fields, and the status widget confirmed success. Invalid ports triggered errors.

- **Error Handling**: Tested empty fields, short passwords, and corrupted `vault.json`, yielding appropriate TUI error messages.

- **Clipboard**: Clicking table rows copied passwords, verified by pasting.

- **Docker**: Ran with a mounted `vault` directory, ensuring data persistence.

  Testing confirmed robust performance and no data leaks during P2P transfers.

5   Security Analysis

The system's security relies on:

- **AES-256-GCM**: Ensures confidentiality and integrity with a 256-bit key, resistant to cryptographic attacks.

- **PBKDF2-HMAC-SHA256**: Resists brute-force and rainbow table attacks with 100,000 iterations and random salt.
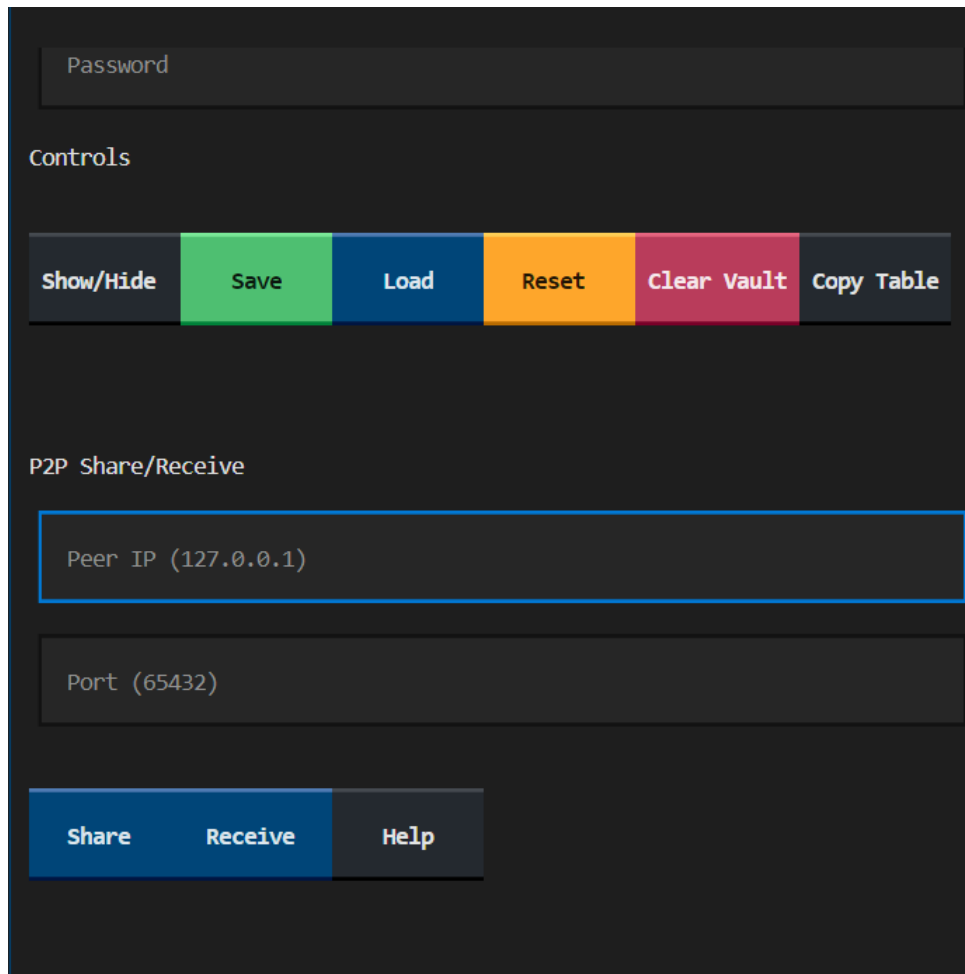
figureVault table with decrypted entries.

- **TLS**: Secures P2P communication, but disabled certificate verification risks man-in-the-middle attacks.

  Vulnerabilities include:

- **Static Master Key**: No rotation increases risk if compromised.

- **TLS Verification**: Lack of verification weakens P2P security.

- **Local Storage**: `vault.json` is encrypted but vulnerable without file system encryption.

  Mitigations involve key rotation, enabling TLS verification, and secure file storage.

6   Alignment with Project Requirements

The system meets requirements as follows:

- **P2P Messaging**: `share_password` and `receive_password` enable secure exchange.

- **Shared Password**: Master password secures vault via PBKDF2.

- **Encryption**: AES-256-GCM and TLS exceed 56-bit key requirement.

- **Key Derivation**: PBKDF2-HMAC-SHA256 ensures security.

- **Padding**: AES-GCM and TLS eliminate padding needs.

- **GUI**: Textual TUI is user-friendly, with potential for ciphertext display.

- **Connection Handling**: TLS-wrapped sockets support P2P.

- **Non-Repeating Ciphertext**: Random nonces ensure uniqueness.

- **Key Management**: TLS session keys provide P2P key rotation.

7   Extra Credit Considerations

The extra credit tasks were explored:

- **Custom Encryption Algorithm (5 pts)**: Combine AES-GCM and ChaCha20, XORing outputs to obscure the process. This increases security against algorithm-specific attacks but doubles computational cost. Implementation would modify `crypto_utils.py` to integrate ChaCha20 and XOR operations.

- **No Pre-Shared Password (5 pts)**: Implement Diffie-Hellman key exchange in `p2p.py` to negotiate a session key, authenticated via certificates. This supports insecure channels, requiring TUI updates for key exchange status.

  These were not implemented but are feasible with additional development.

8   Future Improvements

Proposed enhancements include:

- **Key Rotation**: Prompt master password updates every 90 days, re-encrypting the vault.

- **Ciphertext Display**: Show ciphertext in the TUI for shared entries.

- **TLS Verification**: Enable $\texttt{ssl.CERT}_{R}EQUIRED with a trust store. \textbf{Diffie-Hellman}: Support non-pre-shared password scenarios.$

- **LAN Discovery**: Use `zeroconf` for peer detection.

- **Encrypted Backups**: Export vault as an encrypted archive.

- **Auto-Lock**: Clear TUI after inactivity.

9   Conclusion

The Secure Instant Point-to-Point Password Manager provides a robust, decentralized solution for password storage and sharing, using AES-256-GCM, PBKDF2, and TLS. The Textual TUI ensures usability, and P2P sharing meets secure messaging requirements. Testing validated functionality, and security analysis highlighted strengths and addressable limitations. Extra credit explorations and future improvements, like key rotation and TLS verification, enhance the system's potential, demonstrating cryptographic applications in real-world security.

References

[1] Python Cryptography, "Cryptography Library," https://cryptography.io, 2025.

[2] Python Software Foundation, "SSL: TLS/SSL Wrapper for Socket Objects," https://docs.python.org/3/library/ssl.html, 2025.

[3] Docker Inc., "Docker: Accelerated Container Application Development," https://www.docker.com, 2025.