**Course**: CSCI 6221 - Advanced Software Paradigms

George Washington University

**Authors:**  
Owen Chibanda  
Jasmine Sciarillo  
Aryan Bhosale

|  |  |
| --- | --- |
| A picture of a winding road and trees  **Secure Password Vault**  A Rust-Based Local Password Manager with Military-Grade Encryption | Abstract  This document provides comprehensive technical documentation for the Secure Password Vault, a locally-managed password storage system built using the Rust programming language. The system implements industry-standard cryptographic primitives including AES-256-GCM for authenticated encryption and Argon2id for password-based key derivation. Unlike cloud-based password managers, this solution prioritizes user privacy and data sovereignty by storing all credentials locally on the user's device. The documentation includes detailed architectural diagrams, security analysis, implementation details, and operational guidelines. Six comprehensive diagrams illustrate the system architecture, module dependencies, data flows, cryptographic operations, file structure, and state management. This documentation is intended for developers, security auditors, system administrators, and end users requiring deep understanding of the system's capabilities and security guarantees. |

Contents

[1. Executive Summary 2](#_Toc212389197)

[1.1 Project Overview 2](#_Toc212389198)

[1.2 Key Features and Capabilities 2](#_Toc212389199)

[2. System Architecture and Design 3](#_Toc212389200)

[2.1 Layered Architecture Overview 3](#_Toc212389201)

[2.2 Module Dependencies 3](#_Toc212389202)

[2.3 Operation Data Flow 4](#_Toc212389203)

[3. Cryptographic Implementation 6](#_Toc212389204)

[3.1 Cryptographic Operations Flow 6](#_Toc212389205)

[3.2 AES-256-GCM Specifications 7](#_Toc212389206)

[4. Vault File Structure 8](#_Toc212389207)

[4.1 JSON File Format 8](#_Toc212389208)

[5. Vault Lifecycle Management 9](#_Toc212389209)

[5.1 State Transitions 9](#_Toc212389210)

[6. Security Best Practices 10](#_Toc212389211)

[6.1 Master Password Guidelines 10](#_Toc212389212)

[6.2 Threat Model Summary 10](#_Toc212389213)

[7. Conclusion 11](#_Toc212389214)

# 1. Executive Summary

## 1.1 Project Overview

Secure Password Vault is a command-line password manager that provides enterprise-grade security for personal credential storage. Built with Rust for memory safety and performance, it addresses the growing concern about cloud-based password storage by giving users complete control over their sensitive data.

## 1.2 Key Features and Capabilities

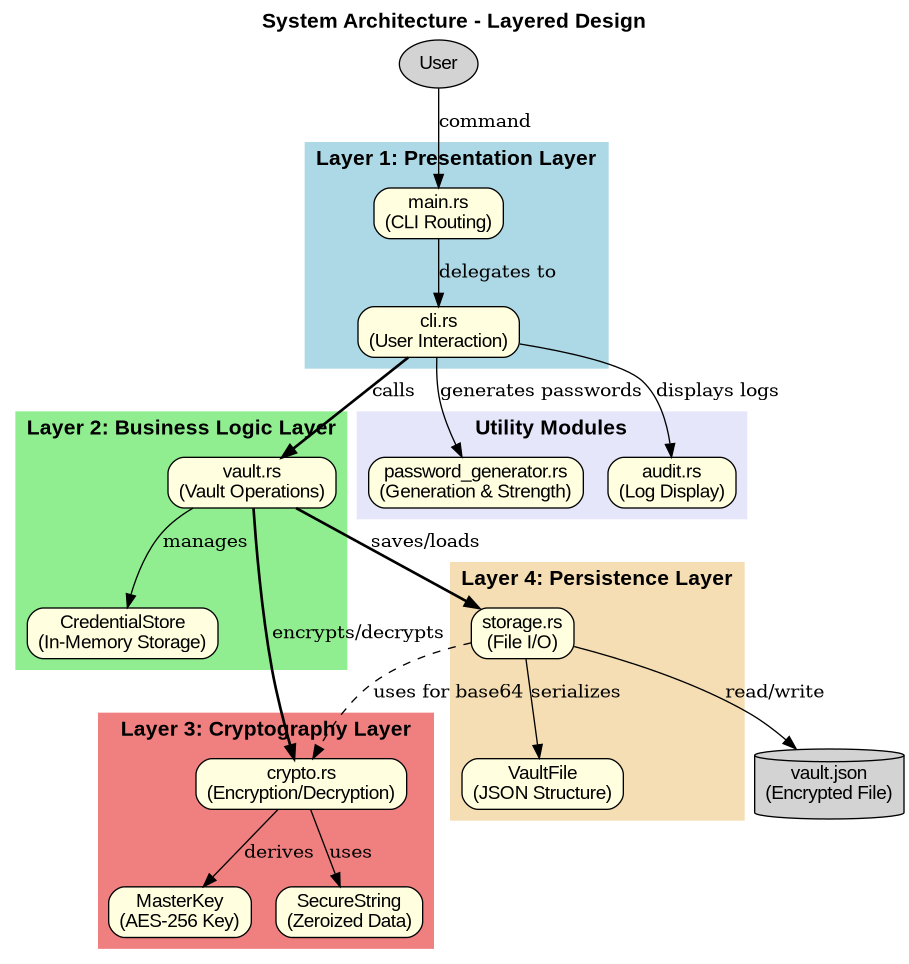
* **Military-Grade Encryption:** AES-256-GCM provides authenticated encryption with 256-bit keys
* **Memory-Hard Key Derivation:** Argon2id resists GPU and ASIC attacks
* **Zero-Knowledge Architecture:** Master password never stored on disk
* **Memory Safety:** Rust prevents buffer overflows and use-after-free
* **Comprehensive Audit Logging:** All operations tracked with timestamps
* **Cross-Platform Support:** Works on Windows, macOS, and Linux

# 2. System Architecture and Design

## 2.1 Layered Architecture Overview

The application follows a strict layered architecture with clear separation of concerns. Figure 1 illustrates the complete system architecture showing all modules, their relationships, and data flow paths.

Figure 1: System Architecture - Layered Design



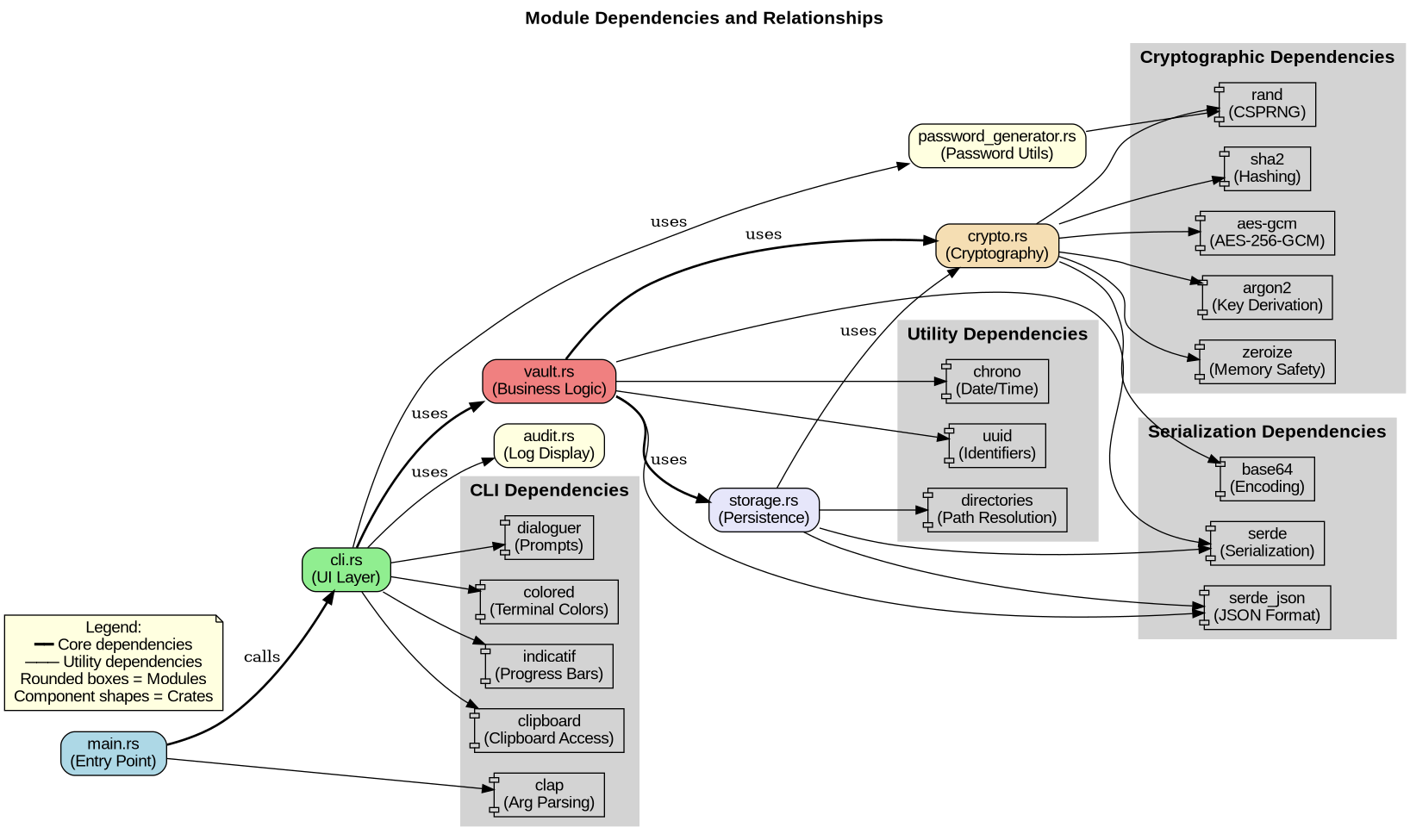
The architecture consists of four primary layers:

* **Layer 1 - Presentation:** CLI interface, argument parsing, user prompts
* **Layer 2 - Business Logic:** Vault operations, credential management
* **Layer 3 - Cryptography:** Encryption, key derivation, secure memory
* **Layer 4 - Persistence:** File I/O, serialization, atomic writes

## 2.2 Module Dependencies

Figure 2 shows the complete dependency graph including all internal modules and external crates. This visualization helps understand the project structure and how components interact.

Figure 2: Module Dependencies and Relationships



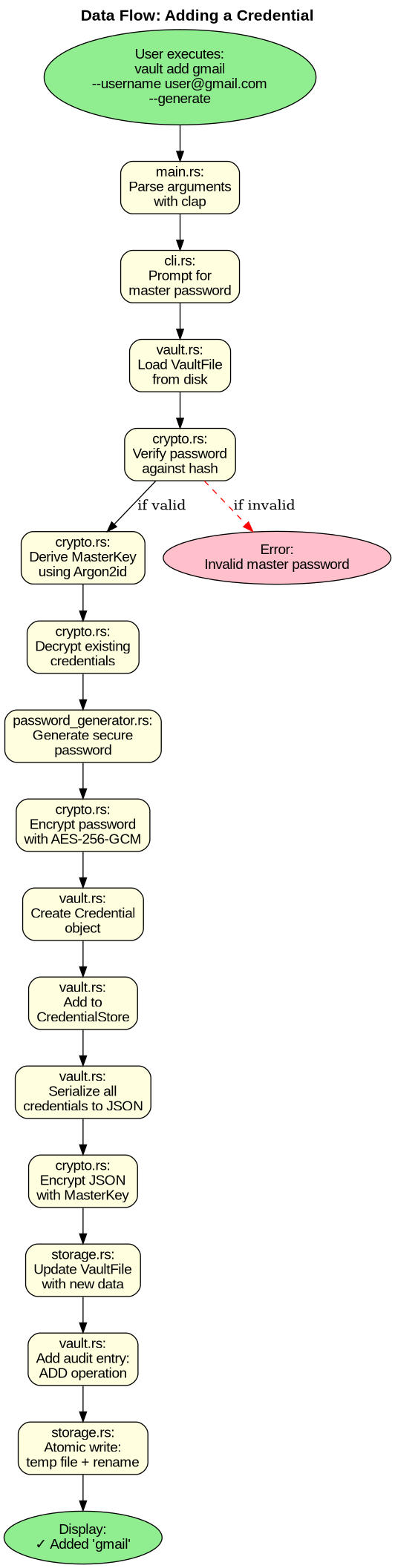
The diagram illustrates:

* Core module relationships (bold lines)
* External crate dependencies grouped by category
* Cryptographic dependencies (aes-gcm, argon2, rand, zeroize)
* CLI dependencies (clap, dialoguer, colored)
* Utility dependencies (serde, chrono, uuid)

## 2.3 Operation Data Flow

Understanding the data flow through the system is crucial for security analysis and debugging. Figure 3 illustrates the complete sequence of operations when adding a new credential.

Figure 3: Data Flow - Adding a Credential



The flow demonstrates:

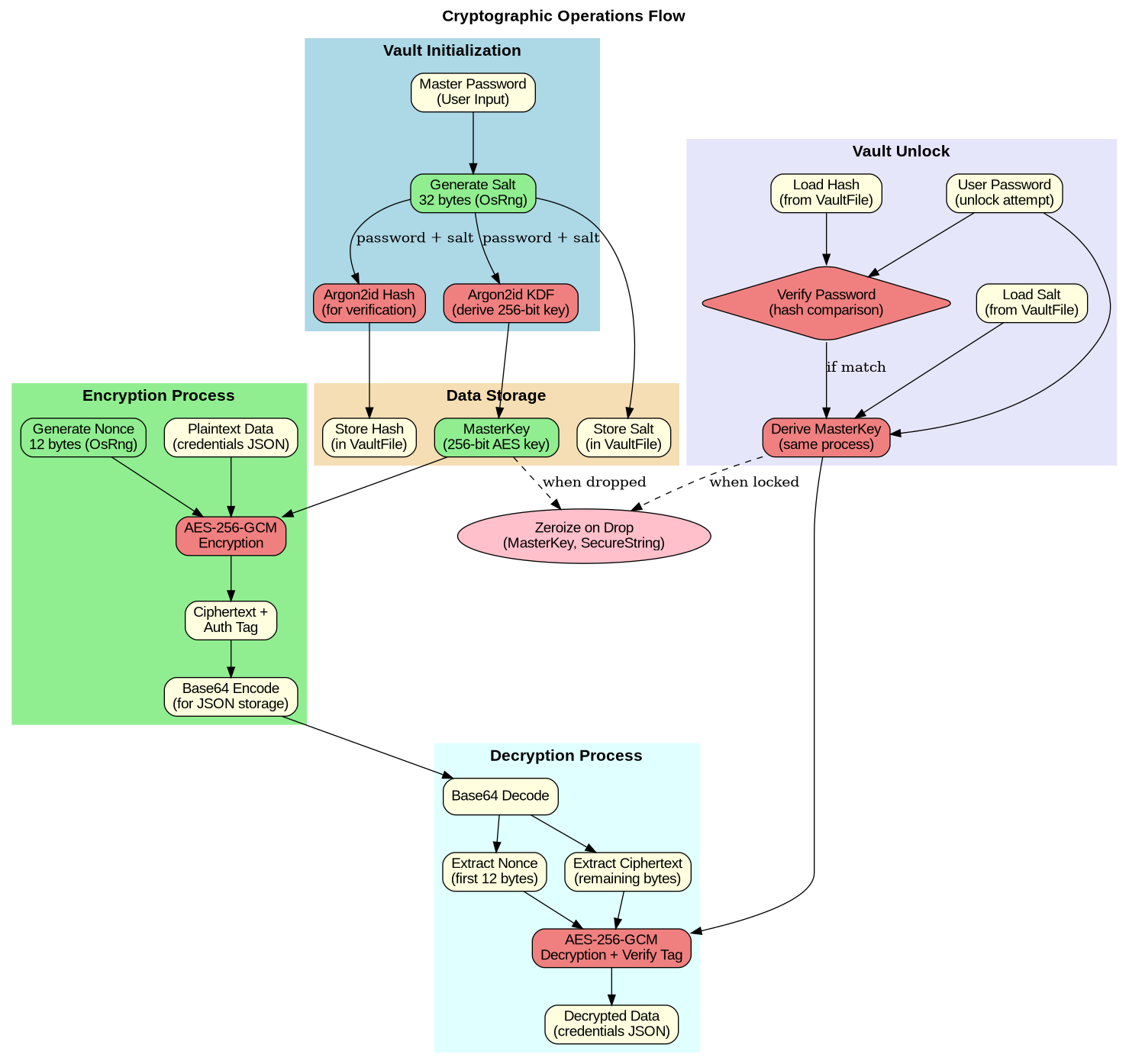
* 17 distinct steps from user command to successful completion
* Master password verification before any operation
* Encryption of individual passwords and entire credential store
* Atomic file operations ensuring data integrity
* Audit log updates for security monitoring

# 3. Cryptographic Implementation

## 3.1 Cryptographic Operations Flow

Figure 4 provides a comprehensive view of all cryptographic operations in the system, from vault initialization through encryption and decryption cycles.

Figure 4: Cryptographic Operations Flow



Key cryptographic processes shown:

* **Initialization:** Salt generation, Argon2id hashing for verification, and key derivation
* **Encryption:** Nonce generation, AES-256-GCM encryption with authentication tag
* **Unlock:** Password verification, key re-derivation from stored salt
* **Decryption:** Base64 decoding, nonce extraction, authenticated decryption
* **Memory Safety:** Zeroize-on-drop for MasterKey and SecureString

## 3.2 AES-256-GCM Specifications

| **Parameter** | **Value / Description** |
| --- | --- |
| **Algorithm** | AES-256-GCM (Galois/Counter Mode) |
| **Key Size** | 256 bits (32 bytes) - Quantum-resistant planning |
| **Nonce Size** | 96 bits (12 bytes) - Random per encryption |
| **Authentication Tag** | 128 bits (16 bytes) - Detects tampering |
| **Standards** | NIST FIPS 197, NIST SP 800-38D |

# 4. Vault File Structure

## 4.1 JSON File Format

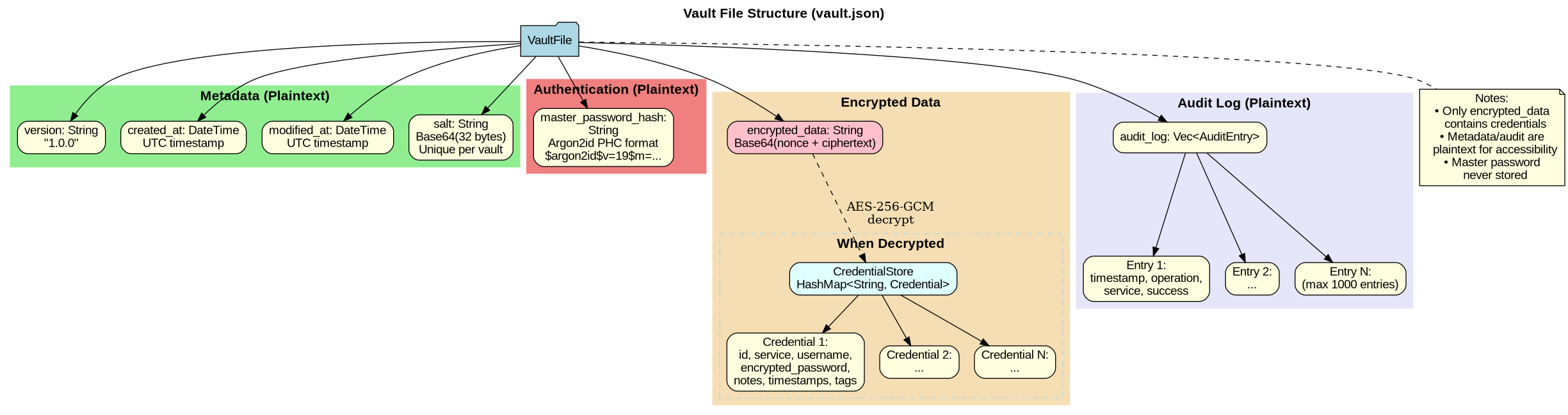
The vault is persisted as a JSON file with a well-defined structure. Figure 5 shows the complete file organization including metadata, encrypted data, and audit log.

Figure 5: Vault File Structure (vault.json)

The structure contains:

* **Metadata (Plaintext):** Version, timestamps, salt for key derivation
* **Password Hash (Plaintext):** Argon2id hash for password verification only
* **Encrypted Data:** Base64-encoded AES-256-GCM ciphertext containing all credentials
* **Audit Log (Plaintext):** Operation history for security monitoring

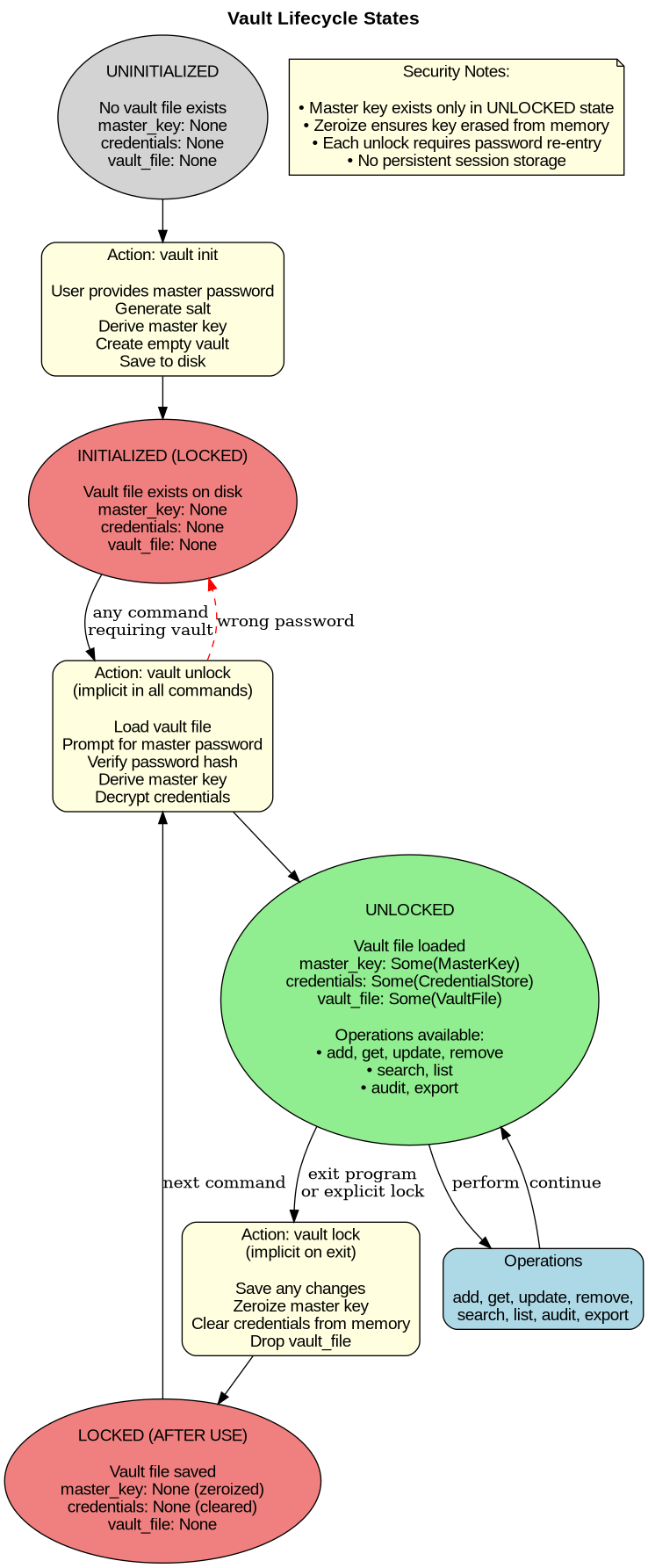
Security Note: Only the encrypted data field contains sensitive credential information. All other fields are either non-secret (salt, timestamps) or cannot be used for decryption (password hash).

# 5. Vault Lifecycle Management

## 5.1 State Transitions

The vault progresses through distinct states during its lifecycle. Figure 6 illustrates all possible states and the operations that trigger state transitions.

Figure 6: Vault Lifecycle States



State Descriptions:

* **UNINITIALIZED:** No vault exists. Requires 'vault init' command.
* **LOCKED:** Vault file exists but not accessible. Master key = None.
* **UNLOCKED:** Master key derived, credentials decrypted. All operations available.

Security Implications:

* Master key exists only in UNLOCKED state
* Zeroize ensures key is erased when transitioning to LOCKED
* Each unlock requires password re-entry (no persistent sessions)
* Vault automatically locks on program exit

# 6. Security Best Practices

## 6.1 Master Password Guidelines

* **Minimum 16 characters** (20+ for high-security needs)
* **Character diversity:** Mix uppercase, lowercase, numbers, symbols
* **Avoid common patterns:** No dictionary words, dates, keyboard patterns
* **Never reuse:** Must be unique from all other passwords
* **Physical backup:** Store written copy in secure location

## 6.2 Threat Model Summary

Protected Against:

* Stolen vault file (encryption protects data)
* Weak passwords (Argon2id memory-hardness)
* Data tampering (GCM authentication tag)
* Memory dumps (zeroization)
* Buffer overflows (Rust memory safety)

Not Protected Against:

* Keyloggers (requires OS-level protection)
* Malware on host system
* Physical access to unlocked vault
* Social engineering attacks

# 7. Conclusion

The Secure Password Vault demonstrates how modern programming languages and cryptographic libraries can be combined to create robust security solutions. The architectural diagrams provided throughout this document illustrate the clean separation of concerns, comprehensive security measures, and thoughtful design decisions that make this system trustworthy.

By leveraging Rust's memory safety guarantees, industry-standard encryption algorithms (AES-256-GCM, Argon2id), and a well-architected layered design, the system provides a reliable platform for credential management. The visual representations of architecture, data flow, and cryptographic operations enable both developers and security auditors to understand and verify the system's behavior.

This project serves as both a practical tool for password management and an educational resource demonstrating secure software development principles, cryptographic implementation best practices, and the benefits of memory-safe programming languages.

End of Documentation

© 2025 Owen Chibanda, Jasmine Sciarillo, Aryan Bhosale

George Washington University