IronClad Vault (Enterprise Edition)

Technical Specification v4.4.3

Architecture: Local-First / Hybrid Client-Server

November 26, 2025

Abstract

Overview: IronClad Vault is a secure, high-performance file storage system designed for environments requiring strict data sovereignty (e.g., legal, healthcare, defense). Operating on a Zero-Trust, Local-First model, it eschews cloud dependencies in favor of a custom TCP tunneling protocol, hierarchical Role-Based Access Control (RBAC), and a physical "Megakey" recovery mechanism.

Contents

1 System Architecture

The system is built in **Go (Golang)** and operates in three distinct modes using a single binary executable.

1.1 Operational Modes

- **Desktop Mode:** A standalone GUI application using the Fyne toolkit for local file management.
- **Sentinel Server (Headless):** A GUI-wrapped daemon that unlocks the vault in memory and listens for secure remote connections.
- **Remote Client:** A lightweight GUI client that connects to the Sentinel via an encrypted TCP tunnel.

1.2 Data Flow Architecture

The system follows two primary data paths depending on the user's connection method:

Local Desktop Path

$$\text{User} \xrightarrow{\text{Drag \& Drop}} \textbf{GUI Client} \xrightarrow{\text{Stream (64KB Chunks)}} \textbf{AES-256-GCM Engine} \xrightarrow{\text{Encrypted Blob}} \textbf{Secure Storage (Disk)}$$

Remote Admin Path

$$\text{Remote User} \xrightarrow{\text{TCP Tunnel}} \textbf{Sentinel Server} \xrightarrow{\text{Decrypt (RAM)}} \textbf{Memory Buffer} \xrightarrow{\text{Re-Encrypt (Session Key)}} \textbf{Tunnel Stream} \xrightarrow{\text{Encrypted Traffic}} \text{Remote User}$$

2 Cryptographic Standards

IronClad adheres to modern cryptographic standards, prioritizing authenticated encryption and memory hardness.

- **Data Encryption:** AES-256-GCM (Galois/Counter Mode). Provides both confidentiality and integrity.
- **Key Derivation (Passwords):** Argon2id (Time=2, Memory=64MB, Threads=4). Resists GPU/ASIC brute-force attacks.
- **Key Derivation (Megakey):** SHA-256. The 1MB entropy file is hashed to produce a 32-byte Key Encryption Key (KEK).
- **Randomness:** `crypto/rand` (OS-level CSPRNG) is used for all Nonce and Key generation.

2.1 Key Wrapping Architecture

The system utilizes a hierarchical key wrapping strategy:

1. **Master Vault Key (MVK):** A random 32-byte AES key generated at setup. This key encrypts all files and metadata. *It never touches the disk in plain text.*

2. **User KEK:** Derived from the user's password via Argon2id. Used to encrypt the MVK for storage in the `users` table.
3. **Megakey KEK:** Derived from the 1MB recovery file via SHA-256. Used to encrypt the MVK for emergency recovery.

2.2 Megakey Mechanism

To prevent lockout without backdoors, a 1,048,576-byte (1MB) binary file is generated during setup.

- **Storage:** Saved only to the user's external storage. Never stored inside the vault.
- **Unlock Process:** The server reads the file, hashes it to 32 bytes, and attempts to decrypt the wrapped Master Key stored in the configuration.

3 Networking & Tunneling Protocol

To bypass complexity and vulnerabilities associated with self-signed HTTPS certificates in local intranets, IronClad implements a custom **Secure TCP Tunnel on port 9000**.

3.1 Connection Handshake

1. **Sentinel Start:** Server generates a random 32-byte Session Key (displayed in GUI log).
2. **Connection:** Client connects via raw TCP.
3. **Authentication:** The client must possess the Session Key (Pre-Shared Key model) to communicate.

3.2 Binary Packet Structure

Every transmission (Request or Response) follows this strict binary format:

Segment	Size	Description
Length	4 Bytes	`uint32` (Big Endian) length of the following payload.
Nonce	12 Bytes	Random AES-GCM Nonce (unique per packet).
Ciphertext	Variable	AES-256-GCM encrypted payload (JSON).

Table 1: TCP Packet Format

3.3 Chunked Transfer for Large Files

To handle files larger than available RAM (e.g., 10GB video), the system uses a streaming approach:

1. **Command:** Client sends `get_file` or `upload_file`.
2. **Metadata Exchange:** Parties exchange JSON metadata (Filename, Size).
3. **Stream Loop:**

- Sender reads 64KB from disk.
- Decrypts file from storage (Master Key).
- Re-encrypts chunk for tunnel (Session Key).
- Transmits JSON packet: `{"chunk": "BASE64_STRING"}`.

4. **Termination:** Sender transmits `{"status": "eof"}`.

4 Access Control & Security Model

4.1 Clearance Levels

Access control is enforced at the API/Tunnel level before any file operation is permitted.

Level	Designation	Description
1	Public	General visibility.
5	Internal	Standard operational data.
7	Secret	Sensitive/restricted data.
10	Top Secret	Executive/Admin only.

Table 2: Integer-based Clearance Levels

Logic: `Read Access: if User.AccessLevel >= File.MinAccessLevel`

4.2 User Management

Only users with `role="admin"` can invoke `add_user`, `update_user`, `reset_password`, or `delete_user`.

5 Data Persistence & Hygiene

5.1 Database Schema

Operational data is stored in `vault_metadata.db`:

- **users:** Stores usernames, roles, access levels, and the unique wrapped Master Key for that user.
- **files:** Stores randomized `stored_name` (e.g., `enc_183471.dat`) and metadata.
- **Privacy Upgrade:** The `original_name` column is encrypted with the Master Key to prevent metadata leakage if the DB file is stolen.
- **audit_log:** Immutable log of actions (LOGIN, IMPORT, EXPORT, DELETE).

5.2 Security Measures

- **Storage Location:** `./secure_storage/`
- **Auto-Locking:** After any write operation, the system calls `os.Chmod(path, 0400)` (Read-Only) to prevent accidental deletion by the OS.
- **Secure Deletion:** When deleting via the app:

1. `chmod 0600` (Unlock).
2. Overwrite content with cryptographic noise (1 pass).
3. `os.Remove` (Unlink).

- **Memory Hygiene:** The application explicitly zeroes out the Master Key (`SecureZero`) upon logout or shutdown to minimize RAM footprint.