⍰ Security Overview Document

1. Purpose

This document provides an overview of the encryption methods, key management, and integrity verification mechanisms used in the file storage and retrieval system. It explains how sensitive data is protected at rest and in transit, ensuring confidentiality, integrity, and authenticity.

2. Encryption Methodology

2.1 Algorithm: AES-GCM

- AES (Advanced Encryption Standard) with Galois/Counter Mode (GCM) is used for encryption.

- AES-256 (32-byte key) ensures strong encryption resistant to brute-force attacks.

- GCM mode provides confidentiality, integrity, and authenticity.

2.2 Nonce (Initialization Vector)

- Each file is encrypted with a unique 12-byte nonce.

- Nonces are randomly generated for every upload.

- The nonce is prepended to the ciphertext before storage.

3. Key Derivation & Management

3.1 Master Password

- A single master password is defined in .env (MASTER_PASSWORD).

- This is never stored in plaintext in the system.

3.2 PBKDF2 Key Derivation

- AES encryption key is derived from the master password using PBKDF2-HMAC-SHA256 with 200,000 iterations and a random 16-byte salt.

- The resulting derived key (32 bytes) is used with AES-GCM.

3.3 Salt Handling

- The random salt is stored in salt.bin.

- It ensures that even if two deployments use the same password, the derived key is unique.

- If salt.bin is deleted or replaced, previously encrypted files cannot be decrypted.

4. File Integrity & Metadata

4.1 Integrity Verification

- For each uploaded file, a SHA-256 hash of the plaintext is computed.

- The hash is stored in metadata.json.

- During download/decryption, the system recomputes SHA-256 and compares with the stored value.

- If mismatch occurs, a warning about data tampering is raised.

4.2 Metadata Storage

- metadata.json contains: original filename, upload timestamp, SHA-256 integrity hash, file size in bytes.

- The actual encrypted files are saved separately in the uploads/ directory.

5. Key & Secret Handling

- Master Password: Must be kept secret (only in .env, never hardcoded).

- Flask Secret Key: (FLASK_SECRET) is used for session cookies and flash messages.

- Salt: Persisted in salt.bin. Deleting it breaks decryption capability.

- Symmetric Key: Derived in-memory only during runtime. It is never stored on disk.

6. Security Considerations

1. Brute-Force Resistance: PBKDF2 with 200k iterations slows down attackers attempting to guess the password.

2. Tamper Detection: AES-GCM provides built-in authentication. SHA-256 adds an extra layer of verification.

3. Replay Protection: Unique random nonces ensure that even identical files produce different ciphertexts.

4. Deployment Safety: .env must be protected with file permissions. Never commit .env or salt.bin to public repositories.

5. Production Hardening: Disable debug=True in Flask for production use. Use HTTPS to protect data in transit.

7. Threat Model

Protected Against:

- File leakage if server storage is compromised (files are encrypted).

- Tampering with encrypted files (AES-GCM + SHA256 detect this).

- Password re-use across deployments (different salt ensures uniqueness).

Not Protected Against:

- Weak master password chosen by user.

- Insider threats with access to .env.

- Active runtime memory extraction (key is stored in memory during runtime).