**Security Overview**

This document explains the encryption methods, key handling, and security controls used in the **Secure File Sharing** project. It also lists recommended hardening steps and a validation checklist.

**1) Threat Model**

**Assets**

* User files uploaded via the portal
* Encryption keys
* User identities and sessions

**Primary Threats**

* Unauthorized access to files at rest (server compromise or storage theft)
* Leaked/stolen encryption keys
* Integrity tampering (malicious modification of files)
* MITM during file transfer
* Abusive uploads (malware, oversized files)

**Out of Scope (for this MVP)**

* Nation‑state adversaries
* Advanced browser exploitation
* Insider threats with OS‑level access to the key store

**2) Encryption Design**

**Algorithm**

* **AES‑256‑GCM** (preferred) using PyCryptodome.
  + Provides **confidentiality** (encryption) and **integrity** (auth tag).
  + Each encryption operation uses a **unique 96‑bit nonce** generated with a CSPRNG.

If you implemented AES‑CBC initially, migrate to **AES‑GCM**. CBC requires separate HMAC for integrity and is easier to misuse.

**File Encryption Flow (at rest)**

1. On upload, the server:
   * Generates a **random 12‑byte nonce** per file.
   * Encrypts the file stream with **AES‑256‑GCM** using the user’s key.
   * Stores **ciphertext + nonce + auth tag** (e.g., as a header or sidecar .meta).
2. On download, the server:
   * Looks up the user’s key, reads nonce & tag, then decrypts.
   * If the **auth tag fails**, abort and return HTTP 400.

**File Format Options**

* **Single blob**: [nonce | ciphertext | tag] concatenated.
* **Sidecar metadata**: save {nonce, tag, original\_filename, mime} as JSON next to the encrypted blob.

**Large Files**

* Use **chunked/streaming encryption** to avoid loading the whole file into memory. Each chunk must get a **unique nonce** or derive a per‑chunk nonce.

**3) Key Management**

**Key Types**

* **Master Key (optional, recommended)**: One per environment, used to encrypt per‑user keys (“key‑encrypting key”).
* **Per‑User Data Key**: A 256‑bit AES key used to encrypt that user’s files.

**Key Generation**

* Use a CSPRNG: os.urandom(32) for AES‑256 keys.
* Encode keys as base64 when storing in files/JSON.

**Where Keys Live (MVP)**

* **Per‑user key files** under ./keys/<username>.key (file‑permission restricted to the service account).
* Do **not** commit keys to Git. Ensure .gitignore contains keys/.

**Better Storage (recommended for production)**

* **Environment variable + file‑based** master key (e.g., in .env or OS secret store), used to encrypt per‑user keys at rest.
* **External secret managers** (preferred): AWS KMS, GCP KMS, Azure Key Vault, or HashiCorp Vault.

**Key Rotation**

* Support rotating per‑user keys by:
  1. Generating a new key K2.
  2. Re‑encrypting newly uploaded files with K2 (lazy rotation).
  3. Optionally re‑encrypting old files in the background.
  4. Keeping key metadata per file so you can locate the correct key.

**Access Control to Keys**

* Keys are only read on the server side after **session authentication**.
* File ownership is enforced by storing files under uploads/<username>/... and resolving paths **only** via server‑side username from session, never from user input.

**4) Transport Security**

* Run the app behind **HTTPS** (TLS 1.2+). In dev, terminate TLS with a reverse proxy (Nginx/Caddy) or use a platform that provides TLS.
* Set SESSION\_COOKIE\_SECURE=True in production so cookies only travel over HTTPS.

**5) Authentication & Session Security**

* Use server sessions (Flask session cookie).
* Configure in production:
  + SESSION\_COOKIE\_HTTPONLY=True (mitigate XSS cookie theft)
  + SESSION\_COOKIE\_SAMESITE='Lax' (or Strict)
  + Rotate SECRET\_KEY and keep it out of Git.
* Rate‑limit login attempts; lock out or add CAPTCHA on repeated failures.

**6) Upload Security & Validation**

* **Allowlist file types** (e.g., png, jpg, jpeg, gif, pdf, txt).
* **Max size** limit (e.g., 16MB) using MAX\_CONTENT\_LENGTH.
* Sanitize filenames with werkzeug.utils.secure\_filename.
* Store encrypted blobs without executing them (never serve from a path that the web server executes).
* Consider AV/malware scanning (e.g., ClamAV) for enterprise contexts.

**7) Integrity & Metadata**

* AES‑GCM’s **auth tag** gives integrity. Decryption should fail if the file was modified.
* Track metadata: original name, size, content type, upload time, and key id used.
* Optionally compute and store a **SHA‑256 hash** of the plaintext or ciphertext for audit/verification.

**8) Logging & Auditing**

* Log: user, action (upload/download/delete), filename (normalized), size, outcome, and timestamp.
* **Never** log raw keys, nonces, or auth tags.
* Consider a separate, append‑only audit log for admin review.

**9) Backups & Disaster Recovery**

* Back up **encrypted blobs only**, not plaintext.
* Back up the **key store** (or key‑encrypting material) separately and securely.
* Practice restore drills to validate you can decrypt old data after rotations.

**10) Example Snippets (PyCryptodome, AES‑GCM)**

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

Encrypt

def encrypt\_bytes(plaintext: bytes, key: bytes):

nonce = get\_random\_bytes(12) # 96‑bit nonce for GCM

cipher = AES.new(key, AES.MODE\_GCM, nonce=nonce)

ciphertext, tag = cipher.encrypt\_and\_digest(plaintext)

return nonce, ciphertext, tag

Decrypt

def decrypt\_bytes(nonce: bytes, ciphertext: bytes, tag: bytes, key: bytes):

cipher = AES.new(key, AES.MODE\_GCM, nonce=nonce)

return cipher.decrypt\_and\_verify(ciphertext, tag)

For files, read/write in chunks and prepend/store the nonce and tag alongside the ciphertext.

**11) Environment & Deployment Hardening**

* Keep SECRET\_KEY, master keys, and config in environment variables or a secret manager.
* Restrict filesystem permissions for uploads/ and keys/ (only the app user can read/write).
* Run the app under a **non‑root** user.
* Use a reverse proxy and enable **HTTPS** + **HSTS**.
* Turn off DEBUG in production.

**12) Validation Checklist (pre‑release)**

* All files are encrypted at rest with AES‑GCM.
* Unique nonce per encryption; nonces never reused with the same key.
* Keys are generated via CSPRNG and never committed to Git.
* keys/ and uploads/ are in .gitignore and permission‑restricted.
* MIME/type allowlist and max upload size enforced.
* Sessions: HttpOnly, Secure (prod), SameSite set.
* HTTPS enforced in prod; no plaintext credentials.
* Logs exclude sensitive secrets and cover key events.
* Backups contain only ciphertext; restore tested.
* README and this SECURITY.md describe the design and assumptions.

**13) Future Enhancements**

* Replace in‑memory users with a database (hashed passwords via Argon2/bcrypt).
* Use **KMS / Vault** for key storage + automatic rotation.
* Add **per‑file keys** (envelope encryption) and store encrypted data keys with each file.
* Implement **two‑factor authentication**.
* Add **download links with expiry** and signed URLs.
* Integrate **malware scanning** and content‑disarm (CDR) if targeting high‑risk environments.