# Task 3 Given by Future Interns

# Secure File Share System Using AES – Architecture, Code & Security

# Overview

**Stack:** Python 3, Flask 3, PyCryptodome, Jinja2, HTML/CSS, Dotenv
**Core Goal:** Confidential file storage & retrieval with authenticated encryption at rest; zero plaintext persistence during upload; decryption-on-download.

> **Note on variants:** You may have two code variants from our conversation. 1) **Recommended build** (more secure): AES-256-GCM, per-file random *salt* and *nonce*, PBKDF2-HMAC-SHA256 key derivation, no plaintext files written to disk on upload, temporary plaintext deleted after response. 2) **Simplified build** (for study): AES-CFB with a fixed KDF salt and no authentication tag; decrypts to a ".dec" file. This document specifies the **recommended AES-GCM design** and then lists differences if you use the simplified build.

---

## 1) System Architecture

### 1.1 High-level Components

- **Client (Browser):** Uploads a file via HTML form; downloads decrypted file on demand.
- **Web Server (Flask):** Routes for upload, list, and download. Performs encryption/decryption and enforces size/type limits.
- **Crypto Module (PyCryptodome):** Provides AES and PBKDF2 primitives.
- **Storage (Filesystem):** Persists only **encrypted** files in `uploads/`. Temporary plaintext exists only in memory (on upload) or short-lived on disk (for download response) and is then deleted.

### 1.2 Data Flow (AES-GCM design)

```
[User Browser]
    | 1) POST /upload (multipart/form-data)
    v
[Flask Upload Route]
    - Read file bytes in memory
    - Derive key: PBKDF2(passphrase, random_salt, 200k iters)
    - Generate random nonce (12 bytes)
    - AES-256-GCM encrypt -> ciphertext + tag
    - Persist: MAGIC|salt|nonce|tag|ciphertext  => uploads/<name>.enc
    - No plaintext saved
    v
[Storage (uploads/*.enc)]

Download path:
[User Browser]
    | 2) GET /download/<name>.enc
    v
[Flask Download Route]
    - Read .enc, parse header (MAGIC|salt|nonce|tag|ciphertext)
    - Re-derive key with salt
    - AES-GCM decrypt+verify tag
    - Write a temp plaintext file -> send_file(...)
    - After response: delete temp plaintext
```

### 1.3 Deployment View

- **Dev:** Run with Flask built-in server (`python app.py`).
- **Prod:** Run with a WSGI server (gunicorn/waitress) behind Nginx/Apache, **HTTPS** required.
- **Secrets:** `.env` file or OS secrets store (Windows Credential Manager, Kubernetes secrets, etc.).

---

## 2) Code Walkthrough (AES-GCM build)

Files: `app.py`, `templates/`, `static/style.css`, `uploads/`, `decrypted/`.

### 2.1 Configuration

- `MAX_CONTENT_LENGTH = 10 * 1024 * 1024` – 10 MB limit.
- `ALLOWED_EXTENSIONS = {"txt","pdf","png","jpg","jpeg","gif","csv","json","xml","docx","xlsx","pptx","zip"}`
- `.env` contains AES_PASSPHRASE and an optional `FLASK_SECRET_KEY` used for session flashing.

### 2.2 Crypto Helpers

- **Key Derivation**
  - `PBKDF2(passphrase, salt, dkLen=32, count=200_000, hmac_hash_module=SHA256)`
  - Produces a 256-bit key (**AES-256**).
- **Encrypt**
  - Random 16-byte `salt` and 12-byte nonce per file.
  - `AES.new(key, AES.MODE_GCM, nonce=nonce)`

    - cipher.encrypt_and_digest(plaintext) returns (ciphertext, tag).
    - Stored format: MAGIC|salt|nonce|tag|ciphertext to <name>.enc.
- **Decrypt**
    - Parse header, re-derive key with same salt, decrypt_and_verify(ciphertext, tag).
    - Integrity failure (tamper/Wrong key) raises an exception and returns an error page.

### 2.3 Upload Route (/upload)

- Validates file presence, filename, type, and size.
- Reads file **bytes in memory** (avoids plaintext on disk).
- Encrypts using AES-GCM.
- Saves **only** encrypted blob to uploads/<name>.enc.

### 2.4 Files Listing (/files)

- Renders a list of encrypted files (*.enc) with download links.

### 2.5 Download Route (/download/<path:enc_filename>)

- Validates suffix .enc and sanitizes filename.
- Reads blob, decrypts, writes a **unique temp file** (UUID prefix), serves it with send_file(..., as_attachment=True).
- Registers @after_this_request cleanup to **delete** the temp file post-response.

### 2.6 Error Handling

- Returns informative 400/404 pages without leaking secrets or stack traces in production.

---

## 3) Security Measures

### 3.1 Cryptography

- **AES-256-GCM** for confidentiality + integrity/authentication.
- **Per-file random salt** for PBKDF2 prevents cross-file key reuse.
- **Per-file random nonce** ensures GCM uniqueness; never reused.
- **200k PBKDF2 iterations** harden against offline passphrase guessing.

### 3.2 Key & Secret Handling

- Passphrase stored **outside code** in .env (Windows: don't commit to Git).
- Derive per-file keys via PBKDF2 rather than storing raw keys.
- Optionally rotate passphrase: new files use new passphrase; re-encrypt old files if needed.

- For production, prefer a **KMS/HSM** or OS secret store over `.env` files.

### 3.3 File Handling

- **No plaintext persisted** at upload.
- Temporary plaintext on download is **randomly named** and **deleted** immediately after response.
- `secure_filename(...)` prevents path traversal.
- File extension allow-list and **10 MB** max size reduce risk.

### 3.4 Transport & Server

- Use **HTTPS** end-to-end in production.
- Run behind a hardened WAF/reverse proxy.
- Disable Flask debug in production; restrict error details.

### 3.5 Logging & Privacy

- Avoid logging filenames or paths if they may be sensitive.
- Never log keys, salts, nonces, or tags.

### 3.6 Threat Model → Mitigations (summary)

| Threat | Impact | Mitigation |
|---|---|---|
| Stolen storage (disk theft) | Confidentiality | AES-GCM at rest; no plaintext files saved |
| Tampered `.enc` files | Integrity | GCM tag verification fails and aborts |
| Brute-force passphrase | Confidentiality | PBKDF2-HMAC-SHA256 (200k iters) + strong passphrase policy |
| Path traversal | RCE / data exfil | `secure_filename`, enforced `.enc` suffix |
| Oversized uploads | DoS | `MAX_CONTENT_LENGTH` limit |
| MITM during transit | Confidentiality/Integrity | Enforce HTTPS in prod |
| Plaintext remnants | Confidentiality | In-memory upload; temp files auto-deleted |

---

## 4) How to Run (Windows)

1. Extract project → open PowerShell in folder.

2. Create venv & install deps:

```
python -m venv .venv
.venv\Scripts\activate
pip install -r requirements.txt
```

3. Configure secrets:

```
copy .env.example .env
notepad .env   # set a strong AES_PASSPHRASE
```

4. Start server:

```
python app.py
```

5. Visit `http://127.0.0.1:5000` → Upload, list, and download files.

---

## 5) Validation & Testing Plan

### 5.1 Functional Tests

- Upload files of each allowed type; verify `.enc` exists and no plaintext remains.
- Download each file; verify it opens.

### 5.2 Integrity & Confidentiality Tests

- Compute SHA-256 of original vs downloaded file (should match):

```
CertUtil -hashfile .\my.pdf SHA256
CertUtil -hashfile .\Downloads\my.pdf SHA256
```

- Flip random bytes inside an `.enc` file; download should **fail** with an error.
- Change `AES_PASSPHRASE`; old files should not decrypt (by design).

### 5.3 Negative Tests

- Upload >10MB file → rejected.
- Upload disallowed extension → rejected.
- Request `/download/<name>` without `.enc` suffix → 400.

### 5.4 Basic Security Review Checklist (OWASP ASVS style)

☐ Secrets not in VCS
☐ HTTPS enabled in prod
☐ Safe filename handling
☐ Input size limits
☐ Strong passphrase policy
☐ No sensitive data in logs

---

## 6) Operations: Key Rotation & Backup

### 6.1 Key Rotation Strategy

- **Soft rotation:** change passphrase → new files use new key; old files still decrypt with old passphrase if you keep it.
- **Hard rotation:** decrypt each `.enc` with old passphrase and re-encrypt with new; script this offline.

### 6.2 Backup Strategy

- Back up only **encrypted** files from `uploads/`.
- Back up the **current passphrase** (or KMS policy) in a separate secure vault.

---

## 7) Future Enhancements

- User authentication (Flask-Login / OAuth) + per-user namespaces.
- Server-side streaming encryption/decryption for very large files.
- Envelope encryption with a DEK per file and KEK in a KMS (Azure Key Vault / AWS KMS / GCP KMS).
- Audit logging (hashed/append-only) and retention policies.
- Virus scanning on upload (ClamAV service) before encryption.
- Content Security Policy (CSP) headers; secure cookies; rate limiting.

---

## 8) Differences if Using the Simplified AES-CFB Build

- **Cipher:** AES-CFB (confidentiality **only**, no authenticity). Consider upgrading to GCM.
- **KDF Salt:** Fixed salt → less ideal; use per-file random salt.
- **Decryption:** Produces a `.dec` file that remains on disk unless manually deleted → add cleanup.
- **Action:** For internship submission, prefer the AES-GCM project; or refactor CFB code to the GCM pattern above.

---

## Appendix A – API Endpoints (recommended build)

- `GET /` → redirects to `/upload`.
- `GET|POST /upload` → upload and encrypt file.
- `GET /files` → list encrypted files.
- `GET /download/<path:enc_filename>` → decrypt and download original.

## Appendix B – File Format (recommended build)

```
0..3    : MAGIC bytes ("SFv1")
4..19   : salt (16 bytes)
20..31  : nonce (12 bytes)
32..47  : tag (16 bytes)
48..end : ciphertext
```

## Appendix C – Quick Demo Commands

```
# Upload via curl
curl -F "file=@C:\\path\\to\\report.pdf" http://127.0.0.1:5000/upload

# List encrypted files (browser)
start http://127.0.0.1:5000/files
```

---

**End of Document**

By Navneet Bhatt\