

Secure File Sharing System – Security & Implementation Report

Project: Secure File Sharing System

Tools Used: Python Flask, PyCryptodome (AES), HTML/CSS/JS, Postman, curl,

Prepared by: *RAWLINGS ODIERO*

Date: August 2025

Task: 3



FutureInterns

1. Executive Summary

This internship project focuses on the design and implementation of a **Secure File Sharing System** that allows users to upload and download files with robust security controls. The core objective is to ensure that all files remain confidential, tamper-proof, and protected during both storage and transfer.

The system leverages **AES encryption** to safeguard data at rest and in transit. Files are encrypted immediately upon upload, stored in their encrypted form on the server, and decrypted only when accessed by authorized users. This approach prevents unauthorized access, even if the storage medium is compromised.

The project simulates real-world client requirements in industries where secure data exchange is essential, such as **healthcare, legal services, and corporate environments**. In these sectors, compliance with data protection regulations (e.g., GDPR, HIPAA) and client confidentiality agreements is critical, making strong encryption and key management mandatory.

Beyond the encryption layer, the system incorporates **secure file handling practices, basic key management, and integrity verification** to detect any unauthorized modifications. It also includes a simple web interface for intuitive file uploads and downloads, making it accessible for both technical and non-technical users.

This project delivers not only a functional prototype but also demonstrates secure development practices that can be expanded into a production-grade solution with additional features such as authentication, access control, and centralized key vault integration.

2. Objectives

The primary objectives of this project are to design and implement a secure, efficient, and user-friendly file sharing system with a strong emphasis on **data confidentiality, integrity, and secure handling**.

3. Tools & Technologies Used

The project leverages a combination of backend frameworks, cryptography libraries, and frontend technologies to create a secure and functional file-sharing platform

TOOL / TECHNOLOGY	PURPOSE
1. Python Flask	Backend web framework for handling file upload/download routes and encryption logic.
2. PyCryptodome	Python cryptography library used for implementing AES-256-GCM encryption and decryption.
3. HTML / CSS / JavaScript	Frontend user interface for file upload/download functionality.
4. dotenv	Secure storage of environment variables, including encryption keys.
5. Postman	API testing tool for verifying upload and download endpoints.
6. curl	Command-line tool for testing API requests and encryption output.
7. Git & GitHub	Version control and collaboration platform for managing project code.
8. Checksum Tools (SHA256)	Used to verify file integrity before and after encryption/decryption.

4. System Setup & Environment Configuration

The development environment was configured on a secure, externally managed Python environment to ensure dependency isolation and security.

4.1 Environment Creation

The project was initialized in an externally managed Python environment to avoid conflicts with system-level Python installations.

```
(kali㉿kali)-[~/Desktop/secure-file-share]
$ python3 -m venv venv

(kali㉿kali)-[~/Desktop/secure-file-share]
$ source venv/bin/activate

(venv)-(kali㉿kali)-[~/Desktop/secure-file-share]
$ pip install --upgrade pip
Requirement already satisfied: pip in ./venv/lib/python3.13/site-packages (25.1.1)
Collecting pip
  Downloading pip-25.2-py3-none-any.whl.metadata (4.7 kB)
  Downloading pip-25.2-py3-none-any.whl (1.8 MB)
    1.8/1.8 MB 3.1 MB/s eta 0:00:00
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 25.1.1
    Uninstalling pip-25.1.1:
      Successfully uninstalled pip-25.1.1
Successfully installed pip-25.2
```

4.2 Project Folder Structure

Next, the folder structure was initialized with separate directories for templates, static files, uploads, and configuration files.

```
(venv)-(kali㉿kali)-[~/Desktop/secure-file-share]
$ mkdir -p templates static uploads

(venv)-(kali㉿kali)-[~/Desktop/secure-file-share]
$ touch app.py crypto_utils.py models.py config.py
```

```
(venv)-(kali㉿kali)-[~/Desktop/secure-file-share]
$ touch requirements.txt .env.example .gitignore README.md

(venv)-(kali㉿kali)-[~/Desktop/secure-file-share]
$ touch templates/base.html templates/index.html templates/files.html

(venv)-(kali㉿kali)-[~/Desktop/secure-file-share]
$ ls
app.py  config.py  crypto_utils.py  models.py  README.md  requirements.txt  static  templates  uploads  venv
```

4.3 Dependency Management

Dependencies were defined in **requirements.txt** to ensure consistency across environments.

```
(venv)-(kali@kali)-[~/Desktop/secure-file-share]
└─$ pip install -r requirements.txt

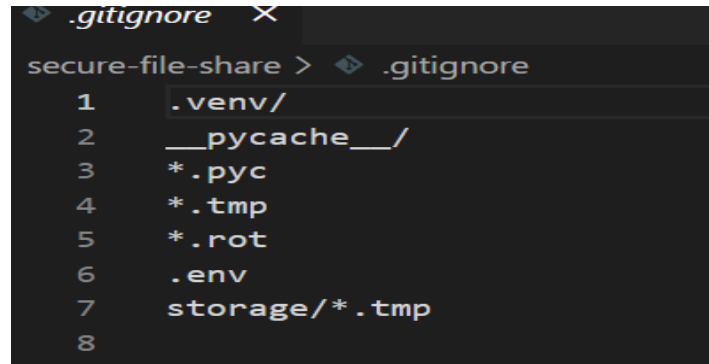
Collecting Flask==3.0.3 (from -r requirements.txt (line 1))
  Downloading flask-3.0.3-py3-none-any.whl.metadata (3.2 kB)
Collecting SQLAlchemy==2.0.32 (from -r requirements.txt (line 2))
  Downloading SQLAlchemy-2.0.32-py3-none-any.whl.metadata (9.6 kB)
Collecting Flask_SQLAlchemy==3.1.1 (from -r requirements.txt (line 3))
  Downloading flask_sqlalchemy-3.1.1-py3-none-any.whl.metadata (3.4 kB)
Collecting python-dotenv==1.0.1 (from -r requirements.txt (line 4))
  Downloading python_dotenv-1.0.1-py3-none-any.whl.metadata (23 kB)
Collecting pycryptodome==3.21.0 (from -r requirements.txt (line 5))
  Downloading pycryptodome-3.21.0-cp36-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.4 kB)
Collecting itsdangerous==2.2.0 (from -r requirements.txt (line 6))
```

```
(venv)-(kali@kali)-[~/Desktop/secure-file-share]
└─$ pip list
```

Package	Version
blinker	1.9.0
click	8.2.1
Flask	3.0.3
Flask-SQLAlchemy	3.1.1
itsdangerous	2.2.0
Jinja2	3.1.6
MarkupSafe	3.0.2
pip	25.2
pycryptodome	3.21.0
python-dotenv	1.0.1
SQLAlchemy	2.0.32
typing_extensions	4.14.1
Werkzeug	3.1.3

4.4 Git & Ignore Rules

A `.gitignore` file was added to exclude sensitive files (e.g., `.env`, `__pycache__`/, `uploads/`) from version control.



```
.gitignore
secure-file-share > .gitignore
1 .venv/
2 __pycache__/
3 *.pyc
4 *.tmp
5 *.rot
6 .env
7 storage/*.tmp
8
```

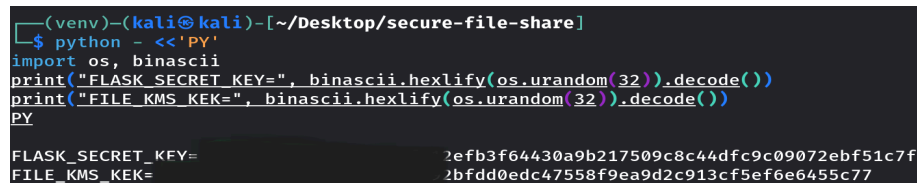
4.5 Generating Secrets & Configuring Environment Variables

To ensure secure key management, cryptographic secrets were generated dynamically and stored in `.env`.

This produced two random **256-bit keys**:

- `FLASK_SECRET_KEY` → Used by Flask for session signing.
- `FILE_KMS_KEY` → Master key for AES file encryption.

The keys were then stored in `.env`, and an `.env.example` file was committed for documentation.



```
(venv)-(kali@kali)-[~/Desktop/secure-file-share]
└─$ python - <<'PY'
import os, binascii
print("FLASK_SECRET_KEY=", binascii.hexlify(os.urandom(32)).decode())
print("FILE_KMS_KEY=", binascii.hexlify(os.urandom(32)).decode())
PY
FLASK_SECRET_KEY=                2efb3f64430a9b217509c8c44dfc9c09072ebf51c7f
FILE_KMS_KEY=                    2bfdd0edc47558f9ea9d2c913cf5ef6e6455c77
```

4.6 Core Application Files

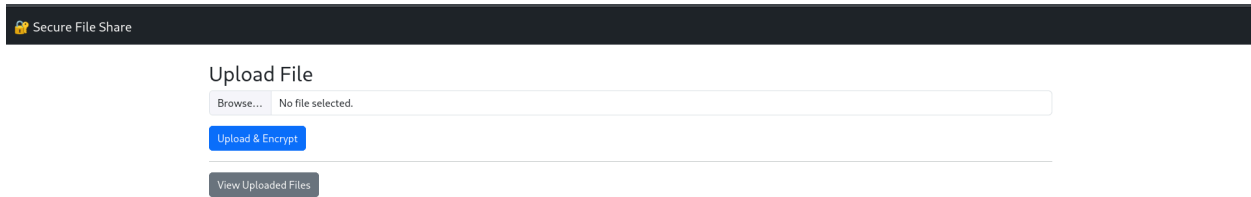
The following files were developed to implement the system:

- `config.py` → Configuration & environment variable loading
 - `models.py` → Database models (SQLAlchemy)
 - `crypto_utils.py` → AES encryption/decryption helpers
 - `app.py` → Flask application (routes, file handling, encryption logic)
 - `templates/*.html` → UI templates for file upload & download
-

4.7 Running the Application

Finally, the Flask server was started for local testing:

```
(venv)-(kali@kali)-[~/Desktop/secure-file-share]
$ python -m flask run
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [17/Aug/2025 11:19:48] "GET /files HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2025 11:19:53] "GET / HTTP/1.1" 200 -
```




5. System Architecture

The **Secure File Sharing System** is designed with a layered architecture focusing on **confidentiality, integrity, and controlled access**.

Workflow

- File Upload (Client → Server)**
 - The User uploads a file via the Flask web portal.
 - The file is immediately encrypted using **AES-256-GCM** before storage.
 - Metadata (filename, upload date, encrypted file path) is recorded in the database.
- File Storage**
 - Encrypted files are stored in the `uploads/` directory.
 - Plaintext files never touch the disk, ensuring confidentiality at rest.
- File Download (Server → Client)**
 - The User requests a file via the portal.
 - The system decrypts the file on-the-fly using the encryption key.
 - The decrypted file is streamed to the client.
- Key Management**
 - Keys are stored securely in `.env` as environment variables.
 - `FILE_KMS_KEK` (Key Encryption Key) secures Data Encryption Keys (DEKs).
 - Keys are never hardcoded into the application or committed to GitHub.

Figure 1.

 Secure File Share

Uploaded Files

Original Name	Uploaded At	Action
secrets.txt	2025-08-13 17:42:38	<button>Decrypt & Download</button>
passwordlist.txt	2025-08-14 13:37:45	<button>Decrypt & Download</button>
test.txt	2025-08-15 14:04:17	<button>Decrypt & Download</button>

Back to Upload

Figure 2.

```
(venv)-(kali@kali)-[~/Desktop/secure-file-share]
$ ls -l uploads
total 556
-rw-rw-r-- 1 kali kali 569236 Aug 14 09:37 passwordlist.txt.enc
```

Figure 2.1

```
(venv)-(kali@kali)-[~/Desktop/secure-file-share/uploads]
$ xxd passwordlist.txt.enc | head
00000000: 7800 9e3d 5207 d6d4 5c16 f0f9 1b95 bd58  x...=R...\.....X
00000010: 4adf 08c9 dc4d 425c 5f9e 86a9 f9fe f44a  J....MB\_.....J
00000020: 99e1 d0b4 87fb e06e 520c 7f48 7d8d 7715  ....nR..H}.w.
00000030: a39e d89c 8e23 1e16 544f 79be 824a 66ce  ....#..TOy..Jf.
00000040: 992c 1fb5 286d 420d a2f3 4a39 b961 f94e  .,..(mB...J9.a.N
00000050: 09a1 54c1 a873 231c 3879 3df5 7baf a696  ..T..s#.8y={...
00000060: 9468 7205 b8c4 96b3 79c4 2e7c ce70 d35f  .hr.....y..|.p._
00000070: e27b 4488 fbc8 944d a709 5201 9cf1 5fbb  .{D....M..R..._.
00000080: 02cf 1f5b 1289 2a1c c799 7f05 0894 ee38  ...[...*.....8
00000090: 1725 ca02 af98 d9ec f3d9 65e2 b6ab dce1  .%.....e.....
```


Figure 3

```
(kali㉿kali)-[~/Desktop/secure-file-share/uploads]
$ ls -l
total 4
-rw-rw-r-- 1 kali kali 47 Aug 13 13:42 secrets.txt.enc
```

Figure 3.1

```
(kali㉿kali)-[~/Desktop/secure-file-share/uploads]
$ cat secrets.txt.enc
S/wsYee%g piHfJ!l
```

Figure 3.2

```
(kali㉿kali)-[~/Desktop/secure-file-share/uploads]
$ xxd secrets.txt.enc | head
00000000: b909 8c02 999b 8e02 f0b9 e453 0fd0 2faf  ....S../.
00000010: 77fd e1af 7395 5991 6525 b1a0 7fc5 cb67  w...s.Y.e%....g
00000020: 2070 6948 d566 9ec1 a24a dbba 9421 6c   piH.f...J...!l
```

Figure 4

```
(venv)-(kali㉿kali)-[~/Desktop/secure-file-share/uploads]
$ xxd test.txt.enc | head
00000000: fb64 e810 b063 07ac 2d01 a837 3741 7481  .d...c...-..77At.
00000010: 7cd9 a500                                     |...
```

Upon successful upload of the files and storage, the process demonstrated that:

Files are Encrypted at Rest:

- Upon upload, each file is automatically encrypted using **AES-256 in GCM mode** before being written to disk.
 - At the storage layer (**uploads/**), files appear only as ciphertext (**.enc** format) and cannot be opened or interpreted without the encryption key.
 - This ensures that even if the storage directory is accessed directly, no sensitive information is exposed furthermore these files cannot be tampered with or read in plain text.
-

6. Security Implementation

- **AES-256-GCM:** Used for encryption and decryption, ensuring confidentiality and integrity.
- **Key Management:** Keys stored in `.env`, never hard-coded.
- **Database Security:** Only metadata stored (file names, timestamps); ciphertext stored separately.
- **Tamper Protection:** GCM authentication tags verify data integrity during decryption.

7. Integrity Verification

To confirm files were unaltered during storage and transfer, SHA-256 checksums were generated and compared and results matched, confirming no corruption during encryption/decryption.

Figure 1.1

```
<tr>
  <td>secrets.txt</td>
  <td>2025-08-13 17:42:38</td>
  <td>
    <a href="/download/1" class="btn btn-success btn-sm">Decrypt & Download</a>
  </td>
</tr>

<tr>
  <td>passwordlist.txt</td>
  <td>2025-08-14 13:37:45</td>
  <td>
    <a href="/download/2" class="btn btn-success btn-sm">Decrypt & Download</a>
  </td>
</tr>

<tr>
  <td>test.txt</td>
  <td>2025-08-15 14:04:17</td>
  <td>
    <a href="/download/3" class="btn btn-success btn-sm">Decrypt & Download</a>
  </td>
</tr>
```

Figure 2.1

```
(venv)-(kali㉿kali)-[~/Desktop/secure-file-share/uploads]
$ curl -OJ http://127.0.0.1:5000/download/2
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100  555k  100  555k    0     0  12.0M      0 --:--:-- --:--:-- --:--:-- 12.3M
```

```
(venv)-(kali㉿kali)-[~/Desktop/secure-file-share/uploads]
$ sha256sum passwordlist.txt passwordlist.txt.enc
d4223652f52493446401ecec72da52e42043205db5563601dde30688d797818d passwordlist.txt
bdc bde8706ccc0d033b7e906b14835aa58b241b9b853e47ab922ec4fc7c054fd passwordlist.txt.enc
```

Figure 3.1

```
(venv)-(kali@kali)-[~/Desktop/secure-file-share]
$ curl -OJ http://127.0.0.1:5000/download/3
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %             0         0              0             0             0             0
100    20    100    20    0    0    354    0    --:--:--  --:--:--  --:--:--    357

(venv)-(kali@kali)-[~/Desktop/secure-file-share/uploads]
$ sha256sum test.txt test.txt.enc
c87e2ca771bab6024c269b933389d2a92d4941c848c52f155b9b84e1f109fe35  test.txt
b0232591d2cc2885fe4f1fe282980adc138b46481faf91dc4962ad171260ee1b  test.txt.enc
```

7. File Integrity Tampering Test

As part of the security validation, an intentional tampering test was performed to assess the system's integrity controls. A valid file (`passwordlist.txt`) was first uploaded and stored in encrypted form. The ciphertext was then deliberately modified (7800 to 8000) at the storage layer by altering a few bytes directly within the encrypted file.

```
(venv)-(kali@kali)-[~/Desktop/secure-file-share/uploads]
$ xxd passwordlist.txt.enc | head
00000000: 7800 9e3d 5207 d6d4 5c16 f0f9 1b95 bd58  x..=R...\.....X
00000010: 4adf 08c9 dc4d 425c 5f9e 86a9 f9fe f44a  J....MB\_\.....J
00000020: 99e1 d0b4 87fb e06e 520c 7f48 7d8d 7715  ....nR...H}.w.
00000030: a39e d89c 8e23 1e16 544f 79be 824a 66ce  ....#...TOy..Jf.
00000040: 992c 1fb5 286d 420d a2f3 4a39 b961 f94e  ,...(mB...J9.a.N
00000050: 09a1 54c1 a873 231c 3879 3df5 7baf a696  ..T..s#.8y=. {...
00000060: 9468 7205 b8c4 96b3 79c4 2e7c ce70 d35f  .hr.....y...|.p._
00000070: e27b 4488 fbc8 944d a709 5201 9cf1 5fbb  .{D....M..R...._
00000080: 02cf 1f5b 1289 2a1c c799 7f05 0894 ee38  ...[...*.....8
00000090: 1725 ca02 af98 d9ec f3d9 65e2 b6ab dce1  .%.....e.....

(venv)-(kali@kali)-[~/Desktop/secure-file-share/uploads]
$ hexedit passwordlist.txt.enc

(venv)-(kali@kali)-[~/Desktop/secure-file-share/uploads]
$ xxd passwordlist.txt.enc | head
00000000: 8000 9e3d 5207 d6d4 5c16 f0f9 1b95 bd58  ...=R...\.....X
00000010: 4adf 08c9 dc4d 425c 5f9e 86a9 f9fe f44a  J....MB\_\.....J
00000020: 99e1 d0b4 87fb e06e 520c 7f48 7d8d 7715  ....nR...H}.w.
00000030: a39e d89c 8e23 1e16 544f 79be 824a 66ce  ....#...TOy..Jf.
00000040: 992c 1fb5 286d 420d a2f3 4a39 b961 f94e  ,...(mB...J9.a.N
00000050: 09a1 54c1 a873 231c 3879 3df5 7baf a696  ..T..s#.8y=. {...
00000060: 9468 7205 b8c4 96b3 79c4 2e7c ce70 d35f  .hr.....y...|.p._
00000070: e27b 4488 fbc8 944d a709 5201 9cf1 5fbb  .{D....M..R...._
00000080: 02cf 1f5b 1289 2a1c c799 7f05 0894 ee38  ...[...*.....8
00000090: 1725 ca02 af98 d9ec f3d9 65e2 b6ab dce1  .%.....e.....
```

The integrity mechanism worked as designed. When a download was attempted, the decryption process failed and returned an **error 500** as illustrated in the screenshot, indicating that the authentication tag verification had not passed. This outcome confirmed that **the system is capable of reliably detecting unauthorized modifications**, thereby preventing the delivery of corrupted or malicious files to users.

```
File "/home/kali/Desktop/secure-file-share/venv/lib/python3.13/site-packages/Crypto/Cipher/_mode_gcm.py", line
567, in decrypt_and_verify
    self.verify(received_mac_tag)
    ~~~~~^~~~~~
File "/home/kali/Desktop/secure-file-share/venv/lib/python3.13/site-packages/Crypto/Cipher/_mode_gcm.py", line
508, in verify
    raise ValueError("MAC check failed")
ValueError: MAC check failed
127.0.0.1 - - [15/Aug/2025 09:42:21] "GET /download/2 HTTP/1.1" 500 -
```

Internal Server Error

The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.

8. Risks and Mitigations

While the system provides a strong baseline for secure file sharing, several risks remain that should be addressed in production environments:

- **Risk 1: Key Exposure**
 - **Description:** If the encryption key stored in the environment file (`.env`) is leaked, all encrypted files could be compromised.
 - **Mitigation:** Implement a secure key management system (KMS) such as AWS KMS, HashiCorp Vault, or Azure Key Vault. Regularly rotate keys and restrict access through role-based controls.
- **Risk 2: Insecure Transmission**
 - **Description:** Without proper HTTPS/TLS configuration, files in transit may be intercepted.
 - **Mitigation:** Deploy the Flask application behind HTTPS with TLS 1.2/1.3 enabled. Use a valid certificate from Let's Encrypt or a trusted CA.
- **Risk 3: Unauthorized Access**
 - **Description:** Currently, the system does not enforce user authentication, allowing any visitor to upload or download.
 - **Mitigation:** Add authentication and role-based access controls (RBAC). Each user should only be able to access their own encrypted files.

- **Risk 4: File Injection or Malware Upload**
 - *Description*: Attackers may attempt to upload malicious files to the server.
 - *Mitigation*: Enforce strict file type validation, limit maximum file size, and integrate malware scanning tools (e.g., ClamAV).
 - **Risk 5: Denial-of-Service (DoS) via Large Uploads**
 - *Description*: Excessively large files may be uploaded to exhaust server resources.
 - *Mitigation*: Configure upload size limits, rate limiting, and monitoring to detect anomalies.
-

9. Recommendations

To strengthen this system for real-world deployment, I recommend the following key improvements:

1. **Integrate Secure Key Management**: Move from static `.env` secrets to a centralized KMS.
 2. **Enable Strong Authentication**: Implement login functionality with salted password hashing and session management.
 3. **Audit Logging**: Maintain detailed logs of file uploads, downloads, and tampering attempts for forensic analysis.
 4. **Automated Backup & Recovery**: Encrypted files should be backed up securely to prevent accidental loss.
 5. **Continuous Security Testing**: Conduct penetration testing, fuzzing, and automated vulnerability scans regularly.
-

10. Conclusion

This internship project successfully delivered a **secure file sharing system** that emphasizes confidentiality, integrity, and controlled storage of sensitive data. Files are encrypted at rest with **AES-256 GCM**, ensuring both privacy and tamper detection. Testing demonstrated that files could be uploaded, securely stored, and retrieved without data loss, while tampered files were reliably detected and blocked.

Although the system is functional and secure in a controlled environment, further enhancements (key management, authentication, malware scanning) are required for production use. Overall, the project met its objectives and provided valuable hands-on experience with **cryptography, secure web development, and risk mitigation strategies** relevant to modern cybersecurity practices.

11. Appendices

This section provides selected code snippets, configuration references, and command examples used during the development and testing of the secure file sharing system.

11.1 Code Snippets

The following critical code excerpts demonstrate how encryption, decryption, and integrity verification are implemented in the system:

(a) Security Headers ensures protection against clickjacking, MIME sniffing, and cross-site scripting (XSS).

```
25 # Security headers
26 def set_security_headers(resp: Response) -> Response:
27     resp.headers["X-Content-Type-Options"] = "nosniff"
28     resp.headers["X-Frame-Options"] = "DENY"
29     resp.headers["Referrer-Policy"] = "no-referrer"
30     resp.headers["Content-Security-Policy"] = "default-src 'self'; style-src 'self'; script-src 'self'"
31     return resp
32
33 def load_index():
34     if not os.path.exists(INDEX_FILE):
35         return {}
```

(b) File Upload with AES Encryption demonstrates that every uploaded file is encrypted before being written to disk.

```
85         "size_bytes": 0,
86         "uploaded_at": datetime.datetime.utcnow().isoformat() + "Z",
87         "ext": ext,
88     }
89     try:
90         total = encrypt_stream_to_file(f.stream, stored_path, PASSPHRASE)
91         meta["size_bytes"] = total
92     except Exception as e:
93         flash(f"Encryption failed: {e}")
94         if os.path.exists(stored_path):
95             try: os.remove(stored_path)
96             except: pass
97         return redirect(url_for("index"))
98
99     idx = load_index()
100     idx[file_id] = meta
101     save_index(idx)
102     flash("Upload & encryption successful.")
103     return redirect(url_for("index"))
104
105 @app.get("/download/<file_id>")
106 def download(file_id):
107     idx = load_index()
108     if file_id not in idx:
109         return "Not found", 404
110     record = idx[file_id]
111     stored_path = record["stored_path"]
112     if not os.path.exists(stored_path):
113         return "File missing on server", 410
114
115     # Decrypt to a temp file (streaming to avoid memory spikes)
```

(c) File Download & Integrity Check Ensures that tampered or corrupted files cannot be retrieved.

```
118     decrypt_stream_to_file(stored_path, tmp_path, PASSPHRASE)
119 except IntegrityError:
120     if os.path.exists(tmp_path):
121         os.remove(tmp_path)
122     return "Integrity verification failed. The file may be corrupted or the key is wrong.", 409
123 except Exception as e:
124     if os.path.exists(tmp_path):
125         os.remove(tmp_path)
126     return f"Decryption failed: {e}", 500
127
128 # Send and then remove the temp file
129 try:
130     return send_file(tmp_path, as_attachment=True, download_name=record["original_name"])
131 finally:
132     try:
133         os.remove(tmp_path)
134     except Exception:
135         pass
```

(d) Crypto Utility: The `crypto_utils.py` module provides AES-based encryption and decryption, ensuring confidentiality and integrity of uploaded files.

```
crypto_utils.py X
secure-file-share > crypto_utils.py > decrypt_stream_to_file
1  import os, struct
2  from Crypto.Cipher import AES
3  from Crypto.Random import get_random_bytes
4  from Crypto.Protocol.KDF import scrypt
5
6  # Chunk size for streaming reads/writes
7  MAX_CHUNK = 64 * 1024
8
9  MAGIC = b"SFS1" # file format identifier
10 SALT_LEN = 16
11 NONCE_LEN = 12
12 TAG_LEN = 16
13
14 class IntegrityError(Exception):
15     pass
16
17 def _derive_key(passphrase: str, salt: bytes) -> bytes:
18     if not isinstance(passphrase, str):
19         raise ValueError("Passphrase must be a string.")
20     # scrypt: strong KDF; adjust N/r/p for your hardware/security needs
21     key = scrypt(passphrase.encode("utf-8"), salt, key_len=32, N=2**15, r=8, p=1)
22     return key
23
24 def encrypt_stream_to_file(in_stream, out_path: str, passphrase: str) -> int:
25     """
26     Encrypts data from a readable stream to an output file using AES-GCM.
27     File format:
28     [MAGIC(4)][SALT(16)][NONCE(12)][CIPHERTEXT...][TAG(16)]
```

11.2 Command-Line Tests

To validate encryption and integrity, the following commands were executed:

Upload a sample file

echo "This is a testfile" > test.txt

curl -F "file=@test.txt" <http://127.0.0.1:5000/upload>

curl -F "file=@test.txt" <http://127.0.0.1:5000/> -v (redirect)

Verify download

curl -OJ <http://127.0.0.1:5000/download/><file_id>

Compute SHA-256 checksum before and after

sha256sum testfile.txt

sha256sum downloaded_testfile.txt


```

(venv)-(kali@kali)-[~/Desktop/secure-file-share/uploads]
└─$ echo "This is a test file" > test.txt

(venv)-(kali@kali)-[~/Desktop/secure-file-share/uploads]
└─$ curl -F "file=@test.txt" http://127.0.0.1:5000/ -v
* Trying 127.0.0.1:5000...
* Connected to 127.0.0.1 (127.0.0.1) port 5000
* using HTTP/1.x
* POST / HTTP/1.1
> Host: 127.0.0.1:5000
> User-Agent: curl/8.14.1
> Accept: */*
> Content-Length: 218
> Content-Type: multipart/form-data; boundary=-----VuWQfCgYONr2f06Gi5L6aA
>
* upload completely sent off: 218 bytes
< HTTP/1.1 302 FOUND
< Server: Werkzeug/3.1.3 Python/3.13.5
< Date: Fri, 15 Aug 2025 14:04:17 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 199
< Location: /files
< Vary: Cookie
< Set-Cookie: session=eyJfZmxhc2hlcYI6W3siIHQI0IsibWVzc2FnZSI6IkpZbGUGdX8sb2FkZWQgYW5kIGVudY3J5cHRlZCBzdWNjZXRnZnVsbHkhIl19XX0.aJ8-YQ.QKsHuZ3PYutcgU6Ys38ffda-WnQ; HttpOnly; Path=/
< Connection: close
<
<!doctype html>
<html lang=en>
<title>Redirecting...</title>
<h1>Redirecting...</h1>
<p>You should be redirected automatically to the target URL: <a href="/files">/files</a>. If not, click the link.
* shutting down connection #0

```

11.3 Source Code Reference

The complete project codebase is hosted on GitHub. For verification purposes, reviewers may focus on the following:

- **Security Headers** → *app.py*
- **Upload & Encryption Logic** → *app.py*
- **Download & Integrity Enforcement** → *app.py*
- **Crypto Utilities** → *crypto_utils.py* (AES-GCM implementation & custom *IntegrityError*)
- **Configuration Management** → *.env* and *config.py*

This ensures that all reported findings are backed by auditable, working code.
