Security Overview – Encryption & Key Management

1. Purpose of This Document

This document explains the **security mechanisms** used in the Secure File Upload and Download Portal, with a focus on:

- Encryption methods used to protect files
- How encryption keys are generated and stored
- How data is protected both **at rest** and **in transit**

The goal is to ensure confidentiality, integrity, and secure handling of sensitive data.

2. Encryption Method Used

2.1 Advanced Encryption Standard (AES)

The system uses **AES (Advanced Encryption Standard)**, which is a **symmetric encryption algorithm** widely accepted as an industry standard.

Key characteristics:

- Same key is used for encryption and decryption
- Fast and efficient
- Approved by NIST
- Used in real-world applications (TLS, disk encryption)

2.2 AES Mode: AES-GCM

The implementation uses **AES in Galois/Counter Mode (GCM)**.

AES-GCM provides:

- **Confidentiality** – protects file contents
- **Integrity** – detects unauthorized modifications
- **Authentication** – ensures data has not been tampered with

Unlike basic AES modes (e.g., ECB or CBC), GCM is resistant to common cryptographic attacks.

3. Encryption Workflow

3.1 File Upload (Encryption at Rest)

1. User uploads a file via the web interface
2. The server reads the file as binary data
3. AES-GCM encrypts the file using a secret key
4. A unique **nonce** and **authentication tag** are generated
5. The encrypted data (nonce + tag + ciphertext) is stored on disk

🔒 **Result:** Stored files are unreadable without the AES key

3.2 File Download (Decryption on Demand)

1. User requests to download a file
2. The encrypted file is read from storage
3. AES-GCM verifies the authentication tag
4. If verification passes, the file is decrypted
5. The decrypted file is sent to the user

⚠ If the encrypted file is altered, decryption fails automatically

4. Key Management Strategy

4.1 AES Key Handling

- The AES key is **never hardcoded** in the source code
- The key is stored securely in an environment variable:
- `AES_KEY`
-
- The key is loaded at runtime using `python-dotenv`

This approach prevents accidental exposure of the encryption key.

4.2 Environment File (`.env`)

The `.env` file contains:

```
AES_KEY=<secure-random-key>
```

Security measures:

- `.env` is excluded using `.gitignore`
- Not uploaded to GitHub

- Not visible in the web interface
- Only accessible to the server runtime

5. Protection of Data in Transit

To secure data while it is being transmitted between the client and server:

- HTTPS is enabled using Flask's ad-hoc SSL certificate
- All uploads and downloads occur over an encrypted TLS channel

🔐 This prevents:

- Man-in-the-middle attacks
- Packet sniffing
- Data interception

6. GitHub & Source Code Security

Sensitive files and folders are excluded from version control:

```
.env

__pycache__/

static/files/
```

Benefits:

- Prevents leakage of encryption keys
- Prevents exposure of user-uploaded data
- Ensures only clean source code is published

7. Security Strengths

✔ Industry-standard encryption (AES-GCM)
✔ Secure key storage using environment variables
✔ Encrypted data storage (at rest)
✔ HTTPS encryption (in transit)
✔ Proper Git hygiene

8. Security Limitations

- Single AES key used for all files
- No user-based authentication

- No automatic key rotation
- Not designed for production-scale deployment

These limitations are acceptable for a learning-focused implementation.

9. Future Security Improvements

- Per-user encryption keys
- Hardware Security Module (HSM) or key vault
- Key rotation policies
- Multi-factor authentication
- Secure access logging

Note : My apologies not being able to give a video demonstration with audio , there was a issue on my end because of which i couldnt record audio