## ****SECURITY OVERVIEW DOCUMENT****

### ****1. Introduction****

In the digital age, the ability to share files securely over the internet is critical. Sensitive information can be exposed if transmitted or stored without proper protection, leading to privacy violations, identity theft, and compliance breaches. To address these concerns, we developed a **Secure File Sharing System** using **Python Flask** and **AES encryption** (via the PyCryptodome library).

This document outlines the core security mechanisms built into the system, explains how data is protected both at rest and in transit, and provides a detailed overview of the encryption implementation, file handling, key management, and recommended practices to further harden the application.

### ****2. System Architecture and Workflow****

The secure file sharing system consists of three main components:

****1.Client Interface (HTML/Flask UI)**:**

* Allows users to upload files via a web interface.
* Uses secure POST requests to transfer files to the server.

**2.Server Application (Flask)**:

* Handles file uploads and downloads.
* Encrypts files using AES before storage.
* Decrypts files upon authorized download.

**3.Encryption Module (PyCryptodome)**:

* Implements AES encryption in ECB mode (basic version).
* Pads data to match AES block size requirements.
* Ensures files are encrypted at rest.

**3. Encryption Implementation**

#### ****3.1 Algorithm Used****

* **AES (Advanced Encryption Standard)**:

1. Symmetric key algorithm approved by NIST.
2. Block size: 128 bits.
3. Key size used: 128 bits (16 bytes).
4. Mode of Operation: **ECB (Electronic Codebook)** for the base version, though other modes are recommended for production (CBC, GCM).

#### ****3.2 Padding and Block Alignment****

* AES operates on fixed-size blocks.
* To encrypt files of arbitrary size, we pad the data manually to a multiple of 16 bytes.
* Padding scheme used: simple space-padding (b' ') for simplicity.
* def pad(data):

return data + b' ' \* (16 - len(data) % 16)

#### ****3.3 Encryption Flow****

* Read the uploaded file in binary.
* Pad the data.
* Encrypt with AES using a symmetric key.
* Save the encrypted file to disk and delete the original.

#### ****3.4 Decryption Flow****

* Locate the encrypted file.
* Read and decrypt the binary data.
* Remove padding (strip spaces).
* Serve the decrypted file to the user for download.

### ****4. File Handling and Storage Security****

#### ****4.1 File Upload Security****

1. Uploaded files are saved temporarily and immediately encrypted.
2. Original unencrypted files are deleted after encryption.
3. MIME type and file extension checks can be added to prevent malicious uploads (e.g., .php, .exe)

#### ****4.2 Encrypted File Storage****

* All files in the uploads/ folder are stored encrypted.
* Decrypted files are written to a separate decrypted/ folder only during download and are automatically deleted afterwards (optional clean-up script can be used).

#### ****4.3 Access Control****

1. Flask route /download/<filename> allows file retrieval.
2. **Future improvements:**
3. Implement user authentication (JWT or session tokens).
4. Check file ownership before allowing downloads.

### ****5. Key Management****

#### ****5.1 Symmetric Key Storage****

* In this implementation, the AES key is hardcoded for simplicity:
* key = b'ThisIsASecretKey'
* **Drawbacks** of hardcoded keys:
* Anyone with access to the code can decrypt files.
* Not scalable for multiple users or file-specific keys.

#### ****5.2 Best Practices for Key Management****

1. Store keys in environment variables or external config files:
2. import os
3. key = os.environ.get('AES\_KEY').encode()
4. Use a **key vault** or **HSM (Hardware Security Module)** in production.
5. Rotate keys periodically and track key usage.
6. Consider using **per-user keys** or **per-file keys** with metadata.

### ****6. Security Threats & Mitigations****

| **Threat** | **Risk** | **Mitigation Strategy** |
| --- | --- | --- |
| Plaintext storage | Sensitive data exposed if disk is accessed | Files are AES-encrypted at rest |
| Data interception during transit | Man-in-the-middle attacks | Use HTTPS (TLS) in production |
| Hardcoded key exposure | Anyone with source code can decrypt files | Move key to env variable or external config |
| File injection | Malicious files uploaded | Validate file types and sanitize filenames |
| Unauthenticated access to download | Anyone can guess file name and download | Add user login/authentication layer |
| No expiration for decrypted files | Decrypted files may linger | Use temp storage and delete after serving |

### ****7. Recommendations and Future Enhancements****

To make the system truly production-ready and enterprise secure, we recommend:

**1.Use AES in CBC or GCM Mode**

* ECB mode is deterministic and may leak data patterns.
* CBC introduces IV (Initialization Vector) for added randomness.
* GCM adds authentication tags for tamper detection.

**2.File Authentication**

* Add file hash or HMAC to detect tampering or modification.

**3.User Authentication System**

* Add login system using Flask-Login, JWT, or OAuth.
* Associate each file with a user and limit access

**4.Logging and Monitoring**

* Log file uploads, downloads, and failed access attempts
* Use a security information and event management (SIEM) tool

**5.Rate Limiting & CSRF Protection**

* Protect endpoints from abuse using Flask-Limiter.
* Use Flask-WTF for CSRF token validation.

**6.Clean-Up Mechanism**

* Periodically delete decrypted files from the decrypted/ directory.
* Set expiration for temporary download links.

### ****8. Conclusion****

The Secure File Sharing System provides a foundational implementation for encrypting files at rest and ensuring safe handling of uploaded and downloaded documents. The current system uses AES with a symmetric key and basic Flask-based routing to demonstrate secure file operations.

However, for enterprise-grade deployment, additional security layers such as proper key management, authentication, HTTPS, and advanced encryption modes must be integrated. This project serves as a starting point for learning and building real-world secure systems, promoting awareness of best practices in encryption and web application security.