Security Overview Document

Project: Secure File Sharing System (Flask + AES)

Intern: Zikra Abdusemed Mohammed

Date : Nov 8 ,2025

Introduction

The **Secure File Sharing System** is a Flask-based web application designed to enable users to upload and download files securely.
The main focus of this system is **data protection** — ensuring that all uploaded files are encrypted before storage and safely decrypted when retrieved.

The project uses the **Advanced Encryption Standard (AES)** algorithm to protect files both **at rest** (while stored on the server) and **in transit** (during upload/download operations).

Encryption Method

AES (Advanced Encryption Standard)

The application uses **AES-256**, a symmetric encryption algorithm widely adopted for government and enterprise-grade data protection.

- **Symmetric Encryption:** The same key is used for both encryption and decryption.
- **Key Size:** 256 bits (32 bytes), offering a high level of security.
- **Block Size:** 128 bits (16 bytes).
- **Mode of Operation: CBC (Cipher Block Chaining)** — each block of plaintext is XORed with the previous ciphertext block before being encrypted.

Advantages of AES-CBC:

- Ensures that identical plaintext blocks produce different ciphertexts.
- Prevents simple pattern analysis in encrypted data.
- Well-tested and supported in standard crypto libraries.

Implementation Library:
`PyCryptodome` — a Python library providing cryptographic primitives including AES, hashing, and random number generation.

Encryption Process Overview

Below is the flow of how files are encrypted and decrypted in the system:

Encryption Steps

1. User uploads a file via the web interface.
2. The system generates or retrieves the AES key.
3. A **random Initialization Vector (IV)** (16 bytes) is created using `get_random_bytes()`.
4. The file content is padded to match the AES block size (16 bytes).
5. The AES cipher encrypts the file using **AES-256-CBC** mode.
6. The IV is prepended to the encrypted file (so it can be reused for decryption).
7. The original plaintext file is deleted from disk immediately after encryption.

Decryption Steps

1. The user requests a download by providing the encrypted file name.
2. The system reads the encrypted file and extracts the first 16 bytes as the IV.
3. The AES key and IV are used to decrypt the ciphertext.
4. The decrypted plaintext file is temporarily stored and then served to the user for download.

Key Management

Key Generation

- A 256-bit (32-byte) AES key is generated once using `get_random_bytes(32)`.

The key is saved locally in a secure file: `/keys/aes.key`

- If the key file doesn't exist, the system automatically creates it.

Key Storage

- The `/keys/` directory is **excluded from Git tracking** (`.gitignore` file).
- Only the server-side process (Flask app) has read access to this directory.
- Keys are stored in **binary format**, not human-readable text.

Key Usage

- The same AES key is used to encrypt and decrypt files.
- The key is never sent over the network or exposed in logs.

Key Security Recommendations

To improve key security in future versions:

- Store keys using **environment variables** or a **key vault service** (e.g., AWS KMS, HashiCorp Vault).
- Use **password-based key derivation** (PBKDF2) to generate keys from user credentials.
- Implement **key rotation** for long-term security.

File Handling Security

Security Goal	Implementation
Confidentiality	Files are encrypted using AES-256 before storage
Integrity	Each file uses a unique IV; data corruption breaks decryption
No Plaintext Storage	Plaintext files are deleted after encryption
Key Secrecy	Keys stored securely, never exposed to clients
Secure Transfer	Flask handles upload/download via HTTPS (recommended)

Security Testing Summary

Test Case	Description	Result
File upload followed by download	Ensures encryption & decryption are correct	✅ Pass
Open encrypted file in text editor	Confirms data is unreadable without AES key	✅ Pass
Attempt decryption with wrong key	Fails to produce valid file	✅ Pass
Verify key excluded from repository	`.gitignore` confirmed functional	✅ Pass
Tamper with `.enc` file	Causes decryption failure, proving integrity	✅ Pass

---

Potential Vulnerabilities & Mitigation

Potential Risk	Description	Mitigation
Key exposure	If `/keys/aes.key` is accessed by an attacker	Restrict file permissions; consider environment variables
Man-in-the-middle attacks	During upload/download on HTTP	Use HTTPS for secure transfer
File tampering	If encrypted files are modified	Add integrity check (e.g., SHA-256 hash)
Key reuse	Same key encrypts multiple files	Future improvement: generate unique keys per session/file
Plaintext persistence	Temporary plaintext file not deleted	Code removes unencrypted files after encryption

---

Future Security Enhancements

1. **User Authentication:** Add login system (Flask-Login) for controlled access.
2. **Key Rotation:** Periodically change AES keys to limit exposure risk.
3. **Hash Validation:** Use SHA-256 to verify decrypted file integrity.
4. **Password-Based Encryption:** Derive keys from user passphrases (PBKDF2).
5. **Audit Logging:** Track file operations and user actions.
6. **Transport Security:** Enforce HTTPS with SSL certificates.

Conclusion

This Secure File Sharing System demonstrates core principles of **data confidentiality, key management, and encryption best practices** using AES.
It simulates real-world secure file handling workflows — suitable for sectors like **healthcare, legal, or corporate environments**, where sensitive information must remain protected at all times.

By implementing AES-256 encryption and safe key management, this project establishes a strong foundation for more advanced security systems in the future.

Walkthrough video link

FUTURE_CS_03_VIDEO