SECURITY ARCHITECTURE & IMPLEMENTATION GUIDE

Project: Secure File Sharing System (AES-256)

Date: December, 2025.

Author: Zannu Opeyemi Emmanuel

1. Task Summary

This document outlines the security architecture of the Secure File Sharing System designed for the Future Interns Cybersecurity program. The system is a web-based application that ensures the confidentiality and integrity of user files through client-authenticated encryption standards.

The primary security goal is to ensure that files stored on the server are mathematically indecipherable to unauthorized entities, including system administrators or attackers who might gain physical access to the storage disk.

2. Cryptographic Standards

The core security mechanism is based on the **Advanced Encryption Standard (AES)**.

- **Algorithm:** AES-256 (256-bit key length).
- **Mode of Operation:** Galois/Counter Mode (GCM).
- **Library:** PyCryptodome (Python).

Why AES-GCM?

We selected GCM (Galois/Counter Mode) over older modes like CBC (Cipher Block Chaining) or ECB (Electronic Codebook) for the following reasons:

1. **Authenticated Encryption (AEAD):** GCM provides both data confidentiality (secrecy) and data integrity (authenticity) simultaneously. It generates an "Authentication Tag" alongside the ciphertext.
2. **Performance:** GCM is parallelizable, making it significantly faster for high-throughput file uploads compared to CBC.
3. **Attack Resistance:** It eliminates padding oracle attacks that plague CBC mode.

3. System Architecture & Data Flow

The diagram below illustrates how data is transformed from a readable file into a secure, encrypted object before storage.

3.1 Upload Process (Encryption)

When a user uploads a file, the following sequence occurs in memory (RAM) before the file ever touches the disk:

1. **Nonce Generation:** A unique 16-byte *Nonce* (Number used once) is cryptographically generated. This ensures that even if the same file is uploaded twice, the encrypted output will look completely different.
2. **Encryption:** The file data is encrypted using the 256-bit Master Key and the Nonce.
3. **Tagging:** A 16-byte *MAC (Message Authentication Code)* tag is generated. This tag is a mathematical checksum of the encrypted content.
4. Storage: The system concatenates these elements into a single blob:

[NONCE (16 bytes)] + [TAG (16 bytes)] + [CIPHERTEXT]

3.2 Download Process (Integrity Check)

When a download is requested:

1. **Parsing:** The system reads the Nonce and Tag from the file header.
2. **Verification:** The AES-GCM engine uses the Key, Nonce, and Tag to verify the Ciphertext.
3. **Fail-Safe:** If even a single bit of the file has been altered (tampered with) on the disk, the verification fails, and the system **refuses** to decrypt the file.

4. Key Handling & Management

Proper key management is critical to the security of the cryptosystem.

- **Key Generation:** A 256-bit symmetric key is generated using Crypto.Random.get_random_bytes(32). This ensures high entropy and unpredictability.
- **Storage (Prototype):** For this internship simulation, the key is stored in a local file (master.key) which is strictly isolated from the web root and excluded from version control (.gitignore).
- **Production Plan:** In a live environment, this key would be migrated to a cloud-based Key Management Service (AWS KMS or Azure Key Vault) to allow for key rotation and strict access logging.

5. Threat Mitigation

Threat Scenario	**Mitigation**	**Status**
Server Theft	If an attacker steals the hard drive, they cannot read the files without the key.	✅ Protected
Data Tampering	If an attacker modifies an encrypted file, decryption fails (Integrity Check).	✅ Protected
Network Sniffing	If an attacker intercepts the file *during* upload.	⚠️ Requires TLS (HTTPS)