Secure File Sharing System

Name: Oyekale Olamide David

Task 3: Secure File Sharing System (Flask + AES)

Program: Future Interns Cybersecurity Internship

Date: September 2025

Project Summary

The Secure File Sharing System was developed to support safe file upload, storage, and retrieval through encryption. It employs AES-256 to ensure confidentiality, preventing unauthorized access to stored files. While the system provides a strong baseline, it demonstrates notable weaknesses in key management, access control, transport security, and integrity validation.

This whitepaper presents an evaluation of the system's current security model, outlines vulnerabilities, analyzes threats, and offers recommendations for strengthening the solution to production-grade standards.

1. Introduction

File sharing platforms are frequent attack targets due to the sensitive nature of the data they handle. Without proper safeguards, they expose users and organizations to breaches, theft of intellectual property, and compliance failures.

The project introduces a prototype file sharing system for educational purposes. It relies on symmetric encryption (AES) for file confidentiality but lacks several layers of security found in real-world deployments.

The objectives of this security overview document are to:

1. Assess the existing implementation.
2. Identify potential risks.
3. Recommend improvements aligned with modern cybersecurity practices.

2. Technical Overview

2.1 System Architecture

The system is composed of the following components:

• Frontend: A Bootstrap-based web interface for file uploads and downloads.
• Backend: A Flask application managing file processing and encryption.
• Encryption Engine: AES-256-CBC implemented using Python's cryptography library.
• Storage: Encrypted files stored in the uploads/ directory; plaintext files are never persisted to disk.

2.2 Encryption Process

• A hardcoded 256-bit symmetric key is embedded in the application.
• A random Initialization Vector (IV) is generated for each file.
• Files are padded, encrypted with AES-CBC, and stored as [IV] + [Ciphertext].
• For decryption, the IV is extracted and AES reverses the process.

2.3 Decryption Options

The system supports two decryption methods:

• Direct Decryption: File decrypted and streamed to the user on demand.

• File Reconstruction: File temporarily written to a decrypted/ directory before being served.

3. Security Strengths

• Strong Cryptography: AES-256-CBC is a widely trusted standard.

• Randomized IV: Guarantees ciphertext uniqueness, even for identical files.

• No Plaintext Storage: Files remain encrypted on disk at all times.

• Simplicity: Minimal complexity reduces misconfiguration risks.

4. Threat Model

Potential adversaries include:

• Network Attackers: Intercepting unencrypted traffic (e.g., MITM).

• Malicious Users: Exploiting the system to access files they do not own.

• Server Intruders: Compromising the backend host.

• Insider Threats: Administrators or developers with access to encryption keys.

Assets at risk:

• Uploaded files (confidentiality).

• User data/identities (if authentication is added).

• Encryption keys (core of security).

5. Risk Analysis

5.1 Key Management

Current State: Encryption key hardcoded in source code.
Risk: If leaked, all encrypted files are exposed.
Impact: Catastrophic.

5.2 Transport Security

Current State: Operates over HTTP.
Risk: Files, credentials, and keys can be intercepted.
Impact: High.

5.3 Integrity & Authenticity

Current State: AES-CBC ensures confidentiality but not integrity.
Risk: Attackers can modify ciphertext undetected.
Impact: Medium.

5.4 Access Control
Current State: No authentication or authorization.
Risk: Any user can upload or download files.
Impact: High.

5.5 Operational Security
Current State: Runs on Flask's development server with minimal monitoring.
Risk: Weak hosting security in production.
Impact: Medium.

6. Recommendations

6.1 Cryptography
• Replace AES-CBC with AES-GCM (adds confidentiality + integrity).
• If GCM unavailable, use HMAC (SHA-256).

6.2 Key Management
• Employ a Key Management Service (AWS KMS, HashiCorp Vault).
• Rotate keys periodically.
• Consider per-user encryption keys.

6.3 Transport Security
• Deploy behind Nginx with TLS 1.2+ enforced.
• Redirect all HTTP traffic to HTTPS.
• Use strong cipher suites.

6.4 Authentication & Access Control
• Add user accounts secured with bcrypt/argon2.
• Implement Role-Based Access Control (RBAC).
• Restrict file access to respective owners.

6.5 Monitoring & Logging
• Enable detailed audit logs for uploads/downloads.
• Monitor anomalies in key usage and server activity.
• Integrate with IDS/IPS tools.

6.6 Deployment Security
• Replace Flask dev server with Gunicorn/uWSGI.
• Apply firewall rules and OS hardening.
• Conduct periodic penetration testing.

7. Future Improvements

• Integrate Multi-Factor Authentication (MFA).
• Add file integrity verification with digital signatures.
• Explore client-side end-to-end encryption.
• Extend usability with desktop and mobile clients.

8. Conclusion

The Secure File Sharing System highlights the role of encryption in safeguarding sensitive information. Although AES-256 provides robust confidentiality, shortcomings in key management, access control, and transport security introduce serious risks.

To achieve production readiness, the system must adopt a layered defense strategy: secure key lifecycle management, strong authentication, TLS-enforced transport, integrity checks, and continuous monitoring. With these improvements, the platform can evolve into a reliable, enterprise-grade solution for secure digital asset protection.