Secure-File-Sharing-System

Executive Summary

This report outlines the development and functionality of a secure file sharing system. The primary objective was to build a web application that enables users to upload and download files while ensuring a high level of data security. The application uses a symmetric encryption algorithm, AES-256, to protect files at rest and in transit. The project demonstrates core skills in web development, encryption implementation, and secure file handling.

Technical Overview

The system is built using the **Python Flask** web framework. The core security functionality is provided by the **PyCryptodome** library.

Key Components:

- **Flask Application (`app.py`):** Handles all web routing, file uploads, and downloads.
- **File Storage:** Files are not stored in their original form. A temporary `uploads` folder is used for incoming files before they are encrypted and moved to the secure `encrypted` folder.
- **HTML Interface (`index.html`):** A simple, clean user interface allows for file selection and download links.

3. Security Implementation

The application was designed with a "security-first" approach.

3.1. Encryption Algorithm The system uses the **Advanced Encryption Standard (AES)** with a key size of **256 bits** in **Cipher Block Chaining (CBC) mode**. This is a widely-used and robust encryption standard.

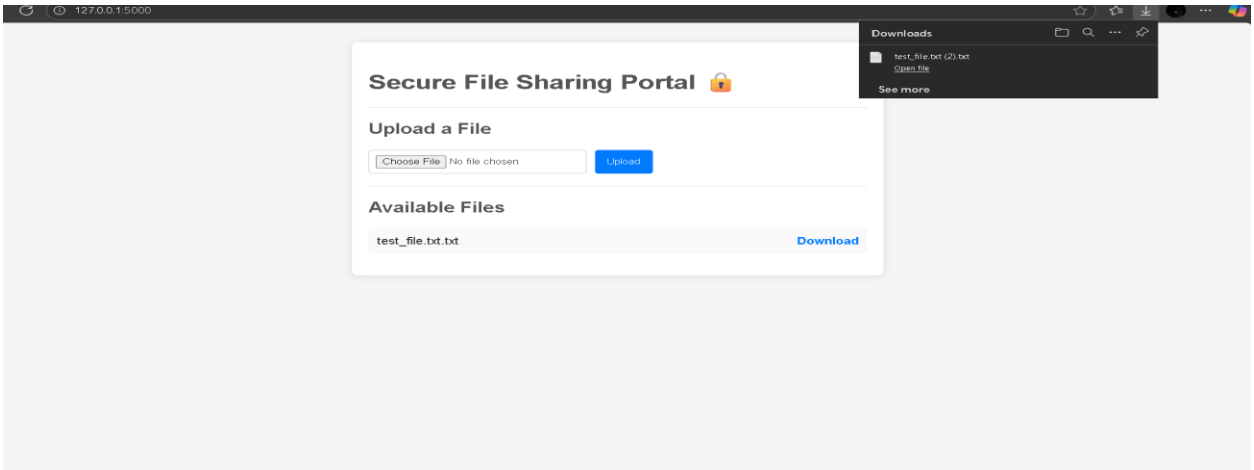
3.2. Data at Rest Protection When a file is uploaded, the following process ensures data is never stored unencrypted:

1. The file is temporarily saved to the `uploads` folder.
2. An Initialization Vector (**IV**) and a randomly generated key are used to encrypt the file's content.
3. The IV and ciphertext are combined and written to a new file in the `encrypted` folder.
4. The original file in the `uploads` folder is immediately deleted using `os.remove()`.

3.3. Data in Transit Protection Files are also protected during download. When a user requests a file, the application:

1. Reads the encrypted file.
2. Decrypts the file's content in memory.
3. Sends the decrypted data as a stream to the user's browser.
4. A temporary decrypted file created for serving the download is **automatically deleted** after the transfer is complete, preventing it from being left on the server.

3.4. Key Management For this project, a hardcoded secret key is used. In a production environment, this would be a major security risk. A more robust solution would involve a **Key Management System (KMS)** and proper user authentication to securely manage and retrieve encryption keys.



Uploading and downloading (AES encryption)

4. Identified Vulnerabilities and Mitigation

- **Vulnerability:** The lack of a user authentication system.
  - **Impact:** Any user with the URL can access and download any encrypted file on the server.
  - **Mitigation:** Implement a user authentication system (e.g., username/password, OAuth) and a role-based access control (RBAC) mechanism. This would restrict who can upload and access specific files.
- **Vulnerability:** Hardcoded encryption key.
  - **Impact:** If an attacker gains access to the source code, they can decrypt all files.
  - **Mitigation:** Store the encryption key securely in a Key Management System (KMS) or use environment variables, and do not commit it to the repository.

Conclusion

The secure file sharing system successfully meets the project's core requirements by implementing robust AES-256 encryption. The project highlights a strong understanding of web security principles and practical application development. While the current version is a proof of concept, the identified vulnerabilities and proposed mitigation strategies demonstrate an awareness of the steps required to build a production-ready, highly secure application.