SECURE FILE SHARING SYSTEM

CYBERSECURITY INTERNSHIP TASK-03

FUTURE INTERNS

Title: Secure File Sharing System

Intern Name: Gagana G A

Date: 19/06/2025

INTRODUCTION

In today's digital landscape, secure data transfer is paramount—particularly in sensitive sectors such as healthcare, law, and finance. As part of my cybersecurity internship with Future Interns, I designed and implemented a Secure File Sharing System that allows users to upload and download files safely, ensuring data confidentiality through robust encryption mechanisms.

OBJECTIVE OF THE TASK

The main goals of the task were:

- To develop a basic but secure web portal for file sharing.
- To implement encryption logic for uploaded files using AES.
- To ensure files are safely decrypted only when downloaded.
- To securely manage and handle encryption keys using environment variables.
- To simulate real-world secure data sharing for internship deliverables.

TOOLS AND TECHNOLOGIES USED

Backend: Node.js + Express.js
Encryption: Node.js Crypto module (AES-256)
File Handling: Multer (middleware for handling multipart/form-data)
Deployment: Localhost (for development)

Step 1: NodeJS installation

Download Node.js

- Visit the official Node.js website: https://nodejs.org
- Download the **LTS (Long Term Support)** version suitable for your operating system (e.g., Windows Installer .msi file).

Install Node.js

- Run the downloaded installer.
- Follow the setup wizard:
  - Accept the license agreement.
  - Choose the installation path (default is usually C:\Program Files\nodejs\).
  - Select the components (default selection is recommended).
  - Confirm and install.

Verify the Installation

After installation, open **Command Prompt** or **PowerShell** and run:
bash
node -v
npm -v
These commands will return the installed versions of **Node.js** and **npm** (Node Package Manager), confirming successful installation.

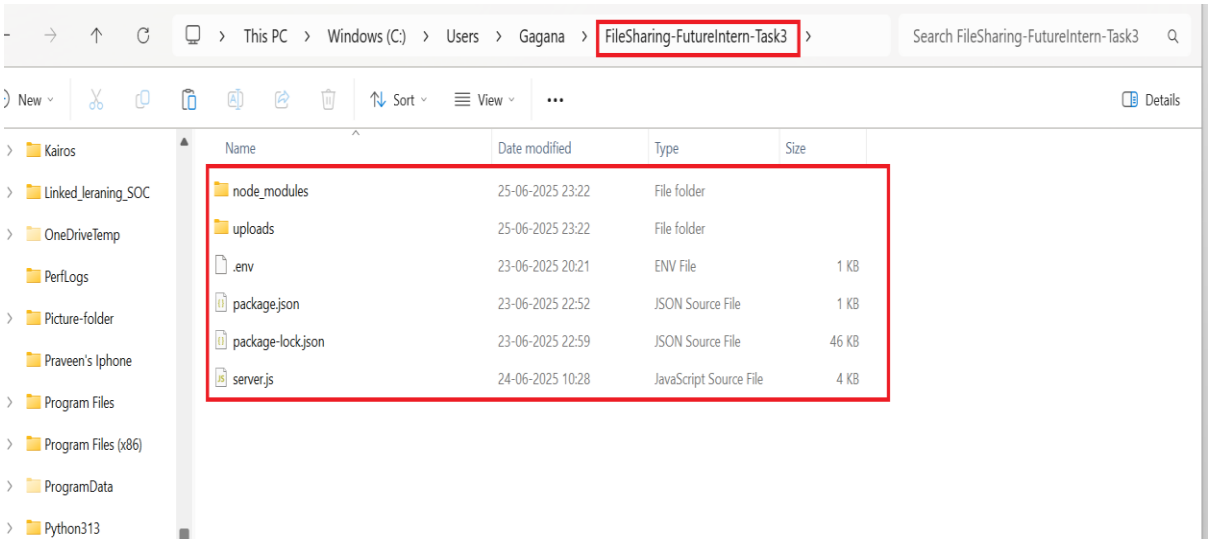
Initialize Project

Navigate to your project folder and initialize a Node.js project:
bash
mkdir secure_file_share
cd secure_file_share
npm init -y: This will generate a package.json file to manage dependencies.

Step 2: Creating the Project Structure

A structured project folder named FileSharing_FutureInern_Task3 was established, encompassing the following key components:

- **server.js:** Initializes Express, connects routes and DB.
- **.env:** Store sensitive values like encryption key, MongoDB URI.
- **uploads:** Folder where AES-encrypted files are stored.
- **package. Json:** The project name, version, and description.
- **package_lock.json :** Records the **exact versions** of all installed packages and their dependencies.
- **node_modules:** Stores the **actual code** for third-party libraries your project uses (like express, mongoose, crypto, etc.).



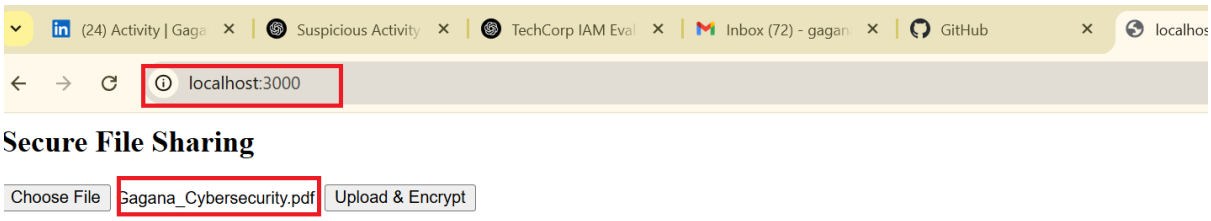
Step 3: Developing AES Encryption Logic

Process Overview:

1. **File Encryption on Upload**:
   - When a user uploads a file, it is passed through an AES encryption stream before being saved to disk.
   - The encrypted file is stored in a secure directory with limited access.
2. **File Decryption on Download**:
   - Upon download request, the encrypted file is read, decrypted in real-time, and served to the user securely.
   - This ensures the file is only available in its original form during transit and never stored in plaintext.

Step 4: Testing the application

To ensure the secure file sharing system functions as intended and meets security objectives, the application was tested across the following areas



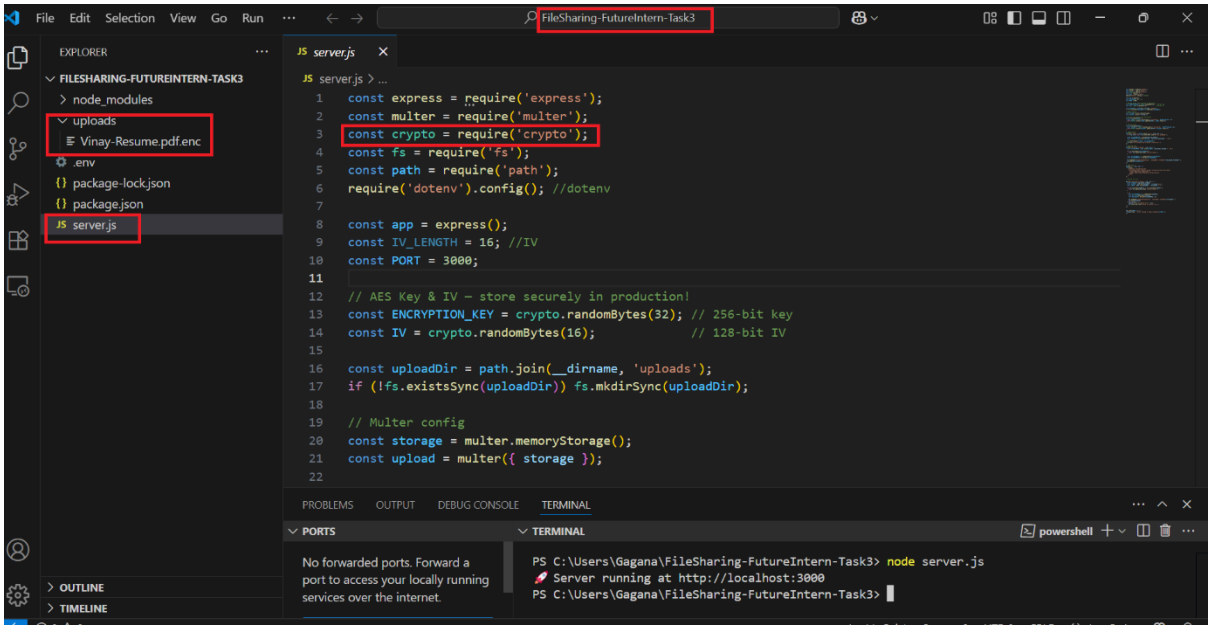
1. File Upload Functionality

- Uploaded different file types (e.g., .txt, .pdf, .jpg) through the UI and Postman.
- Verified that files were **encrypted** before being saved to the server using AES encryption.

2. File Download and Decryption

- Downloaded uploaded files to confirm they were **successfully decrypted** and restored to their original format.
- Validated file integrity by comparing original and decrypted files.

3.The following diagram demonstrates that upon successful upload, the file is securely stored in the designated uploads directory on the server.



Step 5: After uploading file Encrypted

The above message indicates that:

- The file transfer to the server was completed without errors.
- Encryption was applied using a secure AES algorithm.
- The file is now stored securely and cannot be read without decryption.

SECURITY MEASURES IMPLEMENTED

To ensure the safe transfer and storage of user files, the following security mechanisms were integrated into the system:

1. **AES Encryption (Advanced Encryption Standard)**
   All files uploaded by users are **encrypted using AES-256** before being saved on the server.
   During download, files are **decrypted on the fly**, ensuring data confidentiality during storage and transmission.
2. **Secure File Storage**
   Encrypted files are stored in a protected directory, inaccessible through the public web interface.
   Directory listing and direct file access were disabled to prevent unauthorized access.
3. **Key Management**
   Encryption keys are securely stored within the server environment and are **not hardcoded** in the source code.
   Environment variables were used to manage sensitive data like secret keys.
4. **Input Validation**
   Filenames and user inputs are sanitized to prevent **injection attacks** and **directory traversal vulnerabilities**.
5. **HTTPS (optional/ready for implementation)**
   The application can be configured to run over **HTTPS** to ensure secure data transmission over networks.
6. **Error Handling**
   Sensitive information is never exposed in error messages.
   Proper response codes are returned to avoid information leakage.
7. **Access Control (basic)**
   Only authorized users can upload or download files (can be extended to include authentication in future versions).

Hands-on Experience Gained

During the development of the **Secure File Sharing System**, I gained valuable practical experience in both **backend development** and **security implementation**. Key areas of hands-on learning include:

File Upload and Download Handling

- Learned how to configure and manage file uploads using **Multer** in a Node.js/Express environment.
- Implemented secure file retrieval mechanisms ensuring users can safely download stored files.

AES Encryption and Decryption

- Gained experience using the **crypto module** in Node.js to:
  - Encrypt files using **AES-256** before saving them to disk.
  - Decrypt files on demand during download, maintaining data confidentiality.

Secure Project Architecture

- Designed a modular backend structure for scalability and security.
- Used **environment variables (.env)** to securely manage secret keys and configurations.

Node.js Project Setup

- Practiced setting up a complete Node.js application from scratch:
  - Initialized with npm init
  - Installed required dependencies
  - Managed folder structures, routes, and server files

Postman API Testing

- Used **Postman** to simulate upload/download requests and verify encryption and decryption processes.
- Understood the importance of HTTP status codes and secure API responses.

Security Best Practices

- Applied basic security measures such as:
  - Input validation
  - Directory protection
  - Safe key handling
- Recognized potential vulnerabilities and how to mitigate them in file handling systems.

Conclusion:

The **Secure File Sharing System** was successfully developed to facilitate safe upload and download of files with a strong emphasis on data protection. By implementing **AES encryption**, we ensured that files are securely encrypted before being stored and decrypted only upon retrieval. This safeguards data both **at rest and in transit**, mitigating the risk of unauthorized access or data breaches.
The project demonstrates how encryption techniques can be seamlessly integrated into a Node.js-based file management system, making it a practical solution for secure file transfers in real-world environments.