Internship:
Future Interns –
Cybersecurity Internship
Intern Name:
Dakshayani Sindiri

TASK 3 – SECURE FILE SHARING SYSTEM

GitHub Repository: https://github.com/dakshayanisindiri-98/FUTURE_CS_03
LinkedIn Profile: https://www.linkedin.com/in/dakshayani-sindiri-a55037302
Date of Completion: SEP-2025

Table of Contents

Executive Summary

This document is an extended incident report for Task 3 of the Future Interns Cybersecurity Internship. The project implements a Secure File Sharing System using Node.js and AES-256-CBC encryption. Files uploaded to the server are encrypted before storage and decrypted only on explicit request. The system includes a simple frontend, Postman-tested APIs, and secure handling of secrets using environment variables. This report details design, implementation, testing, challenges, security considerations, and recommended future enhancements.

Objectives

The project objectives were to:

- Design a backend service capable of receiving file uploads and encrypting files at rest.
- Provide secure decryption and download functionality.
- Use AES-256-CBC for strong symmetric encryption.
- Protect cryptographic keys using environment variables and a proper .gitignore.
- Validate functionality using Postman and a browser-based frontend.
- Document the solution and produce deliverables for submission.

System Components & Architecture

The Secure File Sharing System consists of three main components:
1. Frontend: A small static website (index.html, style.css, script.js) served from the Express server. Provides file upload UI and a decryption/download form.
2. Backend: Node.js + Express server (server.js) handling file uploads via Multer, performing AES encryption and decryption using Node's crypto module, and returning file responses.
3. Storage: Local filesystem folders for encrypted files (uploads/) and temporary decrypted files (decrypted/).

Data Flow:

User -> Frontend -> POST /upload -> Server (Multer) -> Encrypt -> Save uploads/*.enc -> Respond with filename.
User -> Frontend or Postman -> POST /decrypt { filename } -> Server -> Decrypt -> Temporary decrypted file -> Download -> Delete temporary file.

Folder Structure

```
secure-file-share/            # Root project folder
│
├── public/                   # Frontend files
│   ├── index.html              # Main frontend page
│   ├── script.js             # Handles file upload & decrypt logic
│   └── style.css               # Styling for frontend
│
├── uploads/                   # Encrypted files (auto-created)
│   └── [filename].enc
│
├── decrypted/                  # Temporary decrypted files (auto-created)
│   └── decrypted-[timestamp].txt
│
├── screenshots/                # Screenshots for documentation
│   ├── upload-response.png
│   ├── decrypt-response.png
│   └── postman-testing.png
│
├── .env                       # Environment variables (NOT uploaded to GitHub)
├── .gitignore                  # Files/folders to be ignored by Git
├── README.md                        # Project documentation
├── server.js                  # Main Node.js backend file
├── package.json                   # Dependencies & scripts
└── package-lock.json                # Auto-generated dependency lock file
```

Explanation

- public/ → Contains the UI files (HTML, CSS, JS).
- uploads/ → Stores encrypted files after upload.
- decrypted/ → Stores temporary decrypted files (auto-deleted after download).
- screenshots/ → Screenshots for GitHub and documentation.
- .env → Your secret keys and configuration (never pushed to GitHub).
- .gitignore → Ensures uploads/, decrypted/, node_modules/, and .env stay private.
- server.js → Core backend logic for encryption, decryption, and routes.
- package.json → Tracks Node.js dependencies like express, multer, etc.

Encryption Design

Encryption algorithm: AES-256-CBC.
Key length: 32 bytes (256 bits). IV length: 16 bytes (128 bits).
Key and IV are loaded from environment variables (ENCRYPTION_KEY and IV). For production, a secure secret manager is recommended.

Important implementation notes:
- Both encryption and decryption must use the exact same algorithm, key, and IV.
- Files are processed using streaming (createReadStream / createWriteStream) piped through

cipher/decipher to avoid loading entire files into memory.
- Temporary decrypted files are deleted after download to minimize risk of data leakage.

Implementation Details

Key source files: server.js, public/index.html, public/script.js, public/style.css.

server.js (summary)

Key behavior implemented in server.js:
- Serve static files from public/.
- POST /upload uses Multer to accept 'file' field, encrypts the uploaded file and saves as uploads/<name>.enc, then deletes original.
- POST /decrypt accepts JSON body { filename }, validates existence, decrypts to decrypted/<tempfile>, serves file as download and deletes temp file.
- All endpoints include try/catch to prevent server crash; detailed logging is used for debugging.

Frontend (index.html, script.js, style.css)

The frontend is intentionally lightweight. It provides:
- An upload form that posts the selected file as FormData 'file' to /upload.
- A decrypt form where the user pastes the encrypted filename returned by the upload step and requests /decrypt.
- Basic feedback messages and a modern clean design via style.css.

Testing & Validation

Testing was performed using both the browser frontend and Postman. Key test cases included:
- Uploading small text files and verifying resulting .enc file exists in uploads/.
- Decrypting uploaded files and confirming content equality via manual inspection and file hashes (SHA-256).
- Negative tests: attempting to decrypt non-existing files, verifying server returns 404; uploading without file field returns 400.

Sample Postman requests used:

1) POST http://localhost:5000/upload -> form-data key=file
2) POST http://localhost:5000/decrypt -> raw JSON {"filename":"uploads/XXXX.enc"}

Test Results:

1. Upload text file 'hi.txt' => uploads/XXXX.enc created. PASS
2. Decrypt uploads/XXXX.enc => decrypted/hi.txt content matches 'hi dakshu'. PASS
3. Upload unsupported large file > limit (if size limit used) => REJECT or fail as expected. (Optional)

Challenges and Solutions

- MulterError: Field name missing — Fixed by ensuring the HTML input has name='file' and FormData uses same key.
- ERR_OSSL_WRONG_FINAL_BLOCK_LENGTH — Fixed by ensuring correct ENCRYPTION_KEY length (32) and IV length (16) and reencrypting files with corrected values.
- File path mismatches (uploads/ vs filename only) — Standardized API responses to return 'uploads/<filename>.enc' and added server-side checks to accept either form.
- Port conflicts (5000 already in use) — Checked netstat to find offending PID and stopped it, or changed PORT in .env.

Security Considerations

- Do not commit .env to version control; use .gitignore.
- Use strong, randomly generated ENCRYPTION_KEY and IV in production; consider a secrets manager (AWS Secrets Manager, HashiCorp Vault).
- Serve the app over HTTPS in production to protect data in transit.
- Implement authentication & authorization before allowing decryption to prevent unauthorized downloads.
- Apply file type validation and size limits to mitigate abuse.
- Consider storing encrypted blobs in cloud storage (S3) with server-side encryption rather than local filesystem.
- Keep logs minimal and avoid logging sensitive plaintext data.

Recommendations & Future Work

- Integrate user authentication (JWT/OAuth) and role-based access control.
- Switch from static IV to per-file random IVs stored securely alongside the ciphertext.
- Rotate encryption keys periodically and implement key versioning.
- Use secure cloud storage (S3) and managed encryption keys (KMS).
- Add rate-limiting and logging/alerting for suspicious activity.
- Create automated tests and CI workflow to validate functionality on push.

Screenshots & Evidence

All screenshots produced during testing are included in the repository under the screenshots/ folder. Typical screenshots to include are server terminal output, frontend UI, upload response, encrypted file in uploads, decrypt response, and final decrypted file contents.

How to Run (Detailed)

1. Clone the repository: git clone <repo-url> && cd secure-file-share
2. Install dependencies: npm install
3. Create a .env file with ENCRYPTION_KEY (32 chars) and IV (16 chars) and PORT value
4. Start the server: node server.js
5. Open the browser: http://localhost:5000
6. Use the UI to upload and decrypt files, or use Postman for API testing.

Appendices

Appendix A: Frontend sample code (snippets)

script.js (snippet)

```
// Upload handler
const form = document.getElementById('uploadForm');
form.addEventListener('submit', async (e) => {
  e.preventDefault();
  const fd = new FormData();
  fd.append('file', document.getElementById('fileInput').files[0]);
  const res = await fetch('/upload', { method: 'POST', body: fd });
  const data = await res.json();
  document.getElementById('uploadResult').innerText = 'Encrypted: ' + data.file;
});

async function decryptFile() {
  const filename = document.getElementById('filename').value;
  const res = await fetch('/decrypt', { method: 'POST', headers: {'Content-Type':'application/json'}, body:
JSON.stringify({ filename }) });
  if (res.ok) { const blob = await res.blob(); /* trigger download */ }}
```

style.css (snippet)

```
body { font-family: Arial, sans-serif; background: #f7f9fc; }
.container { max-width: 800px; margin: auto; padding: 20px; }
button { background:#3498db; color:white; padding:10px 15px; }
```

Appendix B: Files included in repository

- public/index.html
- public/script.js
- public/style.css
- server.js

- README.md
- .gitignore
- package.json
- package-lock.json
- screenshots/ (all images)

Appendix C: Full frontend file samples

script.js

```
/* script.js - frontend logic */

document.getElementById('uploadForm').addEventListener('submit', async (e) => {
  e.preventDefault();
  const fd = new FormData();
  fd.append('file', document.getElementById('fileInput').files[0]);
  const res = await fetch('/upload', { method: 'POST', body: fd });
  const data = await res.json();
  document.getElementById('uploadResult').innerText = 'Encrypted file: ' + data.file;
});

async function decryptFile() {
  const filename = document.getElementById('filename').value;
  const res = await fetch('/decrypt', {
    method: 'POST', headers: {'Content-Type':'application/json'}, body: JSON.stringify({ filename })
  });
  if (res.ok) {
    const blob = await res.blob();
    const url = URL.createObjectURL(blob);
    const a = document.createElement('a'); a.href = url; a.download = filename.replace('.enc',''); a.click();
  } else {
    const err = await res.json(); document.getElementById('decryptResult').innerText = 'Error: ' +
(err.error || 'Decryption failed');
  }
}
```

style.css

```
/* style.css - basic styling */

body { font-family: Arial, sans-serif; background: #f7f9fc; margin:0; }
.container { max-width:800px; margin:auto; padding:20px; }
.card { background:#fff; padding:20px; border-radius:8px; box-shadow:0 2px 6px rgba(0,0,0,0.1);
margin-bottom:20px; }
button { background:#3498db; color:#fff; padding:10px 15px; border:none; border-radius:6px;
cursor:pointer; }
```

References & Acknowledgements

References:
- Node.js Crypto Documentation: https://nodejs.org/api/crypto.html
- Express Documentation: https://expressjs.com/
- Postman Learning Center: https://learning.postman.com/

Acknowledgements:
Thank you to Future Interns for the internship program and guidance. This task helped strengthen practical skills in secure development and encryption.