Security Overview – Secure File Sharing Application (Detailed)

1. Executive Summary

The Secure File Sharing Application is a Flask-based web system designed to provide **confidential, integrity-protected storage and retrieval of files**. The application ensures that uploaded files are encrypted before being stored on disk and are only decrypted momentarily during authorized download requests.

The system implements **modern authenticated encryption (AES-256-GCM)**, secure key handling through environment variables, and strict file lifecycle controls. This document provides a detailed technical and security-focused explanation suitable for **cybersecurity internships, SOC documentation, academic submissions, and entry-level security assessments**.

---

2. Security Objectives

The primary security goals of the application are:

1. **Confidentiality** – Prevent unauthorized access to file contents
2. **Integrity** – Detect any tampering or corruption of encrypted files
3. **Key Protection** – Prevent exposure of cryptographic keys
4. **Controlled Exposure** – Minimize the time plaintext data exists on the server
5. **Input Safety** – Prevent filename-based and path-based attacks

---

3. System Components and Trust Boundaries

3.1 Frontend (index.html)

- Provides a basic HTML interface for file upload
- No cryptographic operations occur on the client
- All security-critical operations are handled server-side

Trust boundary: - User-controlled input enters the system via file upload

---

3.2 Backend Application (app.py)

The Flask backend performs the following security-relevant tasks:

- Validates HTTP request methods

- Sanitizes uploaded filenames using `secure_filename()`
- Encrypts files before storage
- Decrypts files only during download
- Ensures temporary decrypted files are deleted

This layer represents the **core security enforcement point** of the application.

3.3 Configuration Layer (config.py & .env)

- Handles sensitive configuration such as encryption keys
- Ensures secrets are not embedded in source code
- Loads configuration securely at runtime

3.4 Cryptographic Module (crypto_utils.py)

- Implements AES-256-GCM encryption and decryption
- Generates nonces internally
- Verifies authentication tags automatically

This module ensures cryptographic correctness and isolation from business logic.

---

4. Cryptographic Design

4.1 Encryption Algorithm Selection

The application uses:

- **Algorithm:** Advanced Encryption Standard (AES)
- **Key Size:** 256 bits
- **Mode:** Galois/Counter Mode (GCM)

Reasons for choosing AES-256-GCM:

- Industry-standard and widely audited
- Provides both encryption and authentication
- Resistant to padding oracle and bit-flipping attacks
- Efficient and suitable for file encryption

4.2 Encryption Workflow (Upload Path)

1. User uploads a file via HTTP POST
2. Server reads the file as raw bytes
3. AES-GCM cipher is initialized with the secret key

4. A cryptographically secure **random nonce** is generated
5. File data is encrypted
6. Authentication tag is generated
7. Encrypted output is stored as:

```
[ NONCE (12 bytes) | AUTH TAG (16 bytes) | CIPHERTEXT ]
```

Security guarantee: - The plaintext file is never written to disk - Every encryption operation is unique due to random nonce

---

4.3 Decryption Workflow (Download Path)

1. Encrypted file is read from storage
2. Nonce and authentication tag are extracted
3. AES-GCM verification and decryption is attempted
4. If verification fails, decryption aborts
5. If successful, plaintext is reconstructed
6. File is sent to the user
7. Temporary decrypted file is deleted immediately

Security guarantee: - Any tampering with encrypted data is detected - Plaintext exposure is limited to milliseconds

---

5. Key Management and Handling

5.1 AES Secret Key Properties

- Symmetric encryption key
- Length: **32 bytes (256 bits)**
- Used for both encryption and decryption

5.2 Secure Key Storage

- Key is stored in an **environment variable**
- Loaded using `python-dotenv`
- Not hardcoded in any source file

Example:

```
AES_SECRET_KEY=abcd1234abcd1234abcd1234abcd1234
```

5.3 Key Validation and Enforcement

- Application fails fast if key is missing
- Key length is validated before use
- Prevents insecure key sizes

This ensures cryptographic operations are always performed safely.

5.4 Key Security Best Practices Followed

- `.env` file excluded from version control
- Keys never logged or returned in responses
- Keys never transmitted to clients
- Keys remain only in server memory

---

6. Secure File Handling and Storage

6.1 Encrypted Storage

- Files are stored only in encrypted form
- Direct disk access does not reveal contents
- Stored files are useless without the AES key

6.2 Temporary Decryption Strategy

- Decrypted files are written only for download
- Automatically deleted using Flask response hooks
- Minimizes attack window and data leakage risk

---

7. Input Validation and Attack Prevention

7.1 Filename Sanitization

- Uses Werkzeug's `secure_filename()`
- Prevents directory traversal and injection attacks

Blocked attack examples:

```
../../windows/system32
../app.py
```

7.2 HTTP Method Enforcement

- Upload endpoint only allows POST
- Prevents misuse through direct URL access

---

8. Threat Model and Mitigations

Threat	Description	Mitigation
Unauthorized file access	Attacker gains disk access	AES-256 encryption
File tampering	Attacker modifies ciphertext	AES-GCM authentication
Key exposure	Key leaked via source code	Environment-based storage
Path traversal	Malicious filename input	secure_filename()
Plaintext leakage	Files stored unencrypted	Temporary decryption only

---

9. Limitations

- No user authentication or authorization
- Single global encryption key
- No audit logging
- No HTTPS enforcement at application level

These limitations are acceptable for learning and controlled environments but should be addressed for production use.

---

10. Recommended Future Enhancements

- User authentication and role-based access control
- Per-user or per-file encryption keys
- Password-based key derivation (PBKDF2 / Argon2)
- HTTPS and secure cookies
- File size limits and rate limiting
- Secure key storage using Vault or HSM
- Audit logs and alerting

---

11. Conclusion

The Secure File Sharing Application demonstrates **strong foundational security practices** including authenticated encryption, secure key handling, and careful file lifecycle management. By combining

AES-256-GCM with environment-based key storage, the system protects data confidentiality and integrity while remaining simple and extensible.

This design is well-suited for educational use, cybersecurity training, and as a base for more advanced secure storage solutions.