Project title: Optimizing file system

Submitted to: Dr. Gurbinder Singh Bar

Subject code: CSE316

Group Details:

1.Naga Srinivasa Reddy

2.Mazin Zidan tp

3.Sakshi Verma

Section: K23WG

1. Project Overview

The Optimizing File Systems seeks to offer a secure environment for file storage, fast data retrieval, and system recovery. The system combines advanced authentication, protection of files, and threat detection with improved file system performance via optimized free-space management, directory structures, and file access mechanisms. The system will maintain recovery for the purpose of data integrity restoration as well as enhancement of read/write performance with the aim of efficiently carrying out operations.

Goals:

- Secure user authentication through passwords and two-factor.
- Efficient and secure storage of files along with encryption and access control.
- Further boosting file system performance with sophisticated free-space management and access methods.
- Advanced data recovery methods for catastrophic or corrupt failures of the system.
- Identification and protection from the most common forms of attacks (buffer overflow, worms, and unauthorized access).

Expected Outcomes:

- A functioning, secured, and optimized file management system.
- Increased read and write speeds with reduced latencies for efficient file management.
- Deliver robust recovery mechanisms to recover lost or corrupted data.
- Secure operations on files with advanced threat identification and counter-actions.

Scope:

- Master functionalities: Safe storage of files, authentication of users, efficient access processes.
- Security features: Encryption, access permissions, threat detection, and recovery.
- User roles: Administrator and normal users with varied privileges and levels of access.

Challenges in Previous Systems and Solutions in the New System:

1.Ineffective Management of Free Space:

- Previous Problems: Traditional systems suffer from fragmentation and slow performance.

- Solution: Bitmap-based and index-based allocation methods allow dynamic handling of free space, thereby reducing fragmentation and enhancing performance.

2.Ineffective Authentication Modes:

- Previous Problem: The systems completely reliant on passwords are vulnerable to brute-force attacks.
- Solution: Secure with hashing, e.g., PBKDF2/bcrypt, and use of two-factor authentication (2FA).

3.Missing Encryption:

- Previous Issue: Plaintext data can be accessed illegally.
- Solution: With AES-256 Encryption in place, the information is secured during storage and transit.

4.Limited Threat Detection:

- Previous Issue: Minimal monitoring allows the worm to be inside the system without being observed.
- Solution: ClamAV gives real-time threat monitoring using malware detection along with heuristic analysis.

5.Insufficient Data Recovery Mechanisms:

- Previous Issue: Regular file systems do not maintain effective recovery, and data gets lost during any crash.
- Solution: Guarantee fast data restoration by implementing journaling, checkpointing, or snapshot-based recovery.

6.Slow File Access:

- Previous Issue: Sequential searches with horrible caching system are killing the speed.
- Solution: Using caching, read-ahead, and good indexing optimized access for any fast read/write.

2. Module Wise Breakdown

Module 1: User Authentication and User Management

Purpose:

To ensure user access to data in a secure, safe, and sequestered environment.

Features:

- Password-based authentication (secure password storage by either PBKDF2 or bcrypt hashing).
- Two-factor authentication (either through email OTP or Time-based One-time Password - TOTP).
- User sign-up, login, logout, and control over different roles for user management (that is, admin and regular users).
- Applications session management and enforcement of timeout. This is to avoid unauthorized persistence of session(s).
- Password recovery through secure email verification.
- Prevent brute force attack implementations- rate-limiting and CAPTCHA.

Module 2: Secure and Optimized File Management

Purpose:

To provide a secure instance for the storage of files with optimized access to files, thereby raising performance and reliability.

Features:

- File operations: upload, download, delete, rename, and share.
- Optimized free space management which includes:
  - Bitmap allocation to search quickly on free-space.
  - Linked list allocation for dynamic free-space.
  - Indexed allocation for quick access and little fragmentation.

- Efficient directory structures that include:
  - Navigation by hierarchy in tree-based directories.

- End to end encryption including:

    - AES-256 encryption for storage and transfer of files.
    - Secure key management using an HSM (Hardware Security Module) or software vaults.
- Metadata handling:
    - Properties of files like size, type last access, creation date.
    - User activity auditing and tracking through an access log.
    - Hash maps for quick lookup of files and directories.
- File access optimization includes:
    - Caching and read-ahead techniques to hasten up accessing files.
    - Lazy loading to boost the performance of the system for large data operations.
- Role-based access control (RBAC) for managing user permissions (read, write, execute).

Performance Optimization Techniques:

- Use asynchronous I/O-based faster read and write operations.
- The data compression through storage space also enhances the speed of access (such as GZIP, LZ4).
- Implementing caching mechanisms to rapidly access both files and metadata.
- Simultaneous requests of users handle balancing so that the system will be reliable.
- Just-in-time deletion for improved performance by avoiding file deletions in peak offensively busy hours.
- Parallel processing supports large-scale file operations highly optimally.

Module 3: Threat Detection, Monitoring, and Recovery

Purpose:

Identify and eradicate security threats while enabling comprehensive data recovery.

Features:

- Malware detection:
    - Signature-based scanning through either ClamAV or custom signature database.
    - Heuristic Analysis: Attempts to find new and unknown threats.
- Buffer Overflows prevention:
    - Input validation and boundary checks to prevent memory corruption.
    - One able to prevent coding mistakes while it keeps an eye on the real-time activity.
- Activity recording and alerting:
    - Recording event occurrences into logs like file, user, and system events.

    - Real-time notification on suspicious activities like unauthorized entry and boundary data volume transfer.
- Routine integrity checking;
    - File checksum verification using SHA-256 checks algorithm.
    - Routine audit trails and an automated set of integrity checks.
- File System Recovery Techniques:
    - Checkpointing saves the state of the system after certain intervals.
    - Journaling keeps track of the write operations to be able to later bring back the system into the previous state.
    - Recovery through snapshots for a faster return to earlier phases.
    - Automatic repair of inconsistencies using transaction logs and validation routines.

Advanced Security Features:

- File integrity monitoring (FIM): will track and alert unauthorized modifications.
- Outcome prevent data loss (DLP): prohibit any unauthorized file sharing with sensitive data.
- Secure file shredding: permanent file deletion beyond any recovery.
- Implementing secure multi-party access: has cryptographic sharing protocols.
- User behavior analytics will help detect any anomaly and mitigate insider threats.
- Multi-level encryption layers are necessary for enhanced data protection.

3. Functionalities

Authentication and User Management:

- User Registration: Create new accounts with encrypted password storage and two-factor setup.
- Secure Login: Authenticate users securely with multi-factor support and session tracking.
- Role Management: Admin or user definition with right access levels and privileges.
- Session handling: Timeouts should be implemented on sessions with a secure token-based session identifier.
- Password Recovery: Secure mechanism for password reset with email verification. Secure and Optimized File Handling:

Secure and Optimized File Handling:

- File Operations: Secure file uploads, downloads, deletions, renaming, and sharing.
- File Sharing: Role-based access controls give security to share files selectively.

- Free-Space Management: bitmap, linked list, and indexed techniques for better efficiency in allocation.
- Directory Optimization: Hierarchical and hash-based lookup for faster navigation. Encryption: AES-256 is automatically applied for encrypt and decrypt during file operations.
- Metadata Management: Storing and managing file attributes, user logs, and access records.

Threat Detection, Monitoring, and Recovery:

- Malware Scanning: Signature-based and heuristic analysis techniques scan the malware for identification and blockage of offending content.
- Input Validation: Boundary checks as a way of mitigating vulnerabilities to buffer overflow attacks.
- Activity Logs: Detailed logs kept, adjusted with alerts setup on abnormal system activity.
- Recovery Mechanisms: Supports journaling and snapshot-based recovery for a robust system restoration.
- Automatic Repair: Detects and repairs inconsistencies by using transaction logs and periodic checks.

4. Technology Used

Programming Languages:

- Python: For backend logic, encryption, and threat detection.
- JavaScript: For the frontend interface and user interaction.

Libraries and Tools:

- Flask or Django: In building the web application and API services.
- PyCryptodome: As a library for AES encryption and cryptographic operations.
- SQLAlchemy: Database management through ORM-based management.
- Flask-Login: A library for use in authentication and session management.
- ClamAV: For signature-based malware scanning.

Other Tools:

- Docker: For containerized deployment and environment isolation.
- Git: For managing version control and collaborative development.

- JWT: For implementing secure session tokens.

5. Execution Plan

Phase 1: Project Setup and Environment Configuration

1. creation of a link between the two lovers: a virtual environment for python.
2. Instantiate either a Flask or Django Project.
3. Write a database schema (User profiles, file metadata, access logs).

Phase 2: Implement Authentication and User Management

1. Create endpoints for user registration and login.
2. Hash passwords (using either PBKDF2 or bcrypt).
3. Make two-factor authentication.
4. Using JWT manage user sessions securely.

Phase 3: Safe and Optimized File Handling Module Designing

1. Add encrypted and decrypted file upload/download.
2. There must be role-based access control for access control.
3. Efficient utilization of free space management (indexed allocation and bitmap) must be employed.
4. Make data structures like directories with rapid searching capabilities (trees, hash tables).
5. Create metadata tracking and display functionality.
6. Use secure file sharing mechanisms.

Phase 4: Threat Detection, Monitoring, and Recovery

1. Use ClamAV for malware scanning.
2. Use input validation for file and metadata fields.
3. Develop logging and alerting for unusual activity.
4. Journaling must be provided to recover from a crash and for integrity verification.

5. Data recovery automatically from checkpoints should be done.

Phase 5: Testing and Security Verification

1. Authentication and file operations must be tested using unit testing.
2. Test security like simulating buffer overflow attacks.
3. Encryption and decryption integrity must be tested.
4. Test file recovery features using a crash simulation.

Phase 6: Deployment and Documentation

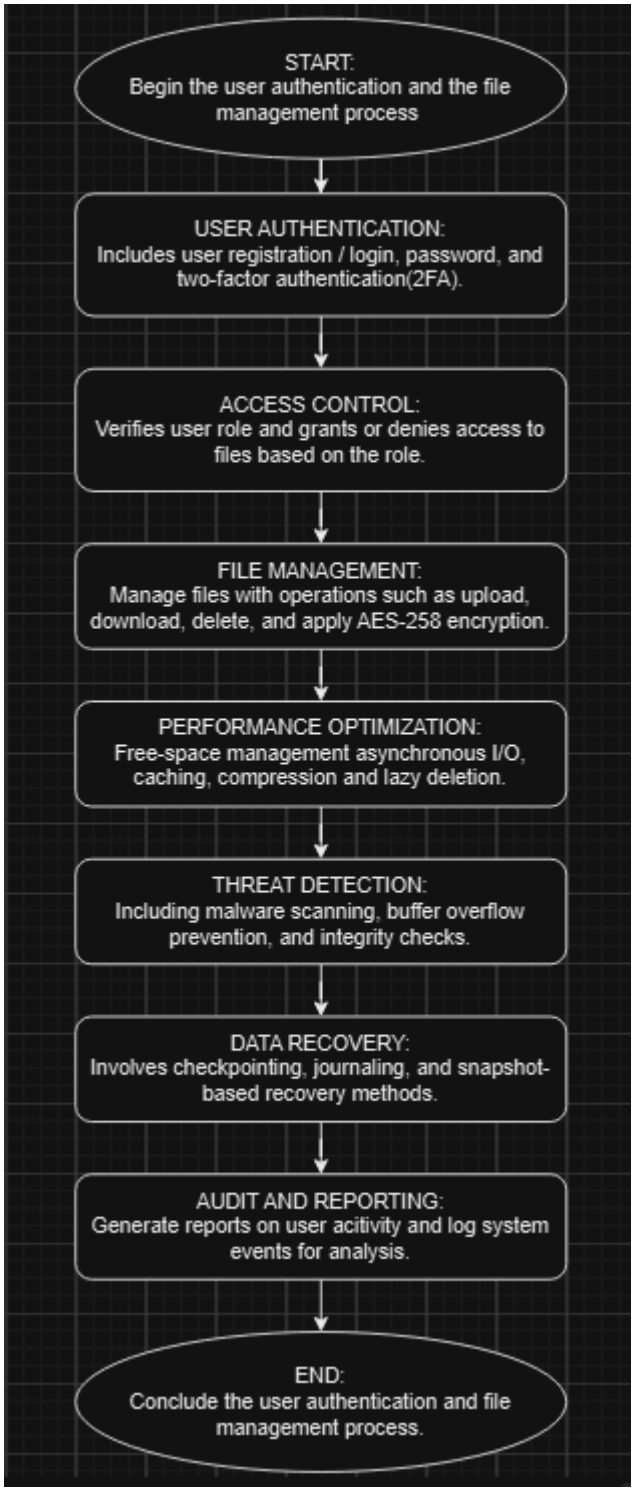
1. Set up a docker for easy deployment.
2. Prepare documentation (installation procedures, usage, and security policies).
3. Deploy to a secure server (AWS, DigitalOcean, etc.).

Hints to Enhance Efficiency

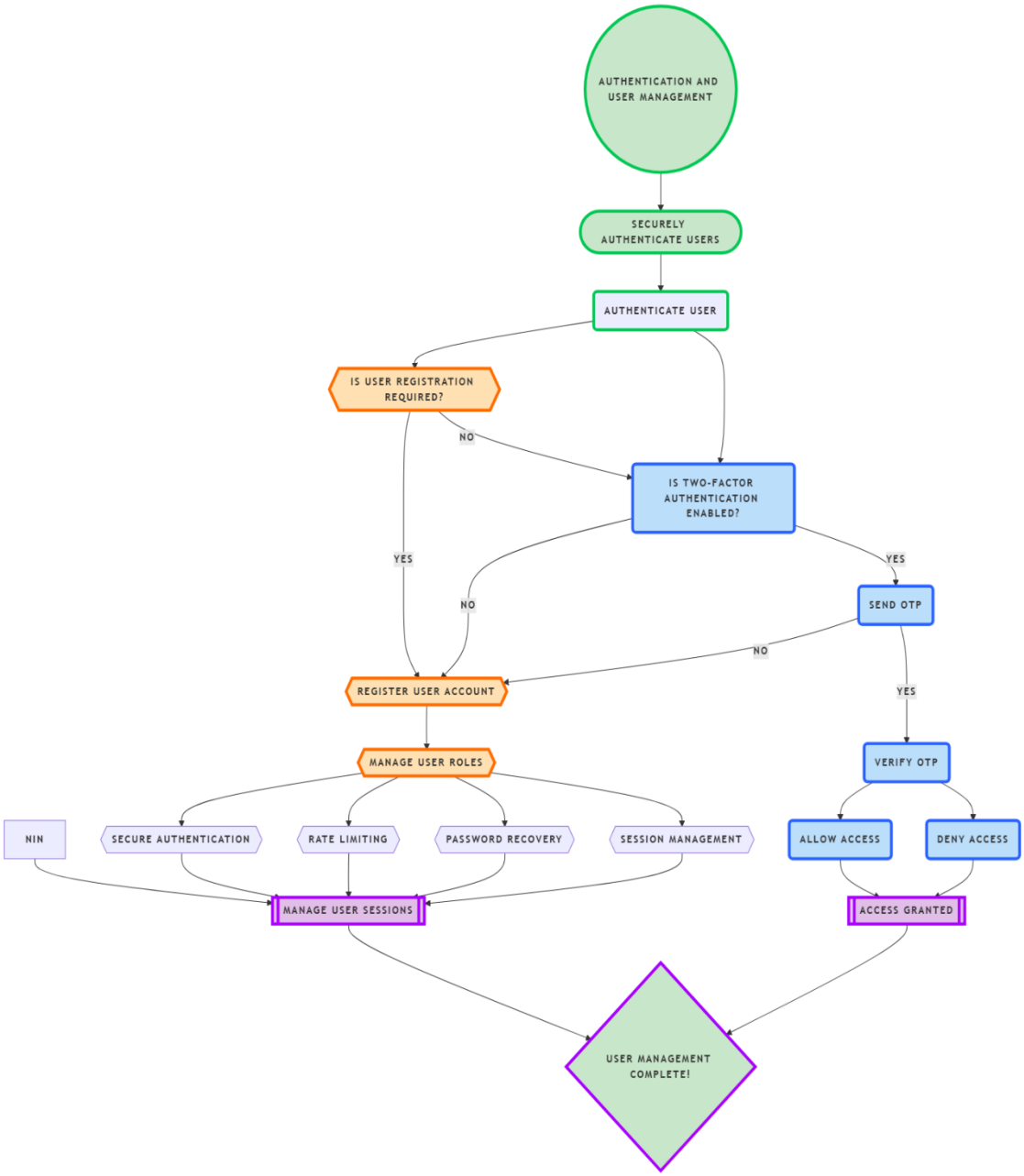
- Write modular code to ease debugging and enhancing in the future.
- Automate tests and security scans done routinely.
- Follow secure coding principles and for valid input.
- Look into libraries and dependencies that are updated to hide vulnerabilities.

6.FLOWCHART

Simple flow chart


START:
Begin the user authentication and the file management process
USER AUTHENTICATION:
Includes user registration / login, password, and two-factor authentication(2FA).
ACCESS CONTROL:
Verifies user role and grants or denies access to files based on the role.
FILE MANAGEMENT:
Manage files with operations such as upload, download, delete, and apply AES-258 encryption.
PERFORMANCE OPTIMIZATION:
Free-space management asynchronous I/O, caching, compression and lazy deletion.
THREAT DETECTION:
Including malware scanning, buffer overflow prevention, and integrity checks.
DATA RECOVERY:
Involves checkpointing, journaling, and snapshot-based recovery methods.
AUDIT AND REPORTING:
Generate reports on user acitivity and log system events for analysis.
END:
Conclude the user authentication and file management process.

Mechanism flowchart



7. Revision Tracking on GitHub

• Repository Name: Optimizing-file-system

• GitHub Link: GNSReddy/Optimizing-file-system

8. Conclusions and Future Directions

The Secure and Optimized File Management System assures confidentiality, integrity, and availability of data in storage and retrieval. Also, it has specialized security mechanisms, such as encryption, two-factor authentication, real-time detection of threats, and others. The system exhibits a cohesive high-performance package in that it integrates optimized storage management and explicit recovery techniques, thereby being not only high performing but resilient against failures and digital cyber threats.

Future work may cover:

- Incorporation of AI-based anomaly detection for preemptive threat analysis;
- Improving consistency for handling large scales of datasets in distributed settings;
- Enriching user experience through sophisticated search and categorization of files;
- Use blockchain technology in logging activities to ensure an immutable record of events; and
- Adoption of state-of-the-art filing compression algorithms to optimize the storage footprint.

9. References

1. Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of Applied Cryptography.

2. Tanenbaum, A. S., & Bos, H. (2014). Modern Operating Systems.

3. Garfinkel, S., & Spafford, G. (2002). Practical UNIX and Internet Security.

10. Applications

The Optimizing File System is applicable in many domains and industries where secure data handling, quick access, and stable recovery are paramount. Some of the most important environments where this system is extremely useful are:

Corporate and Enterprise Environments

Use Case: Secure storage for internal documents and restricted access for employees.

Benefit: Insures that sensitive corporate data such as financial statements and customer information are encrypted and only accessed by authorities.

Example: The HR system of a company requiring role-based access to restrict the visibility of employee records

Legal and Compliance

Use Case: Non-tampering archiving and attending access regarding legal documents and contracts.

Benefit: Making sure records cannot be tampered with alongside audit trails for compliance.

Example: Law firms access to case files while recording access for legal liability.

Educational Institutions

Use Case: Preserving and safeguarding student records, grades, and scholarship work.

Benefit: Provides controlled access to students and professors while safeguarding academic sensitive information.

Example: Colleges enabling professors to control research documents while students view their records.

Healthcare Industry

Use Case: Secured storage and retrieval of patient records (e-medical records), bureaucracy-evidence compliance.

Benefit: Protect patient confidentiality, per access control and encryption, and other benefitpower, like regulatory acts (for example, HIPAA).

Example: A hospital system managing medical history, tests, and prescriptions.

Government and Public Sector

Use Case: Ideal for the classified matters and citizen data, including regulatory files.

Benefit: It provides national security through encryption, restricted access, and disaster recovery.

Example: Government departments that handle multi-factor authentication sensitive citizen information.

Media and Entertainment

Use Case: Effective secure management of large media files (video, audio, and graphics).

Benefit: Such files can be manipulated in a fast manner, while at the same time encryption safeguards the intellectual property against unauthorized access.

Example: A production studio that uses the system to store raw video files and project files.

Research and Development

Use Case: Protecting confidential research information and intellectual property.

Benefit: Promotes safe collaboration and safeguards research from data compromises.

Example: An academic research institution using the platform to save proprietary scientific information and studies.

Cloud Storage Services

Use Case: Allow users to upload, store and share their files in a secure cloud-based environment.

Benefits: Users get secure upload, sharing, and retrieval of files from anywhere.

Example: The cloud provider may offer encrypted personal storage countered with malware scanning.

Problem Statement:

In the computer world, an organization as well as an individual is facing endless challenges in keeping and recovering from loss of access due to non-existent securities available at that time. Increasing threats from within and outside create some necessity for an excessive file that may perform all secure storing, accessible retrieving, and recoverable functions in the high performance system. The Secure and Optimized File Management System thus takes up the challenge of propounding both advanced authentication mechanisms and fast file handling, together with their robust threat and recovery protocols. Data is also made confidential during retrieval as well as during storage operation through optimized operations coupled with pro-effective threat mitigation strategies.

Here's a **Optimizing File System** using **Flask**, implementing **user authentication**, **file encryption**, **secure uploads/downloads**, and **access control**. It ensures that only authenticated users can upload, download, and manage files.

Features in the code:

1. User Authentication

- **User Registration**: Allows new users to register with a unique username and a hashed password (using werkzeug.security for password hashing).
- **User Login**: Users can securely log in using their credentials.
- **User Logout**: Provides a logout endpoint to securely terminate user sessions.
- **Access Control**: Only logged-in users can access protected endpoints (@login_required decorator).

2. File Security

- **File Encryption**: Files are encrypted using **Fernet symmetric encryption** before storage, ensuring confidentiality.
- **Secure File Upload**: Allows users to upload files; each file is encrypted and stored under a unique user-based identifier.
- **Secure File Download**: Decrypts and delivers requested files securely while ensuring only the owner can access them.

3. Threat Detection

- **Suspicious Activity Logging**: Logs potential threats by monitoring access attempts to sensitive endpoints (e.g., admin routes).
- **IP Monitoring**: Detects and logs unusual activity from external IP addresses (ignoring localhost for simplicity).

4. Activity Monitoring

- **File Action Logging**: Records all file-related actions (upload, download) along with timestamps and user IDs.
- **User Activity Logs**: Tracks user registration, logins, and logouts for security auditing.
- **Failed Login Detection**: Logs failed login attempts to detect potential brute-force attacks.

5. System Recovery

- **Admin Log Access**: Authorized admin users can retrieve activity logs for forensic analysis via the /logs endpoint.

- **Action History**: Maintains a persistent database (FileLog table) with records of all user-file interactions.

6. Optimizations & Security Measures

- **File Isolation**: Users can only access their own files—ensures isolation of private data.
- **Temporary Files**: Uses temporary decrypted files for downloads to avoid persistent unencrypted data.
- **Database Initialization**: Automatically initializes the database schema if not already set.

7. Logging System

- **Detailed Logs**: Logs critical system events (registrations, logins, uploads, downloads, and threats) in security.log.
- **Audit Trail**: Comprehensive logging supports auditing and post-incident reviews.

Code using python:

```
from flask import Flask, request, jsonify, send_file
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager, UserMixin, login_user, logout_user, login_required, current_user
from werkzeug.security import generate_password_hash, check_password_hash
import os
from cryptography.fernet import Fernet
import logging
from datetime import datetime

# Flask App Configuration
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///file_manager.db'
app.config['SECRET_KEY'] = os.urandom(24)
```

```
app.config['UPLOAD_FOLDER'] = 'uploads'
os.makedirs(app.config['UPLOAD_FOLDER'], exist_ok=True)

db = SQLAlchemy(app)
login_manager = LoginManager(app)
login_manager.login_view = 'login'

# Encryption Key (Store securely in production)
key = Fernet.generate_key()
cipher = Fernet(key)

# Logging Configuration
logging.basicConfig(filename='security.log', level=logging.INFO, format='%(asctime)s %(message)s')

# User Model
class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(150), unique=True, nullable=False)
    password = db.Column(db.String(256), nullable=False)

class FileLog(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, nullable=False)
    filename = db.Column(db.String(256), nullable=False)
    action = db.Column(db.String(50), nullable=False)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

```
# User Registration
@app.route('/register', methods=['POST'])
def register():
    data = request.json
    username = data.get('username')
    password = data.get('password')

    if User.query.filter_by(username=username).first():
        return jsonify({"error": "User already exists"}), 409

    hashed_password = generate_password_hash(password)
    new_user = User(username=username, password=hashed_password)
    db.session.add(new_user)
    db.session.commit()

    logging.info(f"User registered: {username}")
    return jsonify({"message": "User registered successfully"}), 201

# User Login
@app.route('/login', methods=['POST'])
def login():
    data = request.json
    user = User.query.filter_by(username=data.get('username')).first()

    if user and check_password_hash(user.password, data.get('password')):
        login_user(user)
        logging.info(f"User logged in: {user.username}")
        return jsonify({"message": "Login successful"})
```

```
    logging.warning(f"Failed login attempt: {data.get('username')}")
    return jsonify({"error": "Invalid credentials"}), 401

# User Logout
@app.route('/logout', methods=['POST'])
@login_required
def logout():
    logging.info(f"User logged out: {current_user.username}")
    logout_user()
    return jsonify({"message": "Logout successful"})

# Secure File Upload
@app.route('/upload', methods=['POST'])
@login_required
def upload():
    if 'file' not in request.files:
        return jsonify({"error": "No file provided"}), 400

    file = request.files['file']
    if file.filename == '':
        return jsonify({"error": "Empty filename"}), 400

    # Encrypt and Save the File
    encrypted_data = cipher.encrypt(file.read())
    file_path = os.path.join(app.config['UPLOAD_FOLDER'],
f"{current_user.id}_{file.filename}")
    with open(file_path, 'wb') as f:
        f.write(encrypted_data)

    # Log upload action
    log_action(current_user.id, file.filename, "upload")
```

```
    logging.info(f"File uploaded by {current_user.username}: {file.filename}")
    return jsonify({"message": "File uploaded securely"}), 201

# Secure File Download
@app.route('/download/<filename>', methods=['GET'])
@login_required
def download(filename):
    file_path = os.path.join(app.config['UPLOAD_FOLDER'],
f"{current_user.id}_{filename}")

    if not os.path.exists(file_path):
        logging.warning(f"File not found for download: {filename} by
{current_user.username}")
        return jsonify({"error": "File not found"}), 404

    with open(file_path, 'rb') as f:
        decrypted_data = cipher.decrypt(f.read())

    download_path = f"temp_{filename}"
    with open(download_path, 'wb') as f:
        f.write(decrypted_data)

    # Log download action
    log_action(current_user.id, filename, "download")

    logging.info(f"File downloaded by {current_user.username}: {filename}")
    return send_file(download_path, as_attachment=True, download_name=filename)

# Threat Detection: Log suspicious activity
@app.before_request
```

```
def detect_threats():
    user_ip = request.remote_addr
    if user_ip == '127.0.0.1':
        return  # Ignore localhost for simplicity

    if 'admin' in request.url:
        logging.warning(f"Potential threat detected: Access attempt to admin endpoint from {user_ip}")

# Action Logger
def log_action(user_id, filename, action):
    log = FileLog(user_id=user_id, filename=filename, action=action)
    db.session.add(log)
    db.session.commit()

# System Recovery: Fetch activity logs
@app.route('/logs', methods=['GET'])
@login_required
def get_logs():
    if current_user.username != 'admin':
        return jsonify({"error": "Unauthorized"}), 403

    logs = FileLog.query.all()
    return jsonify([{ "user_id": log.user_id, "filename": log.filename, "action": log.action, "timestamp": log.timestamp } for log in logs])

# Initialize Database
with app.app_context():
    db.create_all()

if __name__ == '__main__':
```

```
app.run(debug=True)
```