Project Report

Team29

COMP3334 Computer System Security

LIAN JIONGING 22097661D

LI JIONGYI 22101759D

ZHEN ZHANG Alex 22103518D

FENG RUIZHE 22108436D

Table of content

1. Abstract.....3
2. Introduction.....3
3. Threat Model.....4
3.1 Adversaries.....4
3.1.1 Passive Adversary.....4
3.1.2 Unauthorized User.....4
3.2 Defend from adversary attacks.....5
3.3.1 Defense for Passive Adversary.....5
3.3.2 Defense for Unauthorized User.....5
3.3 Attack Vectors.....5
3.4 Potential Impacts.....6
4. Algorithms Designed to Implement Functionalities.....6
4.1 Functionality Requirements and Implementations.....6
4.1.1 User Registration.....6
4.1.2 User Login.....12
4.1.3 Password Reset.....14
4.1.4 Upload File.....16
4.1.5 Download File.....18
4.1.6 Edit File.....20
4.1.7 Share File..... 23
4.1.8 View Logs.....24
4.1.9 Libraries.....25
4.1.10 Challenges.....26
5. Test Cases.....27
5.1 SQL Injection Attacks.....27
5.2 Unauthorized users Prevention.....27
6. Future Works.....28
6.1 Graphical User Interface (GUI).....28
6.2 Scability.....28
6.3 Performance Optimization.....28
7. References......28

Contribution Table

Student ID	Name	Overall contribution
22097661D	LIAN JIONGING	25%
22101759D	LI JIONGYI	25%
22103518D	ZHEN ZHANG Alex	25%
22108436D	FENG RUIZHE	25%

1. Abstract

The database system for the project aims to ensure confidentiality, integrity, and access control for user data and file management. It incorporates cryptographic algorithms, secure authentication mechanisms, and defensive design principles to protect against passive adversaries, unauthorized users, and SQL injection attacks. By encrypting sensitive data, using hashed passwords, and implementing access control policies, the database prevents unauthorized access to files and critical operations. The report includes an in-depth analysis of the database architecture, algorithms, and security measures, along with test cases that demonstrate its resilience against various attacks. Recommendations for future improvements, such as enhanced scalability and performance optimization, are also provided.

2. Introduction

Background

In today's digital landscape, databases storing sensitive user data are prime targets for cyberattacks, with threats ranging from passive eavesdropping to unauthorized access and SQL injection. Existing systems often lack comprehensive defenses against these evolving threats, leading to frequent data breaches and loss of user trust. **This project addresses these challenges by designing a secure, client-server database system** that prioritizes confidentiality, integrity, and access control.

Objectives:

1. **Secure Authentication**: Implement multi-factor authentication (MFA) with OTPs to prevent credential theft.

2. **End-to-End Encryption**: Use AES-GCM for file encryption

3. **Attack Resilience**: Mitigate SQL injection, brute-force attacks, and nonce reuse risks through defensive coding practices.

4. **Auditability**: Maintain granular logs for accountability and forensic analysis.

The system leverages SQLite for lightweight database management and Python's cryptographic libraries to balance security with efficiency. By focusing on modern cryptographic standards and proactive threat modeling, this solution aims to set a benchmark for secure file storage systems.

3. Threat Model

3.1 Adversaries

3.1.1 Passive Adversary:

The Server acts as a passive participant in the file management system, monitoring communications and data transfers without altering the transmitted information. The primary goal of the passive adversary is to decrypt user-uploaded files. They analyze network traffic patterns to capture sensitive information, such as encryption keys, posing a significant risk to user confidentiality while appearing legitimate.
Example: A malicious third party eavesdropping on file transfers.

Abilities:

- **Monitors Network Traffic:** Captures encrypted messages and metadata (packet size, timing, frequency) to infer sensitive information.
- **Analyzes Stored Data:** Looks for vulnerabilities in encryption schemes, such as weak algorithms or poor key management
- **Observes Communication Channels:** Identifies relationships between users, enabling potential attacks like deanonymization
- **Employs Traffic Correlation:** Tracks user behavior and uncovers communication endpoints.
- **Exploits Protocol Weaknesses:** Launches replay or timing attacks without interfering with the system.

3.1.2 Unauthorized User:

An unauthorized user poses a direct threat by attempting to access legitimate user files through various methods. They may exploit application vulnerabilities, such as weak authentication or poor access controls, or target the user's machine. Techniques like phishing and social engineering can lead to unauthorized access by tricking users into revealing their credentials. This adversary is a persistent threat to data integrity and confidentiality.
Example: Hacker brute-forcing login credentials.

Abilities:

- **Monitors Network Traffic:** Captures encrypted messages and metadata (packet size, timing, frequency) to infer sensitive information.
- **Analyzes Stored Data:** Looks for vulnerabilities in encryption schemes, such as weak algorithms or poor key management.
- **Observes Communication Channels:** Identifies relationships between users, enabling potential attacks like deanonymization.
- **Employs Traffic Correlation:** Tracks user behavior and uncovers communication endpoints.
- **Exploits Protocol Weaknesses:** Launches replay or timing attacks without interfering with the system.

3.2 Defend from Adversary Attacks

3.2.1 Defense for Passive Adversary:

- **Encryption with AES-GCM**:
    - Files are encrypted using AES-GCM with **unique nonces** for each encryption operation. A cryptographically secure random number generator (CSPRNG) ensures nonce uniqueness. A counter-based mechanism tracks nonces to prevent reuse.
    - **Key Management**: Encryption keys are rotated to mitigate risks from potential key compromise. Keys are stored in secure, isolated environments (e.g., Hardware Security Modules - HSMs) to prevent unauthorized access.

3.2.2 Defense for Unauthorized User:

- **Multi-Factor Authentication (MFA):** Requires multiple verification forms (e.g., OTP) during login. The API includes OTP generation and verification.
- **Input Validation and Sanitization:** Validates and sanitizes all user inputs to prevent injection attacks. For instance, the (sanitize_filename) function is used.
- **Rate Limiting:** Implements rate limiting on login attempts to mitigate brute force attacks.
- **User Education:** Trains users to recognize phishing attempts and maintain secure credentials.
- **Regular Security Audits:** Conducts audits and penetration testing to identify and fix vulnerabilities.

3.3 Attack Vectors

- **Network Attacks**: Interception of data during transmission, which could lead to the exposure of sensitive information.

- **Social Engineering**: Manipulating users through deceptive tactics to gain access to systems or sensitive data.
- **Application Vulnerabilities**: Exploiting bugs or security weaknesses within the software to gain unauthorized access.

3.4 Potential Impacts:

- **Data Breach**: Unauthorized access to sensitive user data, leading to potential misuse.
- **Data Integrity Compromise**: Alteration of data, resulting in misinformation and loss of trust.
- **Reputation Damage**: Erosion of user trust and confidence due to security incidents, impacting the organization's credibility.

4. Algorithms Designed to Implement Functionalities

4.1 Functionality Requirements and Implementations

4.1.1 User Registration

```python
def handle_register(request, ip_address=None):
    """
    Handle user registration request
    """
    # Validate inputs
    username = validate_input(request.get("username"), r'^[a-zA-Z0-9_]{3,32}$')
    password_hash = request.get("password_hash")

    if not username or not password_hash:
        return {"status": "error", "message": "Invalid username or password format"}

    try:
        conn = get_db_connection()
        cursor = conn.cursor()

        # Check if username already exists
        cursor.execute("SELECT user_id FROM users WHERE username = ?", (username,))
        if cursor.fetchone():
            return {"status": "error", "message": "Username already exists"}

        # Generate a salt and final password hash
        salt = secrets.token_hex(16)
        final_password_hash = hashlib.sha256((password_hash + salt).encode()).hexdigest()

        # Generate OTP secret for MFA
        otp_secret = pyotp.random_base32()

        # Insert new user
        cursor.execute("""
    INSERT INTO users
    (username, password_hash, salt, creation_date, last_login, otp_secret)
    VALUES (?, ?, ?, ?, ?, ?)
""", (username, final_password_hash, salt, datetime.datetime.now(), None, otp_secret))
        user_id = cursor.lastrowid
        conn.commit()

        # Log the registration
        log_action(user_id, "user_register", f"New user registered: {username}", ip_address)

        # Generate OTP provisioning URI for QR code
        otp_uri = pyotp.totp.TOTP(otp_secret).provisioning_uri(
            name=username,
            issuer_name="SecureStorage"
        )
```

```python
        return {
            "status": "success",
            "message": "User registered successfully",
            "user_id": user_id,
            "otp_secret": otp_secret,
            "otp_uri": otp_uri
        }
    except Exception as e:
        logger.error(f"Registration error: {e}")
        return {"status": "error", "message": "Registration failed"}
    finally:
        close_connection(conn)
```

Building Blocks:

- **SQLite**: For efficient and lightweight database management, ensuring easy setup, portability, and integration with minimal overhead. SQLite provides ACID compliance and is ideal for small to medium-sized applications.
- **bcrypt**: To implement secure hashing algorithms for password storage, ensuring that user credentials are protected even if the database is compromised. Bcrypt also incorporates salting to further strengthen password security against brute-force attacks.

Workflow:

User Input:

- The system prompts the user to enter a username and password.
- Input validation is performed to ensure that the username and password meet predefined requirements.

Duplicate Check:

- The application queries the database using an SQL SELECT… FROM…WHERE… statement to check if the entered username already exists.
- If a duplicate username is found, the system notifies the user and requests a new username.

Password Hashing:

- If the username is available, the system hashes the password using bcrypt.
- A unique salt is generated for each password to ensure that even identical passwords result in different hashes, protecting against rainbow table attacks.

Database Insertion:

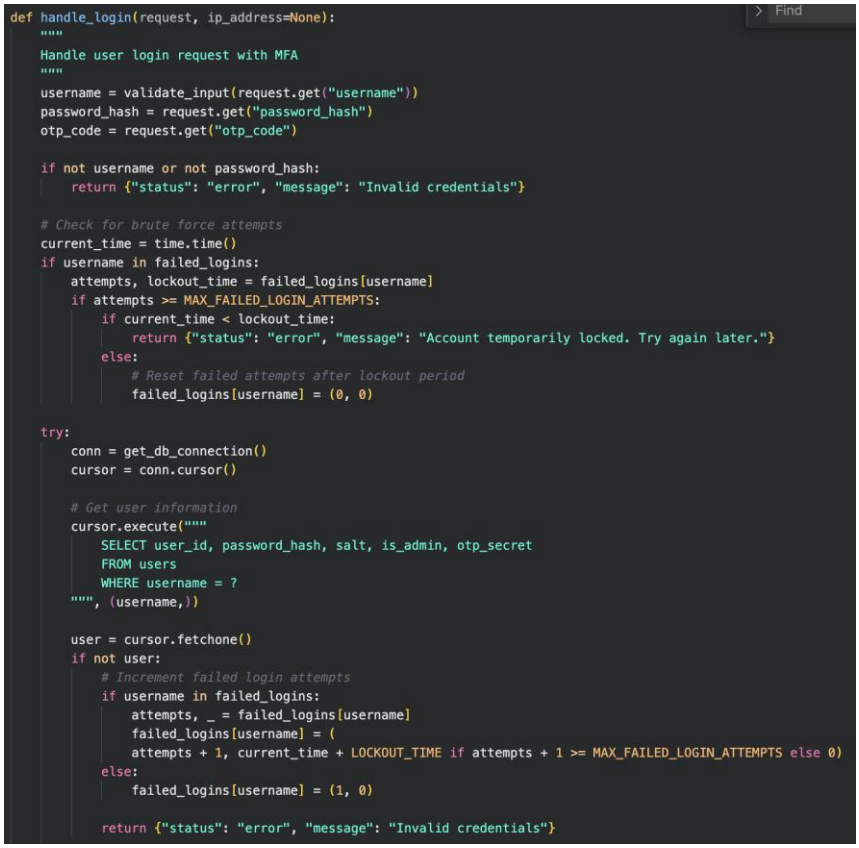
- An SQL INSERT statement is executed to add the new user to the database, storing the username and the hashed password.

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    user_id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    salt TEXT NOT NULL,
    creation_date TIMESTAMP NOT NULL,
    last_login TIMESTAMP,
    is_admin BOOLEAN DEFAULT 0,
    reset_token TEXT,
    reset_token_expiry TIMESTAMP,
    otp_secret TEXT
)
''')
```

Confirmation Message:

- After successful registration, the system displays a confirmation message to the user.

4.1.2 User Login



```python
def handle_login(request, ip_address=None):
    """
    Handle user login request with MFA
    """
    username = validate_input(request.get("username"))
    password_hash = request.get("password_hash")
    otp_code = request.get("otp_code")

    if not username or not password_hash:
        return {"status": "error", "message": "Invalid credentials"}

    # Check for brute force attempts
    current_time = time.time()
    if username in failed_logins:
        attempts, lockout_time = failed_logins[username]
        if attempts >= MAX_FAILED_LOGIN_ATTEMPTS:
            if current_time < lockout_time:
                return {"status": "error", "message": "Account temporarily locked. Try again later."}
            else:
                # Reset failed attempts after lockout period
                failed_logins[username] = (0, 0)

    try:
        conn = get_db_connection()
        cursor = conn.cursor()

        # Get user information
        cursor.execute("""
            SELECT user_id, password_hash, salt, is_admin, otp_secret
            FROM users
            WHERE username = ?
        """, (username,))

        user = cursor.fetchone()
        if not user:
            # Increment failed login attempts
            if username in failed_logins:
                attempts, _ = failed_logins[username]
                failed_logins[username] = (
                attempts + 1, current_time + LOCKOUT_TIME if attempts + 1 >= MAX_FAILED_LOGIN_ATTEMPTS else 0)
            else:
                failed_logins[username] = (1, 0)

            return {"status": "error", "message": "Invalid credentials"}
```

```
# Verify password
expected_hash = hashlib.sha256((password_hash + user['salt']).encode()).hexdigest()
if expected_hash != user['password_hash']:
    # Increment failed login attempts
    if username in failed_logins:
        attempts, _ = failed_logins[username]
        failed_logins[username] = (
        attempts + 1, current_time + LOCKOUT_TIME if attempts + 1 >= MAX_FAILED_LOGIN_ATTEMPTS else 0)
    else:
        failed_logins[username] = (1, 0)

    return {"status": "error", "message": "Invalid credentials"}

# Verify OTP if provided
if otp_code:
    otp_secret = user['otp_secret']
    totp = pyotp.TOTP(otp_secret,interval=300)
    if not totp.verify(otp_code):
        # Increment failed login attempts
        if username in failed_logins:
            attempts, _ = failed_logins[username]
            failed_logins[username] = (
            attempts + 1, current_time + LOCKOUT_TIME if attempts + 1 >= MAX_FAILED_LOGIN_ATTEMPTS else 0)
        else:
            failed_logins[username] = (1, 0)

        return {"status": "error", "message": "Invalid OTP code"}

# Create a new session
session_id = create_session(user['user_id'], username)

# Update last login time
cursor.execute("""
    UPDATE users
    SET last_login = ?
    WHERE user_id = ?
""", (datetime.datetime.now(), user['user_id']))

conn.commit()
```

```
    # Log the login
    log_action(user['user_id'], "user_login", f"User logged in: {username}", ip_address)

    # Reset failed login attempts
    if username in failed_logins:
        failed_logins.pop(username)

    return {
        "status": "success",
        "message": "Login successful",
        "session_id": session_id,
        "user_id": user['user_id'],
        "is_admin": user['is_admin']
    }
except Exception as e:
    logger.error(f"Login error: {e}")
    return {"status": "error", "message": "Login failed"}
finally:
    close_connection(conn)
```

Building Blocks:

- **SQLite**: To retrieve stored user credentials and associated metadata for validation.
- **OTP Generation**: Integrates a secure OTP (One-Time Password) generation mechanism, such as the Time-Based One-Time Password (TOTP) algorithm, to enhance login security.
- **API**: To securely deliver OTPs to the user's registered.
- **Verification System**: Implements a reliable method to verify user credentials (username and password) alongside the OTP, ensuring multi-factor authentication (MFA).

```
==== File Management System ====
1. Register
2. Login
3. Admin Login
4. Exit
Please select (1-4): 2
Please enter your username: johnny
Please enter your password: 123456
sending johnny Sending OTP...
OTP sent successfully!
Please enter OTP:
```

```
❯ /opt/local/bin/python3.13 "/Users/lijiongyi/Desktop/Simple-network-storage-project 2/get_otp.py"
[2025-04-13 21:02:05] OTP receiver started, waiting for verification codes...
[2025-04-13 21:02:05] Type 'exit', 'quit', or 'stop' to shut down the receiver
[2025-04-13 21:02:05] OTP receiver started, listening on port: 8888
[2025-04-13 21:02:05] Waiting for OTP codes...
[2025-04-13 21:03:11] Received OTP for user johnny: 634804
[2025-04-13 21:03:11] OTP valid for: 5 mins
```

- **Session Management**: Ensures secure session creation and management after successful login.

4.1.3 Password Reset

```python
def handle_reset_password(request, ip_address=None):
    """
    Handle password reset request
    """
    (variable) session_id: Any
    session_id = request.get("session_id")
    username = validate_input(request.get("username"))

    # verify session validation
    if not validate_session(session_id, username):
        return {"status": "error", "message": "Invalid or expired session, please log in again"}

    old_password_hash = request.get("old_password_hash")
    new_password_hash = request.get("new_password_hash")

    if not old_password_hash or not new_password_hash:
        return {"status": "error", "message": "missing password"}

    try:
        conn = get_db_connection()
        cursor = conn.cursor()

        # get password information
        cursor.execute("""
            SELECT user_id, password_hash, salt
            FROM users
            WHERE username = ?
        """, (username,))
```

```
    user = cursor.fetchone()
    if not user:
        return {"status": "error", "message": "user not found"}

    # verify old password
    expected_hash = hashlib.sha256((old_password_hash + user['salt']).encode()).hexdigest()
    if expected_hash != user['password_hash']:
        return {"status": "error", "message": "password is not correct"}

    # create new salt and hash
    new_salt = secrets.token_hex(16)
    final_new_hash = hashlib.sha256((new_password_hash + new_salt).encode()).hexdigest()

    # update password
    cursor.execute("""
        UPDATE users
        SET password_hash = ?, salt = ?
        WHERE username = ?
    """, (final_new_hash, new_salt, username))

    conn.commit()

    # log the password reset
    log_action(user['user_id'], "password_reset", "password is reset", ip_address)
```

```
        # tell client need to re-login
        return {
            "status": "success",
            "message": "password reset successfully",
            "require_relogin": True  # need to re-login
        }
    except Exception as e:
        logger.error(f"password reset error: {e}")
        return {"status": "error", "message": "password reset failed"}
    finally:
        close_connection(conn)
```

Building Blocks:

- **Enhanced Security Checks**: Incorporates additional layers of verification to confirm the user's identity, such as validating the old password

Workflow:

Session check:

- The program checks user authority and passes if user is identified.

Password Reset Request:

- The user initiates the password reset process by providing:
    - Their username or email address.
    - Their current (old) password.
    - A new password they wish to set.
- Input validation is performed to ensure all fields are provided and meet the required format.

Old Password Validation:

- The application retrieves the stored hashed password for the user from the database using an SQL SELECT query.
- The provided old password is hashed using bcrypt and compared with the stored hash.
- If the old password is incorrect, the system notifies the user of the failed validation and does not proceed further.

Password Update:

- Upon successful validation of the old password, new password, and optional security checks, the system hashes the new password using bcrypt.
- The hashed password is updated in the database using an SQL UPDATE query.
- The system ensures that the old password hash is removed or stored securely in a password history table to prevent reuse.

4.1.4 Upload File

```
def handle_upload_file(request, ip_address=None):
    """
    Handle file upload request with simplified interface
    """
    username = validate_input(request.get("username"))
    session_id = request.get("session_id")

    if not username:
        return {"status": "error", "message": "Invalid username"}

    if not validate_session(session_id, username):
        return {"status": "error", "message": "Invalid or expired session, please log in again"}

    # Get user_id from username
    try:
        conn = get_db_connection()
        cursor = conn.cursor()

        cursor.execute("SELECT user_id FROM users WHERE username = ?", (username,))
        user = cursor.fetchone()
        if not user:
            return {"status": "error", "message": "User not found"}

        user_id = user['user_id']

        # Get filename and data from request
        filename = validate_input(request.get("filename"), r'^[a-zA-Z0-9_\-. ]{1,255}$')
        encrypted_data_str = request.get("data")  # This contains all the encrypted package
```

```
        if not filename or not encrypted_data_str:
            return {"status": "error", "message": "Missing filename or file data"}

        # Decode the encrypted package
        try:
            encrypted_package_json = base64.b64decode(encrypted_data_str).decode()
            encrypted_package = json.loads(encrypted_package_json)

            # Extract data from the package
            ciphertext = base64.b64decode(encrypted_package["ciphertext"])
            nonce = encrypted_package["nonce"]  # Keep as base64 for storage
            tag = encrypted_package["tag"]      # Keep as base64 for storage
            metadata = encrypted_package.get("metadata", {})

            original_filename = metadata.get("original_filename", filename)
            file_size = len(ciphertext)

            # Generate a unique filename to prevent overwriting
            safe_filename = f"{int(time.time())}_{filename}"
            file_path = os.path.join("files", safe_filename)

            # Ensure the files directory exists
            os.makedirs("files", exist_ok=True)
```

```
        # Save the encrypted file
        with open(file_path, 'wb') as f:
            f.write(ciphertext)

        # Create file record
        cursor.execute("""
            INSERT INTO files
            (filename, original_filename, owner_id, upload_date, last_modified, file_size, file_path, is_deleted)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
        """, (
            safe_filename,
            original_filename,
            user_id,
            datetime.datetime.now(),
            datetime.datetime.now(),
            file_size,
            file_path,
            0
        ))

        file_id = cursor.lastrowid

        # Store encryption information
        cursor.execute("""
            INSERT INTO file_keys
            (file_id, key_encrypted, iv)
            VALUES (?, ?, ?)
        """, (file_id, tag, nonce))  # Using tag as key_encrypted and nonce as iv
```

```
            conn.commit()

            # Log the upload
            log_action(user_id, "file_upload", f"Uploaded file: {original_filename}", ip_address)

            return {
                "status": "success",
                "message": "File uploaded successfully",
                "file_id": file_id
            }

        except json.JSONDecodeError:
            return {"status": "error", "message": "Invalid encrypted package format"}
        except KeyError as e:
            return {"status": "error", "message": f"Missing required encryption data: {str(e)}"}

    except Exception as e:
        logger.error(f"File upload error: {e}")
        return {"status": "error", "message": "File upload failed"}
    finally:
        close_connection(conn)
```

Building Blocks:

Session check:

- The program checks user authority and passes if user is identified.
- **File Handling Techniques**: Utilizes efficient and robust file handling methods to manage file uploads, storage, and retrieval.
- **Data Encryption**: Implements strong encryption protocols, such as Advanced Encryption Standard (AES), to secure files both during transmission and storage, protecting sensitive data from unauthorized access.
- **Validation and Permissions**: Ensures that only authorized users can upload, access, or modify files by validating user roles and permissions.

Workflow:

Session check:

- The program checks user authority and passes if user is identified.
  User Permission Validation:
- Every file operation begins with a validation step(session id) to ensure the user has the required permissions for the action.
- User roles and permissions are checked in the database (e.g., admin, standard user).
- Unauthorized users are denied access, and an error message is displayed.
  File Upload Request:
- The user selects a file to upload through the application interface.
  File Encryption:
- Before the file is permanently stored, it is encrypted using AES (e.g., AES-256) with a unique encryption key.
- The encryption key is securely managed
- The encrypted file is saved to the file system or database.

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS files (
    file_id INTEGER PRIMARY KEY AUTOINCREMENT,
    filename TEXT NOT NULL,
    original_filename TEXT NOT NULL,
    owner_id INTEGER NOT NULL,
    upload_date TIMESTAMP NOT NULL,
    last_modified TIMESTAMP NOT NULL,
    file_size INTEGER NOT NULL,
    file_path TEXT NOT NULL,
    is_deleted BOOLEAN DEFAULT 0,
    FOREIGN KEY (owner_id) REFERENCES users(user_id)
)
''')
```

4.1.5 Download File

```
def handle_download_file(request, ip_address=None):
    """
    Handle file download request, allowing both owner and users with share permissions
    """
    username = validate_input(request.get("username"))
    file_id = request.get("file_id")
    session_id = request.get("session_id")

    if not username or not file_id:
        logger.debug(f"Invalid input: username={username}, file_id={file_id}")
        return {"status": "error", "message": "Invalid username or file ID"}

    if not validate_session(session_id, username):
        return {"status": "error", "message": "Invalid or expired session, please log in again"}

    try:
        file_id = int(file_id)  # make sure file_id is integer
    except (TypeError, ValueError):
        logger.debug(f"Invalid file_id: {file_id}")
        return {"status": "error", "message": "Invalid file ID"}

    try:
        conn = get_db_connection()
        cursor = conn.cursor()

        # get user_id
        cursor.execute("SELECT user_id FROM users WHERE username = ?", (username,))
        user = cursor.fetchone()
        if not user:
            logger.debug(f"User not found: username={username}")
```

```
            return {"status": "error", "message": "User not found"}
        user_id = user['user_id']

        # Check if the user is the file owner or has sharing permissions
        cursor.execute("""
            SELECT f.file_id, f.filename, f.original_filename, f.file_path, f.owner_id, u.username as owner_username
            FROM files f
            JOIN users u ON f.owner_id = u.user_id
            WHERE f.file_id = ? AND f.is_deleted = 0
            AND (f.owner_id = ? OR EXISTS (
                SELECT 1 FROM file_permissions p
                WHERE p.file_id = f.file_id AND p.user_id = ?
            ))
        """, (file_id, user_id, user_id))

        file = cursor.fetchone()
        if not file:
            logger.debug(f"File not found or no permission: file_id={file_id}, user_id={user_id}")
            return {"status": "error", "message": "File not found or you don't have permission to download"}

        # Get the encryption key and IV
        cursor.execute("SELECT key_encrypted, iv FROM file_keys WHERE file_id = ?", (file['file_id'],))
        key_data = cursor.fetchone()
        if not key_data:
            logger.debug(f"Encryption key not found for file_id={file['file_id']}")
            return {"status": "error", "message": "Encryption key not found"}
```

```python
    # Reading encrypted files
    if not os.path.exists(file['file_path']):
        logger.error(f"File path does not exist: {file['file_path']}")
        return {"status": "error", "message": "File not found on server"}
    with open(file['file_path'], 'rb') as f:
        encrypted_data = f.read()

    # verify and clean nonce and tag  Base64 encode
    try:
        nonce_clean = base64.b64encode(base64.b64decode(key_data['iv'], validate=True)).decode('utf-8')
        tag_clean = base64.b64encode(base64.b64decode(key_data['key_encrypted'], validate=True)).decode('utf-8')
    except base64.binascii.Error as e:
        logger.error(f"Invalid Base64 data: iv={key_data['iv']}, key_encrypted={key_data['key_encrypted']}, error: {e}")
        return {"status": "error", "message": f"Invalid encryption data: {str(e)}"}

    # Prepare encryption package
    encrypted_package = {
        "ciphertext": base64.b64encode(encrypted_data).decode('utf-8'),
        "nonce": nonce_clean,
        "tag": tag_clean,
        "metadata": {
            "original_filename": file['original_filename'],
            "file_type": os.path.splitext(file['original_filename'])[1].lstrip('.').lower(),
            "owner_username": file['owner_username']  # Add owner username
        }
    }
```

```python
    # Record download operations
    log_action(user_id, "file_download", f"Downloaded file: {file['original_filename']}", ip_address)

    # Encoding JSON and validating
    try:
        json_str = json.dumps(encrypted_package)
        logger.debug(f"Encrypted package JSON: {json_str[:100]}...")
    except TypeError as e:
        logger.error(f"JSON encoding error: {e}, package content: {encrypted_package}")
        return {"status": "error", "message": f"JSON encoding error: {str(e)}"}

    # Returns a Base64-encoded JSON packet
    return {
        "status": "success",
        "message": "File download successful",
        "data": base64.b64encode(json_str.encode('utf-8')).decode('utf-8')
    }

except sqlite3.Error as e:
    logger.error(f"Database error: {e}")
    return {"status": "error", "message": f"Database error: {str(e)}"}
except FileNotFoundError as e:
    logger.error(f"File not found: {e}")
    return {"status": "error", "message": f"File not found: {str(e)}"}
except Exception as e:
    logger.error(f"Unexpected error: {type(e).__name__}: {e}")
    return {"status": "error", "message": f"Unexpected error: {str(e)}"}
finally:
    close_connection(conn)
```

Building Blocks:

- **Database Connection**: Establishes a secure connection to the SQLite database to facilitate reliable file access and management.
- **SQL Queries**: Executes specific SQL queries to efficiently retrieve encrypted data based on the provided filename and the associated user credentials.
- **Data Decoding**: Utilizes base64 encoding techniques to accurately decode the encrypted data, transforming it into a readable format.

Workflow:

Authentication:

Session check:

- The program checks user authority and passes if user is identified.
- The user initiates the process by providing their username along with the desired filename for the download.

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS file_permissions (
    permission_id INTEGER PRIMARY KEY AUTOINCREMENT,
    file_id INTEGER NOT NULL,
    user_id INTEGER NOT NULL,
    granted_by INTEGER NOT NULL,
    granted_date TIMESTAMP NOT NULL,
    FOREIGN KEY (file_id) REFERENCES files(file_id),
    FOREIGN KEY (user_id) REFERENCES users(user_id),
    FOREIGN KEY (granted_by) REFERENCES users(user_id),
    UNIQUE(file_id, user_id)
)
''')
```

Check existence:

- The system checks for the existence of the encrypted data and verifies whether the user has the necessary shared access rights.

Decode file:

- If the data is located, the system fetches it and proceeds to decode the data to make it accessible.
- Finally, the system returns a success message then encode the files again and returns a success message.

4.1.6 Edit File

```python
def handle_edit_file(request, ip_address=None):
    """
    Handle file edit request for .txt files
    """
    username = validate_input(request.get("username"))
    file_id = request.get("file_id")
    encrypted_data_str = request.get("data")  # Base64-encoded encrypted package

    if not username or not file_id or not encrypted_data_str:
        logger.debug(f"Invalid input: username={username}, file_id={file_id}")
        return {"status": "error", "message": "Missing username, file ID, or file data"}

    try:
        file_id = int(file_id)  # Ensure file_id is integer
    except (TypeError, ValueError):
        logger.debug(f"Invalid file_id: {file_id}")
        return {"status": "error", "message": "Invalid file ID"}

    try:
        conn = get_db_connection()
        cursor = conn.cursor()

        # Get user_id
        cursor.execute("SELECT user_id FROM users WHERE username = ?", (username,))
        user = cursor.fetchone()
        if not user:
            logger.debug(f"User not found: username={username}")
            return {"status": "error", "message": "User not found"}
        user_id = user['user_id']
```

```python
        # Check if user owns the file and it's a .txt file
        cursor.execute("""
            SELECT f.file_id, f.filename, f.original_filename, f.file_path, f.owner_id, f.file_size
            FROM files f
            WHERE f.file_id = ? AND f.is_deleted = 0 AND f.owner_id = ?
        """, (file_id, user_id))
        file = cursor.fetchone()
        if not file:
            logger.debug(f"File not found or not owned: file_id={file_id}, user_id={user_id}")
            return {"status": "error", "message": "File not found or not owned"}

        # Get encryption key and IV
        cursor.execute("SELECT key_encrypted, iv FROM file_keys WHERE file_id = ?", (file['file_id'],))
        key_data = cursor.fetchone()
        if not key_data:
            logger.debug(f"Encryption key not found for file_id={file['file_id']}")
            return {"status": "error", "message": "Encryption key not found"}

        # Verify file type from filename
        _, ext = os.path.splitext(file['original_filename'])
        if ext.lower() != '.txt':
            logger.debug(f"File is not a .txt file: file_id={file_id}, ext={ext}")
            return {"status": "error", "message": "Only .txt files can be edited"}
```

```
    # Decode new encrypted package
    try:
        encrypted_package_json = base64.b64decode(encrypted_data_str).decode()
        encrypted_package = json.loads(encrypted_package_json)
        ciphertext = base64.b64decode(encrypted_package["ciphertext"])
        nonce = encrypted_package["nonce"]  # Base64 string
        tag = encrypted_package["tag"]      # Base64 string
        metadata = encrypted_package.get("metadata", {})
    except (json.JSONDecodeError, KeyError) as e:
        logger.error(f"Invalid encrypted package: {str(e)}")
        return {"status": "error", "message": "Invalid encrypted package format"}

    # Verify metadata consistency
    if metadata.get("original_filename") != file['original_filename']:
        logger.debug(f"Metadata filename mismatch: received={metadata.get('original_filename')}, expected={file['original_filename']}")
        return {"status": "error", "message": "Filename mismatch in metadata"}

    # Update file content
    try:
        with open(file['file_path'], 'wb') as f:
            f.write(ciphertext)
    except IOError as e:
        logger.error(f"Failed to write file: {file['file_path']}, error: {e}")
        return {"status": "error", "message": "Failed to update file content"}

    # Update files table
    cursor.execute("""
        UPDATE files
        SET last_modified = ?, file_size = ?
        WHERE file_id = ?
```

```
        """, (datetime.datetime.now(), len(ciphertext), file_id))

        # Update file_keys table
        cursor.execute("""
            UPDATE file_keys
            SET key_encrypted = ?, iv = ?
            WHERE file_id = ?
        """, (tag, nonce, file_id))

        conn.commit()

        # Log the edit action
        log_action(user_id, "file_edit", f"Edited file: {file['original_filename']}", ip_address)

        return {
            "status": "success",
            "message": "File edited successfully",
            "file_id": file_id
        }

    except sqlite3.Error as e:
        logger.error(f"Database error: {e}")
        return {"status": "error", "message": f"Database error: {str(e)}"}
    except Exception as e:
        logger.error(f"Unexpected error: {type(e).__name__}: {e}")
        return {"status": "error", "message": f"Unexpected error: {str(e)}"}
    finally:
        close_connection(conn)
```

Building Blocks:

- **Database Connection**: Establishes a connection to the SQLite database to effectively manage and manipulate file data.
- **SQL Queries**: Executes queries to retrieve the file ID associated with the provided filename and user, followed by updates to the encrypted data.
- **Data Encoding**: Utilizes base64 encoding to securely encode the new data before storing it in the database.

Workflow:

User Input:

- The user begins the process by providing their username along with the filename of the file they wish to edit.

Session check:

- The program checks user authority and passes if user is identified.

File Identification:

- The system checks for the existence of the file ID based on the provided filename and verifies the ownership of the file.

Data Update:

- If the file ID is found, the function proceeds to update the encrypted data with the new information provided by the user.

Response Handling:

- Finally, the system returns a success message indicating that the update was successful, or an error message if the file is not found in the database.

4.1.7 Share File

```python
def handle_share_file(request, ip_address=None):
    """
    Handle file sharing request
    """
    username = validate_input(request.get("username"))
    session_id = request.get("session_id")
    user_id = validate_session(session_id)
    session_id = request.get("session_id")

    if not user_id:
        return {"status": "error", "message": "Invalid or expired session"}

    if not validate_session(session_id, username):
        return {"status": "error", "message": "Invalid or expired session, please login again"}


    file_id = request.get("file_id")
    share_with_username = validate_input(request.get("share_with_username"))

    if not file_id or not share_with_username:
        return {"status": "error", "message": "Missing required information"}

    try:
        conn = get_db_connection()
        cursor = conn.cursor()

        # Check if user owns the file
        cursor.execute("""
            SELECT file_id, original_filename
            FROM files
            WHERE file_id = ? AND owner_id = ? AND is_deleted = 0
```

```python
        """, (file_id, user_id))

        file = cursor.fetchone()
        if not file:
            return {"status": "error", "message": "File not found or you don't have permission to share"}

        # Get the user ID of the user to share with
        cursor.execute("""
            SELECT user_id, username
            FROM users
            WHERE username = ?
        """, (share_with_username,))

        share_user = cursor.fetchone()
        if not share_user:
            return {"status": "error", "message": "User to share with not found"}

        # Check if already shared
        cursor.execute("""
            SELECT permission_id
            FROM file_permissions
            WHERE file_id = ? AND user_id = ?
        """, (file_id, share_user['user_id']))

        if cursor.fetchone():
            return {"status": "error", "message": "File already shared with this user"}
```

```
    # Create permission record
    cursor.execute("""
        INSERT INTO file_permissions
        (file_id, user_id, granted_by, granted_date)
        VALUES (?, ?, ?, ?)
    """, (file_id, share_user['user_id'], user_id, datetime.datetime.now()))

    conn.commit()

    # Log the sharing
    log_action(
        user_id,
        "file_share",
        f"Shared file: {file['original_filename']} with user: {share_user['username']}",
        ip_address
    )

    return {"status": "success", "message": "File shared successfully"}
except Exception as e:
    logger.error(f"File sharing error: {e}")
    return {"status": "error", "message": "File sharing failed"}
finally:
    close_connection(conn)
```

Building Blocks:

- **Database Connection**: Establishes a connection to the SQLite database to manage and facilitate file sharing operations.
- **SQL Queries**: Executes queries to retrieve the file ID associated with the provided filename and shares it with another specified user.

Workflow:

User Input:

- The user provides their username along with the filename they wish to share with another user.

Session check:

- The program checks user authority and passes if user is identified.

File Identification:

- The system checks for the existence of the file ID based on the provided filename and verifies the ownership of the file.

Share Insertion:

- If the file ID is found, the function inserts the share information into the database, allowing the designated user access.

Response Handling:

- Finally, the system returns a success message indicating that the file has been successfully shared, or an error message if the file is not found in the database.

4.1.8 View Logs

```python
def handle_view_logs(request, ip_address=None):
    """
    Handle admin request to view audit logs
    """
    username= validate_input(request.get("username"))

    if not username:
        return {"status": "error", "message": "Invalid or expired session"}

    try:
        conn = get_db_connection()
        cursor = conn.cursor()

        # Check if user is admin
        cursor.execute("""
            SELECT is_admin
            FROM users
            WHERE user_id = ?
        """, (username,))

        user = cursor.fetchone()
        if username!='admin':
            return {"status": "error", "message": "Admin privileges required"}

        # Get logs with limit and pagination
        limit = request.get("limit", 100)
        offset = request.get("offset", 0)
```

```python
        cursor.execute("""
            SELECT l.log_id, l.user_id, u.username, l.action_type, l.action_details,
                   l.ip_address, l.timestamp, l.signature
            FROM audit_logs l
            LEFT JOIN users u ON l.user_id = u.user_id
            ORDER BY l.timestamp DESC
            LIMIT ? OFFSET ?
        """, (limit, offset))

        logs = [dict(row) for row in cursor.fetchall()]

        # Log the viewing of logs
        log_action(username, "view_logs", f"Viewed audit logs (limit: {limit}, offset: {offset})", ip_address)

        return {
            "status": "success",
            "message": "Logs retrieved successfully",
            "logs": logs
        }
    except Exception as e:
        logger.error(f"View logs error: {e}")
        return {"status": "error", "message": "Failed to retrieve logs"}
    finally:
        close_connection(conn)
```

Building Blocks:

- **Auditing Mechanism**: Implements a logging system that records user actions and system events for accountability and monitoring.

Workflow:

Action Recording:

- Each significant action taken by users such as logins, file uploads, and deletions is recorded in a dedicated log database to maintain a detailed history.

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS audit_logs (
    log_id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER,
    action_type TEXT NOT NULL,
    action_details TEXT,
    ip_address TEXT,
    timestamp TIMESTAMP NOT NULL,
    signature TEXT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
)
''')
```

Log Details Capture:

- The logs capture essential details, including the timestamp of the action, the user ID associated with the action, the specific action performed, and any relevant metadata that provides context.

Administrator Access:

- Administrators have the ability to access these logs for auditing purposes, which helps in identifying anomalies, tracking user behavior, and detecting any unauthorized activities.

4.1.9 Libraries

Functionalities	Libraries
User Register	Sqlite3, hashlib, secrets, re
User Login	Sqlite3, hashlib, pyotp, secrets, re
Reset Password	Sqlite3, hashlib, secrets, re
Upload File	Os, Crypto.Cipher.AES, Crypto.Random, base64, sqlite3
Download File	Os, Crypto.Cipher.AES, base64, sqlite3
Edit File	Os, Crypto.Cipher.AES, Crypto.Random, base64, sqlite3
Share File	Sqlite3, secrets
View Logs	Sqlite3, datetime, os

Library	Purpose
`sqlite3`	For database management and operations.
`os`	For operating system interactions, such as file handling.
`hashlib`	For hashing passwords and data.
`secrets`	For generating cryptographically secure random numbers.
`datetime`	For managing date and time.
`Crypto.Cipher.AES`	For AES encryption and decryption.
`Crypto.Random`	For generating random bytes.
`Crypto.Protocol.KDF`	For key derivation functions.
`Crypto.Hash`	For cryptographic hashing.

`socket`	For network communication between client and server.
`json`	For data serialization and deserialization.
`base64`	For encoding and decoding binary data.
`pyotp`	For implementing time-based OTP authentication.
`re`	For input validation using regular expressions.
`time`	For managing delays and time-related functions.

4.1.10 Challenges

1. Secure OTP Generation and Validation
Details: OTPs are generated and sent to users for authentication. Ensuring the OTPs are securely generated and not intercepted is critical.
Challenges:

- Storing OTPs securely to prevent unauthorized access.
- Preventing OTP reuse

Solution: OTPs are stored temporarily in the database with timestamps and invalidated after use or expiration.

2. Securing Communication Between Client and Server
Details: Sensitive data, including OTPs, is transmitted between the client and server.
Challenges: Protecting data from interception or tampering during transmission.
Solution: Encrypting all data in transit.

3. Preventing Brute Force and Replay Attacks
Details: Brute force attempts to guess OTPs or reuse intercepted OTPs are potential threats.
Challenges: Managing multiple failed attempts and ensuring OTPs are not reused.
Solution: Rate limiting is implemented for OTP validation, and OTPs are invalidated after one use.

5. Test Cases:

5.1 SQL Injection Attack:

```
sql_patterns = [
    "--", ";--", ";", "/*", "*/", "@@", "@",
    "EXEC", "EXECUTE", "INSERT", "DROP", "DELETE", "UPDATE", "SELECT",
    "UNION", "CREATE", "ALTER", "TRUNCATE", "DECLARE", "WAITFOR"
]
```

Test: Injected malicious SQL queries into input fields like **username**.

Result: All queries were sanitized, and no unauthorized access was achieved.

5.2 Unauthorized users unable to read uploaded files:

Test: Attempted to access uploaded files using an unauthorized user account.

Result: Access was denied with an "Unauthorized access" message, confirming that unauthorized users cannot read uploaded files.

6. Future Works:

6.1 Graphical User Interface (GUI):

- Implementing a user-friendly GUI to replace the current command-line interface for better usability.

6.2 Scalability:

- Modifying the architecture to support large-scale deployment with multiple servers handling file storage and user management.

6.3 Performance Optimization:

- Improving the efficiency of encryption and decryption processes for handling larger files or increasing the speed of file operations.

7. Reference:

[1]K. Abhishek et al., "A Comprehensive Study on Two-factor Authentication with One Time Passwords," in Computer Networks & Communications (NetCom), New York, NY: Springer New York, 2013, pp. 405–415. doi: 10.1007/978-1-4614-6154-8_40

[2]A. S. Almazyad, Y. Ahmad, and D. Slezak, "A New Approach in T-FA Authentication with OTP Using Mobile Phone," in SECURITY TECHNOLOGY, PROCEEDINGS, The Netherlands: Springer, 2009, pp. 9–17. doi: 10.1007/978-3-642-10847-1_2

[3]S. W. Boyd, A. D. Keromytis, M. Yung, M. Jakobsson, and J. Zhou, "SQLrand: Preventing SQL Injection Attacks," in APPLIED CRYPTOGRAPHY AND NETWORK SECURITY, PROCEEDINGS, Germany: Springer Berlin / Heidelberg, 2004, pp. 292–302. doi: 10.1007/978-3-540-24852-1_21

[4]K. Rana, "Classification of SQL Injection Attacks And Using Encryption As A Countermeasure," International journal of advanced research in computer science, vol. 2, no. 1, 2011.

[5]Shyam Babu Sah and S. Gawade, "Enhancing video encryption: AES and blowfish algorithms with random password generation," ACCENTS transactions on image processing and computer vision, vol. 9, no. 25, pp. 9-, 2023, doi: 10.19101/TIPCV.2023.924002