COMP3334 – Project Report

Course code: COMP3334

Team No.: Team 58

Student ID	Name	Overall Contribution
[redacted]	[redacted]	25%
[redacted]	[redacted]	25%
[redacted]	[redacted]	25%
[redacted]	Chung Cheuk Hang	25%

Abstract

This report details the design and implementation of a secure online storage system for COMP3334, featuring a client-server architecture with HTTPS communication secured by a self-signed SSL certificate. Core functionalities include user authentication, file encryption, access control, and auditing, enhanced by multi-factor authentication (MFA) via email OTP. Using Python with libraries like Flask, Cryptography, and SQLite, our system protects against passive adversaries (server and unauthorized users) while meeting all project requirements. We demonstrate security through encryption, secure key management, and practical test cases.

1.1 Introduction

Online storage systems have become essential for managing data in modern life, yet they pose significant security risks due to the sensitive nature of uploaded content. This project aims to develop a secure online storage application with a client-server architecture that protects user data from unauthorized access and ensures operational integrity. Our solution addresses core requirements—user management, data encryption, access control, and log auditing—while incorporating multi-factor authentication (MFA) via email OTP and SSL encryption to enhance security. This report outlines our approach, implementation, and evaluation.

1.2 Background

The purpose of online storage systems is to allow users to manage files, such as upload files, delete files, share files with another user, and manage their own accounts. To protect user privacy and data integrity, the system should ensure that all data should not be leaked, even the system side. For example, MFA Authentication allows the system to ensure the true user is operating his/her own account’s action. This report assumes the threat models and explains how the system protects data from attackers.

2. Threat Models

Adversaries

Adversary	Who	Abilities	Goals
Passive Server	The machine running the Server program.	Monitors communication and stored data from a Client. Reads messages.	Decrypt the Client's uploaded files.
Unauthorized User	An individual attempting to access the system without proper authorization.	May use a legitimate user's computer.	Access the online files of other users.

3. Algorithms and Design

3.1 Core Functionalities

1. User Management

- **Registration:** Users input a unique username, email, and password. The password is hashed using PBKDF2-HMAC-SHA256 with a random 16-byte salt. An RSA key pair (2048-bit) and a 32-byte master key are generated; the master key is encrypted with a key derived from the password, and the private key is encrypted with the master key. Email syntax is validated using checkmail.py.
- **Login:** Verifies the hashed password and initiates MFA with a 6-digit OTP sent via email.
- **Password Reset:** Re-hashes the password with a new salt, re-encrypting the master and private keys.
- **Tools:** cryptography (PBKDF2, RSA), checkmail.py (email validation).
- **Workflow:** Client sends hashed credentials and encrypted keys to the server over HTTPS, stored in SQLite.

2. Data Encryption

- **Upload:** Files are encrypted with AES-GCM (256-bit key, 12-byte IV, 16-byte tag) using a random file key, which is then encrypted with the master key.
- **Download:** Client retrieves the encrypted file and key, decrypting the file key with the master key, then the file itself.
- **Tools:** cryptography (AES-GCM).
- **Workflow:** Encrypted data is sent to the server via HTTPS, ensuring the server cannot access plaintext.

3. Access Control

- **Ownership**: SQLite queries check the owner_username field to restrict edit/delete to file owners.
- **Sharing**: The file key is decrypted with the owner's master key, re-encrypted with the recipient's RSA public key, and stored in the shared_files table.
- **Tools**: cryptography (RSA).
- **Workflow**: Sharing requests are validated server-side, ensuring only owners can share.

4. Log Auditing

- **Logging:** Operations (e.g., login, upload) are timestamped and stored in SQLite with username and details, accessible by the admin account.
- **Tools:** SQLite.
- **Workflow:** Each critical action triggers a log entry, ensuring non-repudiation.

5. General Security

- **File Name Validation:** Strips "../" and ".." from filenames to prevent directory traversal.
- **SQL Injection:** Uses parameterized SQLite queries.
- **Communication:** HTTPS with localhost.crt and localhost.key secures client-server data exchange.

3.2 Extended Functionality

We implemented four extended functionalities—Multi-Factor Authentication (MFA), SSL Encryption, Password Strength, and Secure File Deletion—exceeding the minimum requirement to bolster security.

1. Multi-Factor Authentication (MFA)

- **Design**: After password verification, a 6-digit OTP is generated, emailed to the user, and must be entered within 5 minutes. The OTP is stored in SQLite with an expiration timestamp and verified server-side.
- **Library**: smtplib (SMTP via mailservice.py), random (OTP generation).
- **Workflow**:
    1. User enters password; server validates hash.
    2. OTP generated and emailed using credentials from .env.
    3. User inputs OTP; server checks against stored value and expiration.
- **Purpose**: Adds a second authentication layer, thwarting unauthorized access even if passwords are compromised.

2. SSL Encryption

- **Design**: Client-server communication is secured with HTTPS using a self-signed SSL certificate (localhost.crt) and private key (localhost.key). The certificate includes a Subject Alternative Name (SAN) for localhost, ensuring compatibility with modern TLS requirements.
- **Tools and Library**: OpenSSL (certificate generation), Flask (ssl_context), requests (certificate verification).
- **Workflow**:
    1. Server runs with ssl_context=('localhost.crt', 'localhost.key') at https://localhost:5000.
    2. Client uses requests with verify="localhost.crt" to authenticate the server and encrypt data in transit.
- **Purpose:** Protects against eavesdropping by passive adversaries, ensuring confidentiality and integrity of data exchanged.
- **Technical Challenge:** Flask initially rejected OpenSSL-generated certificates due to missing SAN and key usage extensions, causing server startup failures.
- **Solution**: After trial and error, we used this OpenSSL command to generate a compatible certificate:

```
openssl req -x509 -out localhost.crt -keyout localhost.key \
  -newkey rsa:2048 -nodes -sha256 \
  -subj '/CN=localhost' -extensions EXT -config <( \
   printf "[dn]\nCN=localhost\n[req]\ndistinguished_name =
dn\n[EXT]\nsubjectAltName=DNS:localhost\nkeyUsage=digitalSignature\nextendedKeyUsage=se
rverAuth")
```

- **Outcome**: Flask accepted the certificate, enabling secure HTTPS communication.

3. Password strength

- **Design**: Enforces strong passwords during registration and reset using the password-strength library. Passwords are evaluated for entropy, requiring a minimum strength score (e.g., "strong") based on length, complexity, and character variety.
- **Library**: password_strength (PyPI: https://pypi.org/project/password-strength/).
- **Workflow**:
    1. User inputs password.
    2. Client checks strength with PasswordStats; if weak, prompts re-entry.
    3. Strong passwords proceed to hashing and storage.
- **Purpose**: Mitigates weak password vulnerabilities, enhancing account security.
- **Implementation Note**: Integrated into client.py's register and change_password functions, rejecting passwords below a defined threshold.

4. Secure File Deletion:

- **Design:** Before deleting encrypted files from the server, overwrites them with random data to prevent recovery. Uses multiple passes (e.g., 3) of random bytes matching the file size.
- **Library:** os (file operations), os.urandom (random data).
- **Workflow:**
    1. User requests file deletion.
    2. Server locates encrypted file (e.g., files/<file_id>.enc).
    3. Overwrites file with random data (3 times).
- Deletes file and SQLite entry.
- **Purpose:** Ensures deleted files cannot be recovered by adversaries with server access, enhancing data privacy.

5. Rate Limit:

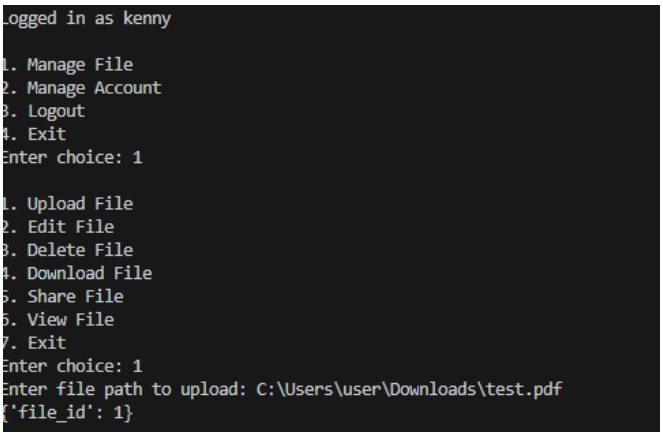
- **Design:** Enforces multiple time-based limits simultaneously: 100 requests per day, 20 per hour, 5 per minute, and 2 per second.
- **Library:** Flask-Limiter (PyPI: https://pypi.org/project/Flask-Limiter/)
- **Workflow:**
    1. The Limiter object is created and bound to the Flask app instance.
    2. For each incoming request, if any limit is exceeded, a 429 Too Many Requests response is returned, and the request is blocked.
- **Purpose:** Protects the application from abuse, preventing denial-of-service (DoS) attacks and ensuring fair resource usage.

Test Cases

Test Case 1: Uploaded file protection

Before the file was uploaded to the server, all uploaded files were encrypted by file key using AES. Moreover, the file keys were encrypted by the Client's master key. Therefore, unauthorized users and the server side cannot view the file's contents.

Test.pdf owned by Kenny



Encrypted file in server

Authorized users can download and read the original file

```
Logged in as kenny

1. Manage File
2. Manage Account
3. Logout
4. Exit
Enter choice: 1

1. Upload File
2. Edit File
3. Delete File
4. Download File
5. Share File
6. View File
7. Exit
Enter choice: 4
Enter file ID to download: 1
File downloaded successfully
```

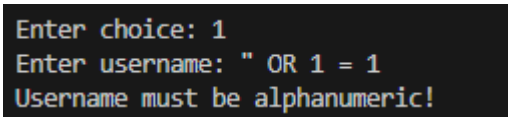
Unauthorized users cannot download the file

```
Logged in as paco

1. Manage File
2. Manage Account
3. Logout
4. Exit
Enter choice: 1

1. Upload File
2. Edit File
3. Delete File
4. Download File
5. Share File
6. View File
7. Exit
Enter choice: 4
Enter file ID to download: 1
{'error': 'Unauthorized'}
```

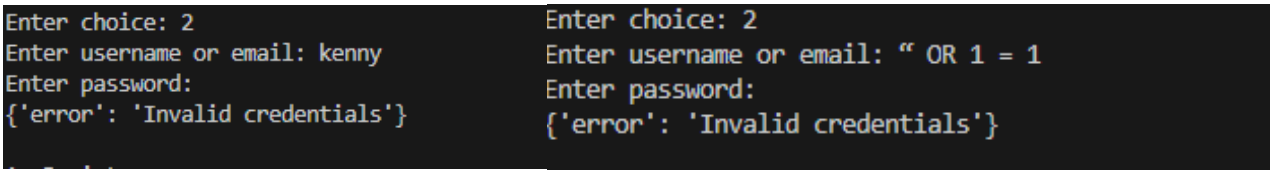
Test Case 2: SQL Injection Attacks

All the inputs related to the database were checked in a parameter manner. Therefore, the attacker does not change the SQL command, such as a comment attack, to gain private data from the database.

Test Registration Input



Test Login Input (input “ OR 1 = 1)



Future Works

1. Encrypt the database

2. make the files directory in the server inaccessible by others except by the server program

3. Implement rate limiting and account lockout to prevent attackers from deciphering accounts

4. Provide a Version control function to improve data availability

5. Limit each uploaded file size to 2GB to protect the server

6. Implement the user account deletion and data purge to minimize the risk of data leaking

7. Implement the file previews to make it more convenient for users.

8. Implement real-time notification for detect suspicious activity

Reference

[1] A. -A. Saifee, “Flask-Limiter”, *Flask-Limiter*. [Online]. Available: https://flask-limiter.readthedocs.io/en/stable/ [Accessed: Apr 5, 2025].

[2] Individual Contributor, *Cryptograph*. [Online]. Available: https://cryptography.io [Accessed: Apr 5, 2025].

[3] Internet Security Research Group, Certificates for localhost, *Let’s Encrypt*. [Online]. Available: https://letsencrypt.org/docs/certificates-for-localhost/ [Accessed: Apr 5, 2025].

[4] M. Vartanyan, “Password Strength”, *Python Package Index*. [Online]. Available: https://pypi.org/project/password-strength/ [Accessed: Apr 5, 2025].

[5] OpenSSL Software Foundation, *OpenSSL*. [Online]. Available: https://www.openssl.org [Accessed: Apr 5, 2025].

[6] Pallets, “Flask Documentation (stable)”, *Flask*. [Online]. Available: https://flask.palletsprojects.com/en/stable/ [Accessed: Apr 5, 2025].

[7] R. Subramanian, “Secure File Deletion using Python”, *Medium*. [Online]. Available: https://ramanantechpro.medium.com/secure-file-deletion-using-python-d0456a0cfc98 [Accessed: Apr 5, 2025].

[8] SQLite, *SQLite Home Page*. [Online]. Available: https://www.sqlite.org [Accessed: Apr 5, 2025].