Cloud file storage system: Report

Architecture overview

Client-Server Model

The secure file-sharing application is designed with a client-server architecture, emphasising security and user privacy. The system consists of two main components:

Client Application: A Python-based desktop application using PyQt5 for the graphical user interface (GUI). It allows users to register, authenticate, upload, download, and share files securely. Having a local application rather than a web application also allows for data to be stored locally on a user's machine and leverages the security features of the physical machine.

Server Application: A web server built with Python and Django, utilising the Django REST Framework (DRF) to provide RESTful API endpoints for client interactions.

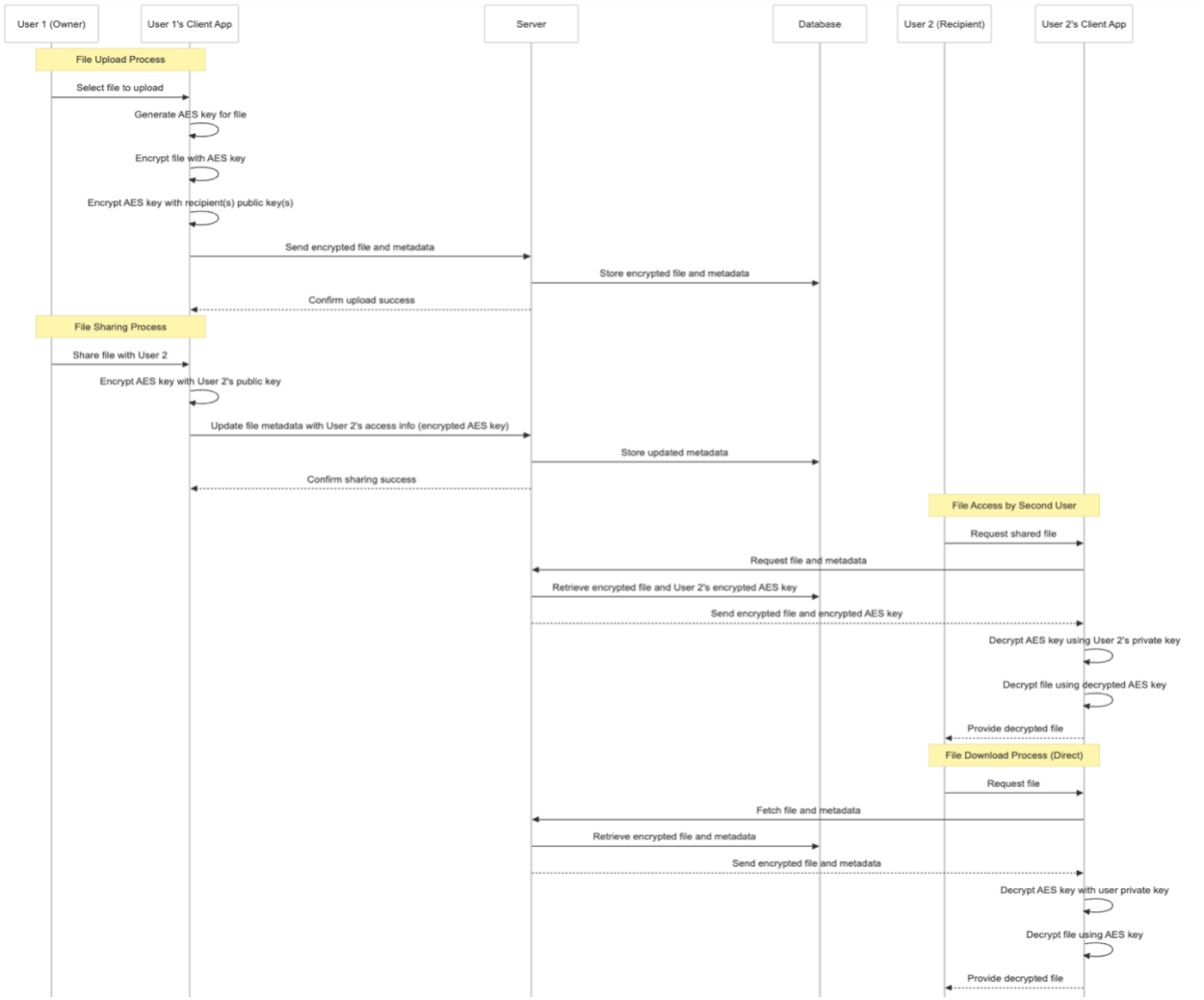

Figure 1 - Sequence diagram showing the file upload and file sharing process between two authenticated users

Security features

End-to-End Encryption

- **Asymmetric Encryption (RSA)**:

    - Each user generates a 2048-bit RSA key pair upon registration.
    - The private keys are stored on the user's local machine that they registered the account with (../user_keys/{username}_private_key.pem).
    - **Public Keys**: Uploaded to the server and stored in the database for key exchange.
- **Symmetric Encryption (AES)**:
    - Files are encrypted using AES with a 256-bit key in Cipher Feedback (CFB) mode.
    - A unique AES key is generated for each file upload.

Zero-Trust Model

- The server stores only encrypted files and cannot access decryption keys.
- All encryption and decryption occur on the client side.
- The server facilitates file storage and sharing without knowledge of file contents.

Secure Communication

- All client-server communications are secured using HTTPS with TLS encryption.
- Self-signed certificates are used during development and are replaceable with CA-signed certificates in production.

Access Control

- **Authentication**: Users authenticate via token-based authentication using Django's built-in system.
- **Authorisation**: File access is controlled through cryptographic keys rather than server-side permissions.

Justification of Design and Technologies Used

Cryptographic choices

RSA Encryption:

- Selected for its robust security and widespread acceptance.
- Enables secure exchange of AES keys during file sharing.
- A key size of 2048 bits balances security with performance.

AES Encryption:

- Chosen for efficient symmetric encryption of large files.
- AES-256 provides strong security that is suitable for sensitive data.
- CFB mode allows encryption of data streams of arbitrary length.

Key Management:

- Private keys remain on the client side to maintain user control.
- Public keys stored on the server facilitate secure key exchange.
- Certification for HTTPS is done through self-certified certificates, in a professional deployment, we would use a more official organisation for certification

Cryptography library

- The cryptography library is a well-managed library that provides encryption methods for RSA and AES Encryption, making it an ideal option for this project.

Design Decisions

Zero-Knowledge Server:

- By handling all encryption client-side, the server cannot decrypt user data. This Aligns with the zero-trust approach, enhancing security even if the server is compromised.

Local Key Storage:

- Storing private keys locally prevents exposure of sensitive keys over the network. This also allows users to maintain sole control over their private keys.

Token-Based Authentication:

- Provides a secure method for session management. Tokens reduce the risk associated with transmitting credentials multiple times.

User Experience Considerations:

- The GUI simplifies complex cryptographic operations for the user. Clear prompts and messages guide users through the application's functionalities.

Evidence of System functionality

User Registration and authentication

When you click register, you are presented with a widget that will ask you to enter a username and password. When you click register, if the server is running correctly, there should be a message saying, “user registered successfully” When you try and log in, but the information is wrong, you will receive an alert saying, “failed to login: invalid credentials”. Within the database, we are storing and comparing a hashed version of the password using Django’s built in password hashing library. This allows us to have a more trustworthy and secure system for storing our user data.

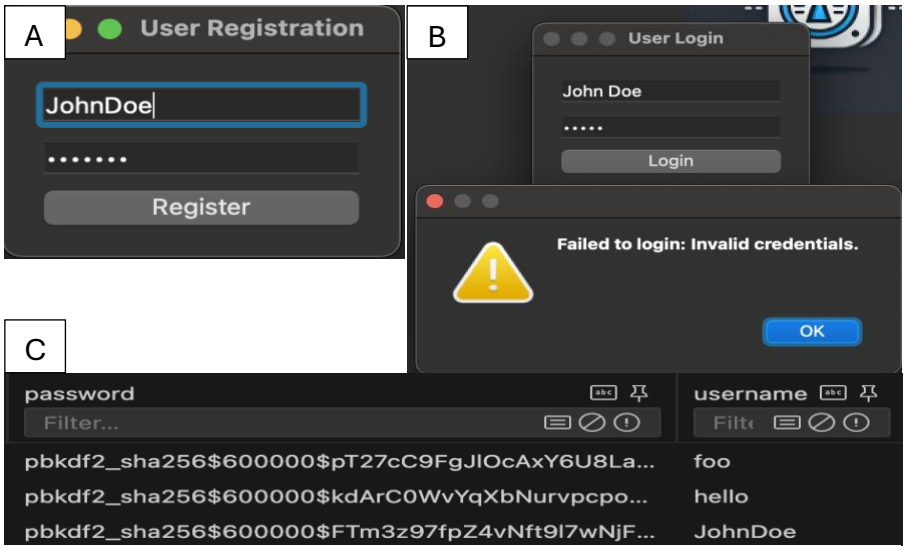

Figure 2 - A. User registration widget, B. User Login when credentials are invalid, C. Database entry of users

File upload and Encryption

After logging in, you have the option to upload a file. When you select this option, you will be able to select a file. When selected, the file is encrypted with AES. And the file, along with a version of the AES key encrypted with the user’s public key is also sent to the server. Upon trying to read the file on the server, the file says it is corrupted due to the encryption on the file. Within the database, the data is stored with its filename, location on the server, and the encrypted AES key.

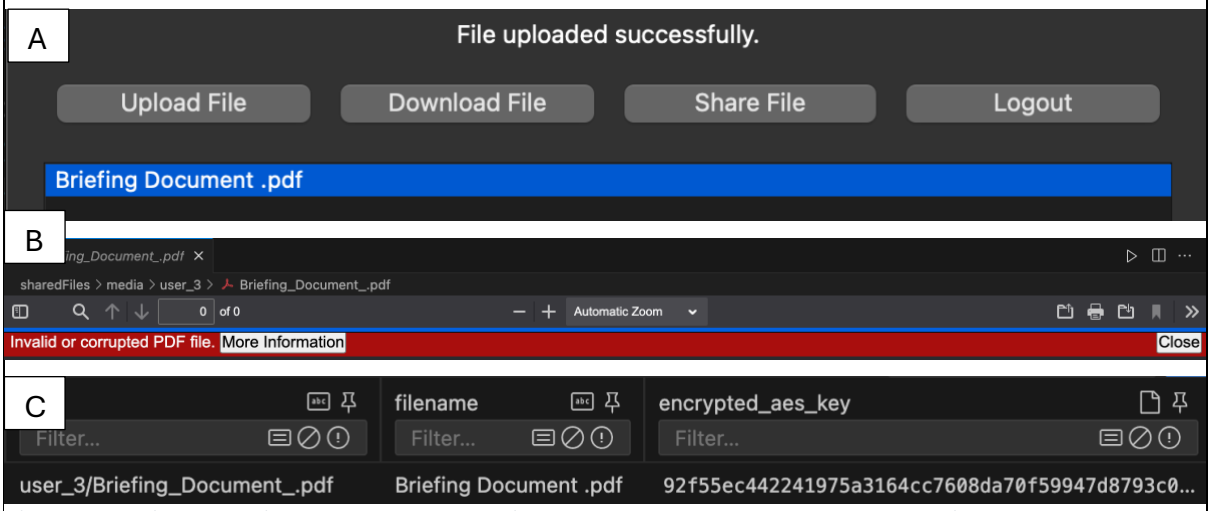

Figure 3 - A. Widget showing the newly uploaded file. B. Document stored on the server showing a corrupted message when attempted to read. C. Database entry of the file

Secure File Sharing

With the file selected, you can enter the username of who you want to share this file with. The code will then find their public key, decrypt the AES key and make a newly encrypted version with the new user's public key, all locally. The file is then uploaded, and the new AES key is stored in the shared_file table in the database.

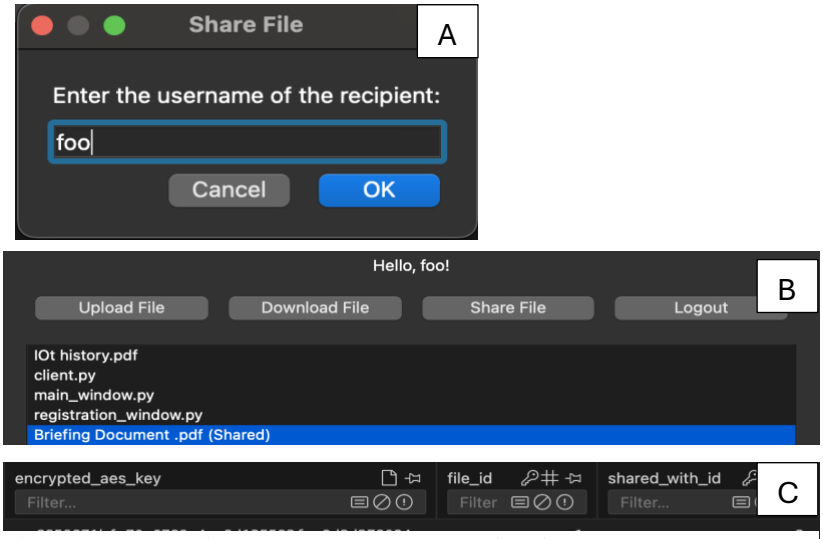

Figure 4 - A. Entry widget for the user to share a file with. B. The newly shared file present in the user's storage page. C. Database entries of the shared files

File Download and Decryption

When logged in, you can select a file and choose "download" When selected you can choose where to download the file to. After choosing where to download the file, the AES key is retrieved, and decrypted using the user's private key, then the file is decrypted using this AES key, and then saved to the correct location.

Once downloaded, you will see a widget saying, ‘file downloaded and decrypted successfully.’

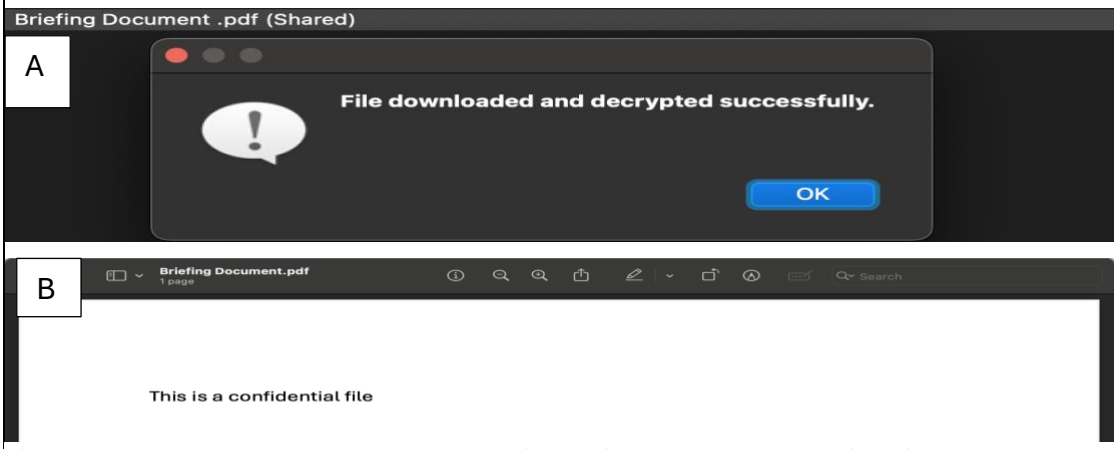

Figure 5- A. Response after successfully downloading the file. B. The unencrypted file being opened and readable.

Conclusion

The secure file-sharing application successfully implements end-to-end encryption, ensuring that users can store, retrieve, and share files securely over a network. By utilising robust cryptographic algorithms and a zero-trust model, the system meets the requirements of providing secure cloud storage without relying on the cloud provider's trustworthiness. The design choices balance security and usability, demonstrating how cryptography can be effectively applied to enhance data privacy and access control.

References

Django Documentation. https://docs.djangoproject.com/
Django REST Framework. https://www.django-rest-framework.org/
PyQt5 Documentation. https://www.riverbankcomputing.com/static/Docs/PyQt5/
Cryptography Library. https://cryptography.io/en/latest/