University of St Andrews
School of Computer Science

November 18, 2024

CS4203

Secure File Server

Student ID: [redacted]

1 Overview and Threat Model

The system consists of a client and a server, both implemented using Python. The server is built with Flask and handles requests over HTTPS, enforcing mutual TLS (mTLS) for secure client-server communication. The client handles user authentication, file encryption/decryption, and key management.

Users can use a client to register using a username and password and can log in once registered. To further increase security, 2-factor-authentication (2FA) using `Flask-Mailman`[1] and SendGrid[2] was set up as part of the registration step, but after having problems with emails not arriving, the code sending the mail was commented out and to simulate 2FA, the 6-digit is written to `data/server/<uuid>/2fa code.txt` when running the server on your local machine.

Once logged in, the user is presented with the following operations.

```
INFO:client:# Please choose an action:
INFO:client:  upload <filename>
INFO:client:  download <filename>
INFO:client:  show users
INFO:client:  show files
INFO:client:  share file <filename> <username>
INFO:client:  get shared <filename>
INFO:client:  remove user
INFO:client:  logout
```

The client offers some basic security measures to prevent misuse, such as not permitting the download, share file, or get shared option, if the user has not even uploaded a file yet. The client also accepts only exactly these inputs and will sanitise any user input to allow only alphanumeric characters, underscores, hyphens, and periods, and strips of any directory components. With regard to all cryptography in this project, one can only make it more secure by selecting different algorithms, but this report focusses on how and where cryptography is implemented.

For the purpose of this project, the server storage is `data/server/`, the certificate authority (CA) lives in `data/certificates` and the client uses `data/client`. In practice, these would be physically separated machines.

1.1 Threat Model

Eavesdropping: The adversary can intercept communications between the client and the server. To prevent this, mTLS is used to verify both the client and the server and to secure the connection. A short Wireshark analysis showed that TLSv1.3 was being used and handshakes were performed.

[1] https://pypi.org/project/Flask-Mailman/
[2] https://sendgrid.com

Server Compromise: The adversary can gain access to the server's storage, but cannot obtain users' private keys. All files are encrypted using either a unique file encryption key (FEK), or the private RSA key of a client - the only unencrypted file on the server (keys.json) only contains randomly generated user IDs and keys encrypted using the users' public RSA keys. With a sufficient key length (2048 bits or more), it is considered infeasible to encrypt it.

Adversary-in-the-Middle Attacks: The adversary can attempt to impersonate the server or client. The use of a CA for authenticity, mTLS for identity verification, and a key rotation mechanism (`server_cert_rotation.py`) to reduce key exposure. Furthermore, a secret API token is used as an additional authentication for the client (Sec.2.1).

Unauthorized Access: The adversary can try to authenticate as a legitimate user without valid credentials. This is prevented on the client side by setting a minimum password length to enforce a strong password. 2FA provides an additional layer of security.

(D)DOS Attack: The server can be subject to a (D)DOS attack. To mitigate this, `Flask-Limiter`[3] is used to limit the request rate that the server can request. This library has limitations in that it is not effective if the attacker uses multiple IPs and, in practice, it would be necessary to use DDOS protection from known hosting providers.

1.2 Key Management

User Master Key: Derived from the user's password and unique salt using Scrypt, providing resistance against brute-force attacks. Storing the Salt on the client device acts like a 2FA machanism, so the service is bound to the local device the user registered on.

Symmetric file encryption key): Generated uniquely per file using Fernet, ensuring forward secrecy of file contents.

Symmetric Key for User Registry: Used to encrypt the user registry, encrypted individually for each user with their public key.

User Keypair: Used for signing files, checking validity and prevent tampering, as well as authentication for mTLS and encryption of user registry FEK

Server Keypair: used for mTLS handshake and signed by the CA. Further improvements would using the server private key to sign content like the user registry to add an additional layer to detect tampered files.

2 Implementation Details

2.1 Server Implementation

The server is a Flask application that accepts HTTP POST and GET requests over HTTPS. It enforces mTLS to authenticate clients using X.509 certificates signed by a trusted CA. The CA is modelled by a folder (`data/certificates/`) in the project directory. As an additional layer of security, the server checks the header of client requests for the presence of a secret token, which ensures that the user uses an official client. The secret token is loaded into the environment context at the start of the server and client. In practice, this can be achieved by using secret management services such as AWS Secrets Manager[4] to supply the server and client with the token and store it securely.

[3] https://flask-limiter.readthedocs.io/en/stable/
[4] https://aws.amazon.com/secrets-manager/

```
API_TOKEN = os.getenv("API_TOKEN", "MYSECRETTOKEN")
...
token = request.headers.get("Authorization")
if token != API_TOKEN:
    return jsonify({"message": "Unauthorized"}), 401
```

Upon startup, the server generates its own certificate and private key if they do not exist and loads them into an SSL context, setting `context.verify_mode = ssl.CERT_REQUIRED`. This enforces TLS for all requests. The server certificate is signed by the same CA where the client needs to sign its certificates to establish trust.

2.2 User Registration

The server maintains a list of registered unique user identifiers (UUIDs). When the first user registers, the server automatically creates a user registry. This registry is encrypted using a symmetric key K_{sym} generated by the server, which is immediately encrypted with the first user's public key (RSA with OAEP padding) and stored. The server adds the encrypted FEK for the user registry to the according UUID in `data/server/users/keys.json`:

```
{
  "user1_id": "E_pub_key_user1(K)",
  "user2_id": "E_pub_key_user2(K)",
  "..."
}
```

For subsequent registrations of clients, the server requests already registered clients to encrypt the FEK that was used to encrypt the user registry with the public key of the newly registered user. For the purpose of this assignment, the existing *active* client is just represented by the first registered user, and the key is accessed by accessing `data/client/users/<uuid>/private_key.pem`. In a real deployment, the server would complete the registration and the new user would be able to use the system without being able to share files. If a previously registered user logs in, his client would automatically check if the procedure is necessary for a new user. The new user is then notified via email, sent using `flask-mailman` and SendGrid[5]. Due to time constraints, I opted for the workaround.

Client Side: The client asks for username, email and password. The client sends the email address to the server, which creates the 2FA code, which is currently being performed locally. The client asks for the 2FA code to be entered. The client generates a local RSA key pair and a unique UUID. Then it generates a certificate sign request (CSR), which is signed by the trusted CA. The client saves the keypair in a specified local directory `/data/client/users/<uuid>/`. The client then sends the UUID, the 2FA code and the signed certificate, which contains the public key, to the server. If the code is wrong, all data gets removed and the server returns an error, which triggers the client to also remove all data. The client also hashes the password and creates a unique salt, which is saved per user. The `bcrypt` library offers a method to verify the password using this hash. The system is designed in a way that, even if an attacker gains access to a clients local directory (and possibly the server), the attacker cannot use this information without the master password to decrypt other parts of the system in a feasible amount of time. This salt is used in combination with the user password to derive the master key of the user using `hashlib.scrypt()`, which is used to encrypt the unique FEKs, which are then used to encrypt the files:

$$K_{\mathrm{FEK}_i}(\mathrm{FileData}_i), \quad \text{where } \mathrm{FEK}_i = K_{\mathrm{MasterKey}}(\mathrm{FEK}_i), \ \ \mathrm{MasterKey} = \texttt{scrypt}(\mathrm{Password}, \mathrm{Salt})$$

[5] https://sendgrid.com

The encrypted files are also signed using the users private key to detect tampering.

2.2.1 File Handling

The server handles file storage and retrieval without knowing the contents of the file. Furthermore, the client hashes the filename to a number that appears random to avoid leaking information through the names of uploaded files and the client usernames are not known to te server, it only sees the UUIDs. The server enforces mTLS and checks the request header for the API token (Sec.2.1) before handling the contents of the message. This ensures only official clients can connect to the server.

Upload The user selects a file to upload (via the filename of a file in `data/client/files/`). The client encrypts the file using the unique FEK created with the Fernet library (AES-128 in CBC mode with HMAC) and signs it using the user's private key. The FEK is encrypted using the master key derived from the user password and the saved salt. This acts like a 2FA for the master key (physically stored salt and external password). The file metadata, which contain the encrypted FEK and the encrypted filename, is then stored in `data/client/<uuid>/`. The client then sends the hashed filename, hashed using SHA-256, and the encrypted file along with the signature as a POST request to the server.

Download When a user wants to download a file, the client loads the local file metadata, produces the hashed filename, and sends a GET request to the server. After successful authentication on the server side, the client receives an encrypted file and a signature, verifies the signature using the public key of the user, and then retrieves the FEK from the local metadata, decrypts it using the derived master key and decrypts the file to `data/client/downloaded_files`.

Sharing A user can share previously uploaded files with other registered users. When sharing a file, the client side takes note and adds the info to the local client metadata:

```
{
    "username": "username",
    "password_hash": "$2b$12$...",
    "unique_id": "user_id",
    "salt": "454afc...",
    "shared_files": {
        "hashed-filename-1": ["recipient_1_id", "..."], "..."}
}
```

To be able to do this, the client requests the public certificate of the recipient (retrieved via the user registry), encrypts the FEK for the selected file using the public key of the recipient, and signs the encrypted FEK using its own private key. The client also modifies the server-side user registry, and encrypts it again. The client then sends the encrypted user registry, which contains the new entry, to the server. After authorisation, the server overrides the existing registry with the updated version, which links the new recipient to the filename of the shared file, by adding the following entry to the recipients UUID.

```
"shared_with_me": [
    "file_id_1": {
    "encrypted_fek": "base64-encoded...",
    "signature": "base64-encoded...",
    "sender_id": "unique-id",
    "sender_name": "username"
    },
    "..."
]
```

If an attacker managed to override or delete this registry somehow, the file sharing mechanism is obviously compromised because all knowledge of shared files is lost. The users can still use the service to retrieve their own files. The attacker cannot decrypt any files. To overcome this, the registry can first be signed by the server itself, so clients can check for tampering, and clients can store backups of this file locally. This feature is also not implemented due to time constraints.

When the recipient logs in and chooses the option `show files`, he will see an output: `Shared files:  <filename> by <username>`, indicating a shared file. The user can then retrieve the shared file using `get shared <filename>`. This method differs from `download <filename>` in that it first retrieves the user registry to retrieve the sender id and the hashed filename, and checks the sender certificate against CA, then internally requests the shared file via the hashed filename from the server. Furthermore, the sender signature is checked using the sender public key, and only then the FEK is encrypted using the recipient's private key, which can then be used to decrypt the file.

So, to summarise, shared files are managed in the user registry with entries specifying which files are shared with which users, ensuring access control is maintained cryptographically, because the keys to encrypt the registry are itself encrypted using user public keys, and even the contents of the registry are not sufficient to decrypt any other files on the server.

3 Limitations

This approach tackles the threats mentioned in Sec.2.1, by way of introducing unique FEKs, deriving a master key from a users password, a locally stored salt, and the RSA keypair for authentication, combined with an API token to authenticate the client application itself. One attack vector that I was unable to fix in time is the fact that the server stores the user registry. This could be solved in future work by distributing it to all clients, without keeping a copy on the server itself. The other is the client side: The client user file (which contains the salt, password hash, and info about the files that have been shared to other users) is currently saved unencrypted on the client device. Although this does not contain sensitive information, this could be improved by using a key derivation function with a constant salt for all users, to derive another key, only from using the password, to encrypt client side information.

4 Conclusion

The implemented system successfully demonstrates a zero-knowledge file server that allows secure storage and sharing of files using cryptographic methods. By ensuring that all sensitive operations are performed client-side and that the server remains oblivious to the contents of the data it stores, the system achieves a high level of security at rest and in transit, suitable for environments where data confidentiality and integrity are critical.