Data Structures

User struct:

Username: string

Password: string

DecKey: PKEDecKey

SignKey: DSSignKey

UserNamespace struct:

Username: string

Filename: string

FileAccess struct:

MetadataLocation uuid

LengthLocation uuid

FileMetaData struct:

SharingTree: map[string][]UserNamespace

PendingInvitions: map[string][]Invitation

Owner: string

Invitation Struct:

InvitationPtr uuid

Username string

SymmetricEncryptedData Struct:

EncSalt: []byte

MacSalt: []byte

EncData: []byte

MacTag: []byte

AsymmetricEncryptedData Struct:

Ciphertext: []byte

Signature: []byte

User Authentication

How will you authenticate users?

When a user registers with his username and password, we generate two pairs of public and private keys for that user, one for public-key encryption and the other for digital signature. We store the public keys in Keystore while keeping the private keys as a field in a User struct. Then we create a User struct with a username, password, and two private keys. We encrypt-then-MAC the User struct with 2 different secret keys that are derived on the fly from the user's password and a randomly-generated salt using the password-based key derivation function. After that, we create a SymmetricEncryptedData struct with the encryption salt, HMAC salt, encrypted user struct, and HMAC tag of the encrypted struct. We marshal the struct and store it in Datastore.

When a user wants to verify his credentials with his username and password, we first get the corresponding SymmetricEncryptedData from Datastore and unmarshal it. We then verify its integrity by checking the HMAC tag. If the struct was not tampered with, we can decrypt it to get the user struct. Finally, we can compare the password inside the struct with the provided one. If the two passwords do not match, the user provides the wrong password. Otherwise, we successfully verify the user's identity.

What information will you store in Datastore/Keystore for each user?

For each user, we store his public key in KeyStore and his encrypted user struct in DataStore.

- KeyStore: {

    UUID(hash('Users/Enc/'+ username)): enc key

    UUID(hash('Users/Verify/'+ username)): verify key

    }

- DataStore: {

    UUID(hash('Users/' + username)): JSON of a SymmetricEncryptedData struct ({

        encSalt,

        macSalt,

        Enc(encKey, user struct),

        HMAC(macKey, Enc(encKey, user struct))

    })

    }

How will you ensure that a user can have multiple client instances (e.g. laptop, phone, etc.) running simultaneously?

To ensure that a user can have multiple client instances running simultaneously, we will get up-to-date information about the user at the beginning of every function and store the user to Datastore at the end whenever we make changes to the user struct. Also whenever we modify a file, we fetch the file from and store the file to Datastore at the beginning and the end of a function respectively to keep the data in Datastore in sync with our changes.

File Storage and Retrieval

How will you store and retrieve files from the server?

We will store files as a list of appended content. When a user first creates a file, we generate a unique key used to encrypt the file metadata along with its content, two random uuids where we store the file metadata (metaLocation) and number of parts the file is divided into (lengthLocation). At the metadata location, we store a FileMetadata struct which contains authorized users, pending invitations, the owner of the file. At the length location, we store the number of parts the file is divided into in a location. The number of parts field will be used to determine the UUID of the next appended piece of content.

To retrieve a file from the server, we first get the number of pieces of content appended in the length location. Starting the counter from 1 to the number of pieces, we are able to determine the UUID of each piece by calculating UUID(hash(metadata location + filename + counter)). From there, we can get the content in each piece and concatenate them in order to get the entire file content.

How will your design support efficient file append?

To append a new piece of content to a file, we only need to get the number of pieces appended so far in the length location. From there, we are able to calculate the UUID in Datastore where we should store the newly appended piece of content as UUID(hash(metadata location + filename + (numberOfParts + 1))). Then we encrypt the new content and store it there. Also, we need to increase the number of pieces in the file metadata by 1. As a result, we don't have to encrypt and decrypt the entire file when appending but only the file metadata and the size of the file metadata only scales with the number of users the file is shared with. Therefore, the append function only scales with the size of the content being appended and the number of authorized users.

File Sharing and Revocation

How will you allow files to be shared with other users?

When a file is created, the owner of the file will generate a secret key, which serves as a password to the file, and two UUIDs where we will store the file metadata struct and the number of parts the file is divided into.

When the file is shared with another user, we create a 3-part invitation, which contains filekey, file access (metadata location and length location) and owner public key. The three parts are chained with each other, so we only need to share the location of the first part. From there, the user the file is shared with can determine the location of the remaining parts. All the three parts are encrypted using the receiver's public key and signed by the sender's private key, so only the receiver can decrypt them using his private key and verify its integrity using the sender's public key.

When the receiver accepts the invitation, he gets added to the list of authorized users and the invitation gets removed from the list of pending invitations. This secret key and the file access containing metadata location and length location stored in Datastore will be encrypted using the receiver's public key and signed by the owner of the file, so only the receiver can decrypt to get the secret key and the location of the file metadata using his private key. With the knowledge of the secret key and the location of where to find the file metadata and the number of parts, the receiver now has full access to the file.

How will you manage file revocation?

When a user's access to a file gets revoked, the record associated with the encrypted key used to decrypt the file by the user gets removed from Datastore. Also, the revoked user gets removed from the sharingTree field of the file metadata.

In addition, anyone with whom the revoked users shared the file also loses access. Their access to the secret key to decrypt the file gets stripped and they will be removed from sharingTree. To be able to accomplish this feature, we keep track of who sends invitations and who accepts invitations in the sharingTree field.

> For example, let's say, A created file.txt and shared it with B and C. Then B shared it with D while C shared it with E. Then
>
> sharingTree = [A : [B, C], B: [D], C: [E], D: [], E: []]

Moreover, any pending invitations associated with all users whose access gets revoked are removed from the pendingInvitations field of the file metadata.

How will you ensure a revoked user can't take any malicious actions on a file?

To prevent revoked users from regaining access to the file by simply storing the secret key from previous requests, the owner of the file will regenerate a new secret key for the file and change the location of the file metadata and the length location by creating new UUIDs. Then the owner has to redistribute the new secret key, metadata location, and length location to the remaining authorized users. Both the owner and the user who the file gets shared with will agree on the location of the secret key and the file metadata as UUID(hash(owner + filename + user)), so the owner knows where to store the secret key and location of file metadata and the user knows where to get them. The owner also has to re-encrypt the file metadata and its content with the newly-generated key. Therefore, revoked users no longer know the locations where the file metadata and its content are stored and the key used to decrypt the file. As a result, they can't take any malicious actions on the file.

Helper Functions

StoreDataSymmetrically(object interface{}, key UUID, password string) (err error): encrypt-then-MAC the object and store it in DataStore at the provided key

GetDataSymmetrically(object interface{}, key UUID, string password) (err error): get the encrypted object corresponding to the key in Datastore

StoreDataAsymmetrically(object interface{}, key UUID, signKey DSSignKey, encKey PKEncKey) (err error): encrypt the object with public-key encryption, sign it with digital signature and store it to Datastore

GetDataAsymmetrically(object interface{}, key UUID, verifyKey DSVerifyKey, decKey PKDecKey) (err error): verify the integrity of the object from Datastore and decrypt it with decKey