Data Structures

Struct User:

Whenever a user will create an account, a user struct will be created. This user struct is independent for each user. The user struct will be encrypted by first applying PBKDF on the password, and then calling HBKDF twice on the PBKDF key to create an Encryption key and HMAC key. The result of both will be stored in the datastore, with the UUID being the Hash(Username).

Struct User-File:

This struct is used to link a given user with the file-entry struct. This struct will contain the UUID for a user's file-entry struct, as well as the decryption key for that respective file-entry struct. This struct is independent for each user.

Struct File-entry:

This is the most important struct. It is essentially a gateway entry to the base file struct. This will contain the UUID of the base file struct, and the encryption and HMAC key to decrypt the base file struct. It will be encrypted with by calling HBKDF twice on a random set of bytes to generate the Encryption key and the HMAC key.

Struct Invitation:

When the owner decides to invite someone else, an invitation struct is created that will contain the file-entry UUID for that specific user, plus the decryption key for that file-entry struct. This invitation struct will also be encrypted and HMAC'ed using a random key and placed in the datastore. As a proof, the owner would sign the struct with his/her digital signature.

User Authentication

1. How will you authenticate users?

**The UUID of each user struct is generated as a hash of the username.

For each initialization of a user, we will encode their password using PBKDF. We then re-encode this with HashKDF to create two separate keys: K1 and K2. We will store an encrypted version of the struct(SymEnc) with K1 and an HMAC version of the struct using K2 to ensure confidentiality and integrity of the user struct. These will be located at the same UUID.

When GetUser is called, the user will provide a password for the specific username. We will access the UUID of the provided username and run the same password PBKDF/HashKDF scheme using the provided password to get the two keys: K1 and K2. We first will access the E encrypted version of the struct stored at the UUID and run SymDec using K1 to retrieve the original struct. We then HMAC the recovered struct and compare it to the HMAC version of the struct stored at the UUID. We can use the function HMACEqaul for this. If the user is verified, return the pointer to the decrypted original struct.

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

Datastore: 1) the encrypted version of the struct 2) the HMAC version of the struct

KeyStore:

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

Every file and user struct we create will be stored in the Datastore, thus if a user wants to run their account on a separate device, they can just retrieve the file struct from the database.

File Storage and Retrieval:
File design:
We have a base file struct. This base file struct contains the number of blocks, and the first block UUID.
Store: If the file does not exist, create a new base file struct starting from the first block UUID until the last one. If the file exists, overwrite the contents.
Load: To load, go to the base file struct, find the first block UUID, and download the blocks for all the blocks (based on the number of blocks value).
Append: From our design, we can easily find the last block by starting from the first_block_UUID, and adding n to the UUID, where n = number of blocks, so first_block_UUID + n. We then append the file contents to first_block_UUID+n+1, basically to the UUID after the last block.

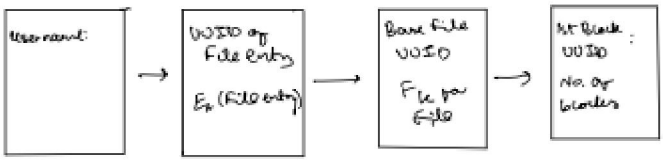
File Sharing and Revocation:
Sharing file: To share the file, a user who has access to the file must create an invitation. This basically entails creating an invitation struct and a new file-entry struct, if the invitee user is in the first layer (see diagram). The invitation struct contains the UUID for the file-entry struct that was just created. If the invitee user is not in the first layer, then the file-entry struct to be referenced would be the owner's file-entry struct (see diagram for details). After accepting the invite, a unique User-File struct will be created for that user.
Shared User Accessing Shared File: Once the above process is complete, then the user who has access would have a file-entry struct, containing the UUID of the file and the decryption key for the file. Using this piece of information, they can access the base file struct and the file.

Revoke User: Diagram explains this the best. If the structure of the file is e.g. D -> A, D -> C, and C -> G, where D created invitations for A and C (D is the owner of the file) and C created an invitation for G. Based on the first layer rule (as explained in the diagram), a new file-entry struct object would be created for A and C. However, G is in the second layer, and as a result, G's file entry struct would be the same as C, it's parent in the first layer. Throughout this whole process, D has a dictionary of usernames and their respective entry UUID for A, D, and G. If D were to revoke C, then there is a two-stage process:

- The existing file will have to be copied over to another UUID.
- D would have to scan through this dictionary and change the Entry UUIDs for all the users who are not revoked and can still access the file, in this case it is only A.

Ensure revoked user cannot take any malicious actions: As per the logic above, when the file UUID is changed, only users who have access to the new file UUID can access the file block.

User Struct User-File Struct File Entry Base File



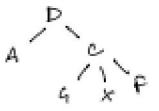
Symmetric

↳ All will be encrypted, HMAC and placed in container.

File Structure

Files will stored in a block format with each block having an arbitrary number of bytes. This makes it easy to find the last block and append. [1] → [2] → [3] ...

Revocation



create invite

⇒ When D → A, a user-file struct for A is created with its own unique entry UUID.

⇒ Similarly, D → C also creates a new user-file struct. When C → G, G will have access to the same File-Entry struct as C i.e. C → G

(---→ when createInvitation() is called.

⇒ Only the users that D (the owner) sent an invite to have a unique File-entry struct.

⇒ D's dictionary of users

↳

Users	Entry UUID
A	A's Entry UUID
C	C's Entry UUID
G	C's Entry UUID

D (the owner) duplicates the content of the file to another UUID and changes A's Entry UUID to reflect the new address of the file. C and G still have the old location, which may be deleted by the owner.

UUID	Encyrpted	Key Derivation	Value at UUID	Decription/Relationship
Hash(username)	Hashed	Symmetric Key: Deterministic - PBKDF, HashKDF - The User Struct itself is Encrypt + HMAC'ed using the keys generated by: (HashKDF(PBKDF(password+salt))	User Struct: 1. Username 2. Two dictionaries - Mapping of all filenames and UserStruct UUIDs + Dk **owned** by this user - Mapping of all filenames and FileEntry UUIDs + Dk **accessible** by this user 3. Password	- Represents the account of each user - For each user, there are two dictionaries containing the UserFile Structs of the files owned by the user along with its decryption key - Another dictionary for the UserEntry Structs which are files that the user does not own but have recieved and accepted an invitation for, along with its decryption key
Hash(filename + username)	Hashed	Public Key Enrcyption: Deterministic - - Create an ecrytption and decryption key using PKEKeyGen() --> Ek, Dk - We will use Ek to encrypt the UserFike Struct - The decryption key will be contained in the UserStruct so that the only person able to decrypt this struct is the owner of the file	UserFile Struct: - this is for a specific file and specific owner 1. Two dictionaries - Mapping of UUIDs of all Entry Structs(i.e. all users that recieved and accepted an inivitation to access this file) - Mapping of decryption keys for each of these Entry Structs 2. Updated UUID of the Base File	- Almost like an ownership card that is only accessible by the owner of the file. - When a user creates a file, it generates a UserFile Struct - Everytime a new user is invited to access this file, that user is added to the dictionary of accessors and recieves an Entry Struct used to access the actual file - Each of these Entry Structs will be encrypted so also their decryption keys included - It also contains the true UUID of the BaseFile Struct(with the file contents). When users are revoked, this UUID will change so the most updated UUID is stored here
Hash(username + recipientUsername+ filename)	Hashed	Diffie Helman Key Exchange: (Ex: Owner A -> User B) - A and B both creates a private key - They will use DHE scheme to trade those keys (mod p) and generate a shared secret key - A will encrypt the invitation with the secret key and B can decrypt it using the same key	Invitation Struct: 1. FileEntry UUID for the new users 2. Decryption key for that struct	- A struct created whenever the owner invites a new user. - This struct will contain the UUID for the new user's File Entry Struct which contains the UUID for the Base File where the content is stores
Hash(filename+username+ usernameParentNode)	Hashed	Public Key Enrcyption: Deterministic - - The Invitation Struct contains the UUID for the new user's FileEntry and the decryption key for that struct - When the user decrypts the Invitation Struct with DHE, they will have access to the decryption key for their FileEntry Struct	FileEntry Struct: 1. UUID for the Base File(may be the old UUID if the user is revoked) 2. Decryption key for the Base File	- Each user who was invited and accepted to access a file obtains a FileEntryStruct which is like a gateway to the BaseFile. - It contains the UUID of the BaseFile, but if this user is revoked, the owner will update the actual UUID of the BaseFile. - This means that this user will only have and be able to change the old BaseFile
Hash(Filename+ allUsersWithCurrentAccess + number of file contents)	Hashed	Public Key Enrcyption: Deterministic - - Create an ecrytption and decryption key using PKEKeyGen() --> Ek, Dk - We will use Ek to encrypt the BaseFile - The decryption key will be contained in the UserFile Struct as well as each of the FileEntry structs so that all users with access can decrypt	BaseFile Struct: 1. Head: UUID of the first set of contents stored 2. Tail: UUID of the most recently stored content	- This is the struct containing the actual content of the file. - When the file is first initialized with User.StoreFile, this struct is created and its UUID will be stored in the UserFile Struct of the owner. - New content will be added to the UUID with the hash of (Filename+ allUsersWithCurrentAccess +number of file contents + 1) - If a user is revoked, the UUID will be changed so that the revoked user only have access to the file located at the old UUID address.

Draft Test Proposal

Design Requirement [3.6.3]: Changes to the contents of a file MUST be accessible by all users who are authorized to access the file.

1) Define a function CreateFile() that creates a file and adds it to the datastore.
2) Define method Dictofusers() which is a dictionary containing all the users and their file UUID contained inside their respective file-entry structs
3) Send invitations to other users using the user.CreateInvitation method
4) Expect all the users to be in the dictionary mentioned in pt. 2
5) Change the contents of the file
6) As per the design of the program, all the users can access the file via the file UUID. Since the file inside that respective file UUID was changed, these changes should be accessible to all users when they go to that file UUID.

Design Requirement [3.6.8]: The client MUST enforce that the file owner is able to revoke access from users who they directly shared the file with.

1) Define a function CreateFile() that creates a file and adds it to the datastore.
2) Create two users, A, and B, and invite them to the file using the User.CreateInvitation method.
3) Revoke user A using the User.RevokeAccess() method.
4) Change the contents of the file and store it in a new location
5) Check A's file UUID in its file-entry struct and expect it to point to the old file without the new contents.

Design Requirement [3.5.2]: The client MUST ensure the integrity of filenames.

1) Define a function CreateFile() that creates a file and adds it to the datastore.
2) Define a function Filename() that outputs the filename for the file
3) Invite users to the file via the User.CreateInvitation method.
4) Change contents of file
5) Revoke one of the users via the User.RevokeAccess method
6) Change the file UUID for the file
7) Throughout all these steps, Filename() should still remain the same as before the users were invited.

Design Requirement [3.6.9]: When the owner revokes a user's access, the client MUST enforce that any other users with whom the revoked user previously shared the file also lose access

1) Create a user X
2) Call User.CreateInvitation method to create an invitation for X for a file, created by a defined method CreateFile()
3) Create another user, Y, and extend an invitation from X via the User.CreateInvitation method.
4) Call User.RevokeAccess method on X to revoke its access to the file
5) Change the file contents by appending through the UserAppendToFile method.
6) Check the file-entry struct for Y (via the owner's dictionary) and locate the file UUID that X has access to
7) Open the file that Y has access to and compare it with the actual file contents that have been changed
8) Expect that the contents would be different.

Design Requirement [3.7.1]: The client MUST allow users to efficiently append new content to previously stored files.

1) Define a function CreateFile() that creates a file and adds it to the datastore.
2) Create a file of size 1000TB of content
3) Call the User.AppendToFile() method and append 1B of content to the file
4) Expect the appending process to take a very small amount of time (max 10-20 seconds), anything more should fail the test. The reason is due to the file being stored in blocks, as the whole file would not need to be downloaded but rather only the last block.

Design Requirement [3.5.1]: The client MUST ensure confidentiality and integrity of file contents and file-sharing invitations

1) Define a function CreateFile() that creates a file and adds it to the datastore.
2) Encrypt the contents of the file and store it in the datastore using symmetric encryption
3) HMAC the contents of the file and store it in the datastore
4) Create several users and invite them all to the file
5) Check if the encrypted version of the file stored in the datastore corresponds to the same value as before, and repeat this process for the HMAC
6) As per the design of the program, any changes to the file will change the file UUID, and so the owner's dictionary will point to which user created the change as their respective file UUID will be different.