Design Document

0. Special Reminders:

Without specification, all asymmetric key pairs are generated by *PKEKeyGen() (PKEEncKey, PKEDecKey, err error)* and *DSKeyGen() (DSSignKey, DSVerifyKey, err error)* , and all symmetric keys are generated by Random Byte Generator. All encrypted contents and HMAC (or digital signature) are concatenated together. When checking the integrity, we just split back the byte-array, since HMAC (or digital signature) always has the same length.

1. Users & User Authentication

1.1 InitUser

Each user has a *struct user,* which is stored in Datastore. It contains four parts: *username*, *password*, and two private keys for sharing.

When we initialize a new user, we create a corresponding *struct user*. The *UUID* is deterministic and generated from the *username*. If this *username* is empty or it already exists, we report errors. The value consists of the encrypted *struct user*, concatenated with a digital signature signed by an RSA private key. The key to encrypt *struct user* is a *password*-based symmetric key. The corresponding RSA public key is stored as ("*username*", public key) in *Keystore*.

1.2 GetUser

Given the *username*, we can search in Keystore and Datastore to find public key and *struct user.* There are four possible search results. (1) If we can find the pair in Keystore but not found in Datastore, the *struct user* must be deleted by attackers. Then we report the attack. (2) If we can find the pairs in both stores, we use public key to verify whether the *struct user* is modified. If yes, report the attack. Else, we decrypt the *struct user* by password and check whether the input password is correct. (3) If we cannot find any pair in both stores, this user must not exist. (4) If we can find in Datastore but not Keystore, the attacker must create a fake *struct user*.

2. File Operation

2.1 Data Structure

Each file is stored as analogous to a linked list and composed of three separate parts: metadata, header and file content. For metadata, the UUID is deterministic by username and filename. The encryption and HMAC keys are also deterministic by username, filename and password. To distinguish the file type, we define two metadata structs. Type 1, for the original file, stores the UUID of the header and its two symmetric keys, the UUID of descendants and its two symmetric keys, and marks itself as type 1. Type 2, for shared file, stores the "communication channel", two symmetric keys, its ascendant and marks itself as type 2. The header of the file, *struct head*, contains the *UUID* and two symmetric keys of the newest *struct file* following it. The *struct file* stores the exact file content, the *UUID* and two symmetric keys of the previous part of this file. The contents in the file will be chained together when the user appends new contents to it.

2.2 StoreFile & LoadFile

When storing the file, we first generate the deterministic UUID and two keys of file *metadata struct*. If the UUID does not exist, the file does not exist, and we create a metadata, head, and *file struct*. If it exists, we check all structures' integrity one by one (metadata, header

and file content). If all structures are not attacked, we create a new *struct file* containing the newest content with NULL "Previous Part". *struct head* now will point to this *struct file*. Notably, for type 2 *metadata struct*, we will first go to the "communication channel" to find the header information and access the file content. Same procedures and relevant attack checking will be applied to *LoadFile*.

When the file is shared and the sender revokes someone, the information in the "communication channel" will become invalid. The revoked user and its descendants cannot access file anymore.

2.3 AppendToFile

When *AppendToFile* is called, a new *struct file* is created, the *content* is stored into the "File content" field. The information storing in *struct head* is the previous part of file. Thus, we put previous part information into new *struct file*'s "*Previous Part*" field and update the content in *struct head* to *UUID* and two keys of this new *struct file.*

3. Users' sharing & revocation of files

3.1 CreateInvitation

When the owner invites someone, it first constructs a "communication channel" in Datastore. The channel contains the UUID and two keys of the header of the sharing file. Then, the owner creates a new UUID-value pair to store the UUID and two keys of this "communication channel". This new UUID is *invitationPtr* and randomly generated. The value is encrypted by the receiver's *sharing_encrypt* public key and signed by caller's *sharing_verify private key.* All users' *sharing_encrypt* and *sharing_verify* public keys are stored in Keystore. Every time, one user shares file with the other user, the receiver's username and corresponding UUID of "communication channel" will be stored in descendants of sharing file. When the user is not the owner, he will not create a "communication channel". Instead, the *invitationPtr* contains the "communication channel" information between the owner and the user.

3.2 AcceptInvitation

Once the receiver (also the caller of this function) accepts the invitation, the receiver uses sender's *sharing_verify* public key to check whether the content has been modified by attacker. If the content hasn't been modified, the receiver decrypts the content using its *sharing_encry private key.* Then, the receiver creates a type 2 metadata struct and store the "communication channel" information (*UUID* and two keys) and sender's name.

3.3 RevokeAccess

When access is revoked from a particular user, the file owner needs to modify the "*Keys*" in his *metadata struct* and all keys in "communication channels" between him and valid receivers. The owner will also delete the revoked user and "communication channel" from *descendant struct*. For example, file owner A shared file with both B and C, C shared file with D, and then when A revokes the access of B, owner A needs to change the "*Keys*" field of the "communication channel" that is generated when A shared file with C, to the new "*Keys*". Now, B with the old "Keys", no longer has access to decrypt the file, the goal of revoking access is achieved. C and D still can access the file thanks to the communication channel.

Structure :

User struct

UUID:

username
password
sharing - encrypt private key
sharing - verify private key

metadata struct:

Type 1: [for owner]

UUID:

UUID of header
2 symmetric keys for header
type = 1
UUID of descendants
2 symmetric keys for descendants

Type 2: [for non-owner]

UUID:

UUID of communication channel
2 symmetric keys for channel
type = 2
parent's username

header struct

UUID:

UUID of newest file content
2 symmetric keys for file content

file content struct

UUID:

file content
UUID of previous content
2 symmetric keys for previous content

share struct

UUID

UUID
2 symmetric keys for this UUID

can be UUID of header or UUID of communication channel.

descendant struct

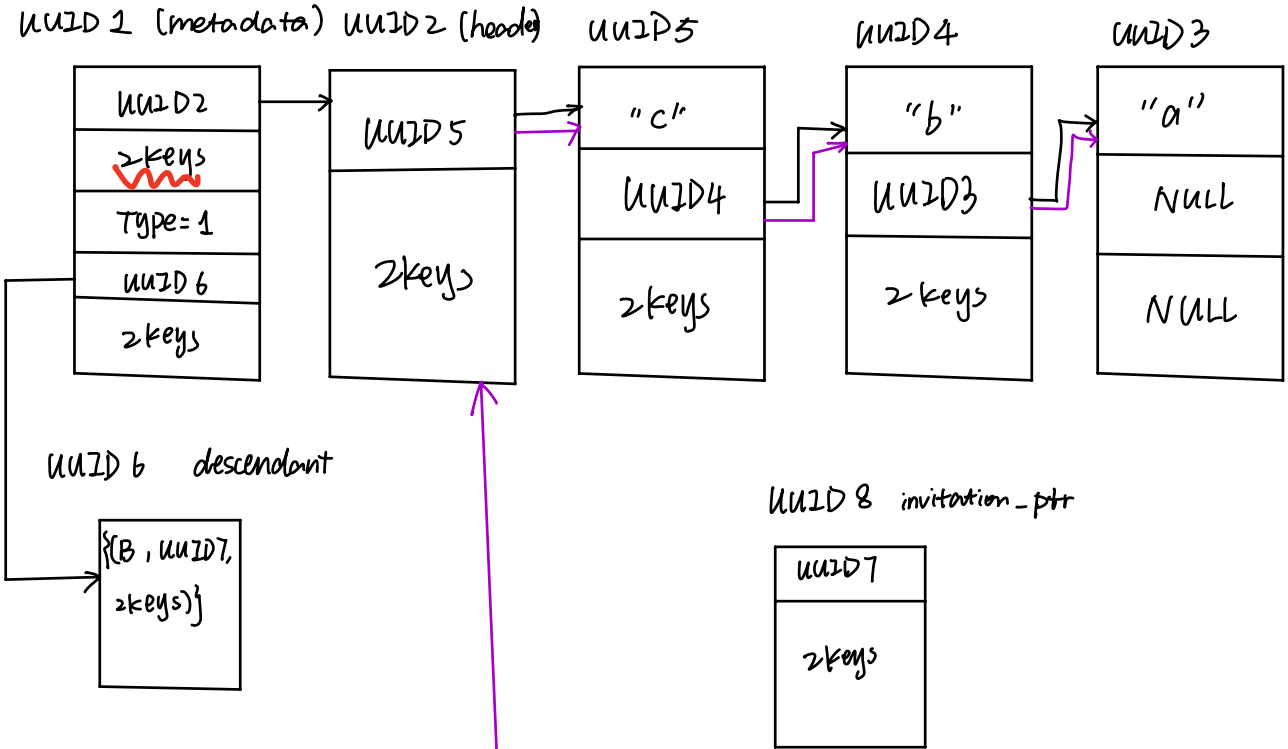
UUID:

List = {[child's name, "communication channel", 2 keys]}

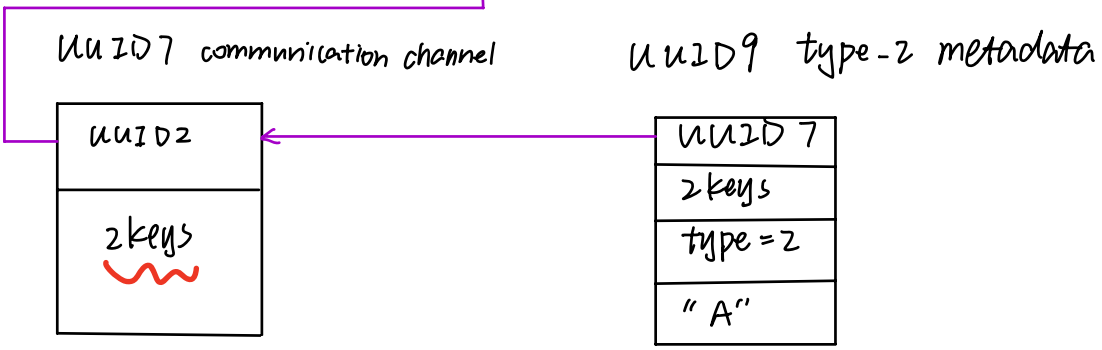
Append & Share . Revoke

Suppose a file "a.txt" has content "a" initially. After two appends, its content is "abc".

Case 1: Owner A.



Case 2: Non-owner B



When A shares to B & C and wants to revoke C, these 2keys will be updated.