Design Document

0. Special Reminders:

Without specification, all RSA key pairs are generated by *PKEKeyGen() (PKEEncKey, PKEDecKey, 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 fixed length.

1. Users & User Authentication

1.1 InitUser

Each user has a *struct user* which is stored in Datastore. It contains six parts: *password*, pointer to file *namespace*, 2 symmetric keys to en(de)crypt and verify the value of *namespace,* 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* (*username* will be hashed into 16 bytes). 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*. We also create a UUID-value pair for file *namespace* and generate two keys to en(de)crypt and verify the value of *namespace.* Two asymmetric key pairs for sharing will be generated and stored in *struct user* and 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. Otherwise, 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

As illustrated, we have a pointer to the *struct namespace* as a field in *struct user*. The *struct namespace* stores 5 dictionaries. The pairs are (*filename*, pointer of *struct head*), (*filename*, encryption key), (*filename*, HMAC key), (*filename*, sender) and (*filename*, receivers), where sender and receivers are dictionaries containing (name, *UUID*) pairs.

The file is stored as analogous to a linked list. 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 user appends new contents to it.

2.2 StoreFile & LoadFile

When storing the file, we first check the *namespace* of the user to verify whether the *namespace* is attacked. We then search the *filename* to check whether this file exists. If it exists,

we go into the *struct head* and *struct file*s to check whether they are attacked. If all contents are not modified, we create a new *struct file* containing the newest content with "*Previous Part*" set to *NULL*. *struct head* now will point to this *struct file*. If the file does not exist, we update *namespace* and create new *struct head* and *struct file*. Same procedures and relevant attack checking will be applied to *LoadFile*. When the file is shared and the sender revokes someone, the symmetric keys in *namespace* are no longer valid. Then, the user needs to go to the *"sender"* field to check, encrypt and update its namespace. If the user cannot find the *UUID* pair in Datastore, the attacker must have deleted it and we report the attack. The same procedure happens in *AppendToFile*.

2.3 AppendToFile

Since there is an efficiency requirement on file appending and file loading, we manage the content of the file into a "linked list"-like structure, as illustrated in section 2.1. When *AppendToFile* is called, a new *struct file* is created, the *content* is stored into the "*File content*" field. The information stored in *struct head* is the previous part of file. Thus, we put information of its previous part 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

The caller first goes into its file *namespace* and find the corresponding two symmetric keys and *UUID*. Then, the caller creates a new UUID-value pair to store the information he just retrieves. This UUID is *invitationPtr* and randomly generated. The value is encrypted by 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 *"receivers"* field of this *filename* in the *namespace* will be added a pair (receiver, *UUID*). For instance, when user A shared file *"a.txt"* with user B the new UUID-value pair containing file info will be created and shared among A and B. (B, *UUID*) will be added to the *"receivers"* field of file *"a.txt"*.

3.2 AcceptInvitation

Once 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* and stores the information (*UUID* and two keys) inside its file *namespace* and adds (sender, *UUID*) into its "*sender*" field.

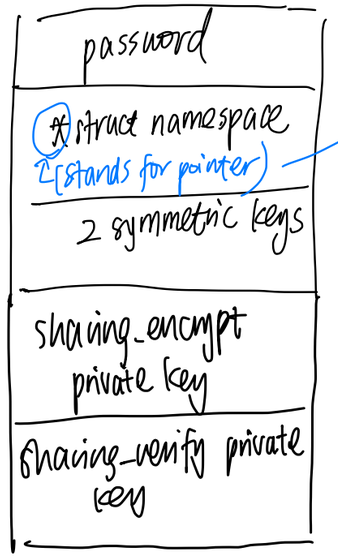
3.3 RevokeAccess

Recall in section 3.1 that every time the user shares a file with another, a new UUID-value pair containing the file access info is created and shared between both users. When access is revoked from a particular user, the file owner needs to modify the "*Keys*" field in UUID-value pair of all the other users that still have access to the file, and this particular user will also be removed from the *"receivers"* field in *struct namespace*. For example, file owner A shared file with both B and C, when A revokes the access of B, owner A needs to change the "*Keys*" field of the UUID-value pair that is generated when A shared file with C, to the new "*Keys*". Now, B still with the old "Keys", no longer has access to decrypt the file, the goal of revoking access is achieved.

Datastore

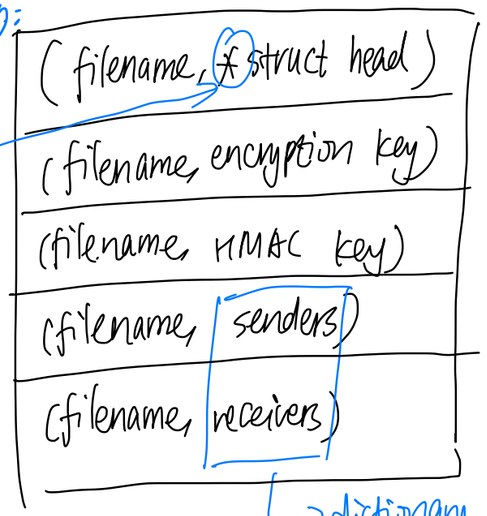
struct user:

UUID:



struct namespace

UUID:



→ dictionary containing (name, UUID) pairs

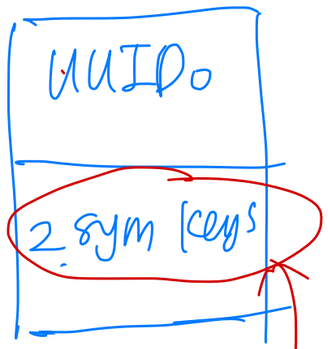
Share: when user B shares a file (owner)

the one in the example at the bottom left

"a.txt" with B, C

(the UUID of a.txt's struct head is UUID0)

then both A, B and C will have a copy of the UUID-value pair

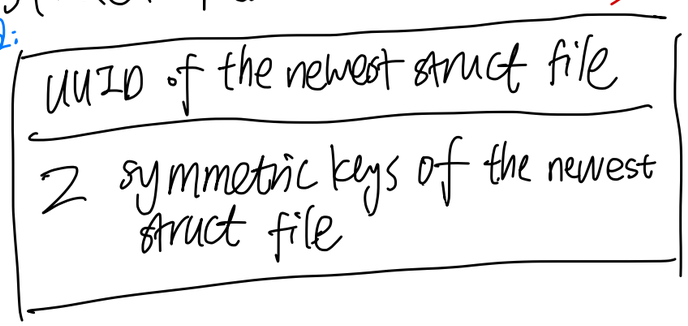


this field will be modified by B for C, changing into a new keys.

File storage structure:

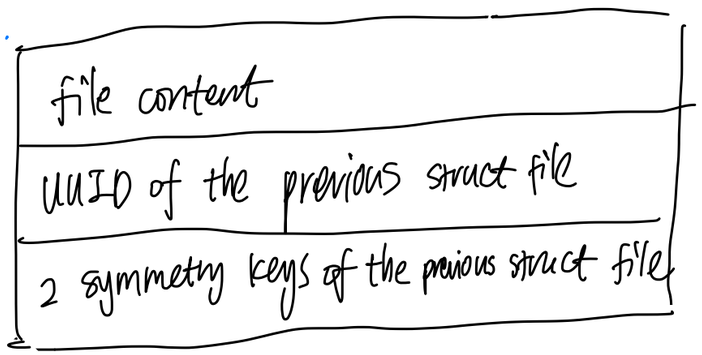
struct head (metadata of file)

UUID:

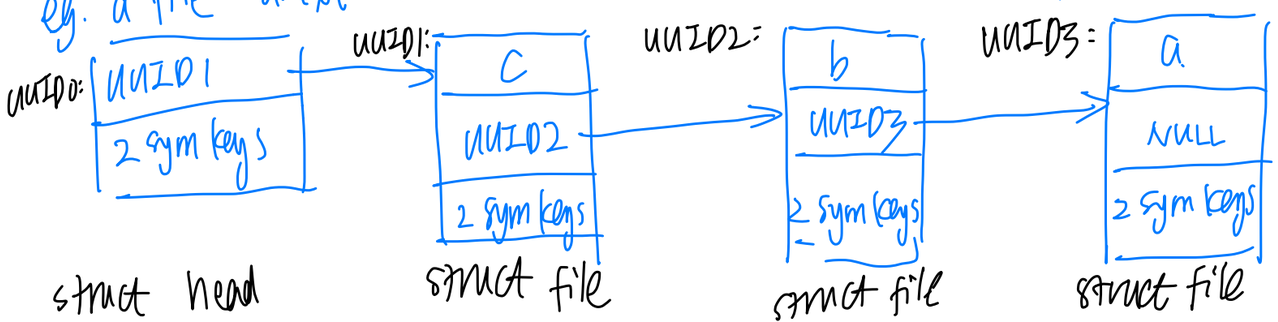


struct file (store exact file content)

UUID:



eg. a file "a.txt" with content "abc" in it (call "appendToFile" 2 times, each time appends 1 character)



Keystore

UUID:

("A_00", log in public key)
("A-01", sharing_encrypt public key)
("A-02", sharing_verify public key.)

↳ public keys for A.