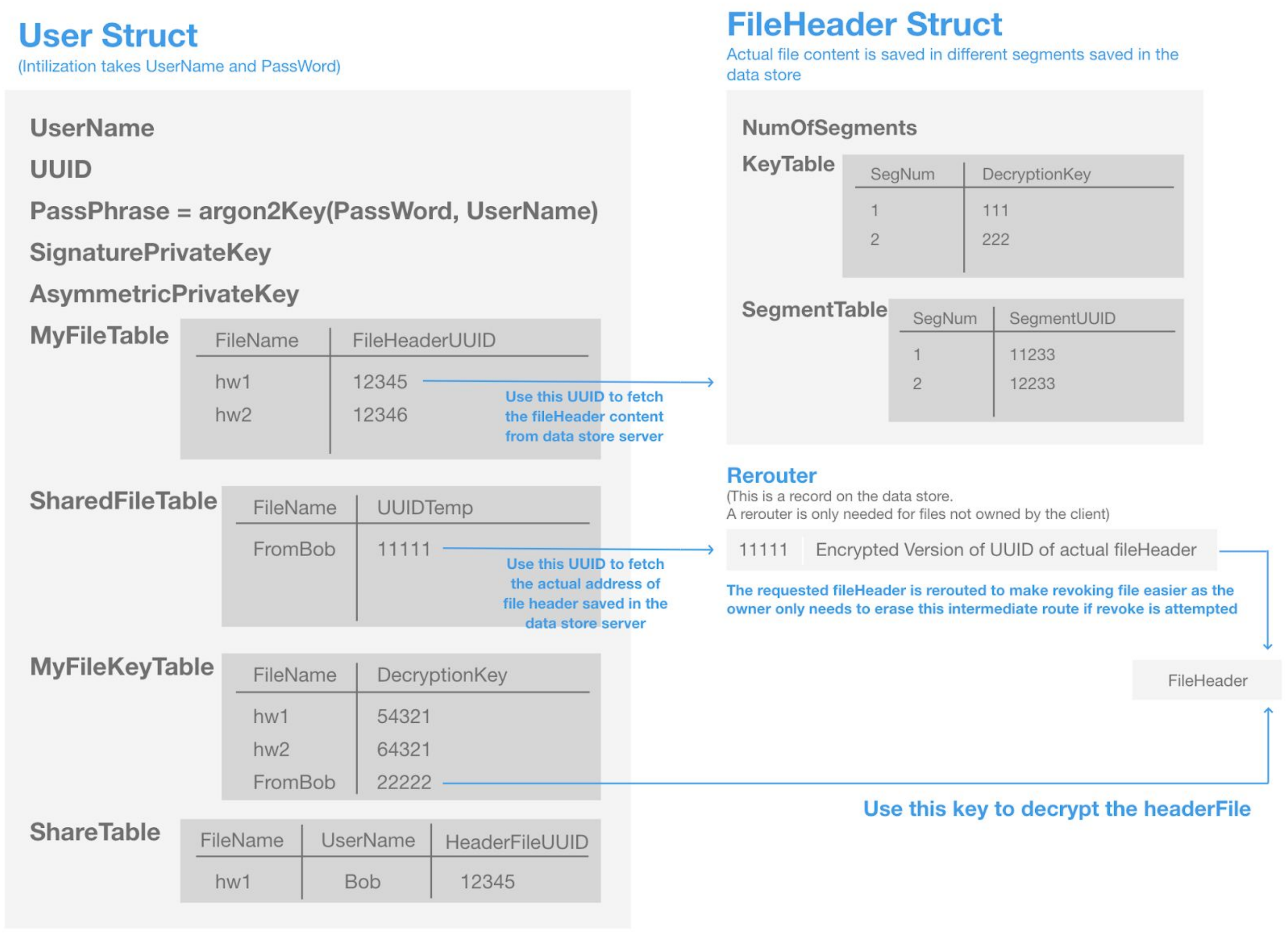
CS 161 Project 2 Secure File Sharing System

Team members: Benny Jiang, Yaxin Lei

Section 1: System Design


User Struct
(Intilization takes UserName and PassWord)
UserName
UUID
PassPhrase = argon2Key(PassWord, UserName)
SignaturePrivateKey
AsymmetricPrivateKey
MyFileTable
Use this UUID to fetch the fileHeader content from data store server
SharedFileTable
Use this UUID to fetch the actual address of file header saved in the data store server
MyFileKeyTable
ShareTable
FileHeader Struct
Actual file content is saved in different segments saved in the data store
NumOfSegments
KeyTable
SegmentTable
Rerouter
(This is a record on the data store.
A rerouter is only needed for files not owned by the client)
11111
Encrypted Version of UUID of actual fileHeader
The requested fileHeader is rerouted to make revoking file easier as the owner only needs to erase this intermediate route if revoke is attempted
FileHeader
Use this key to decrypt the headerFile

MyFileTable

FileName	FileHeaderUUID
hw1	12345
hw2	12346

SharedFileTable

FileName	UUIDTemp
FromBob	11111

MyFileKeyTable

FileName	DecryptionKey
hw1	54321
hw2	64321
FromBob	22222

ShareTable

FileName	UserName	HeaderFileUUID
hw1	Bob	12345

KeyTable

SegNum	DecryptionKey
1	111
2	222

SegmentTable

SegNum	SegmentUUID
1	11233
2	12233

(Note: All numbers in the graph are just example placeholders.)

• How is each client initialized?

When a client is initialized, we will construct a user struct and initialize all the maps and data structures defined below. As a client gets initialized, two pieces of initial client input information: UserName and PassWord.

User Struct:

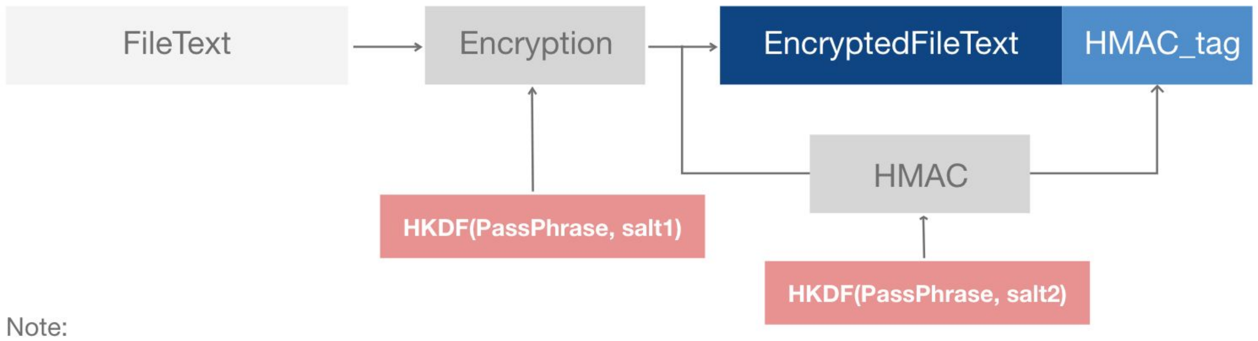
We will store the following information before encryption for a certain user:

1. **Username:** This is initialized by the user.
2. **UUID:** UUID=HMAC(PassPhrase, UserName). This is the UUID of the martialed user struct stored in the data store. Storing this makes updating the user struct very convenient.
3. **PassPhrase**: PassPhrase= argon2Key(PassWord, UserName, uint32).
4. **SignaturePrivateKey:** This is client's private key for generating signatures.
5. **AsymmetricPrivateKey:** This is the asymmetricPrivateKey which comes in pairs with a public key stored in KeyStore. Say private_k and public_k are a pair of asymmetric keys, then private_k will be stored here and public_k will be stored in the key store. If another client wants to share an encrypted file to this client, then he will use public_k to encrypt the UUID of a file together with the decryption key for headerFile for such file.
6. **MyFileTable:** This is a mapping where each FileName of this user is mapped to its FileHead UUID in the DataStore. The FileHead contains UUID and decryption keys of all the segments of this file. (All files in MyFileTable are owned by the client.)

7. **SharedFileTable:** This is a mapping from fileName to a UUIDTemp, which includes all files shared to this client by other users. After getting that the record of UUIDTemp, the content is the UUIDheader of the actual headerFile. We construct this detour to make revoke to certain users easy and possible. (explained in file sharing section)
8. **MyFileKeyTable:** The is a mapping from fileName to decryption key for the headerFile of a file. This includes both owned files and shared files.
9. **ShareTable:** This is a two layer map from FileName to UserName to UUID. This essentially stores which file has been shared to whom and what is the UUID of the files' headerFiles.

• How is a file stored on the server?

File Encryption Scheme



Note:
When storing user structs, PassPhrase = argon2Key(UserName, PassWord)
When storing other types of files, the passPhrase is different.
It is one of the input parameters for the SecureUpload and SecureFetch function.

1. **Secure Upload:** To store any file into the data store server (it could be user struct), we use the above scheme (the possible file types are mentioned in more detail below). Before saving any file, we encrypt it using HKDF(PassPhrase, Salt1), where PassPhrase is an input parameter for the function SecureUpload. Then we append the HMAC tag of the encryptedFileText after the EncryptedFileText to ensure integrity.
2. **Secure Fetch:** Secure fetch follows the reverse of the encryption schema by first checking the HMAC_tag to ensure integrity. If it seems that no one has touched our file through checking, then we decrypt it accordingly.

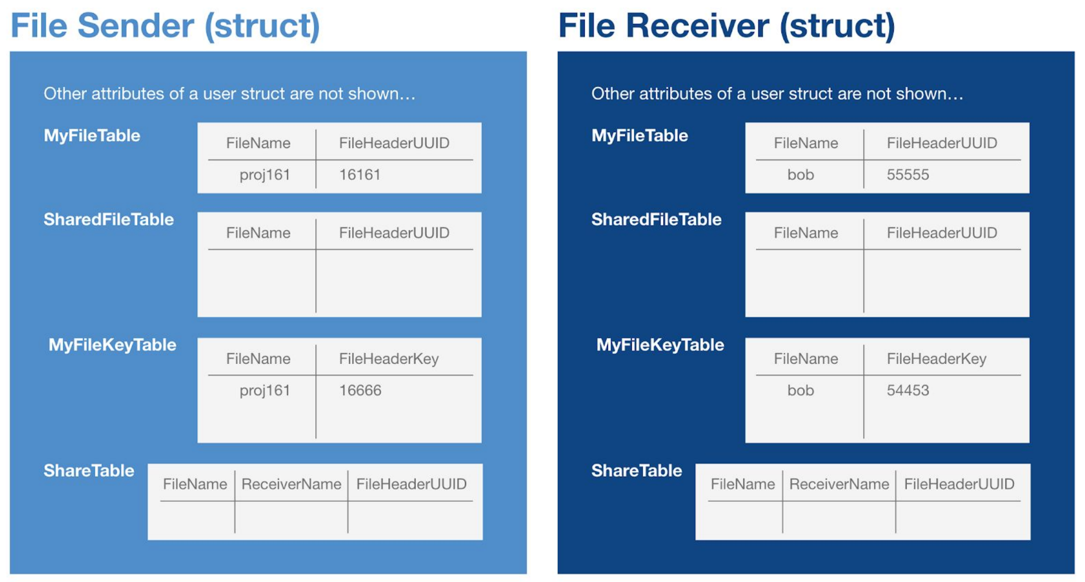
• How are file names on the server determined?

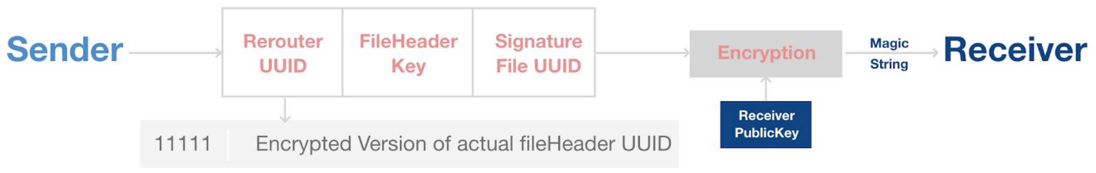
There are a total of five types of files on the data store server. The three most important ones are UserStruct, FileSegment, and FileHeader.

1. **User struct:** One file is the user struct information. In this case the file name is HMAC(PassPhrase, UserName) where PassPhrase = argon2Key(UserName, PassWord). By doing this, we make sure that only the user knows the UUID of his user struct. We also make sure that this would not leak any password/username information by using a secure hashing function HMAC.
2. **File Header:** When a user uploads a file, there will be a file header which stores all relevant information for all the segment of a certain file (explained in the first graph). The fileHeader UUID is randomly generated by calling UUID.new(). This is beneficial for security purposes as different fileHeaders are not sequentially stored. Attacker knowing the UUID of one fileHeader will not let him discover other fileHeader UUIDs.
3. **File Segment**.
4. **Signature.**
5. **Rerouter.** **For 3, 4, 5 listed above. Each file UUID is randomly generated by calling UUID.new() for the reason explained in fileHeader section.**

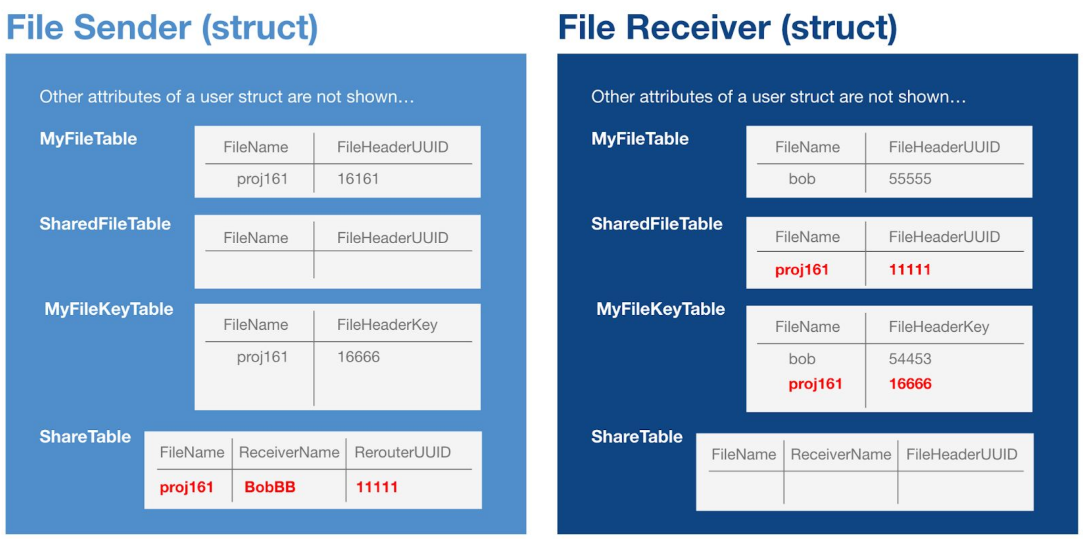
• What is the process of SHARING & REVOKING a file?

As shown in the below graph, which demonstrates the user struct of sender and receiver before the file is sent. All the other instances and information in the user struct is ignored in this image.


File Sender (struct)
Other attributes of a user struct are not shown…
MyFileTable
FileName | FileHeaderUUID
proj161 | 16161
SharedFileTable
FileName | FileHeaderUUID
MyFileKeyTable
FileName | FileHeaderKey
proj161 | 16666
ShareTable
FileName | ReceiverName | FileHeaderUUID
File Receiver (struct)
Other attributes of a user struct are not shown…
MyFileTable
FileName | FileHeaderUUID
bob | 55555
SharedFileTable
FileName | FileHeaderUUID
MyFileKeyTable
FileName | FileHeaderKey
bob | 54453
ShareTable
FileName | ReceiverName | FileHeaderUUID



1. **Sharing and Receiving:** When sender shares a **OWNED** file, the sender creates a rerouter record in the dataStore to store the actual headerFile UUID. Then the sender sends **MagicString = Encryption(Rerouter UUID || FileHeader Key || SignatureFile UUID)** to the receiver. The encryption uses Receiver's public key in the KeyStore, so only the receiver could decrypt it with the private key. The signature message ensures that the sender is indeed the desired sender since only he can sign the string. For the receiver, he receives the MagicString. He first decrypts the message using his own private key, then he checks the signature file. After that, he adds the **rerouter UUID (not the fileHeader UUID)** to his SharedTable. (The below shows how the struct looks like after sharing a sender owned file. Red indicates the changed fields.)



   a. However, to ensure that revoke works properly. **When a sender shares a SHARED file** (not owned). Then all the sender need to do is to share the Rerouter UUID. He will not need to create another Rerouter to the rerouter.

Revoking: By using the Rerouter scheme, since the real FileHeader UUID is not known by the file receiver user struct, all we need to do is to erase whatever record corresponding to the Rerouter UUID.

Testing:

Our testing scheme mainly includes four parts. 1. **Testing for individual functionality**: all functionalities (like sharing, initializing users, revoking, inserting) is tested. 2. **Testing for edge cases**: we test if the system works properly in edge cases like appending empty files, sharing non-existing files, sharing a shared file and then keep sharing it multiple times etc. 3. **Integration test:** we also tested combinations of shares and revoke, chains of shares etc. We also test for creating files with same names. 4. **Testing for attacks:** we test if the system can detect attacks if user struct or files or magic strings are tampered (changed, deleted, inserted) etc.

Section 2: Security Analysis

1. **Peeking data store:**
    a. User struct: Mallory should also not be able to guess where each user struct is stored at, because the user struct UUID is generated by first argon2key then HMAC hashing a combination of the username and user password. Sincer only the user knows the combination of these two, mallory would not easily find where the user struct is stored.
    b. It is possible that mallory scans the whole dataStore at one time stamp, and scans the dataStore after a client has uploaded a file. Then he would be able to know where the file is stored and looking at the difference between the two. If he peeks at such data, he would not be able to know the message in the file because it is fully encrypted. Moreover, each file segment of a certain file is encrypted using a different key, which makes the file even safer against malicious peeks.
    c. Protection against peeking at user password: Throughout the whole system, the user password is never stored. All user password generated content is encrypted and hashed by HMAC as described in the design scheme.
    d. Protection against peeking at magic strings: Since the magic string is encrypted using the public key of the receiver, only the receiver will be able to decrypt the magic string when a file is sent.
2. **Tampering data:**
    a. In our secure fetch and secure upload function, we first encrypted the file and appended HMAC to any file we upload into the dataStore (this includes user struct, files, fileHeaders and anything stored in the dataStore). This makes our system safe against any tampering and malicious change of dataStore files (this includes changes, insertion and deletion).
3. **Known plaintext attack:**
    a. This is a variation of MITM attack. Say A is sending a file “proj161” to B. Mallory might change the magic string and send the new string to B. Since we’ve added the A’s signature in the magic string (then encrypt the whole message), B will be able to notice if the magic string is changed since only A can sign using his private signature key.
4. **ScreenShot reverting attack:**
    a. Another type of malicious attack is that Mallory scans and stores the whole dataStore at a certain time stamp (which is essentially a map from UUID to files). Then after A revoke B’s access from file “cs161”, Mallory attempts to restore the whole dataStore by mapping his scanned version to the current existing UUID’s in the dataStore to destroy this revoke action. This will not work because we’ve deleted the rerouter in dataStore, therefore such rerouter UUID would not exist in the dataStore at all, and the remapping would fail.