Overall System Design

Datastore is a map that stores files using AES-CFB encryption. The files are stored in multiple parts. The first part contains the file metadata which points to the other parts of the file. All other parts contain file data. In a standard file there is only one data part. Appending to a file creates a new data part for each append. During file creation, all files receive a name is datastore that is the result of a hashing algorithm. This ensures that no files collide with each other, even if two users each store a file under the same username.

User is a struct contains the keys that a user need to access their files, sign their files and decrypt messages from other users. The HMAC key and RSA private key are stored directly in the user struct. Also stored in user is the DSKey which points to the file in datastore and EncryptKey which is used for encrypting the user struct. For the file encryption keys we initially used a single key for all of the files owned by a single user. This worked for part one of the project. However, it created problems when we implemented file sharing, because sharing a single file would give a would give another user access to every file the sharer owned. To solve this problem we created a KeyLink struct that contained the Encryption Key and HMAC Key for just one file. We then created a map structure called the FileKeyChain that stored all of a user's KeyLinks. This way a user could share a key for an individual file without giving access to everything.

KeyStore is map structure that contains the RSA public keys for every user. It maps username to PublicKey.

Individual Function Design

InitUser() creates a new user (type User) that contains the username, RSA private key, Datastore Key, EncryptKey, and the file keys stored in a map structure called FileKeyChain. It also uses Argon2 Key Derivation to create a hash of the user password and salted by the username. Argon2 is also used to generate the Encrypt Key and HMAC Key. The User Data structure is then marshaled to JSON, AES-CFB Encrypted using the Encrypt Key and a HMAC tag is generated using the HMAC Key and appended to the Ciphertext then stored in the Datastore using the generated Datastore Key.

GetUser() Validates the entered password against the stored password by creating an Argon2 hash of the entered password salted with the username to generate the Datastore Key, Encrypt Key and HMAC Key. If the username and password are correct the key from the entered password will match the stored key from the original password. The AES-CFB-HMAC User data is then pulled from the Datastore and the HMAC is checked to ensure Integrity. Then the User data is Decrypted, Unmarshaled and the User data structure is returned.

StoreFile() generates a File Encryption Key and File HMAC Key using Argon2 and UUID's and a 16 byte random salt. A KeyLink structure is instantiated to store the File Encryption Key, File HMAC Key and the location on datastore of the File's Metadata. The file metadata contains the file owner's username (RSA Encrypted) and the number of appends to the file. An HMAC tag is appended to ensure file Integrity. The FileMeta and FileData are then encrypted (AES-CFB) using the File Encryption Key. The FileMeta and FileData are stored on the Datastore using a generated UUID as a key. All file parts are tagged with an HMAC to ensure integrity.

LoadFile() takes the filename and pulls the KeyLink from User's FileKeyChain. It then pulls the FileMeta's location from the KeyLink and the File's Metadata is retrieved from the Datastore and verified by the HMAC. The metadata contains the naming convention and number of parts for the rest of the file. All data related parts of the file are then retrieved, decrypted and verified. The file is then and spliced back together and returned to the user.

AppendFile() stores the new part of the file in data store and updates the file's metadata. Much like LoadFile(), this function uses the KeyLink from the User's FileKeyChain to retrieve the file's metadata and verify it with the HMAC key. AppendFile updates the metadata to reflect the new number of file parts. It then stores the updated metadata and the new file part in datastore using the same encryption techniques that are used in StoreFile(). This method allows the metadata to remain a constant size. By only encrypting the new part of the file, the AppendFile function to can work efficiently.

ShareFile() takes the KeyLink from the users FileKeyChain and extracts the Encryption Key and HMAC Key for the file. The recipient's RSA public key is pulled from the Keystore and used to encrypt a message consisting of the concatenation of the Encrypt Key and HMAC Key. The RSA Ciphertext is then signed using the sender's RSA Private Key for Authenticity. We used a SharingRecord struct to store the RSA signature, Location of the File's Metadata in the Datastore and the RSA encrypted keys. The Sharing Record is stored in the Datastore using a generated UUID which is sent to the recipient via an outside channel.

ReceiveShare() uses the messageID sent by the sender to find the Sharing Record in the Datastore. The recipient then verifies the RSA Signature using the sender's RSA Public Key in the Keystore. Once verified, the recipient's RSA Private Key is used to decrypt the message containing the File's Encryption and HMAC Keys. A KeyLink is then instantiated for the recipient to store the File's Metadata location, Encryption and HMAC Keys. The KeyLink is then stored in the recipients FileKeyLink under a filename of their choosing.

RevokeFile() works a lot like LoadFile in that it pulls the file's metadata and decrypts it using the current keys. The file owner is verified by decrypting the ciphertext in the metadata using the RSA Private Key, if it decrypts, then the owner is verified. All the file parts are then pulled from the Datastore and decrypted. New File Encryption and HMAC Keys are then generated and the metadata and file parts are then re-encrypted using the new keys then stored in the Datastore.

Security and Testing of the System

System Security:

The system is secure for two reasons. First, all of the files are encrypted with individual file keys using AES-CFB. Initial values are not reused, therefore the only know way to break the encryption is with brute force. Second, for shared files, the keys to the file are only transmitted after they are RSA encrypted. This means that only the intended recipient can decrypt the message and retrieve the file key.

Three ways that Attacks are Thwarted:

Man in the middle: This sort of attack would involve a third party intercepting the shared message id in transit, then spoofing a new message and sending it to the intended recipient. However, our design would not allow this type of attack to be successful because the messages are encrypted using RSA. Only the intended recipient can decrypt and read a file sharing message.

Compromise the Data server: An attack on the datastore server that was able to gain admin privileges, would allow the attacker access to every encrypted file. This attack would fail because , without the AES key stored in the user struct, the files would be unreadable. No initial values used in the AES encryption are ever repeated, so the attacker can learn nothing from this other than the file lengths.

Decompiling or otherwise gaining access to the source code: By gaining access to the source code an attacker could hope to find a weakness in the security. This type of attack would not succeed because all of the encryption used relies on computational complexity to keep it secure. Even with a complete knowledge of how the system is designed, an attacker would not obtain the encryption keys. The attacker therefore could not decrypt the files with anything less than a brute force attack. This however, could not be completed in any reasonable amount of time.