Design

Part 1

The user struct holds 7 fields that facilitate a proper functioning of the file sharing system. The username, password, HMAC key, CFB encryption key, RSA key, the locations of each file, and the keys for each file.

When initializing the user, we generate all of our keys and update the user struct. Afterwards, we Marshal the userdata, CFB encrypt the marshaled userdata, HMAC the encrypted userdata, and then store the HMAC and CFB encrypted data into the DatastoreSet. This will ensure both confidentiality and integrity of the userdata. We ensure that this data is only accessible to those that have the correct username and password by generating the key to all this data by calling HMAC(userMacKey, username || password). The user's HMAC key and CFB encryption key are both generated by PBKDF2Key(password, username), which ensures that only the user can have access to these keys.

For GetUser, we call PBKDF2Key(password, username) to check if the keys generated by this call indeed gets us the keys to the userdata that we desire. If we do get the correct keys, we can verify HMAC and decrypt the CFB encrypted bytes to get the userdata. After going through the HMAC verification and decryption process, we can unmarshal the decrypted data to produce the User struct.

When calling StoreFile, we generate a secure location and keys for each individual file. Keys are generated by calling PBKDF2Key(filename, password) and file location is generated by calling HMAC(userMacKey, filename). After creating our keys and location, we HMAC and encrypt the file data using our newly generated fileMacKey and fileCFBKey. We choose to store this encrypted file data into a random location to make things easier when we are appending to our file. In order to keep track of this encrypted file data, we create a metadata struct that keeps track of the locations of all components of the file. We store this metadata struct into a random location as well. Finally, we update our FileLocations and FileKeys dictionaries located within our user struct such that filename is mapped to the metadata location and the keys of the file. After this update, reencrypt and HMAC our userdata before storing it into our original location in Datastore.

In AppendFile, we retrieve the location of the metadata and the keys to our file by accessing the dictionaries located within our user struct. We authenticate and decrypt the metadata before calling unmarshal. After doing so, we encrypt the data passed into the function and store that encrypted data into a random location inside Datastore. We store this random location inside of our metadata struct so that the metadata points to the encrypted data. Since we have updated our metadata struct, we marshal the updated metadata, HMAC and reencrypt it before restoring it into the original location in Datastore.

In LoadFile, we similarly retrieve the location of the metadata and keys to our file. We authenticate, decrypt, and unmarshal our metadata. Once we have our metadata, we go through each location stored inside of our metadata to authenticate and decrypt the data stored at each location. We append all of the decrypted data together until we get the entire file.

Part 2

In ShareFile, we retrieve the location and keys of the file by accessing the dictionaries located within our user struct. We create a sharingRecord struct storing the metadata location and the keys for the file that we want to share. We marshal this struct to obtain the plaintext that we will implement RSA on. Then we proceed to RSA sign the plaintext with our private key, RSAEncrypt the plaintext with the recipient's public key, concatenate the two together, and cast those bytes into a string in order to form msgid. We send this msgid to the recipient.

In ReceiveFile, we convert msgid into bytes to obtain the RSA signature and RSA encrypted data concatenated together. We call RSAVerify and RSADecrypt to make sure that the sender has indeed sent the msgid. When we are sure that the decrypted data is the marshaled version of the sharingRecord, we unmarshal this decrypted data. Then we extract the file location and file keys located within the sharingRecord to update the dictionaries located within our user struct. After these updates, we HMAC and reencrypt before restoring the userdata.

For RevokeFile, we relocate the file metadata and generate different keys for HMAC and encryption. After we decrypt the metadata with the information in our dictionaries, we reencrypt and HMAC the metadata with randomly generated keys. We then store this re-encrypted metadata into a new randomly generated location. Then we move on to call datastore delete on the old metadata location. Finally, we update our user struct dictionaries with our newly created locations and keys. Then we re-encrypt and HMAC our updated user struct. We store this updated user struct into Datastore.

Security Analysis

First Attack: Extracting User Data
This attack is relevant to the confidentiality of our information. Suppose we have an attacker trying to extract important user data from datastore. It would be nearly impossible for an attacker to obtain any meaningful information inside of the datastore since the user data is always safely CFB-encrypted before being stored inside of datastore. Brute-forcing to find the encryption key is not a feasible solution for the attacker.

Second Attack: Manipulating information inside Datastore.
This attack is relevant to the integrity of our information. Suppose we have an attacker trying to manipulate the data located inside of Datastore. It would be nearly impossible for an attacker to manipulate the data inside of Datastore while also bypassing the HMAC check that is implemented every time something is stored. Without the key used for HMAC, an attacker will be unable to tamper with the data without being detected.

Third Attack: Sharing malicious data to someone while trying to disguise oneself.
This attack is relevant to the authentication of information being sent through our file-sharing system. Suppose we have an attacker trying to send a file with malicious information to another user while trying to disguise himself as someone else. Without the private key of the user that he wants to disguise himself as, the attacker will be unable to bypass the RSA verification process that confirms the identity of the sender.