Single-User File Storage

1. How is each client initialized?

   We store a big map that leads to every `User` instance in the `Datastore`. The map is accessed under the key of a hardcoded literal: "username", casted to a `[]byte`, so everyone knows how to access the map. The map entry holds a UUID unique to the username used as key. It's randomly generated.

   To make this login UUID secure, we encrypt it with a key generated by the *Argon2Key* function with the actual password as argument. We also use an HMAC for the UUID. Using the password based encryption key, we encrypt a random encryption key, a random HMAC key and the random UUID. The encryption key will be used to decrypt the User structure, the HMAC key will be used to check the integrity of the data stored in the map slot and the UUID will point to the user structure.

2. How is a file stored on the server?

   To store the content of F, we create a `struct`, `fileDataStruct`. A `fileDataStruct` contains three fields: `Indices`, `HMACKeys[][]`, and `ContentUUID`. In `AppendFile`, we tack on additional blocks of ciphertext with additional HMACs. So we store which encrypted block correspond to each HMAC. We also keep track of a random 16 byte UUID, `ContentUUID` that leads to the encrypted content of the file in the `Datastore`. F's `fileDataStruct` is encrypted and HMAC'ed on the server.

   For key management, we create a `struct`, `filePointerStruct`. A `filePointerStruct` contains the random UUID the `fileDataStruct` is stored under, `FileUUID`, the ranodm 16 byte symmetric encryption key used to decrypt the file content stored under `fileDataStruct.ContentUUID`, `Ekey`, and the random 16 byte key used for validating HMAC's on the file, `HMACKey`.

   F's owner uses a `filePointerStructMap`, a map `filename` $\rightarrow$ UUID of a file's `filePointerStruct`, `fileUUID`. To load the file, we retrieve the UUID of F's `fileDataStruct` and the relevant encryption and HMAC keys from the `filePointerStruct`. The `fileDataStruct.ContentUUID` then leads to the encrypted file content. We decrypt the file with `filePointerStruct.Ekey`, and validate each block of ciphertext, separated by `fileDataStruct.Indices`, using `fileDataStruct.HMACKeys`.

3. How are file names on the server determined?

   Filenames are relative to a specific user. When a user has a file shared with them, they can receive it under whatever filename they want, for example. They set up local state, like the `User.ReceivedFiles` field, which uses the filename that they pick for the file. So, the server cannot know the name of the file unless they compromise a `User`, which are encrypted, signed and HMAC'ed.

4. What is the process of sharing a file with a user?

   Case 1: user A wants to share their own F with B, using magic string M. B has public key, denoted as K_b and A has private key, say K_A^{-1}. First, define $E = E_{K_b}(A_{username}||K_{FileEnc}||K_{HMAC}||uuid_{MS})$. And, call $D = DS_{K_A^{-1}}(E)$, and $H = HMAC_{K_{HMAC}}(E||D)$. We will send $M = (E||D||H)$.

   The magic string maps to the UUID of an encrypted `fileDataStruct`, which will contain key information for validating and accessing file information. In `ShareFile` we first define $Data = Enc_{K_{FileEnc}}(uuid_f||K_{FileEnc}||K_{HMAC})$ Then, we define $D' = DS_{K_A^{-1}}(Data)$, and $H' = HMAC_{K_{HMAC}}(E'||Data)$, as in creating M. We will store $(Data||D'||H')$ in the `DataStore` under M. Call this V. A will update a field in the User struct, `ShareMatrix` which has entries `userdata.SharedFiles[F.filename][B.Username] = U` to denote A shared F with B under `U`.

   Case 2: user B wants to share F with user C, after A shared F with B. Set $E^* = E_{K_c}(A_{username}||K_{FileEnc}||K_{HMAC}||uuid_{MS})$, where E_{K_c} is C's public key, and $D^* = DS_{K_A^{-1}}(E^*)$, $H^* = HMAC_{K_{HMAC}}(E^*||D^*)$. Then, we send $M^* = (E^*||D^*||H^*)$.

First, notice that our HMACs do not leak information about the owner of the file since we compute it over the encryption and the DS, which the attacker learns nothing from. Also notice that an attacker cannot match the DS to a user, and so cannot determine the owner of a file.

In `ReceiveFile`, we validate the DS on the encrypted message with the sender's public key, decrypt the public-key encrypted data with the receiver's private key, then verify the HMAC with the key in the plaintext. We retrieve $(sender_{username}, K_{FileEnc}, K_{HMAC})$ and follow $uuid_{MS}$ to the `Datastore`. We repeat the same DS validation for V, decrypt using $K_{FileEnc}$, then validate the HMAC with the K_{HMAC}. V contains $uuid_f$, which points to an encrypted `fileDataStruct`, which we have the appropriate keys to use. This leads us to the `ContentUUID` of a file.

In the `User struct`, we keep an array of `ReceivedFiles`. This is a `map[string]receivedFilePointerStruct`, which maps filenames to *senderusername*, $uuid_f$, $K_{FileEnc}$, K_{HMAC}. We update this appropriately in `ReceiveFile`.

5. What is the process of revoking a user's access to a file?

   Say A wants to revoke B's access to F. In the User struct, we delete the entry for `userdata.SharedFiles[F.filename][B.Username]=` $uuid_f$. Then, we delete the entry for $uuid_f$ from the `Datastore`. Everyone who had F shared with them cannot retrieve the file, and no one has to log in to for `RevokeFile` to work.

6. How were each of these components tested?

   3 contexts were used for testing: Correct user input, Incorrect user input, Malicious usage. In this, we test for regular functionality, and functionality under benign stress from the user: misspelled username/password, misspelled filename, etc. We also tested security by maliciously tampering with the server: replacing every data entry with `nil` and flipping bytes of every entry in the `Datastore`.

Security Analysis

1. **Impersonation/Extension attacks on User roster:** A user who tampers with the map we store users at and attempts to add herself to the map without calling `InitUser` (to override another user, for example), will fail. For the attacker to do anything useful, say, give themselves access over files they don't own, they would have had to decrypt the other users in the `Datastore`, read it, and fake an HMAC on the data. This requires brute forcing encryption keys, HMAC keys, and the login credentials of many users.

2. **File Tampering:** First, adversary S would have to forge the user's password. Then, S would have to forge the appropriate `fileDataStruct` and `filePointerStruct` in the `Datastore`, which requires a forgery of another HMAC key. But to even get to this point, S would also need to forge an RSA key, an encryption key and user's login info. Since we generate `errors.New()` upon the failure of the validation of these HMACs and keys in every file-related method in the API, this is infeasible.

3. **Stealing a file:** If A shares F with B with magic string M, our design will detect immediately if C uses M and $A.Username$. That is, we protect against replay attacks on the magic string. C would need to forge the login of B to do fake a `ReceiveFile` call.

4. **God mode (Meta user):** Consider a `User` who upon revocation, could "save" their file retrieval credentials, and access a file after their access tokens were revoked from the `Datastore`. In `RevokeFile`, we generate encryption key for a file. We update every `filePointerStruct.Ekey` of other recipients of the file, as well as the owner's. The "meta" user will decrypt a garbage `fileDataStruct` and receive garbage at the old `ContentUUID`. But we also generate a new random `filePointerStruct.FileUUID` and new `fileDataStruct.ContentUUID` for the file content - about 50 bits of entropy to be brute forced.