Project 2 Design

Daniel Van Der Maden, Albert Tang

1. System Design:

Our system will heavily rely on a user's `User` struct being securely stored on the Datastore (while detecting for corruptions upon use) as it will contain various user states that are critical for core functionality. We will mention relevant core `User` struct elements, but note that adding extra elements to said struct will not require a change to our system. When we say 'just symmetrically encrypt', we implicitly mean breaking up the data into blocks (and padding as necessary using the 0x100...00 padding scheme) and encrypting each block.

1.1 Client Initialization: (Changes made here: User_UUID derivation and wrapping scheme.)

First define a User Password Hash (UPH) to be `Argon2(pw=<password>, salt=<username>, len=32)`. We will store this in the `User` struct for future use. Next, to obscure the Username for the Datastore's key list, each user will have a `User_UUID = UUID(HMAC(key=UPH, msg=<username>))`. The user can derive this at a later time since they have the password and username for the UPH. Also note that the UPH is unknown to an attacker without the password.

Next, we will create 2 asymmetric key pairs for each user. One key pair will be used for encryption and the other will be used for signing. The private keys will be stored in a user's `User` struct and the public keys will be stored in the keystore under the keys `"enc_<username>"` and `"vfy_<username>"` respectively.

To save a user's `User` struct to the Datastore we will use symmetric encryption and an HMAC. The encryption key is derived from the Argon2 KDF where the password is the byte string for `"enc_<UPH>"` and the salt is the password. Similarly, the HMAC key is derived from the Argon2 KDF with the same salt but the password has "mac" instead of "enc". This creates 2 different keys for the HMAC and encryption. The IV will just be a random byte string. The `User` struct will be encrypted and wrapped in a struct. Said struct will contain a list of encrypted block cyphers such that the first element of the list, C_0, is the IV. The struct will also contain an HMAC of each block concatenated one after the other. This provides integrity for the IV and the encrypted `User` struct as well as confidentiality of the actual `User` struct. Lastly, we will store the wrapped `User` struct on the Datastore with the key being the `User_UUID` defined above.

1.2 Client Get: (Changes made here: wrapping scheme cont.)

We have the user's username and password and can therefore rederive the UPH and `User_UUID.` Using the `User_UUID`, we can fetch the wrapped `User` struct from the Datastore. We can also reconstruct the symmetric keys used to encrypt and mac the `User` struct since said keys were based on the `User_UUID` and UPH. Using said keys, we can unwrap the `User` struct by first verifying that each block cypher concatenated one after the other HMACs to the corresponding HMAC in the wrapper struct (returning an error if it doesn't). Then, we can decrypt the `User` struct using the rederived key. If the `User_UUID` is *not* in the Datastore, we return an error.

1.3 Storing a File: (Changes made here: wrapping scheme cont., metadata layout.)

Note that we have access to all of the elements in the `User` struct since a user calls `StoreFile(..)`. We will first randomly create a `Doc_UUID`. This hides the file name. Next, we will add an entry to a `FileUUIDs` map of filenames to `Doc_UUID` (which is stored in the `User` struct). Next, we will generate a random key, call it `file_key`, which will act as our symmetric key for everyone who has access to the file. Then, we will add the `file_key` to a `FileKeys` map of `Doc_UUID` to `file_keys` (which is also stored in the `User` struct). Next, we will generate a random key to be used as an HMAC key and similarly store it in its own map in the `User` struct.

First, we will encrypt our file with a randomly generated IV and the `file_key` to create cyphers: C_0 .. C_n where C_0 = IV. Then, for each C_i in C_o ... C_n, we will encrypt (with `file_key`) and wrap C_i using the procedure described in 1.1 (except now the encrypted cypher list is just C_i and we are using the file's HMAC key). Next, we will create a list, *L*, of *n+1* randomly generated UUID and for each wrapped C_i, we will store it in the Datastore with the key *L*[i].

To finish it all up, we will store *L* in a metadata struct then encrypt and wrap the struct (like we did in 1.1) and store it on the Datastore with the key being the `Doc_UUID.` Lastly, to save our updated `User` struct, we will re-encrypt and re-wrap the `User` struct (using the exact same procedure in 1.1). Effectively, our scheme double encrypts (and wraps) the file data (each one with a different IV); first for the file data then one for each of the resulting cyphers.

1.4 Loading a File: (Changes made here: wrapping scheme cont., metadata layout cont.)

Note that we have access to all of the elements in the `User` struct since a user calls `LoadFile(..).` Given the filename that we want to load, we will use the user's `FileUUIDs` map to get the underlying document's `Doc_UUID`. Next, we will fetch the respective `file_key` and HMAC key from the user's file key maps. Using all of this, we will fetch, unwrap and decrypt the metadata struct containing *L* (same *L* as described in 1.3) via the process described 1.2. Using *L*, we can similarly fetch, unwrap and decrypt each cypher block to create a list of encrypted block cyphers. Lastly, using said cypher list, we can decrypt the file data using `file_key` and return it. Note that if any unwrapping fail or if the file does not exist for the user, we will return an error.

1.5 Appending to a File:

We will do the same procedure as described in 1.4 to get the unrapped list *L* (same *L* as described in 1.3). Note that doing so, we have the `file_key.` Using *L,* we can fetch the *last* cypher (and delete it from the Datastore), unwrap it and decrypt it to get the last block of the file. Then, we can append the file content to the last block and break up the resulting byte slice into blocks for encryption. We will encrypt and save each block as described in 1.3 (since we have `file_key`) and append the new list of UUIDs to *L* (after removing the last UUID from *L*). Lastly, we update and save the metadata.

1.6 Sharing & Revoking: (Changes made here: message layout due to RSA size limit.)

To share a file, one must simply send the underlying file's `Doc_UUID` and respective file keys (i.e: the `file_key` *and* `HMAC_key`) so that another user can access it. To ensure integrity, authenticity, and confidentiality, the sender will separately encrypt the `Doc_UUID, file_key,` and `fileHmacKey` using the receiver's public key due to RSA's limit on a plaintext's length, then place the three fields into a struct. The sender will sign the struct using their private signature key. On the receiving end, the receiver can check the digital signature of the encrypted message and decrypt it only if it is untampered with and came from the sender. The receiver will *always* add/override the filename entry in their `FileUUIDs` map with the received `Doc_UUID.` They will also *always* add/override the `Doc_UUID` entry in their `FileKeys` map with the received `file_key`.

To revoke a file's access, the file owner (which can be kept track of in the user's `User` struct) can load the file from the Datastore, delete all related file entries in the Datastore, then store the file again. Storing the file again will generate new keys and necessary randomness for the file, so any `magic_string` previously used for sharing will no longer be valid.

1.7 Testing Methodologies:

Each part will have separate unit tests to ensure core functionality and resistance to attacks. Examples of such tests are as follows: If a file has been manipulated, there should be an error returned. If an invalid user (a user who hasn't been initialized or a user who hasn't been invited to a file) tries to access a file, there should be an error returned. If a file has been shared, all the users should be able to access it in accordance to the specifications. Appending, uploading, and downloading should all work in the specified manner for all users that have access to the file, and by no one else. Further, there will be tests that simulate the following three attacks, and the system should behave according to the specifications, i.e. there should be semantic security. Lastly, we will only write tests for the given functions in the skeleton code to ensure that our tests would work on any implementation. However, we will also write private tests for our helper functions.

2. Security Analysis: (Changes made here: updated to reflect scheme changes.)

Our design prevents a malicious Datastore from getting/knowing or silently corrupting a user's `User` struct. This is because our design encrypts the `User` struct using a symmetric key that is ultimately derived from a user's username and password. Since the password is assumed to have high entropy, the malicious Datastore cannot guess the symmetric key with any notably probability. Moreover, our design 'wraps' the encrypted struct, which includes computing an HMAC (using a key derived from the user's username and password) of the encrypted struct's cyphers. Since our design checks the HMAC of the encrypted `User` struct cyphers before decrypting and using it, we can detect any corruption to our user data. Lastly, the key for the `User` struct is a UUID based on the username and password. This prevents a malicious Datastore from learning a user's username.

Our design also prevents a malicious Datastore from getting/knowing or silently corrupting a file's data. This is because our design encrypts a file using a randomly generated symmetric key (thus making it hard to guess). Moreover, each file's encrypted cypher block is encrypted (using a different IV), wrapped, and saved under a randomly generated UUID. This randomizes the order of the cypher block from the Datastore's perspective, thus mitigating targeted file corruptions. The file's metadata struct (containing the list of UUIDs for each file's cypher block) is also encrypted, wrapped before it is saved. Essentially, all file related entries to the Datastore are encrypted and wrapped like they were for the `User` struct, thus our design can detect any corruptions before using a file and prevents a malicious entity from partially reading a file. Lastly, the key for a file's metadata struct is random. This prevents a malicious Datastore from learning any file names.

Our design also prevents an attacker from randomly appending data to a file. More concretely, an attacker cannot just add random cyphers to the datastore and edit a file's metadata to include said cyphers during the file's decryption. This is because the file's metadata is also encrypted (and wrapped) to ensure that malicious people cannot see or edit it.

Our design also prevents a malicious out of band 'share message carrier' from reading or silently corrupting a share message and from injecting their own share message. This is because our design 'encrypts' messages (via RSA) using the receiver's public key so that only the receiver can decrypt the message. Moreover, the encrypted message is signed (using a different key pair) by the sender's private key. This allows the receiver to verify that the message came from the sender using the sender's public key. Since the out of band messenger doesn't know either of the private keys, they cannot do any of the mentioned attacks.