Secure File Storage: Project Design Document

Preliminaries:

Throughout the project, whenever we needed to use symmetric encryption, the following authenticated encryption scheme using two keys was used to ensure both confidentiality and integrity (utilizing Encrypt-then-HMAC). Moreover, since we knew that the K-V pairs stored in the `Datastore` were insecure, for every value V that we encrypted, we made sure to supply the corresponding key K as input to the encryption scheme, which would then be used in the HMAC computation. This was done to ensure that even if an attacker were to swap values between keys, an integrity error would be detected.

$$AE_{k_1,k_2}(D,K) = E_{k_1}(D) \parallel HMAC_{k_2}(K, E_{k_1}(D))$$

During decryption, the HMAC is first validated and only then an attempt is made to decrypt.

These are the three structs we defined and used in our implementation:

```
type User struct {
        Username string
        Privat userlib.PrivateKey
        EncryptionKey []byte
        MacKey []byte
        FilenameKey []byte
}
```

```
type DataNode struct {
        ID uuid.UUID
        EncryptionKey []byte
        MacKey []byte
}
```

```
type FileInfo struct {
        FileIDs []uuid.UUID
        EncryptionKey []byte
        MacKey []byte
}
```

Functionality:

InitUser: `Argon2(password,salt=username)` is used to deterministically generate random bits of size equivalent to `3*AESKeySize`. These bits would be split to form 3 keys: k_a, k_b, k_c. and used for $A = HMAC_{k_a}(username)$ and $B = AE_{k_b,k_c}(User\ struct, A)$.

Further, 3 more keys are randomly generated using `RandomBytes`, each of them specific to that user and stored in the `User` struct. These keys are to be used as encryption key for filenames, encryption and HMAC keys for the `DataNode` struct respectively. The User struct is populated as defined above, marshalled (to be used as input to B) and put in the Datastore with the $K = A$ and $V = B$.

LoadUser: `Argon2(password,salt=username)` is used to deterministically generate random bits and the steps in InitUser are repeated. An attempt is made to retrieve the user record from the `Datastore`, which will succeed only if the correct username and password is used. If the retrieval is successful, the HMAC tag is verified and upon verification and successful decryption, the `User` struct is returned.

StoreFile: At this stage, a layer of indirection is introduced in the architecture: the `DataNode` struct. This acts as an access layer on top of the `FileInfo` struct and the actual files themselves, and enables the implementation of sharing and revocation. For every file, a `DataNode` object is first created as per the definition above, with random encryption and MAC keys – these will be used to encrypt and MAC the `FileInfo` struct that we create next. This is stored in the Datastore with $K = HMAC_{k_{User:FilenameKey}}(filename)$ and $V = AE_{k_{User:EncryptionKey}, k_{User:MacKey}}(marshalled\ DataNode, K)$. After this, we create the `FileInfo` struct with a list of UUIDs for the actual files (this is a list in order to support efficient appends) and random encryption and MAC keys – these are used to encrypt and MAC the actual file content, which is then dumped in the `Datastore` with key = respective UUID. The `FileInfo` struct is stored with $K = DataNode.ID$ and $V = AE_{k_{DataNode:EncryptionKey}, k_{DataNode:MacKey}}(marshalled\ FileInfo, K)$.

LoadFile/AppendFile: LoadFile repeats the above steps to retrieve the files from the bottom layer (the decrypted content from each UUID in the list of UUIDs) while AppendFile adds to the list of existing UUIDs in `FileInfo` and adds the appended content directly to the `Datastore` (with the new UUID).

ShareFile: The idea here is to share the `DataNode` struct with the user with whom the respective file is to be shared since the `DataNode` acts as the intermediate access layer to the file content layers underneath. The marshalled `DataNode` struct for the corresponding filename is encrypted using the public key of the recipient and this ciphertext is then signed over by the private key of the sender. The signature and the ciphertext is together sent as the `msgid`.

ReceiveFile: On receiving the `msgid`, the recipient verifies the signature using the sender's public key and upon successful verification, decrypts the ciphertext using their private key. The decrypted `DataNode` is unmarshalled, encrypted with the user's own encryption and MAC keys and stored in the `Datastore` with the desired filename (as in StoreFile). This architecture allows anyone with access to the file to make updates, which are propagated instantly to all users with access as all the users have access to the same `DataNode` object and subsequently, underlying file content.

RevokeFile: The file is loaded, the existing `DataNode` is deleted from the `Datastore` and a new file is created (as in StoreFile) with the loaded data as the file content. This ensures that anyone except the current user can no longer access the same file as it has been removed from the `Datastore` and reinserted with new `DataNode` and `FileInfo` objects that the other users have no access to.

Testing and Security Analysis: --

In most cases, the names of the following tests are self-explanatory regarding what they are testing and the security tests do include a small description.

Functionality tests: TestGetUser, TestFailGetUser, TestHeavyUser, TestHeavyDNE, TestHeavyUsers, TestLongKeysLargeFiles, TestNonCollison, TestChainSharing, TestAppend, TestEmpty, TestOptimalAppend, TestRevoke, TestRevokeandUpdate

Security tests: TestGetFile (naive attacker), TestRandomMalicious (modifier attacker), TestChainSharingRandomMalicious (modifier attacker), TestAttackThenAppend (modifier attacker), TestAttackThenHeavyAppend (modifier attacker), TestEnsureEncryption (checks if stuff is encrypted), TestLevenshtein (checks if attacker can identify file with > 1/2 probability)

Attack 1: An attacker can attempt a dictionary attack on filenames or usernames.

This does not work on our scheme as we store the $HMAC_k(username/filename)$ in the `Datastore` and therefore, without the symmetric key k, the attacker cannot learn the hash.

Attack 2: An attacker could attempt to switch the values between K-V pairs stored in the `Datastore`.

This attack fails against our scheme as we always include the key K in the HMAC for every value V that we encrypt (as noted in the preliminaries). Always checking the HMAC before decryption ensures that if attackers were to swap between pairs, an integrity error would be thrown instead of returning incorrect information.

Attack 3: An attacker can intercept the `msgid` shared while sharing files and gain access to the shared file.

Our scheme prevents this by encrypting the shared `DataNode` with the public key of the recipient. Since we know that the `Keystore` is secure (does not contain any malicious users), we can be certain that the encrypted message cannot be decrypted by anyone other than the intended recipient (since only the user's private key can decrypt the data). Moreover, the signature over the ciphertext using the sender's private key ensures that if a MITM modifies the `msgid`, it would be detected by the recipient and appropriately handled.