System Overview

All files are stored at random UUIDs and encrypted and HMAC'd with random symmetric keys. Each User struct stores UUIDs, but not file keys. File keys are encrypted with PKE and stored on the Datastore. When sharing a file, the sender stores the file keys on the Datastore for the recipient. The sharing tree is also stored on the Datastore, in that each user stores the list of their children for each file, encrypted for the original owner of the file. When revoking, the owner re-encrypts and uploads the file with new keys, constructs the sharing tree, and distributes the new keys to all users who still have access via the Datastore. To support efficient append, files are stored as a header, which has a list of locations of data blocks. All file names are hashed to provide confidentiality of length, and the User struct is loaded and stored at the start and end of each function to ensure consistency across multiple clients.

InitUser

When a user is initialized, a User struct is created and populated with their username and password. An RSA private/public key pair is generated for encryption, and a separate RSA key pair is generated for digital signatures. The two private keys are stored in the User struct and the two public keys are stored on the Keystore. To support file storage and sharing, a series of three maps are initialized and stored in the User struct as follows: (1) a map from filename → UUID, (2) a map from filename → the owner of the file, and (3) a map from filename → a list of users who this user has directly shared the file with. The User struct is marshalled, encrypted, and HMAC'd with deterministic keys. The encrypted and authenticated User struct is then stored on the Datastore at a deterministic UUID. Finally, to support user authentication, a slow hash of the user's password is generated with a unique, deterministic salt and stored on the Datastore at a unique, deterministic UUID.

GetUser

First, the user is authenticated by retrieving the password hash stored on the Datastore associated with the username and comparing it to the hash generated from the inputted password with the same deterministic salt. Then, a pair of symmetric encryption and HMAC keys are deterministically generated for the User struct, as in InitUser. The encrypted User struct is retrieved from the Datastore, authenticated, decrypted, and unmarshalled.

1. StoreFile

Either, (1) the filename does not yet exist in this user's namespace and it is a new file, or (2) it does exist and the user is attempting to update the file. The user's filename → UUID map is checked for the existence of the filename:

(1) If it is a new file, a random symmetric encryption key and a random HMAC key are generated, as are two random UUIDs, one for the file header, and one for the file data itself. This file metadata is stored in the three maps in the User struct. The random file keys are encrypted with the user's public encryption key, signed with the user's private signing key, and stored on the Datastore for future retrieval at a UUID deterministically generated from the location of the file and the user's username. The updated User struct is then marshalled, encrypted, HMAC'd, and stored on the Datastore.

(2) If it is an existing file, the keys are retrieved from the Datastore, verified, and decrypted. The user's access is then verified by checking if the user has the latest version of the file keys to ensure that a revoked user cannot overwrite the file. A random UUID is then generated for the file data.

In both cases, the file data is then encrypted, HMAC'd, and stored on the Datastore at the random file data UUID, and the file data UUID itself is encrypted, HMAC'd, and stored on the Datastore at the file header UUID.

LoadFile

The file header UUID is retrieved from the filename → UUID map and used to retrieve the symmetric encryption and HMAC keys for the file from the Datastore. The file header is retrieved from the Datastore, authenticated, and decrypted. The file header contains a series of file data UUIDs, so the data at each UUID is retrieved, authenticated, decrypted, and concatenated to form the whole file.

2. ShareFile + ReceiveFile

The user retrieves the symmetric encryption and HMAC keys for the file from the Datastore. This user's access is checked to make sure he is not revoked, then this user encrypts the symmetric keys with the recipient's public encryption key, signs it, and stores it on the Datastore for the recipient at a deterministic UUID generated from the file location and the recipient's username. The access token contains the file location concatenated with the owner of the file. This token is encrypted and HMAC'd with random keys, which are then encrypted with the recipient's public encryption key and signed by the sender (directly encrypting the access token with PKE would not support long usernames). The recipient is added to the user's filename → shared user list map, and the updated User struct is marshalled, encrypted, HMAC'd, and stored on the Datastore. Finally, the shared user list is marshalled and encrypted and HMAC'd with random keys. These keys are then encrypted with the file owner's public key and signed by this user. The encrypted shared user list is then stored on the Datastore, to later be used by the file owner for revocation.

When receiving, the access token is verified with the sender's public key and decrypted with this user's private key to retrieve the file location and owner information, which is used to retrieve the file symmetric keys from the Datastore. These could be signed by either the sender, or the file owner if the owner has revoked another user between the ShareFile and ReceiveFile calls, meaning the keys have been updated. This user then verifies and decrypts the file symmetric keys, encrypts and verifies them with his own keys, and re-uploads them to the Datastore. Finally, this user's maps are updated with the file metadata and the updated User struct is marshalled, encrypted, HMAC'd, and stored on the Datastore.

3. RevokeFile

The user loads the file, re-encrypts and HMACs it with new randomly generated keys, and stores it back on the Datastore. The target user being revoked is removed from this user's filename → shared user list map. The new file keys now need to be distributed to all members of the sharing tree who still have access. Since every time ShareFile is called, the updated shared user list for every user is encrypted for the owner of the file and stored on the Datastore, the owner is able to retrieve these lists and decrypt them to construct the sharing tree. The user then iterates through the tree, replacing the file keys stored on the Datastore for each user with the new ones.

4. AppendFile

Files are stored on the Datastore as a header and a series of data blocks. The header contains the UUID of each data block. Thus, to append to a file, a user only needs to store the new data at a random UUID, load the header, append the UUID of the new data, and store the header back on the Datastore.

Security Analysis

An attacker might attempt to corrupt the user struct in the datastore in order to initiate some malicious behavior. Since the user struct stores important data about a user's files, it is important that a user can rely on the integrity and authenticity of their user information. To ensure that the user struct's authenticity is guaranteed, we generate an HMAC for the user struct before we encrypt and store it on the datastore. When we call getuser(), we verify the HMAC tag of the struct to determine that the data has not been tampered. If the data was compromised, our system would throw an error and prevent the user from using the compromised data. In verifying the authenticity of our user struct data, a user can make sure that the integrity of the data is also secure.

After user A shares a file with user B, an attacker may intercept the access token and try to read the file. However, just the access token alone will not allow the attacker to gain access to the file. Without either user A's or user B's private key, the attacker will only have access to the encrypted version of the file. Due to AES-CTR's IND-CPA property, the attacker will not be able to learn any information about the file, including the filename or even the filename's length.

Consider the following scenario. User A shares a file with user B. User B creates another account B' and shares the file with B'. User A then revokes user B's access to the file. The attacker may try to access the file with user B'. However, in our design, as soon as user A revokes access from user B, the encryption key for the file immediately changes, and the new key is only distributed to those users still authorized to access the file. In this case, user B' would not receive the new key, since the whole access subtree rooted at user B is removed from the list of authorized users when access is revoked from user B.