Design Document
Shreya Nagpal and Shahar Sandhaus

Data Structures

User Struct

Username: the username of the current user
entropy: a 16 byte buffer (generated using Argon2Key) used as the source of entropy for HashKDF
rsa: an instance of the RSAKeys struct, used to store the private RSA keys of the user

Note: Unlike the other structs used in this project, this struct is entirely generated upon user login and is never stored in DataStore in any form.

RSAKeys Struct

DecKey: the user's private RSA key for message decryption
SigKey: the user's private RSA signature key, used to sign data when sending messages to other users

FileSector Struct

Data: a buffer which stores the file data
Next: a UUID pointing to the location of the next sector of the file in DataStore

FileMetadata Struct

First: a UUID pointing to the location of the first sector of the file in DataStore
Last: a UUID pointing to the location of the last sector of the file in DataStore

SharedUser Struct

Username: the user's username
Access: the UUID of the shared FileAccess struct for this user
Key: the key used to encrypt and sign the shared FileAccess struct for this user

FileAccess Struct

Metadata: UUID to the File's Metadata
SharedUsers: array of SharedUser structs
Key: the key used to encrypt and sign the FileSectors and FileMetadata
Redirect: a boolean specifying if this FileAccess struct is a redirection struct pointing to a shared struct

FileInvite Struct

Access: UUID to shared FileAccess struct for this user
Key: the key used to encrypt and sign the shared FileAccess struct for this user

User Authentication

When calling InitUser or GetUser, the first thing we will do is generate User.entropy using the Argon2Key function to derive an entropy source from the User's username and password. We then deterministically generate a UUID (using the username and password) in which RSAKeys will be stored.

For InitUser:

We then have to create an RSAKeys struct and randomly generate the keys that will be stored there. We save the private RSAKeys in the struct, and upload the public keys to KeyStore. Then, we encrypt and sign the RSAKeys struct with the deterministically generated key (using User.entropy and HashKDF). Finally, we upload the signed and encrypted RSAKeys struct to DataStore.

For GetUser:

We go to the deterministically generated UUID and use the deterministically generated key to retrieve the RSAKeys struct from Datastore, verify its integrity, and decrypt it

Further Info:

To ensure that multiple users can simultaneously use the service, the only information stored locally on the client's memory will be an Instance of the User and RSAKeys struct, which remain static and unchanging after user creation. All other user information that is needed for an API call will be fetched at the beginning of the function call. Meaning that no synchronization is needed between concurrent client programs.

File Storage and Retrieval

To circumvent the need to store a dictionary of userspace filenames and corresponding UUID locations, a UUID for each user's file will be created using a combination of a hash of the user's username and a hash of the filename. Similarly, an encryption and HMAC key will be deterministically created from the username/filename hash combination. Therefore, whenever performing file operations, the first action taken by the client will be to generate this UUID and keys.

Creating Files

When creating a file with StoreFile, a new FileAccess struct will be created and stored in the file's deterministic UUID (signed and encrypted with the deterministically generated keys). A random symmetric encryption and HMAC key will be generated and stored in FileAccess.Key (used to encrypt/decrypt and sign/verify FileMetadata and FileSector). FileAccess.Redirect will be set to false. FileAccess.MetaData will be set to a randomly generated UUID which will store the FileMetadata struct of the file. In the FileMetadata struct, we will generate UUIDs for the First and Last sectors. In the First FileSector, we will store the user's content, and set its Next member to point to FileMetadata.Last. Then we will generate a blank FileSector struct to put in FileMetadata.Last (to act as the end of the FileSector linked list).

Overriding Files

Identical to the process for creating files, except we don't need to create a new FileAccess struct. So all we do is override the FileMetadata struct and create a new set of FileSectors

Appending Files

To append to a file, we just go to the last sector of the file (stored in FileMetadata) and put our contents in its Data member, then we set FileSector.Next to a new UUID. Then we upload the FileSector, set FileMetadata.Last to the new UUID, update FileMetadata, then upload a new blank FileSector to FileMetadata.Next.

Reading Files

To read a file, we traverse the linked list of FileSectors (starting at FileMetadata.First) and appending to our return buffer until we reach the last FileSector (the one where the Next member is set to nil).

File Sharing and Revocation

Inviting a user as owner

To create a user invite, the owner must first create a new FileAccess struct that will act as the access point for the shared user, and the users that user shares the file with. The Key member will be initialized to the encryption and HMAC keys for the file. The Metadata member will be set to the UUID of the FileMetadata. The SharedUsers will be set to nil and Redirect will be set to false.

Finally, to invite the user, the owner will create a FileInvite struct which will include the encryption and HMAC keys for the FileAccess struct, and the UUID of the FileAccess struct. This new FileInvite struct will be encrypted with a randomly generated symmetric encryption key, then signed with the sharer's private key. Then, a random amount of padding will be chosen and the three arrays will be appended together and lastly encrypted with another randomly generated symmetric key. The last step of this protocol is to encrypt both symmetric keys with the sharee's public RSA key, append the resulting 256 byte array to our payload, and finally upload the resulting byte array to DataStore. The UUID of the FileInvite struct will be sent to the shared user as the invitationPointer. Lastly, the owner of the file will add the user to the SharedUsers member of their personnel FileAccess struct.

Inviting a user as non-owner

This process is like inviting a user as the owner, except the person sharing won't need to create a new shared FileAccess struct (they will just use the one made by the owner). Just like when inviting as the owner, the user will create a FileInvite struct and populate it with identical information to the FileInvite struct they received from the file owner. They will encrypt the FileInvite struct using the same protocol outlined in the above section, and finally they will upload the encrypted and signed payload to Datastore.

Accepting an Invitation

When accepting a file invitation, the user will read the FileInvite struct (verifying that the signature of it matches the public RSA key of the user who invited them). Then the shared user will create a new FileAccess struct located at the UUID generated by the deterministic algorithm described in the File Storage and Retrieval section of this document. This FileAccess struct will serve as a pointer to the "real" FileAccess struct. To mark this struct as a "pointer", the Redirect field will be set to true. The Metadata field will contain the UUID of the real FileAccess struct. The Key field will contain the encryption and HMAC keys used to encrypt and sign the real FileAccess struct. Therefore, whenever a shared user needs to perform file operations, they will load their FileAccess struct and check the Redirect field. If its false,

then they are the file owner and proceed as normal. If it's true, then they go to the UUID in Metadata and decrypt/verify it using the Key field of their personnel FileAccess struct. Then they use the downloaded struct as the actual access point for the file.

Revoking user access

To revoke a user's access. The owner will first delete that user's FileAccess struct. Then they will generate a new HMAC key and encryption key to be used with the file. Then, the owner will read the entire contents of the file, and reupload it to Datastore at a new location using the new encryption and signature keys. Finally, the owner will update the FileAccess structs of the remaining shared users with the new keys and UUIDs.

Helper Methods

To help deal with securely uploading and downloading from datastore, we have two functions, _DatastoreSet and _DatastoreGet, which operate as such: (these functions are called for all datastore get/set operations except for FileInvite structs) [note: the "key" argument of this method is a 48 byte array, which acts as three separate keys (2 for symmetric encryption/decryption and 1 for HMAC)]

- Upload:
    - Marshal payload
    - Encrypt resulting byte array with bottom 16 bytes of key
    - HMAC Sign with the middle 16 bytes of key
    - Concatenate both byte arrays
    - Generate random amount of padding where each byte of padding equals the total number of bytes of padding
    - Append the padding to the concatenated encrypted data and signature
    - Encrypt the concatenated array with the upper 16 bytes of key
    - The final byte array is then marshaled and uploaded to Datastore
- Download: Perform all operations in revers

We have a helper function _DeriveEntropyFromUserdata which takes the username and password and generates the user entropy, RSAKeys UUID, and RSAKeys encryption/HMAC key.
We have _KeyStoreSet and _KeyStoreGet which set and get keys in keystore at a deterministically generated location based on username and type of key (encryption or verification).
We have _DeriveFileInfo which deterministically generates a file's UUID and encryption/HMAC key.
We have _GetAccess which takes a filename and returns the FileAccess struct for that file.

Draft tests

Requirement: The client SHOULD assume that each user has a unique username, Go functions MUST return an error if malicious action prevents them from functioning properly

1. Create a user
2. Attempt to create another user with the same username
3. Expect an error indicating that the username is already in use

Requirement: The client MUST ensure confidentiality of file contents.

1. Define ReturnUUIDs(), a function that returns the UUID of a user's FileAccess struct, FileMetadata struct, and all related FileSectors in DataStore
2. Create a user User1 and a file File1 owned by User1
3. UUIDs = ReturnUUIDs()
4. Log out and attempt to access information in every UUID in UUIDs
5. Verify that none of the information received contains any plaintext version of the file (in the case of FileSectors), plaintext version of UUIDs and encryption keys (in the case of FileMetadata and FileAccess)

Requirement: The client MUST ensure integrity of file contents.

6. Define ReturnUUID(), a function that returns the UUID of a user's FileAccess struct in DataStore
7. Create a user User1
8. Create a file File1 owned by the user
9. Place malicious files at the UUID returned by ReturnUUID(), effectively swapping the file owned by User1 with a malicious file
10. As User1, attempt to access File1
11. Expect an error indicating that File1 has been tampered with

Requirement: The client MUST NOT assume that filenames are globally unique. For example, user bob can have a file named foo.txt and user alice can have a file named foo.txt. The client MUST keep each user's file namespace independent from one another.

1. Create two users User1 and User2
2. Create files foo.txt for each user
3. Store two different strings in each of the files
4. Verify that when reading both files, each User.LoadFile call returns the correct message

Requirement: The client MUST enforce that there is only a single copy of a file. Sharing the file MAY NOT create a copy of the file.

1. Define ReturnUUIDs(user, file), a function that returns the UUID of a user's FileMetadata struct and all related FileSectors in DataStore
2. Create a user User1 and a file File1 owned by User1
3. Create a user User2 and share File1 as Use1 with User2
4. UUIDs1 = ReturnUUIDs(User1, File1), UUIDs2 = ReturnUUIDs(User2, File2)
5. Expect that UUIDs1 == UUIDs2

Requirement: The client MUST allow users to efficiently append new content to previously stored files.

1. Define TimeAppend(), a function that measures the amount of time that it takes User.AppendToFile() to append to a given file.
2. Create one user, create and store one large (~1gigabyte) file.
3. Y = size of file
4. Append varying amounts of data X (X = 1 byte, 100 bytes, 1 kilobyte, 1 megabyte) to the file.
5. Expect that the result of TimeAppend() scales linearly with the amount of data being appended (TimeAppend() $\propto$ X, TimeAppend() $!\propto$ Y)