CS 161 Project 2 Design Document

Haiming Xu (3034177280) and Ru Pei (3034199185)

Section 1: System Design

User Storage

Users are stored as a struct in the Datastore which contains the user's username, password, owned files, files shared with others (more later), and relevant keys (for MAC, RSA decryption, and signing). Note, for the purposes of this design, MAC always refers to HMAC.

When initializing a user, we first create a `masterKey`, which is simply a symmetric encryption key deterministically generated using `Argon2Key` with the password and salted partially with the username (username + secondary salt). We then populate the struct as follows:

- The MAC key is also deterministically generated from the username and password (but is different from the `masterKey` as we use a different secondary salt).
- The file data structures are initially empty
- Keys for RSA and signing are randomly generated. The public keys are stored in the Keystore (keyed by `username + "_rsaek"` and `username + "_ds"`)

We derive the UUID from the first 16 bytes of the hash of `masterKey`. Since hashes are one-way, we don't have to worry about leaking information. The entire user struct is then marshaled, encrypted with `masterKey`, MACed, and stored in the Datastore. For symmetric encryption, we pad using PKCS#7.

Getting a user is simple: we can recreate the UUID and check if that user actually exists in the Datastore. The UUID will match iff the username and password match. If it does, we unmarshall, verify the integrity of the struct, and decrypt.

File Storage

Files are represented as a LinkedList of `MetaData` nodes. Each one of these nodes includes the UUID of itself (for convenience), the next node, the last node if it's the head, and the UUID of a `File` struct that contains the actual contents of the file. Note UUIDs are essentially pointers in this context.

For each file, we encrypt and MAC independently (with new encryption and MAC keys) and the same keys are used for all contents that belong to the same file, even those that are later appended. Thus, we keep track of a user's owned files via the `Files` attribute in the user struct. This is a map from filename to a `FileKeys` struct that contains an symmetric encryption key, MAC key, and the UUID of the first `MetaData` node.

1. How is a file stored on the server?

- If we're storing a new file
  - We randomly generate an encryption and MAC key, and UUIDs for the head `MetaData` node and `File` struct
  - We initialize a `File` struct and a corresponding `MetaData` node. Marshal, encrypt, MAC, and store both on the Datastore
  - We initialize and add a relevant `FileKeys` struct to a user's `Files`
- If we're overriding an existing file
  - We fetch the same info (keys and UUIDs) as the original file (this is important for sharing: we want all shared users to see the changes and be able to decrypt)
  - We initialize a `File` struct and a corresponding `MetaData` node. Marshal, encrypt, MAC, and store on the Datastore

- We clear the head `MetaData`'s Next and End pointer (if info was appended to the file, it should not exist anymore) and the corresponding `FileKeys`' head pointer

Loading files is as easy as accessing the user's relevant `FileKeys` struct, fetching the head `MetaData` from the Datastore, and running through the LinkedList while decrypting the information.

4. How does your design support efficient file append?

Our design achieves appending in O(size of appended contents). To append, a user will:

1. Fetch the end `MetaData` node directly from the head `MetaData` - O(1)
2. Initialize, marshal, encrypt, MAC, and upload the new `MetaData` and `File` structs to the Datastore - O(size of appended contents) due to encryption
3. Update the end node's next and the head's end - O(1)

Since we're using pointers, this update will also be instantaneously available to all other users this file was shared.

File Sharing

2. How does a file get shared with another user?

The only two things a user needs to access an existing file: encryption information and the address of the head node. Hence, to share, we can simply send a `magic_string` that's the marshaled and encrypted version of all that information + a signature. We do exactly with RSA encryption with slight nuance: to better revoke chain sharing, however, we instead append a dummy node to the first `MetaData` node that's being shared. This node has all the same information but an extraneous `File` UUID (this will be fleshed out when talking about revoking).

On the user end, the user who owns the file creates a new entry into their `SharedWithOthers` attribute. This is a map from filename to a map of shared users to the UUID of their head (or dummy) pointer. Ultimately, this keeps track of all files they have permissions to that they've shared with others.

Receiving a file is simply decrypting the `access_token`/`magic_string`, verifying the signature, unmarshalling, and adding the file to the recipient's `Files`.

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

The point (no pun intended) of having a pointer (`FileKeys`) to a pointer (`MetaData`, which is potentially a dummy) to the file contents is to make this process easy. The owner simply has to upload the file to a new UUID and change the revoked user's head node's next pointer to garbage. Thus, the revoked user won't be able to find the new file or any changes that are made.

Our design also seamlessly facilitates revoking chain sharing. If user1 shares with user2 who then shares with user3, revoking access to user2 also revokes user3. User2's dummy pointer will be invalid, and hence, any pointers dependent on it are too.

Additional Subtleties

We'd like to briefly comment on 2 subtleties.

1. To support the same user logging in on different instances, we fetch the marshalled user information from the Datastore at the start of every operation, and update it at the end
2. Filenames are never exposed by files themselves (the file doesn't even know its own filename). Only the user knows; thus, no information about the names is leaked.

Section 2: Security Analysis

Dictionary Attack

With access to `Hash`, `Argon2Key`, and an understanding of the system implementation (how the UUID is generated), an adversary can create an offline lookup table with many passwords. Without salting, the adversary just needs to match UUIDs in the Datastore with their lookup table to find the password that matches. This then allows the adversary to call `GetUser` and log in as the victim. Hence, we need to introduce a salt to limit the adversary to cracking an individual user at the same time. We choose this unique salt to be partially composed of the username. However, we can imagine that usernames can be short and thus, limit security. To mitigate this, we append a secondary salt to the username as our final salt. The adversary is now limited to making online attacks for a single user (with a decently long salt), which is significantly slower and more secure.

Man In the Middle during Sharing

Sharing is facilitated by returning a `magic_string` with sufficient information in `ShareFile` as the `access_token` in `ReceiveFile`. Our threat model says the adversary can view and change this string. Since we encrypt with RSA, the MITM doesn't gain any information from viewing the string, as the only person able to decrypt it is the intended recipient. Even if they try to alter the `access_token`, since we send a digital signature as well, only the intended sender (from the receiver's end) could create a valid digital signature.

Scrambling/Swapping Datastore UUIDs

Since the adversary effectively controls the Datastore, they can theoretically upload their own file at a different UUID, then scramble/swap UUIDs to have their file appear at the UUID of an existing file. This is a piece of trickery that would allow them to modify the contents of any file (or append with whatever they wish) even without permissions, and is difficult to detect. Since after files are shared, all shared users act as "owners" and can override the file with a new one, none of the owners will know it's a third-party that made the change once they `LoadFile`. They can even communicate amongst each other and have no guarantee another party isn't lying. With independent encryption/authentication for all files, however, for an adversary to be able to overwrite with non-gibberish, they would need to obtain the original encryption/authentication info for the file. If they did, they might as well just manually overwrite (and all security would have been lost for that file regardless).

Access String Reuse

Let's assume an adversary gets shared a file (for whatever reason) and keeps track of the `access_string` provided when they call `ReceiveFile`. Next, the file owner realizes the mistake they made and revokes permissions. Since the adversary keeps the `access_string`, they could try to share this string with a new malicious user (either another individual or another account they create). Without proper design, this could allow the malicious user to retain access to the file at the same level in the sharing tree (i.e. one degree away from the initial sharer) and thus, not even be affected by revoked chain sharing. However, since we change the UUID of the new file, the malicious user will only have access to the old version of the file. Any changes they make will not impact anybody else (they kind of have their own sandboxed version of the file), and hence, this is completely harmless.