System Design

InitUser

Create a unique user UUID from username, and a hashed, salted password using username as hash. Generate required keys and store in either keystore or User (see later functions). Encrypted using a secret, deterministic symmetric key from hashed, salted password for confidentiality, MAC using derived key for integrity. Store marshalled and MACed User into datastore.

GetUser

Generate user's UUID and hashed, salted password from username, password, and keys. Get, unencrypt, and verify MAC of User from datastore using the aforementioned keys.

StoreFile

Generate a random key and derive a mac key from it. After checking if the filename exists for overwriting, create a File object and load in the encrypted file contents. Add a nullUUID as the next and final hop. Reencrypt and MAC the entire File. Store marshalled and MACed File into datastore. Add the file instances to the User's created files table. Update the User in datastore.

AppendFile

Create a new File object for the edit and a new UUID. Find the file you are appending to in either the Received and Created files tables to find their location in datastore and the key. Get the "base" file, or the first File that was created in StoreFile. After extracting, update the FinalEdit on the base to be this edit's UUID, and update the previous FinalEdit's NextEdit to be this edit's UUID. Encrypt the data, entire File, and MAC the File using the same key and add the edit to datastore.

LoadFile

Find the file you are appending to in either the Received and Created files tables to find their location in datastore and the key. Get the "base" file, or the first File that was created in StoreFile. Starting with the base, iteratively extract the Data from each File, using the NextEdit field to find appended data. Unencrypt with key and verify MAC of each edit while iterating.

ShareFile

Create a magic string from random bytes and Share object. Search Received and Created table to get the key and pointer to either my parent's token or the file. Encrypt the key with the receiver's public key and store it in the Share along with the NextHop and FinalHop information. Create a UUID from the magic string to place the Share object in datastore. Send the UUID and its signature to the receiver. Add token information to the user's Shared table.

ReceiveFile

Split the magic string and signature and verify the signature. Create a UUID from the magic string and get the share object. Decrypt the key and store it, along with the NextHop information, in the Received table.

RevokeFile

Search Shared for token. Delete token from datastore and remove from table. This removes the target's children's access too as they cannot use their parent's NextHop.

Questions

1. How is a file stored on the server?

File data is added to a File object, along with information about the next and final edit. The File object is encrypted and MACed and stored in the datastore. “Appends” are represented as edits and a file’s NextEdit field is a pointer to the next edit in the datastore much like a LinkedList. Each edit is encrypted with the same key and MACed with the same key. The FinalEdit pointer points directly to the FinalEdit to make appending a constant time operation as opposed to linear in the number of edits.

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

The sharer creates an access token that contains the file’s UUID, the file’s key (encrypted with the recipient’s public key), and a signature on the token. The magic string is the encrypted UUID of the access token appended to a signature on the UUID. The recipient can use the access token to get the file’s UUID and unencrypt and store the file’s key. Storing this information allows for file retrieval. For subsequent sharing, i.e. if a non-file creator shares with someone else, the sharer’s token’s UUID is stored instead of the file’s UUID to create a tree-like structure that a child must climb to eventually get to the file.

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

As aforementioned, our sharing system buildings a tree of access tokens. Each node must climb its way to the root in order to access the file. Deleting a node deletes his ability to access the parent node, thus revoking access to the target and his/her children.

4. How does your design support efficient file append?

Yes. As aforementioned, using a FinalEdit pointer that points directly to the FinalEdit, we made appending a constant time operation as opposed to linear in the number of edits.

Section 2: Security Analysis

Amortized dictionary attack: When a user creates their account, they must input a unique username and a password of good entropy. In order to prevent the amortized dictionary attack in the case where an adversary compromises the location in which the passwords are stored, we do not store the password in plaintext. Instead, we salt and hash the passwords using the password-based key derivation function provided to us in userlib.go. The salt need not be secret, so we simply set it equal to the user's username. Now, when an attacker attempts to compute the salted hashes for the most common passwords, they must have a separate computation for each guess and each user, which significantly slows them down. Furthermore, we could slow down the hash function by hashing the hash of the hash. We could reuse the same salt here, because the salt information will always be available to the attacker.

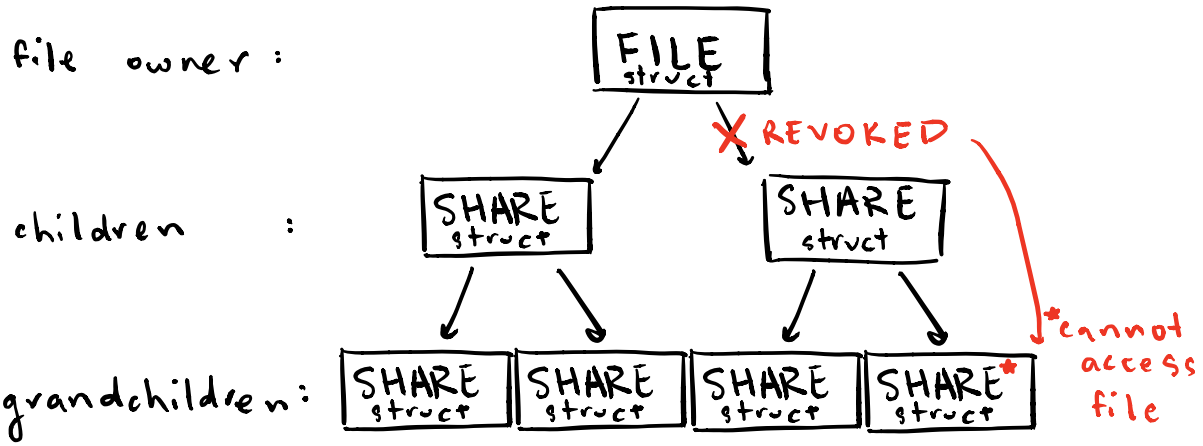
File storage attack: While the datastore is untrustworthy, we must use it store our file information for statelessness. First, we secure the filename by hashing the filename and using 16 bytes of the hash to generate a UUID. Then, we store this UUID as a key in datastore, and set the key's value to a file struct. A file struct contains all relevant information for a single file: a random edit number, a symmetric encryption of the filecontents, a mac of the filecontents, and the UUID of the next file edit (if an edit exists). The file struct itself is not encrypted because it doesn't reveal any sensitive information. Instead, we encrypt the contents with a randomly generated symmetric key, and use the hash based key derivation function to generate another key that we use to secure the mac of the file contents. Now, if an attacker looks into our file struct, they will be unable to read the contents because they do not have the symmetric key (it's only stored in the creator's userstruct) and they will be unable to modify the file contents without raising alarm as the new mac will not match the mac in the file struct.

Sharing attack: Let user A create a file F and share F with user B. Let A create another file F' and share it with user C. What prevents C from obtaining access to F and sharing this with other users? Our solution defends against this with the following. First, when A shares F' with C, C only receives an access token that points them to a share struct of F' in datastore. This share struct contains the UUID of the file shared, a next-hop boolean (used if A was not the file creator), a symmetric key encrypted with C's public key, and a digital signature signed with A's private signing key. With this information, C only knows where the filestruct of F' is, and can only unlock the contents of this specific file struct. Even though A has another file F, C does not know where the file struct of F is because his access token only points him to the file struct of F'. Furthermore, if C knew where the file struct of F' was, the symmetric key he uses to unlock F' will not work for F as the keys are randomly generated for each file. C therefore will be unable to access F even though the creator shared a different file with him, and will also lack the information to share F with other people as he'd need to know not only the location of F, but also the private signing key of A.

System Design

Design Diagram

File Sharing:



Files :