System Design

Question 1: How is a file stored on the server?

We compute[a]: UUID^1, UUID^2, UUID^3, UUID^4; symmetric-key^{ED1}, symmetric-key^{ED2}, symmetric-key^{ED3}, symmetric-key^{ED4}; symmetric-key^{M1}, symmetric-key^{M2}, symmetric-key^{M3}, symmetric-key^{M4}.

We create a counter (counter = 0 when file is first stored) and append it to the file-data[b], then store(file-data, UUID^4, symmetric-key^{ED4}, symmetric-key^{M4}). Next, we need to store the file-pointer (pointer to file-data) which equals struct{UUID^4, symmetric-key^{ED4}, symmetric-key^{M4}} on the DS, so we do store(file-pointer, UUID^3, symmetric-key^{ED3}, symmetric-key^{M3}). Next we take the file-pointer and append it to a master-sheet which equals struct{list-of-file-pointers} that keep tracks of all parts of the file. Next, we need to store the master-sheet on the DS, so we do store(master-sheet, UUID^2, symmetric-key^{ED2}, symmetric-key^{M2}). We then create a master-sheet-pointer (a pointer to the master-sheet), which equals struct{UUID^2, symmetric-key^{ED2}, symmetric-key^{M2}} and store it on the DS by executing store(master-sheet-pointer, UUID^1, symmetric-key^{ED1}, symmetric-key^{M1}). Lastly, we need to be able to access this master-sheet-pointer, as well as indicate who it belongs to, so we create an access-pointer which equals struct{UUID^1, symmetric-key^{ED1}, symmetric-key^{M1}, hashed-username} and we store it in owned_files_map[hashed-file-name][list-of-access-pointers][c] which exists in our user-struct.

Question 2: How does a file get shared with another user?

Sharing for root users: We create a new master-sheet-pointer for the recipient (to emphasize: this new master-sheet-pointer points to the same master-sheet as the root's master-sheet-pointer). We then create a new acces-pointer that points to the new-master-sheet-pointer, i.e. {UUID^5, symmetric-key^{ED5}, symmetric-key^{M5}, hashed-recipient-username}[d]. We then store the new master-sheet-pointer by executing store(master-sheet-pointer, UUID^5, symmetric-key^{ED5}, and symmetric-key^{M5}). When then give them access to this access-pointer by composing the following magic-string: compose-magic-string(UUID^5, symmetric-key^{ED5}, symmetric-key^{M5}, recipient-public-RSA-encryption-key, sender-private-RSA-signing-key). We, as the owner, want access to this master-sheet-pointer as well (to later revoke or repoint), so we append the access-pointer to our list of access-pointers that correspond to the hashed-filename in our owned_files_map.

Sharing for non-root users: When a non-root user shares, they simply give the recipient access to their master-sheet-pointer[e]. They can do so by retrieving their access-pointer which, lets say,

[a] Let $i \in \mathbb{R}$, UUID^i is a randomly generated UUID, symmetric-key^{EDi} is a randomly generated symmetric-key for encryption/decryption, and symmetric-key^{Mi} is a randomly generated symmetric-key for computing a HMAC.

[b] See Attack 1 in Security Analysis to see why we do this.

[c] Maps to a ***list*** of access-pointers because the owner stores the access-pointers of the users that we share with.

[d] NOTE: UUID^1 in Question 1 means something different than UUID^1 in Question 2 (same goes for keys); I simply offer this notation to offer distinction and clarity amongst the UUID's and keys in each individual question.

[e] We do this b/c if the child of the root gives all its children access to its node, then pruning becomes very easy.

equals, struct{UUID^6, symmetric-key^{ED6}, symmetric-key^{M6}, sender-hashed-username} (they do this by querying their received_files_map[hashed-filename][acces-pointer] in their user-struct). Next, they share this access-pointer by creating the magic-string: compose-magic-string(UUID^6, symmetric-key^{ED6}, symmetric-key^{M6}, recipient-public-RSA-encryption-key, sender-private-RSA-signing-key).

Receiving: The recipient can execute decompose-magic-string(magic-string, recipient-private-RSA-decryption-key, sender-public-RSA-verification-key) to retrieve the access-pointer and store it in their received_files_map in their user-struct; they now have access to the file.

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

The user revoking is the root, and they are revoking from their child[f]. We grab the list of file-pointers from the master-sheet associated with the file. Then we grab all the data that each file-pointer points to (file-data^i). Next, we generate a new-file-pointer^i for each file-data^i, i.e. struct{new-UUID^i, new-symmetric-key^{EDi}, new-symmetric-key^{Mi}}, and execute store(file-data^i, new-UUID^i, new-symmetric-key^{EDi}, new-symmetric-key^{Mi}). We create a new-master-sheet which equals struct{list-of-all new-file-pointer^i}. We create a new-master-sheet-pointer, which equals struct{UUID^7, symmetric-key^{ED7}, symmetric-key^{ED7}}; this points to the new master-sheet. We store the new-master-sheet store(new-master-sheet, UUID^7, symmetric-key^{ED7}, symmetric-key^{ED7}). Lastly, the root pulls the list of access-pointers from their owned_files_map[hashed-file-name][list-of-access-pointers], follows each access-pointer to the corresponding master-sheet-pointer, and updates each master-sheet-pointer with the new master-sheet-pointer, doing so for everyone (including himself & excluding the revoked user).

Question 4: How does your design support efficient file append?

To append to a file, a user must follow the pointers and chase down the master sheet. They first grab the access-pointer in their user-struct which equals struct{UUID^8, symmetric-key^{ED8}, symmetric-key^{M8}, hashed-username} and perform get(UUID^8, symmetric-key^{ED8}, symmetric-key^{M8}), giving them the master-sheet-pointer which equals struct{UUID^9, symmetric-key^{ED9}, symmetric-key^{M9}}. Next they get the master-sheet by performing get(UUID^9, symmetric-key^{ED9}, symmetric-key^{M9}). Then we create a file-pointer (for the new file-data we will append), i.e. struct{UUID^{10}, symmetric-key^{ED10}, symmetric-key^{M10}}. We take the file-data that we wish to append and attach a counter to it (counter = the number of current file-pointers). We then execute store(file-data-to-append, UUID^{10}, symmetric-key^{ED10}, symmetric-key^{M10}). We then add that file-pointer to the list of file-pointers on the master-sheet, since master-sheet contains a list of the file-pointers that the file has. Finally, we do a store on the updated-master-sheet: store(updated-master-sheet, UUID^9, symmetric-key^{ED9}, symmetric-key^{M9}). Because we only need to download the master sheet, and upload the encrypted updated master sheet and the encrypted appended portion of the file, appends scales *linearly* with the size of data being appended and is *constant* with respect to the number of users who have access to the file

[f] A user is a root if there is a value corresponding to hashed-file-name in their owned_files_map.

Diagrams

store(data, UUID, symmetric-key-ENC, symmetric-key-HMAC) $\longrightarrow$

marshal the data, encrypt with symmetric-key-ENC, compute HMAC of the encrypted data with symmetric-key-HMAC, concatenate the encrypted data and the HMAC, store it in the DataStore at UUID.

get(UUID, symmetric-key-ENC, symmetric-key-HMAC): $\longrightarrow$

get data from DataStore at UUID, split data into encrypted-data and encrypted-data-HMAC, compute HMAC of encrypted data with symmetric-key-HMAC, verify it's equal to encrypted-data-HMAC, decrypt encrypted-data using symmetric-key-ENC, unmarshal and return it.

compose-magic-string(UUID, symmetric-key-ENC, symmetric-key-HMAC, recipient-public-RSA-encryption-key, sender-private-RSA-signing-key): $\longrightarrow$

concatenate UUID, symmetric-key-ENC, symmetric-key-HMAC (as a byte array), encrypt the byte array with recipient-public-RSA-encryption-key, sign the encrypted data with sender-private-RSA-signing-key, append the two together, convert it to a string, and return it.

decompose-magic-string(magic-string, recipient-private-RSA-decryption-key, sender-public-RSA-verification-key): $\longrightarrow$

convert magic-string to byte-array, split byte-array into encrypted-data and signature, use sender-public-RSA-verification-key to verify the signature, and then use recipient-private-RSA-decryption-key to decrypt the data, thus returning the UUID and keys associated with the access-pointer.

Security Analysis

(Attack 1, Switching Around Appends) attack: An attacker can see the traffic to the DataStore. If a user appends to a file (at a different UUID), the attacker can try and swap multiple appends of the same file on the DataStore. **defense**: When we append, we add a *number* to the end of the appended-data (before we append it) that indicates the position of the appended-data in the entirety of the file data. If the attacks swaps appends, then when we decrypt the data and analyze the *number*, we will see that the *number* has the wrong value and know that the attacker swapped the data. Furthermore, through this design, we are able to actually piece the file back together properly, since the *number* tells us the position of each data as it fits in the file. **(Attack 2, Discovering Who Sent a Magic-String) attack**: The spec indicates that only the recipient should be able to know the sender. An attacker can take the magic-string which has the signature and then take everyone's verifying-key in the KeyStore and try to verify the signature; if it succeeds, then it indicates that the magic-string was sent by that user. **defense**: When storing a user's verifying-key in the KeyStore, we make sure that Key-Value pair in the KeyStore say nothing about who the user is! Instead of using the username + some-string or other variants as the Key to the KeyStore, we use the hash of the username + some-string as the Key. This way, even if an attacker sees that a certain verifying-key works, they won't know which user it belongs to. An additional defense is to encrypt the signature with the recipient's RSA-public-encryption key so that the attacker can't even do this to begin with. **(Attack 3, Amortized Guessing Attack) attack**: Since the adversary can see all the contents on the DataStore, he can try to discover the user's password by performing an amortized guessing attack by computing a bunch of hashes and see which match. **defense**: In order to avoid this we hash the password with salt being the user's username. This will protect against the attack; we couple this with the fact that the hash function we are using (Argon2Key) is slow. Additionally, no where in our design do we leak the username, so the attacker would have to guess both the password and the salt, adding in an extra layer of defense. The last defense is that we don't store the hash of the password on the DataStore like other systems, so they won't be able to do this compute and compare scheme. **(Attack 4, DoS After File Revoked) attack**: When an attacker gets a file revoked from him/her, he can perform a DoS attack by overwriting all the previous file contents. **defense**: We defend this by creating new id's and keys for all the file contents and re-encrypting them all (outlined in Question 4). This way even if the attacker, messes with the content that they had access to, the root and all its shared users will still have the file content :). **(Attack 5, Deletion of Append) attack**: An attacker can see the traffic to the DataStore. If a user appends to a file (at a different UUID), he can mess with the user by deleting that entry. **defense**: Our master-sheet keeps track of all the appends; therefore, when we traverse through our list of file-pointers on our master-sheet and we see that one of the file-pointer's point to a place in the DataStore with no content, we will be able to detect this attack.