David Dominguez Hooper
SID: 24828373
login: cs161-acc

Project 2: Design Document

Section 1 – Design:

- Summary of Design:

My approach is very similar to Design 1 in the Part 1 Design Document.

The client first stores a symmetric key, and the key's signature, on the server using the client's asymmetric public, and private key, respectively. 1. We securely store this on the server under "user/key". (The client needs this key in order to generate HMAC(file name) used in the next steps)**Upload** works by first generating two random symmetric key pairs to encrypt and verify the value of the file. 2. We securely store the value of the file under "user/HMAC(file name)"

We then store the two random symmetric key pairs on the server using the client's asymmetric keys. 3. We securely store these two symmetric keys under "user/HMAC(file name)/keys"

Download works by reversing the above process for upload.

Q2: Sharing:

Share:

To share files with another user, the client must ultimately share the symmetric keys (we will label these sym_keys0) used to decrypt and verify the value of the file. My implementation handles sharing these keys by creating a secure message to the person the client would like to share with. The message is encrypted with the asymmetric keys of the user the client would like to share with. The message contains a single location of these keys on the server, that will be shared with users (keeping this location singular, lets us update the keys at will in the future in case of a revocation). The message also contains two new symmetric key pairs (label sym_keys1) that are used to encrypt and verify sym_keys0. Share also securely stores on the server a list of the children that the client has shared a particular file with under "auth_users" (this is used later for revocation).

Receive Share:

A new user receives a secure (asymmetrically encrypted) message from the previous user they are receiving a shared file from. This message, detailed above, contains the necessary sym_keys1 used to decrypt sym_keys0. The new user uploads to the server the new name for the file that has been shared under "new_user/HMAC(new filename)". This stores a POINTER on the server that redirects to the original "user/HMAC(filename)". The new user also stores a POINTER to the location where the previous user has stored the sym_keys1 to decrypt sym_keys0 which thus decrypts the file's value. There are checks in **Upload** and **Download** to check if the POINTER exists and handles the above to decrypt the file's value successfully.

Transitive sharing: Transitive sharing works because the system uses pointers to redirect download to get the one copy of the sym_keys1 used to decrypt sym_keys0 then used to decrypt the file value.

Q3: Efficient Updates:

I tried to implement a Merkle Hash Tree to keep track of the changes pieces of a file. I created a Tree structure that had left and right sub trees used to grow the tree with more pieces of the file. I would have check the Merkle hash tree representing the already stored file on the server concurrently with Merkle hash tree representing the new file waiting to be uploaded. We would be able to check the hash of the new file with the old file and check to see if the hashes were the same. If they were not, we must continue until the hashes are the same. Once they are the same, we

can update the relevant changed pieces of the file, and thus have efficient updates. The performance of the Merkle Hash Tree would have been O(log n) with n number of hash checks until the system found a matching hash for the old and new pieces of the files.

Q4: Revocation:

The system handles revocation by first re-encrypting the file with new symmetric key pairs sym_keys0'. This is the state that is changed to revoke a file. Since we changed the sym_keys0 to sym_keys0', any user who would like to access the file needs to the new sym_keys0'.

It then proceeds to remove the revoked user from the authorized users list. It then is able to re-encrypt the new sym_keys0' for all the authorized users because the authorized users list also keeps track of the sym_keys1 shared with each new user the client has shared the file with. The system meets all of the revocation requirements because the revoked user and any children the revoked user has shared the file with can no longer decrypt the file because they do not have the new sym_key0'.

Section 2 – Security Analysis:

- The first security attack my system defends against is when a user tries to overwrite the metadata used by the client. I have checks at every point in my code to see if anything ever uploaded to the server has been modified.
- The second security attack my system defends against is when a user tries to overwrite a file for which the user does not have access to. We can check the MAC of the file and see if it has been changed.
- The third attack my system defends against is the attack when a user tries to change the authorized users list meant to keep track of who has access to a file that has been shared. We can check the MAC and signature of the auth_users list to see if it has been modified.
- The fourth attack my system defends against is revocation attacks. If a user tries to revoke another user for which they do not have ownership of the file to, then my system check the mac of that list.