Part 2 Sample Design

Our design mimics the sample insecure client provided in the student framework, and builds upon Design 2 from the solutions for Part 1. As in the solutions to Part 1, when we encrypt something we always perform "authenticated encryption" by first encrypting the value and then computing a MAC on the ciphertext.

To initialize a client, we create three master keys: a symmetric encryption key k_e, a symmetric MAC key k_a, and a symmetric key k_n for name confidentiality. We store these keys on the server encrypted and signed using our public/private key pair, as before.

We also create a "share directory" on the server, encrypted and authenticated using the master keys k_e and k_a. The directory is initially empty, but will be used for tracking users we share files with in the future. We store the directory at the ID `<username>/shares`.

To create a file initially, the client creates two *new* keys (k'_e, k'_a). It then stores the following two files on the storage server:

- The *data node* points to the file contents encrypted and authenticated using k'_e and k'_a, and stored at a randomly generated ID, r_1.
- The *key node* contains the two keys k'_e and k'_a, a pointer to the data node, and the filename, all encrypted and authenticated using the user's master keys k_e and k_a. The key node is stored at the ID `<username>/`r_2 where $r_2 = F_{k_n}(\texttt{filename})$. The function F is chosen as per the solution to Part 1.

To share a file with another user, we create a new *share node* that contains the ID of the data node r_1, along with the encryption and MAC keys used at the data node, k'_e and k'_a. The share node is stored at a randomly generated ID r_3.

We encrypt the share node using two new keys, k''_e and k''_a. We then construct a message m containing these two keys along with the ID of the share node r_3. We send m to the other user after encrypting it with their public key and signing it with our private key.

Additionally, update our share directory with information about the other user. The directory maps each shared file to the list of users we've shared the file with. We append the following information to the list corresponding to the shared file: the name of the other user, the ID of their share node, and the keys k''_e and k''_a used to encrypt the share node. This information will be important to us during revocation.

To receive a shared message, the receiving user first verifies the shared message m, and then extracts the keys k_e'' and k_a'' along with the ID r_3 of the share node.

Next, they create a new *key node* that contains the two keys they have received, and make this key node point to the share node. That is, the share node acts as if it were a data node, but instead of actually containing data it contains a pointer to yet another node (which is either another share node, or actually a data node). The user stores the key node at the ID `<username>/`r_4, where r_4 is the deterministic encryption of `newfilename` using the function F, and `newfilename` is the name assigned to the shared file by the receiving user.

To access a shared file, the user proceeds by decrypting their key node for the file. This will contain symmetric keys and a pointer to a new file. The user then decrypts this file. If it is a data node we stop and read the contents; if it's another share node, we continue in this manner until we reach the data node.

To revoke a user's access, we re-encrypt the file with new symmetric keys, and distribute these new keys to all of our children except the revoked user.

Specifically, we start out by re-generating two new encryption keys and store them in the key node for the file. We then download the entire file, and re-upload it using these new encryption and authentication keys. We place these keys in the key node so that we have access to it in the future.

Then, we use our share directory to iterate over all the users we've shared this file with. For each non-revoked user, we extract the ID of their share node along with the corresponding symmetric keys. We use this information to place the new keys at their share node.

Finally, we delete the revoked user from our share directory. Deleting the user will prevent us from accidentally granting them access to the file when we revoke access from another user in the future. (See test StayAfterRevoke for reasoning.)

This results in the functionality and security properties being satisfied. All of our children (and their children) will still be able to access the new file. However, the revoked user (and all of their children) will not be able to access the file because they will not learn the new encryption key.

Part 2 Tests

KeysAreNotReused This test ensures that when different files are shared with different people, they do not see the same keys. That is, Alice uploads files F and G and shares F with Bob and G with Carol. If Bob and Carol share any keys then you fail this test case.

StayAfterRevoke The following four tests will have similar formats. In each, an adversary will pretend to have valid access to the file. If, when another user is revoked, the adversary tricks the file's owner into giving them access, fail.

This test ensures that a revoked user cannot use known values to trick the file's owner into thinking the revoked user has access to the file, so that the revoked user gains access to the file when a different user is subsequently revoked.

LieAboutBeingShared An adversary attempts to use public information to convince a file's owner that the file was shared with the adversary, so that the adversary gains access to the file when another user is revoked.

ShareTiedToFile An adversary who received a valid share for file A attempts to convice the file owner that they have access to file B, so that the adversary gains access to file B when another user with access to B is revoked.

NoReadAfterRevoke This test verifies that it is not possible for a client to remember encryption keys so that after they are revoked, they can still read the modified file on the server by re-using the old keys they remembered.

OutOfOrderSharing This test checks that a client cannot access files it should not be able to access if it receives shares out of order.

NoReceiveAfterRevoke This test checks that if sharing is revoked before a client receives it, the client cannot receive it anymore.

NoRollbackPartialPuts This test makes sure that updates are always consistent even if an adversary tries to block some writes from occurring. That is, if Alice uploads (F, V) and then (F, V') where V' modifies many portions of F, this test checks if it is possible to stop some writes so that the result of downloading F returns some contents intermediate between V and V'.

NoRollbackPartialPuts2 Simply another way of testing NoRollbackPartialPuts.

NoSharedKeysAfterRevoke After a user's access to a file has been revoked, the user should no longer share any keys in use with the parent. That is, if Alice shares a file with Bob, then after Alice revokes Bob's access, Alice should never try to use a key that Bob knows, and Bob should never try to use a key that Alice knows.

RevokedDataIsReEncrypted Once a user's access to a file has been revoked, the original file must be re-encrypted under a new key. This test verifies that this occurrs and all modifications to the file are done under the new encryption key.

RevokedDataIsReEncrypted2 A different way of testing RevokedDataIsReEncrypted.

ShareMessageDoesntLeakKeys Share messages should be encrypted and keys should not be sent in plaintext in these share messages.

ShareMessagesAreSigned When Alice shares a file with Bob, it should not be possible for Mallory to switch out the message with one of her own so that Bob instead receives a file owned by Malory instead of Alice.

ShareWithSameName If Alice shares a file she calls F with Carol (who calls it F1), and then Bob shares a file he calls F with Carol (who calls it F2), then it should still be possible for Alice and Bob to revoke Carol's access of their own files without breaking Carol's client.

SharingMessageCantBeReplayed This test verifies that a user can't replay a share message to herself after having been revoked in order to re-grant herself access. That is, if Alice shares a file with Bob with share message S, and then later revokes Bob's access, it should not be possible for Bob to call `receive_share` on himself with S again and re-grant himself access.

SharingMessageCantBeReplayed2 Another way of testing SharingMessageCantBeReplayed.

SharingMessageIsEncrypted This test verifies that sharing a file from Alice to Bob does not allow a man in the middle to learn the name of the file being shared.

SilentDropPuts Another way of testing NoRollbackPartialPuts.