Spring 2017 - CS161 - Project 2
Part 2 Design Document
Brandon Teo, 26494656, cs161-aay

System Design (Q1 - Q3)

Snapshot of Untrusted Storage Server:

$alice/dirkeys$	: $E_{alicepublickey}(k_e \| \| k_a) \| \| Signature_{aliceprivatekey}(.)$
$alice/sdirkeys$	: $E_{alicepublickey}(k'_e \| \| k'_a) \| \| Signature_{aliceprivatekey}(.)$
$alice/dir$	: $\{fn_1 : (10111, k_1, k_2); fn_2 : (11001, k_3, k_4)\}_{k_e} \| \| MAC_{k_a}(.)$
$alice/sdir$	: $\{bobfn_1 : 00000\}_{k'_e} \| \| MAC_{k'_a}(.)$
$alice/files/10111$	: [D] $IV_1 \| \| E_{k_1}(SHA256(k_1) \| \| v_1) \| \| MAC_{k_2}(.)$
$alice/files/11001$	: [D] $IV_2 \| \| E_{k_3}(SHA256(k_3) \| \| v_2) \| \| MAC_{k_4}(.)$
$alice/sharewith/bob/00000$	: [P] $alice/files/10111$
$bob/dirkeys$	: $E_{bobpublickey}(k_e \| \| k_a) \| \| Signature_{bobprivatekey}(.)$
$bob/sdirkeys$	: $E_{bobpublickey}(k'_e \| \| k'_a) \| \| Signature_{bobprivatekey}(.)$
bob/dir	: $\{fn'_1 : (00001, k_1, k_2)\}_{k_e} \| \| MAC_{k_a}(.)$
$bob/sdir$	: $\{carolfn'_1 : 10101\}_{k'_e} \| \| MAC_{k'_a}(.)$
$bob/files/00001$	: [P] $alice/sharewith/bob/00000$
$bob/sharewith/carol/10101$	: [P] $bob/files/00001$
$carol/dirkeys$	: $E_{carolpublickey}(k_e \| \| k_a) \| \| Signature_{carolprivatekey}(.)$
$carol/sdirkeys$	: $E_{carolpublickey}(k'_e \| \| k'_a) \| \| Signature_{carolprivatekey}(.)$
$carol/dir$	: $\{fn''_1 : (11111, k_1, k_2)\}_{k_e} \| \| MAC_{k_a}(.)$
$carol/sdir$	: $\{\}_{k'_e} \| \| MAC_{k'_a}(.)$
$carol/files/11111$	: [P] $bob/sharewith/carol/10101$

Remarks:
- The 5 digit id's are a simplification of randomly generated 16 byte id's.
- Every user has different sets of keys for their directories but to minimize confusion we use the same notation for every user in the snapshot.

Preface

This implementation of client.py can be summarized as a snapshot of the untrusted storage server as shown above. The current depiction of the snapshot is that Alice owns two files which are named fn_1 and fn_2 . Alice shares fn_1 with Bob which he names it as fn'_1 . Bob also shares this exact file with Carol which she names it as fn''_1 . The reader can follow the design document and refer to the snapshot above for better interpretation of this system.

Design

The inspiration for this implementation comes from Design 1 of the Part 1 solutions. When a user first initiates his/her client, the client will generate two directories: <username>/dir and

<username>/sdir that are in the form of a Python dictionary and also two pairs of symmetric keys that are used to encrypt the directories and MAC them so that confidentiality and integrity for these directories are achieved. These keys are stored in <username>/dirkeys and <username>/sdirkeys and are encrypted using asymmetric encryption with the user's public key and the user can decrypt these using his/her private key.

The first directory <username>/dir is used to store a $filename : (id, k_1, k_2)$ mapping for EACH file that the user has access to regardless of personally owning the file or being shared by someone else. When we are shared a file or upload a file, we randomly generate a 16 byte id for it and it can then be used to uniquely refer to that file and it does not reveal any information about the filename. k_1 and k_2 here are randomly generated symmetric keys for performing encryption and MAC for this specific file. Every time we finish updating this directory, we encrypt-and-MAC it again using the directory's keys and store it back to the storage server.

Then, to store the exact value of the file, we store a mapping in the form of <username>/files/id : [D] $IV||E_{k_1}(SHA256(k_1)||value)||MAC_{k_2}(.)$ in the storage server. The [D] tag is used to mark that this string is data of a file's content. The value of this file is encrypted using symmetric encryption with its encryption key and a randomly generated IV and also we prepend a cryptographic hash of the encryption key in front of the value to test for swap attacks . We prepend the IV and also include a MAC for it using the MAC key that we can use to perform integrity check. This design makes sure that we achieve confidentiality and integrity for the contents of the file.

The second directory <username>/sdir is used to store sharing information, i.e. to whom and which file a user shares with in the form of a $username||filename : id$ mapping. When a user wants to share a file to another user, he/she will randomly generate another id that is different from the id that is used to associate the file in his/her own directory and stores this information in this second directory for each instance of a file share. We will share the reasoning for doing this in a bit.

The client will then generate a sharename in the form of <user1>/sharewith/<user2>/diff_id. It will then store a mapping of (sharename : [P] <username>/files/id_of_real_file) in the storage server. The [P] tag is used to mark that this string is used as a pointer to "follow" the sharing chain to get to the correct file stored by the owner of the file. This makes it so that whoever has this sharename will be able to access the encrypted contents of this file. Then, this sharename is sent along with the encryption and MAC keys of this file to the user that is being shared the file. Also note that every time after we finish updating this sharing directory, we encrypt-and-MAC it using it's keys and store it back to the storage server.

When the shared user agrees to accept the shared file and receives these 3 items, his/her client will treat it as if it were uploading a new file and first create a new random id and store it along with the given encryption and MAC keys into his/her first directory as a mapping from his/her desired new filename for the shared file. Then, the shared user's client will store a (<user2>/files/new_id : [P] sharename) mapping in the storage server. When the shared user wants to access this shared file, the client will check for the id associated with the shared file in his first directory

and access the value of <user>/files/id in the storage server by "resolving" the [P] tags until it reaches a [D] tag which means it has located the shared file.

When an owner of a file wants to revoke a user of a specific file, the client will simply just look up from the owner's sharing directory the id that was used in the sharename of this instance of sharing. Then, the client will simply remove from the storage server the mapping with the key <owner>/shareswith/<revoked_user>/id. This action essentially blocks the sharing chain from reaching the final destination of the file which means the revoked user can no longer update, download, or do anything to the file. Using this revoking mechanism, an owner of a file will not only be able to revoke the direct sharing user but also all other users that this direct sharing user shared with. In the case of double sharing, i.e. a user C is shared a file by the owner A and a shared user of the file B, it depends on which sharing invite C last chooses to accept. A special consequence of this is that when C accepts A's invitation and then accepts B's invitation and uses the same filename for the shared file, then C will lose access if B ends up being revoked by A. A way to bypass this is to accept both invitations and name them differently.

The previous implementation of generating a different id as mentioned before when generating a sharename is to address the issue that when a user is revoked access of a specific file by the owner, the revoked user cannot use the sharename to trace back to the final destination of the shared file and regain access, i.e. calling receive_share() with a previous sharename. Once the sharename is removed from the storage server, the information to trace back to the final destination of the shared file will be lost forever.

In this implementation, all $E_k(.)$'s refer to an AES-CBC encryption with AES key k and with a fresh random 16-byte IV. All $MAC_k(.)$'s use SHA256-HMAC under key k.

Security Analysis

The system as designed above can protect against the following attacks:

(a) Attack 1

An attacker might want to remove his sharing instance from the sharing directory so that the owner of the file cannot revoke his access to the file, i.e. the attacker removes the mapping of <attacker>$||fn : id$ so that the owner will not be able to get the id to remove the corresponding sharename. This is prevented since we encrypt-and-MAC the sharing directory with different symmetric keys and these keys are also encrypt-and-MAC'd using the owner's public key so the client will know if the sharing directory has been tampered with and raise an IntegrityError.

(b) Attack 2

An attacker might want to spoof a false sharename mapping, i.e. the attacker stores a mapping of alice/sharewith/<attacker>/spoofed_id : [P] alice/files/id even though Alice never intended to share it with him, so that he can ride the sharing chain to access the Alice's file associated with id. This is prevented because even though the attacker can indeed ride the sharing chain to the final destination of the file, he can only get the encrypted packet of it. Since Alice never actively

shared the file with the attacker, the attacker will not have possession of the keys that are required to decrypt the packet and thus the contents of the file are safe.

(c) Attack 3
An attacker might want to change the mapping of alice/sharewith/bob/00000 : [P] alice/files/10101 to [P] alice/files/10111 so that the bob is shared the wrong file without knowing it happened. This is prevented by using file specific keys so that different files can only be decrypted by their own sets of keys even though they are from the same owner. So if the shared user cannot decrypt the file he obtained by following the sharing chain and using the keys that he was given, that means that the files have been tampered with.

(d) Attack 4
An attacker might want to share a file that he is not shared with to another user. This is prevented because the attacker doesn't know the exact file location of the file so he cannot link the sharename he generated to the exact location of the file. Even if the attacker somehow knows this information, he still doesn't have the decryption keys for this file to share with the user he intends to share the file with.

(e) Attack 5
An attacker might want to regain access to a file that was previously shared to him by calling the receive_share() method on an old message that the owner of the file previously shared to him. This will not work because when the owner revoked the attacker, the client will have removed the mapping of his sharename to the exact location of the file. Since the files are represented by random id's, the attacker will not be able to identify which file is the one that he was previously shared with.

(f) Attack 6
An attacker might want to update a file that he is not shared with. This is prevented because in order to update the file, the attacker must have both the correct sharename to ride the sharing chain to the correct location of the file and also the encryption keys specific to that file. Even if the attacker somehow can obtain the exact location of the file, he still cannot update the file without other users with access to the file knowing that it has been tampered with since their decryption keys will not work on the attacker's file.

(g) Attack 7
An attacker might want to decrypt a [D] tagged encrypted file he obtained by observing the keys that users with access to this file will have. This is not possible since all the encryption and MAC keys for all the files the user has access to is encrypted in his/her directory and can be found nowhere else. This means that unless the attacker has the symmetric key to any of the users directories, he will not be able to obtain the decryption keys of the file that he wants to decrypt.