FILIP MIRDITA

CS161-ADP

NO PARTNERS

Project 2-2

DESIGN

This design uses three new keys all generated with get\_random-bytes(). A symmetric encryption key d\_k, and symmetric mac key d\_m, and a symmetric key k\_n which encrypts file names for confidentiality. The keys are generated at different times, one when the client is made, two for each document made.

DATA ON SERVER:

* User/key\_dir – has k\_n , encrypted with user’s public key
* User/file\_uid – has file encrypted with d\_k, and a HMAC using key d\_m of the encrypted file and the file\_uid
* User/file\_uid/key\_dir – has a key dictionary
  + Keys are user names
  + Values are lists with two items
    - d\_k and d\_m encrypted with the key user’s public key
    - list of other users the user has shared the file with

DESCRIPTION OF KEYS

AND THE SECURITY THEY PROVIDE

d\_k: this key is the symmetric “document key” which is used to encrypt and decrypt the file.

d\_m: this is the symmetric document mac key, which is used to compute the HMAC of the encryption of the file and the uid of the file each time it is uploaded and downloaded, to provide integrity.

**Client key, k\_n** is generated when a client is made. There is one for each client. This key is used to provide randomness in the SHA256-HMAC function, which is used to keep the name of a file secret. k\_n for a particular user is stored in user/key\_dir. It is signed, and verified each time it is accessed to prove it wasn’t tampered with.

**Document keys, d\_k and d\_m**: these keys are generated when a file is created. There is a pair of these for each document. The document keys are only stored as encrypted keys on the server; the keys are encrypted with the public key of each user to which the file is shared. The keys for a particular file can be found with that file’s unique ID (uid): uid/key\_dir. The data stored here is a dictionary, where the keys are usernames, and the values are lists. The first value of each list are the keys encrypted with the key user’s public key. The second value of each list is the list of user with which the key user has shared that file.

PROPERTIES

**Upload(file\_name, value):**

“Epilogue”: Retrieve the client’s k\_n, compute HMAC\_k\_n(file\_name) [encrypt file\_name using HMAC with cryptographically safe SHA256 hasing function, and key k\_n for ultimate security]. Then resolve the uid using user/HMAC\_k\_n(file\_name) , which may point to a different uid if the file was shared with the user. Next retrieve d\_k and d\_m for the specific file by indexing uid/key\_dir for the user uploading.

The user will first verify their keys have not been tampered with by verifying an RSA signature on the encrypted keys, then decrypt the keys using their ElGamal private key.

“Prologue”: if the key is not verified, check to see if it was the original owner’s signature on the keys. If it was, the client knows the key was updated due to a revoke, and the client can accept the key as legitimate, decrypt, sign it with their own key, and store it in the file’s key\_dir once more.

With the document’s keys, they can encrypt the value they are uploading with d\_k and using AES encryption in CTR mode, and compute the HMAC of the encrypted value and the uid with d\_m. This is stored under the uid.

**Download(file\_name):**

Same epilogue as before: retrieve and decrypt k\_n, compute and find the location of the file, and retrieve the document’s secret keys, this time only after verifying the file exists (if it doesn’t, return None). Same Prologue. This time the user need only verify the HMAC of the encrypted file with the d\_m, then decrypt the file with d\_k to obtain the value.

**Share(user, file\_name):**

Same epilogue as upload. Encrypt d\_k and d\_m with user’s public keys. The sharing client then signs it and and stores the signature and encrypted keys under the file’s key\_dir using dictionary key “user.” Lastly, add user to the list of clients the user has shared this file with.

Finally, return the uid of the file as the message.

**Receive\_share( from\_user, new\_file\_name, message):**

Same epilogue as before, only there’s a variation in retrieving the keys for the file. Since the keys are signed by from\_user, verify them with their public key. Then, sign the keys with the current client and store the encrypted keys and signature on the server under the client’s name.

Lastly, create a way for the client to access the original file using a pointer to it that is accessible via the client’s new\_file\_name, which is encrypted using the client’s k\_n.

**Revoke(user, file\_name):**

Use the same epilogue as upload. First, verify the user revoking privileges is the original owner of the file, by checking the file’s key\_dir. The original file owner was stored in the key\_dir under ‘ORIGNAL\_OWNER’ , which contains a pair of the original owner’s username and a signature of that username by the original owner. Verify this pair before accepting the owner’s username as legitimate.

Delete the user from the client’s list of user’s the client has shared the file with, then recursively go through the file’s key\_dir, revoking all user’s they’ve shared the file with, etc.

For all remaining clients in the key\_dir, update the keys. Generate new d\_k and d\_m and encrypt them with each client’s public key, then store each value respectively.

Finally encrypt and store the file with the new keys.

ATTACKS

**Attack 1:** If an attacker tried to replace ‘ORIGINAL USER’ with their own username, they would be able to revoke the priveleges of anyone, including the actual original user. To prevent this, my design signs the original user when a file is created.

**Attack 2:** A malicious attacker might send a share message claiming to be from someone else, someone a client trusts. To protect against this attack, the client verifies that the share message and keys were actually sent by the person who claims to have sent them before trying to use the potentially malicious keys.

**Attack 3:** An attacker might try to flip the users in the key\_Dir to share with someone it’s not actually shared with.