Design Report: PA1

CS628

Aditya Rohan(160053)
Paramansh Singh(160474)

InitUser: InitUser function initiates the User Struct with location of list of file inodes (explained in store file), AES and HMAC keys for that list (which are generated randomly), and rsa private-public key pair. The location for this user struct is Argon2Key hash of password with username as its salt. This location can thus can only be determined if the username and corresponding password are known.
The AES key for encryption/decryption, HMAC key for integrity check, of User struct are also computed using Password and two randomly generated keys. These keys are then appended with the encrypted user struct, and then the stored at the computed location along with their checksum.

GetUser: Using the provided username, password we get the data stored at the user struct location. Then using the stored random key and password, we get the HMAC and check integrity of stored data. Then, we decrypt the data to get User Struct. Then, we also check if the private key obtained from the user struct matches with the Public key stored in the keystore.

StoreFile: We maintain a ~~dictionary~~ list of inodes (location of this list is stored in user struct). Each inode consists of hashed filename, location, keys. The location of the file is generated randomly when the user first calls store file. This dictionary is itself encrypted and HMAC is used. The AES and HMAC keys used are stored in the User struct.
Each inode further consists of a list of file blocks. Each file-block contains location of data corresponding to each append. Along with this it contains the AES and HMAC keys used to encrypt/decrypt the data stored and check its integrity. The AES and HMAC keys for file-block list are stored in the file-inode of each file.
Removed the RSA sign part.
Every time a file is uploaded, we generate new location, HMAC and AES keys randomly for the first file block, as well for the data. The file-data is encrypted using AES key for file-data and the checksum computed using HMAC is also appended and stored at the randomly generated location. Then, this location, keys are stored in file-bock structure, which is encrypted and it's checksum is computed along with it. These are then stored at a randomly generated location. Finally, this location and keys are stored in the file-inode structure. This file-inode is then appended to the list of inodes (The corresponding checksum is also updated).

LoadFile: First, we get the list of inodes and search for the file-name in the list of file-inodes. Then, we go the the file-block list and get the location for each data-block. Finally we get the required data from each of the data block. Note that, each of the above mentioned data structure is decrypted and it's integrity is also checked.

AppendFile: Append file works on the same lines as store file, but instead of creating a new file-inode, we find the file-inode corresponding to the current filename. Then we create a new file-block and append it to the list of file-blocks corresponding to the given file (and update the checksum of file-block list). This new file-block contains the location of the appended data-block, and it's AES and HMAC keys.

ShareFile: When the owner or some other collaborator decides to share the file, he encrypts the file's inode struct using the receiver's public key and signs the encrypted message with his own private key and appends the signature to the message.

ReceiveFile: The receiver gets the message string and the sender's name. The last 256 bytes of the message is the sign. To verify the sign he takes the sender's public key from the key store. Then he decrypts the message in chunks of 256 bytes using his own private key. After decryption he has the file inode struct for that file, he replaces the filename hash with the hash of his own filename. Then he adds this file inode struct to his master inode struct or overwrites another file inode with same filename.

RevokeFile: As suggested in the review of our design document, we have implemented this by deleting the old file inode, and storing the old file blocks at a new location.

Testing:

- Data Integrity Check: This checks whether LoadFile gives proper error messages.
  The adversary makes changes in the file data -> Error: checksum failed.
  The adversary replaces <data, checksum> with <data2, checksum2>
  Again checksum error since different HMAC keys as no unauthorised person has access to the file-id.
- File Sharing: Integrity - The adversary modifies the data shared. The signature of the sender's private key results in an error if some contents have been modified.
  Confidentiality - Signed using receiver's public key and hence no one else can decrypt.