CS 161 Fall 2017 Project 2 Design Document

William Smith, SID 26004555, Gera Groshev, SID 26741303

1 Design of System

1. Simple Upload/Download

   User Struct: The User struct contains the following: username, password, a private RSA key, a public RSA key, and two maps – OwnedFiles and ShareFiles – both of which map a Filename (local to the user, and not shared) to a FileMetadata struct. The FileMetadata struct contains private metadata information pertaining to that file, including FileID (a random UUID generated during file storage), EncryptKey and MACKey (both random byte arrays of size 16), all generated during file storage. RSA keys are generated in InitUser, and all other User struct data is populated in InitUser.

   InitUser: To save the user, we generate 2 keys, K_e and K_a using PBKDF2 on the user's password, using SHA256(username) as salt to prevent dictionary attacks (username is globally unique). These serve as the encrypt and MAC keys for the User struct. We then encrypt and HMAC the User struct and save it in the datastore at location "logins/userID", where userID is computed as UUID(HMAC(username, K_a)).

   GetUser: To load a user, we regenerate the K_e and K_a from the username and password, then regenerate userID and load from the datastore, check the HMAC, and decrypt.

   SharedMetadata: In addition to the FileMetadata struct, which is private and saved within the User struct, we have a SharedMetaData struct, which contains the File size, the number of revisions to the file, an array of revision sizes (where the i'th revision size is the size of the i'th revision to the file), and a HMAC of all this data.

   StoreFile: To store a file named $filename$, we generate the FileMetadata (generate 3 random numbers for FileID, K'_e, and K'_a), store it in the user's owned files map at $filename$. We generate a MAC as HMAC(Ka', filedata || 0). The 0 corresponds to the revision number. MACing this way allows us to prevent revision permutation attacks, wherein the malicous server might permute the data of 2 revisions, as well as their metadata. Now, the order must be maintained for the MAC to be correct. We then encrypt and HMAC the file data with K'_e and K'_a, and store it as the first revision in the datastore at location "files/FileID", and then create new revision metadata for the file, and store it at "meta/FileID".

   LoadFile: To load a file, we simply get the private FileMetadata from the user struct, lookup the random FileID, and then Load the file and shared metadata from the datastore at "files/FileID" and "meta/FileID", respectively. We verify the metadata MAC, and then use the metadata to iterate through each revision in the file, each time, verifying the HMAC(encrypted data || revision number), and then decrypting and concatenating the data. We return the aggregated, decrypted data.

   AppendFile: To append to a file, we simply load the file and metadata as discussed earlier, check the metadata MAC, and then encrypt and HMAC(K'_a, encrypted data || revision number) and re-upload the file. The revision metadata is updated with the length of the new revision, the new total length, and increment the number of revisions.

2. Sharing

   ShareFile: To share a file, we retrieve the private metadata of the file from the OwnedFiles or SharedFiles private dictionary within the sender's User struct. Then retrieve the receiver's Public RSA key from the KeyStore and use this to encrypt the metadata. We use the sender's private key to RSASign the encrypted metadata and return encrypted message and signature.

   ReceiveFile: To receive a shared file under the name f_r, we simply get the sender's public key from the KeyStore, verify the RSA signature, then use the receiver's private RSA key (in the receiver User struct) to decrypt the data. We then save this data in the SharedFiles map in the receiver's User struct under the name f_r. Note that the receiver has no kowledge of the sender's filename, as this is not contained in the private metadata contained in the message.

3. Revocation

   RevokeFile: To revoke access to a shared file named f, we first check that f is contained in the OwnedFiles map of the users's file struct. This prevents non-owners from revoking access to a shared file. We then call LoadFile to get the plaintext version of the file. We then generate a random key k_t, encrypt the plaintext with it, and save in the data store at the original location. We **DO NOT** save k_t anywhere. This essentially makes the data at the original location unrecoverable. We then call StoreFile() on the original plaintext, which will handle re-encrypting, re-MAC'ing, and relocating the file, along with regenerating and re-storing all the associated metadata.

2 Analysis of Security

1. **Data Swap Attack**: The malicious server does not know the contents of the file, nor the name of the file, but can see that file/r corresponds to meta/r, and hence perform a swap of the data at file/r1 with the data at file/r2, and to avoid detection, swap the data at meta/r1 with the data at meta/r2. This exploit will fail, however, because we have a unique MAC key corresponding to each individual file which MAC's the file and the associated metadata. This MAC is stored in the user's revisionMetaData struct within the User struct. Since this has not been swapped, the MAC's will fail.

2. **Filename Dictionary Attack**: The malicious server attempts to identify the file by keying on known filenames, or precomputed hash values of known filenames. This will fail, because the filename is generated as a random UUID, which is saved as file metadata within the User struct, and only shared via RSA encryption to shared users. Therefore, to break the filename, the server must break the encryption on the User struct.

3. **ShareFile Man in the Middle Attack**: Some external attacker mounts a man in the middle attack on the communication channel between two users sharing a message. If the attacker tries to eavesdrop, it will fail, due to the data being encrypted by RSA. To decrypt, the attacker would need to know the recipient's private key, which would require breaking the block cipher encryption of his User struct. If the attacker tries to tamper with the message it will fail, due to the data containing an RSA Signature. In order to tamper with it and resign, the attacker would need to know the sender's private key, which would require breaking the block cipher encryption of her User struct.