CS 161 Fall 2017 Project 2 Design Document

Nicholas Kriss, SID 26141698

1 Design of System

1. Simple Upload/Download

   All encryptions are with CFB and all MACs with HMAC, all datastore keys are 32 bytes, encryption keys 16 bytes, hash keys 32 bytes, and IV's 16 bytes.

   For every user, we generate one unique key each for encryption (userEKey), hashing (userHKey), and the location of the user struct (userLKey) by getting 80 bytes from PBKDF2(username + "a" + password), plus a random RSA key. We store the public RSA key in the keystore, the username, password, and RSA private key in the user struct, marshal the struct, encrypt it with userEKey and a random IV, generate an HMAC of IV and the struct with userHKey, and store a combined HMAC, IV, and encrypted, marshaled struct in the datastore at location userLKey. To get a user struct, we generate the user keys, get the data at userLKey, confirm that the stored data is long enough to have a MAC and IV, confirm that the MAC is correct for IV appended to the ciphertext, decode the ciphertext with userEKey and IV, unmarshal the ciphertext and store the unmarshaled data in a user struct. If any of these steps produces an error, we return that error and nil for the struct. We can have multiple users with the same username or with the same password, but not both. I tested for deletion and corruption of the encrypted user info, and to make sure that we only return userdata given a correct username and password.

   When we store a file, we generate the user keys and 4 random file keys for encryption (fileEKey), hashing (fileHKey), the location of the file metadata (metaLKey), and the location of the file (fileLKey). We append fileEKey, fileHKey, and metaLKey as fileKeys, encrypt it with userEKey and a random IV, MAC IV and fileKeys with userHKey, and store the HMAC, IV, and the encrypted fileKeys at loc, where loc is the hash of the password appended to the filename with userHKey. We then encrypt the file data with fileEKey and a random IV, and MAC IV and the encrypted data with fileHKey. We store the IV and encrypted data at fileLKey, and append fileLKey to the HMAC as our metadata, encrypt and MAC the metadata with fileEKey and fileHKey, and store the resulting HMAC, IV and encrypted metadata at metaLKey. If we call StoreFile with an existing file name, that file is overwritten with new keys, metadata, and data.

   When we want to append data, we find the user keys and loc, and get the file keys from loc by decrypting, confirming the MAC, and slicing appropriately so that the keys have the proper size. We then use metaLKey to get the encrypted metadata, which we decrypt and confirm the MAC using fileEKey and fileHKey. We generate a new random location, encrypt the new data with fileEKey and a random IV, and store the IV and the encrypted data at the new location. We find the last HMAC in the metadata and find a new HMAC of this previous HMAC, the new IV, and the new encrypted data. We append the new HMAC and the new location to the metadata, and re-encrypt and re-MAC the metadata with a new random

IV. I tested deletion and corruption of the encrypted keys, metadata, and data, calling on nonexistent files, and the inability of users to access other users' files.

To load a file, we acquire the file keys and metadata the same as for an append. Then, for each HMAC-location pair in metadata, we get the IV and encrypted data from the location, confirm the HMAC is equal to the MAC of the previous HMAC (initially nil), the IV, and the encrypted data, decrypt the data, and append it to the previously decoded data (initially nil). Because of our implementation, we will only ever return nil if the stored data is nil or if the file keys are deleted from the datastore. If the metadata or data are corrupted, deleted, or set to nil (and were not previously nil), or if the file keys are corrupted or set to nil, we will return an error. I tested deletion and corruption of the encrypted keys, metadata, and data, calling on nonexistent files, and the inability of users to access other users' files.

2. Sharing

   We generate the user keys and loc and get the file keys the same as in appending and loading data. We then generate a random 32-byte number to be used as a seed for PBKDF2 to generate temporary encryption, hash, and location keys. We RSA encrypt the seed using the receiver's public key, and generate an RSA signature on the encrypted seed using our private key. We encrypt, MAC, and store the file keys appended to the signature using the temporary keys, and send the receiver the encrypted seed. The receiver decrypts the seed using their private key, generates the temporary keys, and gets the encrypted file keys appended to the signature. After confirming the length of the data and the MAC, the receiver then confirms the signature on the seed using the sender's public key, and if it is valid, they decrypt the file keys and re-encrypt, re-hash and re-store them using the user keys for a chosen filename. If a user calls StoreFile on a shared filename, that user will get new keys, metadata and data and be unable to access the previous shared file. Anyone else with access to the file will still be able to load and append to the original shared file. For ShareFile, I tested for nonexistent receiver, nonexistent file and corrupted key, metadata, and file data. For ReceiveFile, I tested for nonexistent sender, wrong sender, and invalid message ID. I also tested that if A shares with B and B shares with C, then A, B, and C are operating on the same file.

3. Revocation

   We follow the same procedure as in LoadFile, but after decrypting the data at a location from the metadata, we set the datastore at that location to nil. Once we are done with the metadata locations, we also set the datastore at the location of the metadata to nil. We then generate a new set of random file keys and a new metadata key, and follow the procedure for initially storing a file. This also condenses every append into a single data array, stored at a singled location. Any user can revoke a file even if they did not create the file, and doing so will revoke the file from any other user, including possibly the original creator. If someone tries to load a file previously shared with them and then revoked, they will always receive an error. I tested revoke on nonexistent file names, and that if A shares with B, B shares with C, and A revokes access to the shared file, then neither B nor C can access it and receive an error if they try.

   If at any step of any function we find that an HMAC is not the expected value, we return nil and an error. If the datastore returns ok == false when we try to find file keys at loc, we return nil and no error. If any other call to datastore returns ok == false, we return nil and an error or just an error, depending on the function. If any value from the datastore does not have proper length, we return nil and an error or just an error.

2 Analysis of Security

1. An attacker Mallory could pretend to be a user Alice and try to share a file with a victim Bob. To accomplish this, Mallory would have to send a message to Bob that causes Bob to receive a file he thinks is from Alice but was actually created by Mallory. This is prevented by using RSA signatures. When Bob receives a message ID, he finds in the datastore a signature for that message. Bob can use Alice's public key to confirm whether or not Alice sent the original message. Mallory cannot fake the signature, since Alice generates it using her private key. Also, Mallory cannot determine the seed since it is RSA encoded, so he cannot know the temporary keys Bob will generate, and thus cannot properly encode any malicious data, or have his data pass any MAC checks.

2. A malicious server could alter the data stored under a particular filename, replacing the originally stored data. There are many stages of the implementation that prevent this. First, the server does not know what data corresponds to what filename, since we create the store location randomly, encrypt the store location with a random key, and encrypt the random key with a secret key derived from the password. If the server somehow learns what the location of the data for a particular filename, they will not be able to overcame the MAC. The MAC uses a random key, which is encrypted with the user's secret hash key. So the server, given an IV and some ciphertext, cannot alter either and obtain a valid MAC. So when the user checks their file, they will find that the MAC is no longer valid, and fail to return the corrupted data.

3. A malicious server could swap the locations of two appends to a file. Then, when a user tries to load a file, they would have two segments of the file swapped. The implementation prevents this by taking a MAC of both the data and the previous MAC. Say we have a file ”a”, with MAC macA, and we append ”b” and ”c”. When we append ”b”, we take the MAC of macA and ”b” (macB), and store it in front of the location of the ”b” append. When we append ”c”, we take the MAC of macB and ”c” (macC), and store it in front of the location of the ”c” append. If the datastore somehow swap ”b” and ”c”, the user will check whether the MAC of macA and ”c” is equal to macB, which will fail and thus prevent the attack. The datastore cannot swap the MACs, since the metadata is encrypted and there is a MAC on the metadata which uses a secret hash key. Thus the datastore cannot change the metadata without causing the MAC check to fail.