CS161 project2 final design

Section 1: System design

```
struct User { //stored in local memory
  Username: String, Password: String, usersk: []byte
}
struct Userinfo {/*stored in datastore*/
  hashedUsername: []byte, //hash on username
  hashedPassword: []byte, //hash on password with salt
  Salt: []byte,
  Hmac_key: []byte, //encrpted private key of hmac
  PKE_pair: keyPair, // public key encryption
  DS_pair: keyPair, // digital signature
  AccessibleFiles: map[userlib.UUID]userlib.UUID //key:filename, value:acesstoken
}
struct UserFile{/*stored in datastore*/
  ID: uuid, // generated by uuid
  LastSigner:[]byte, //last user to sign
  Filename:[]byte, Content:[]byte, Owner:[]byte,
  Root:ShareNode //tree like structure to store users that acces s this file
}
Invitation struct {/*stored in datestore*/
    Value  []byte, Sender []byte
}
/*stored in local memory */
usersk = hash(username + "-" + password, salt);
filesk = hash(RandomBytes + usersk + filename);
```

Summary:
We choose user symmetric secret key (usersk) based on users's input of their username and password. We need to store the hash version of username and password in the datastore in order to do the login verification. Since hashing a password with a random salt is slow, we can prevent dictionary attack. Then, we choose a random 16 bytes number as a hamc secret key of a user. Since we need to store the hmac secret key in the datastore, we must encrypt it for confidentiality. Then, we can use the usersk to make each field of the Userinfo to be confidential and hmac to make the Userinfo integrity.
We choose a symmetric secret key (filesk) for a file with the same user based on some random bytes, usersk, and filename. Then, each file for the same user can have its own key to do the encryption. Sharing file invitation is easy because we just need to share the filesk to other users. Other users knowing the filesk have no way to get the usersk because hash is oneway. Revoking a file became doable because we just need to update the random bytes. In terms of integrity, we can use user's digital signature secret key to sign the file so that other users who can access the file can verify the file.

How do we store user information to support multiple user sessions?

- We separate User and Userinfo where User only live in local memory and Userinfo is stored in datastore with the key as uuid of hashUsername.
- User struct acts as a key to Userinfo so that we can get the Userinfo from datastore when it's needed. This allows same user with different session logins in at the same time can get the most updated information because we make Userinfo always up to date in datestore.

How is a file stored on the server?

- We store files (UserFile) in datastore with key as uuid of hashed filename.
- If the file is not in the accessible map, we create a filesk for the file. Otherwise, we decrypt the invitation from datastore with the key as acesstoken in accessible map to filesk.
- If we are creating a new file, We initialize share node and store it in the root of the user file. Save the signer name(current user name) as LastSinger.
- Then we marshal the instance and use symmetric encryption to encrypt the marshaled use file instance. We use the user's digital signatures private key to create a signature for the encrypted of the marshaled of the instance user file.
- We store and marshal the encrypted of the marshaled of the instance user file and the signature we just created as a marshaled DataPair.

- We store the marshaled DataPair in the data store with generated uuid from first 16 bytes of the hashed filesk.

How does a file get shared with another user?

- We will generate the signature that from owner's digital signatures private key sign the marshaled invitation with invite and owner's name, and store it with the marshaled invitation in the data store when we first time store the file to the database.
- We get the invitation when we share to another user.
- We generate the access token by get the recipient public key and use PKEEnc to encrypt it.
- Append a new share node with the recipient name under the current user's share node in the user file instance.
- Then, we send the generated access token to the another user.
- The recipient takes the access token and stores it in recipient's accessibleFiles with the fileuuid as the key.

What is the process of revoking a user's access to a file?

- We just need to update the filesk and encrypt the file again with the new filesk.
- We store the encrypted file in the datastore with the owner signature, and remove the old file from datastore.
- We remove the ShareNode that is corresponding to the revoked user in the ShareNode tree of the file, so the children of the removed ShareNode (shared users by the revoked user) are also be removed.
- We use DFS to traverse the ShareNode tree to update the accessToken for the users who are still have the right to access the file.
- Users are revoked have no way to decrypted the file with the old invitation(old encrypted filesk) and Users are not revoked still can use the new filesk to decrypt the file.

How does your design support efficient file append?

- Since we only have one file struct for one file stored in datastore, we only need to update the file.
- We just need to decrypt the invitation to filesk and use it to decrypt the file, and user last signer to verify the integrity of the file with DS.
- We append the new file content onto the old file content and store it back to datastore after the new encryption and signature with the same uuid datastore key.

IND-CPA on file and accssToken:
file:

- We use symmetric encryption (AES-CBC mode) on the file which will be encrypted randomly each time.

accessToken:

- We use 16 byte random bytes padding on the filesk. Then, we use RSA public key encryption on the padded filesk to be an invitation. The accessToken will be first 16 bytes of the invitation. Since random bytes are appended on the filesk before the PKE, the result of the encryption will be random even though PKE is deterministic.

Section 2: Security analysis

Attack1:

We can prevent any dictionary attack on password as long as the attacker does not know the real password. Since we initialize the user with hashed username and hashed password with salt. Even if the attacker knows the username with dictionary attack, it still would be difficult for an attacker to find out the hashed password since the property of the hash function (collision-resistant, one-way, and second preimage resistant), and Argon2 hashing is slow and randomizes the result with salt. Dictionary attacks need to take O(d * n * hash time) to break the password. This prevents all malicious intentions from brute force break the user's password to login to our system.

Attack2:

After we initialize user A, MITM tries to learn and temper the struct user in the datastore. MITM only can learn salt because the username and password are hashed and the private key of PKE, DS, HMAC are confidential and IND-CPA. we encrypt the private keys with CBC with key = hash(username + password, salt). MITM cannot temper the struct user because private keys are HMAC. User A can verify the integrity of the struct.

Attack3:

After User A share a token of file1 to User B, MITM tries to learn the token and temper the it. In order to prevent MITM, our design makes the token confidential, integrity, and IND-CPA. We encrypt to token with user A super secret key f_sk which is only store in local memory and unique. Even MITM somehow get the username of A and filename of file1, MITM still cannot learn anything because f_sk is xor with the uuid of file1. Each user's file has a unique uuid. This is same for the encrypted file_node because we also use f_sk to encrypt the file_node.

Attack4:

After user A revoked user B access to the file1, user B want to continue to use the same token to read and update the file1 that user A holds. Our design prevent this attack because user A will delete the file1 in the datastore and make a copy of file1 with new uuid, store the new file with new encryption (f_sk = hash(h_sk xor new uuid)), and board-casts the new token to the users that still has right to access to the file1 during revoke. If user B uses the old token to get the file from datastore, user B will fail because the key is different. User B only can hold a copy of old file_node and cannot see any update of the file1 that user A currently holds.