CS161 project2 design draft

Section 1: System design

```
/*user data structure that is stored in datastore*/
struct user {
  hash(username): String,
  hash(password, salt): String,
  Salt: String,
  Hmac_key: String, // the private key for hmac, encrpted by u
ser secret key
  //~~~~below are encrypted by user secret key and hmac with h
amc_key
  PKE_pair: (String, String), // public key encryption
  DS_pair: (String, String), // digital signature
  Own_file_list: String, // list of filename
  Share_file_map: (String, String) // (filename, tokens)
  share_user_map: (String, list) // (filename, list of users)
}



/*file data structure that is stored in datastore*/
struct file_node{
  ID: String, // generated by uuid
  Filename: String,
  Content: String, //
  owner_username: String, //
  length_of_file: (Integer), //
}

/*what we store in the datastore*/
```

```
(key, struct user)
(key, (encrypted(file_node), sign(encrypted(file_node))))

/*this is the user super secret key that is only stored in loc
al memory */
String h_sk = hash(username + password, salt);
String f_sk = hash(h_sk xor file.uuid);
```

Summarize:

- InitUser:
  - we initialize the user by hashing the username and hashing the password with salt if the hashed username is not the datastore.
  - also initialize HMAC private key by **SymEnc**(key=h_sk, plaintext=random generator bit), public key encryption (PKE_pair) by **PKEKeyGen**, digital signature (DS_pair) by **DSKeyGen**.
  - we store the PKE_pair, DS_pair, Own_file_list, and Share_file_map as (**SymEnc**(key=h_sk, plaintext=marshal(object)), hmac(object)).
  - we store hash(username) as key, struct user as value in datastore, and send the public keys to keystore.
- GetUser:
  - use the hash(username) to get user struct from datastore
  - if user struct exists, check if the hash(password, salt) matches.
- StoreFile: (How is a file stored on the server?)
  - we first initialize the file_node with the filename, content, and uuid.
  - the private key for file encryption is hash(h_sk xor uuid) = f_sk
  - the key for datastore as **SymEnc**(key=f_sk, filename). this will be unique because h_sk xor uuid is unique.
  - the value for datastore as **SymEnc**(key=f_sk, file_node, but we store the value as pair(value, sign(valye)) in the datastore.
  - **both key token and value are confidential, integrity, and IND-CPA by symmetric encryption (cbc) and digital signature.**
- LoadFile:
  - if it's owner, key is **SymEnc**(key=f_sk, filename)
  - if it's share, key is **SymEnc**(key=PKEDec(key=PKE_sk, tokens), filename)
  - get file_node with getdatatore(key), and verify it with digital signature and decrypt the file_node with either f_sk or PKEDec(key=PKE_sk, tokens).

- AppendFile:
  - We use the hashed filename to check if the file exists, if not, return error. We load the file_node which is similar to loadFile function.
  - We then append the data to file_node.content.
  - Efficiency: O(bytes of data) append + O(number of own files) check own file list + O(1) check share file map.
- ShareFile:
  - We check if the file exists in the own file list and share map, if not, return error.
  - if it's owner, tokens = **PKEEnc**(key=recipient's pk from keystore, f_sk).
  - if it's share, tokens = **PKEEnc**(key=recipient's pk from keystore, **PKEDec**(key=PKE_sk, tokens)).
  - Sign the tokens with the digital signature of the DS_sk, and return the pair.
  - **tokens is confidential and integrity by public key encryption and digital signature. if we want IND-CPA, we might need to update our private, public key of PKE and DS.**
- ReceiveFile:
  - We check if the file exists in the own file list and share map, if yes, return error.
  - We verify the tokens by verifying the digital signature with sender's DS_PK.
  - If the node is verified, and we verify the accessToken was sent by the given sender, we add it to shared file map by (hashed filename, tokens). Otherwise, return error.
- RevokeFile:
  - We check if the file is in owner file list, if not, return error. We load the encrypted node.
  - We delete the node in the datastore, and create a new file_node with a new uuid. Then, call storefile for the same filename and new file_node into the datastore just like the storefile. Then, remove the targetuser from share_user_map.
  - Then loop through share_user_map, share the new token to the users in share_user_map by calling shareFile.
  - We need to make sure users accept the tokens before doing other operations.

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.