DESIGN DOCUMENT
CS 628 ASSIGNMENT-1

NAME- NIRJJHAR ROY(18111409 MS,CSE)
PART-1

The following structures were used(only a logical structure is shown):-

1) *type user struct {*
Username // stores the username
password
Pvt_key_rsa
Owned_file_keys_map[file_name][encryption_keys]
Shared_file_info_map[file_name][(file_location,encryption key)]
}

Design choices for function **InitUser()**:-

1) We will take the username and password. We then calculate a key using username and password using the Argon2. A part of this generated key is denoted as "new_user_hash" and the rest of the part is denoted as "User_Data_store_key_symm"**.** *JUSTIFICATION*- This new_user_hash will serve as the key while storing the new user's record in datastore. The reason we use this key because we don' t want the malicious server to know the exact username.
2) We generate the private and public keys of RSA. Lets denote them as Pvt_key_rsa and Pub_key_rsa.
3) We populate all the entries in the struct "user" , encrypt it using CFB encryption using a symmetric encryption key. This encryption key is derived from Argon2 denoted by "User_Data_store_key_symm". Then we calculate the HMAC using the same key of the encrypted content(denoted by Hash_enc_data). Then we store the (new_user_hash,encrypted data+hmac) key value pair in the data store. *JUSTIFICATION*- So we are in no way storing the password in the malicious server. Also the plaintext data is also not stored(since it is encrypted) and the malicious server won't be able to read the contents. The hmac will help to check integrity.

4) We Then we store (Username, Pub_key_rsa) key value pair in the keystore. *JUSTIFICATION*- Since keystore server is considered trusted we can store the username in its plaintext form.

BIG_KEY= ARGON2(username+password)
new_user_hash= BIG_KEY[low:high]// part of the key is used
User_Data_store_key_symm= BIG_KEY[low2:high2]//a part of the key is used
Hash_enc_data= HMAC(encrypted content)
Content to be stored in data store = key:value= the (new_user_hash:encrypted data+hmac)

Design choices for function **Getuser()**:-

1) We will take the username and password. We then re-calculate a key using username and password using the Argon2(the BIG KEY). We get the key of the "key,value" pair in datastore(denoted by user_hash). We use this user_hash to get the encrypted user data of this user from datastore. Then calculate the HMAC of the received encrypted data(Hash_enc_data) and the received HMAC(recvd_hash) and check if they are equal. If the HMACS match we decrypt and populate the user structure. After decrypting we populate the "user" structure. Decryption is done using the key User_Data_store_key_symm which is derived from the big key(a part of the bug key).

BIG_KEY= ARGON2(given username+password)
user_hash= BIG_KEY[low:high]
User_Data_store_key_symm= BIG_KEY[low2:high2]
Hash_enc_data= HMAC(received encrypted content excluding the received hash)

Design choices for function **Storefile()**:-

1) We generate a random key and lets denote is by "File_Data_store_key_symm". This will be used to encrypt the index_table of the file which we will store as an encrypted array/slice in data store. This slice/array will hold all the locations of the file chunks.

2) Then encrypt the content of the file with CFB encryption using "File_Data_store_key_symm"+"index_number"(basically a combination of the randomly generated key along with the index number of the block). Then we calculate the HMAC using the same key(Data_store_key_symm+"index_number") of the encrypted file content and this HMAC is denoted by "Hash_enc_data".So every different chunk will have a seperate encryption key and location.

3) Index_table[0] will be fill up with location of the first chunk. This location will be obtained by argon2(username+password+filename+index_number of the chunk).. For a new file index_number=0 as it is the first block.

4) Encrypt the index_table(an array/slice) with "File_Data_store_key_symm". Then calculate the HMAC of the encrypted index_table with the same "File_Data_store_key_symm".

5) Now the key of the "key value" pair of datastore is calculated as argon2(username+password+filename) and lets denote is as "File_index_hash". At this location the ("File_index_hash", encrypted_index_table_content +HMAC) will be stored as a key value pair. *JUSTIFICATION*- By doing this we make sure that every different file of every different user has a separate location and the random encryption key for every file makes sure that if the malicious server tries to repostion or swap file positions it will be detected during HMAC checking.

6) Now we have to store the encrypted file content along with HMAC. We store this at the location which is derived from argon2(file_name+username+password+index_number). This index value will be 0 here as we are adding the file's first chunk. So every different chunk will have a separate encryption key and location.

7) We also add this new file entry into the *Owned_file_keys_map* and update the corresponding user data structure. This map will basically store the file name and their corresponding random encryption keys. *JUSTIFICATION*- We do this so that every user has a list owned files and that user cant open any file not owned by him/her. Also using the *Owned_file_keys_map* helps the owner to share the file by sharing the symmetric encryption key to the desired people and when they want to revoke the access , the random key for the file will be changed i.e the file will be re-encrypted by the original owner with a new random key . Also file reposition attack will be detected as the encryption key and HMAC key for every file will be different.

Design choices for function **Loadfile()**:-

1) We will check if the file to be opened is in the list of *Owned_file_hashes_map* or *Shared_file_hashes_map.* If it is present in *Owned_file_keys_map* then
    a) Get the corresponding random encryption key for it from the map and lets denote it by "Owned_file_encryption_key".
    b) Calculate the file index_table location by argon2(username+password+filename) and lets denote it by "Owned_file_index_location"
    c) Fetch the encrypted index table content and HMAC from datastore(with "Owned_file_index_location").
    d) Calculate the HMAC for the received encrypted index_table(which excludes received HMAC) and lets denote it by "calculated_HMAC". The key will be same as the encryption key("Owned_file_encryption_key")

e) If calculated_HMAC== received_HMAC then decrypt the contents with "Owned_file_encryption_key" else return error.
f) Now we have the index_table having all entries for the different file chunks.
g) We traverse thorough each location, fetch the encrypted contents, check the HMAC and decrypted it with ("Owned_file_encryption_key"+index_number) where index_number denotes the file_chunk index. *JUSTIFICATION*- Encrypting each chunk with a different key prevents shuffling attacks.

2) if it is present in *Shared_file_info_map* then
a) Get the file location and the shared random encryption key from the corresponding location of the map. Lets denote them as "Shared_file_index_location" and "Shared_file_encryption_key".(The way the *Shared_file_info_map* map is populated is described in "sharefile()" function.)
b) Fetch the encrypted index_table content and HMAC from datastore(with "Shared_file_index_location").
c) Calculate the HMAC for the received encrypted index_table(which excludes received HMAC) and lets denote it by "calculated_HMAC". The key will be same as the encryption key("Shared_file_encryption_key")
d) If calculated_HMAC== received_HMAC then decrypt the contents with "Shared_file_encryption_key
e) Now we have the index_table having all entries for the different file chunks.
f) We traverse thorough each location, fetch the encrypted contents, check the HMAC and decrypted it with ("Owned_file_encryption_key"+index_number) where index_number denotes the file_chunk index. *JUSTIFICATION*- Encrypting each chunk with a different key prevents chunk swapping attacks.

Design choices for function **Appendfile()**:-

1) We will check if the file to be opened is in the list of *Owned_file_hashes_map* or *Shared_file_hashes_map*. If it is present in *Owned_file_keys_map* accordingly we will get the encryption key of the file index_table denoted by File_encryption_key".
2) We will now get the file index location either by doing argon2(username+password+filename) or from the *Shared_file_info_map*.
3) Then we bring in the encrypted index table content and then check for its HMAC and decrypt it with the encryption key.
4) Now we derive a new location for the newly added content. This location will be (username+password+filename+"index_number") denoted by "new_chunk_location". This is the index number of the newly added chunk which can be obtained getting the length of the index_table slice .
5) We place the encrypted file content to be appended along with the HMAC into the location "new_chunk_location".
6) We will add this new chunk location into the index_table array/slice. Rencrypt this index_table and store in its location in datastore. Here we are basically updating the index_table with a new entry for a new chunk.

PART-2

Structure used:-

```
Struct ShareFile{
        File_Encryption_key
        File_Index_location
}
```

Design choices for function **ShareFile()**:-

1) First we check if the file the user wants to share is in the *Owned_file_keys_map* or *Shared_file_info_map*. If it is *Owned_file_keys_map* then we populate the ShareFile struct as
File_Index_Location= Argon2(owner_user_name+owner_password+file_name to be shared)
File_Encrypted_key= the key for the corresponding file in the map *Owned_file_keys_map*.
2) If it is in the *Shared_file_info_map* then we populate the ShareFile struct as
File_Index_Location= location of the corresponding file index table obtained from the *Shared_file_info_map* by giving the file name.
File_Encrypted_key= the key for the corresponding file index obtained from the map *Shared_file_info_map* by giving the filename.
3) After populating the ShareFile struct we encrypt this struct using the public key of the recipient. Lets denote it as "File_Share_Encrypted_Content"
4) This encrypted content is now signed by the private key of the sender and lets denote is as "msg_sign".
5) Now we pack the encrypted content "File_Share_Encrypted_Content" and the "msg_sign" into a local structure, typecast it into a string and return it as "sharing".

Design choices for function **ReceiveFile()**:-

1) The receiver gets the "msgid".
2) Converts that string to a sequence of bytes and then unmarshals it to a temporary local structure having two fields "File_Share_Encrypted_Content" and the "msg_sign".
3) Then the receiver verifies that "File_Share_Encrypted_Content" and "msg_sign" with the public key of the sender. *JUSTIFICATION-This makes sure that the the owner is actually sending it and also the integrity of whatever the sender is sending.*
4) If the verification succeeds then the receiver decrypts the "File_Share_Encrypted_Content" with the private key of the receiver and unmarshall the content to a ShareFile struct.
5) Now it will have access to the ShareFile struct and we will get the "File_Index_Location" and "File_Encrypted_key".
6) With this we can get the file index_table location and the encrypted file_index_table from data store.
7) This encrypted content from datastore is now decrypted using "File_Encrypted_key".
8) We will also add one entry to *Shared_file_info_map* of this receiver. We will be adding the "filename_used_by_receiver":[("File_Index_Location"," File_Encrypted_key")] as the key:value pair of this map.

Design choices for function **RevokeFile()**:-

1) We get the location of the file for which we want to revoke. This can be obtained from argon2(filename+username+password) i.e the key of the key value pair in datastore.
2) We fetch the entire encrypted file content from the data store. Now we decrypt the contents using the existing symmetric key.
3) Now we call StoreFile(filename, decrypted_contents)- JUSTIFICATION- Basically we are re-encrypting the file contents with a new random key so that the previous users with whom the file was shared and who had the old encryption key , now won't be able to access the file as the encryption key is now changed.