Computer System Security(CS628A)
Design Document

Hariom(18111019)
Shikhar Barve(18111065)

Structures Used:

1.
```
User Struct{
    Username : Argon2(username,password)
    Pri_RSA  : CFB_encrypt(password,RSA_Private_key)
    ListFiles: List(All File Headers)
    HMAC     : this.HMAC(username||password)
}
```

Saved as Argon2(Username,Password)→User Struct

2.
```
File_Header Struct{
    File_Meta     : Hash(PrivOwner||MAC_key)
    Content_key   : RSAEncrypt(PubKeyOwner,SharedKey)
    MAC_key       : RSAEncrypt(PubKeyOwner,key_for_HMAC_of_File_Meta)
    HMAC          : this.HMAC(password||filename)
}
```

Saved as Hash(Username,filename)→File_Header struct

3.
```
File_Meta Struct{
    List_of_UUIDs : List_of_UUIDs of all appends
    HMAC          : this.HMAC(File_Header.MAC_key)
}
```

Saved as Hash(PrivOwner||File_Header.MAC_key) → File_Meta struct

4.
```
File_content Struct{
    Content   : CFB_encrypt(File_Header.Content_key,content)
}
```

Saved as File_Content_UUID → File_Content struct

5.
```
SharingRecord Struct{
    Content_key : RSA_encrypt(PubKey_Receiver,File_Header.Content_key)
    MAC_key     : RSA_encrypt(PubKey_Receiver,File_Header.MAC_key)
    File_Meta   : RSA_encrypt(PubKey_Receiver,File_Header.File_Meta)
    Signature   : RSA_sign(PriKey_Sender,this)
}
```

This will be passed as JSON string argument in ShareFile()

Maintaining Integrity and Preventing Swapping Attacks

Every structure will be stored in Datastore as JSON string file. Every time a structured file is fetched from Datastore and stored in struct object we will compare computed HMAC of all fields in that structure(except HMAC field) to HMAC field of that struct to ensure integrity. Every time a field is changed in a file HMAC will be recalculated. If at any point HMAC or RSA verification fails, we throw a custom ERROR. If a file is not found we throw a generic ERROR. We may choose to encrypt all the file structures.

To prevent swapping we compute HMAC of any structure file with function of either partial or full key at which it is stored in Datastore. For *File_Content* we always compare the UUID of fetched file and the key of Datastore for that file.

When StoreFile() is invoked we check whether a file with same name exists. If not then we create a new $File\_Header$, populate the fields. Generate a new $File\_Meta$, save the UUID of encrypted content in $File\_Meta$ and store the encrypted content at it's UUID in datastore.

When LoadFile() is invoked we go to the $File\_Header$ get all fields. We then get list of all UUIDs from that $File\_Meta$. We decrypt content at all UUID's and show it to user.

When AppendFile() is invoked we will fetch the *Content_key* and encrypt the content, put it in a file and save UUID. This UUID will be updated in the list of UUIDs in corresponding $File\_Meta$ pointed by the $File\_Header$. HMAC will be recalculated for $File\_Meta$.

To share file a user will create an instance of *sharingRecord* which contains MAC_key, *Content_key*, $File\_Meta$ and *Signature*. Receiver will set up a $File\_Header$ structure and populate it with shared data fields and use his own file name to put the header in datastore for future use. Receiver will verify the origin of message using Sender's Signature and Public Key.

To revoke access to a file we ask user for the MAC_key provided by owner and his private key. We hash them together and goto the location of $File\_Meta$ pointed by that hash and revoke access by re-encrypting all the segments of file with new MAC_key and *Content_key*. If the location doesn't contain $File\_Meta$ then we know user is not the owner.