Wesley Hsiao and Eashaan Katiyar
Design Doc

Part 1: Function Designs
InitUser:
We create a wrapper struct called wrapper. wrapper includes a salt generated by RandomBytes() used for hashing the password, creating a key to encrypt the user struct. We also generate a UUID by passing the username into uuid.FromBytes(). Then, we create a user struct, which stores Argon2Key(username + password, salt) as a symmetric key, and we also create a filesmap stored at a new uuid. We also need to store a public key in Keystore, so we call PKEKeyGen(), storing the private key in our user struct, and the public key in the keystore using KeystoreSet() on the pair UUID, Public Key. We do the same thing for digital signatures. We marshal this data structure to json and then encrypted using symEnc(), which will use the symmetric key and an IV generated by RandomBytes(). Lastly, we create a wrapper struct to hold the encrypted user struct, a mac of the encrypted struct, and the aforementioned salt and marshal it. we then call userlib.DataStoreSet(UUID, wrapper) to store all the info in the datastore for retrieval by the getuser function. We return a pointer to the unencrypted user structure to the caller.

GetUser:
We call uuid.FromBytes() using the username as the argument to obtain a UUID. We will then call the function userlib.DatastoreGet() on that UUID to retrieve the wrapper. We then hash the password using the salt contained in the wrapper and additionally use the mac to verify our user struct has not been tampered. If the macs are identical, we decrypt the user struct using symDec(). We then demarshal the json to user struct, and return its pointer.

User.StoreFile:
Inside user struct we store a uuid to the filesmap. We create a metadata structure describing the file called filinfo, which contains the uuid of the file, the key to unencrypt the file, and a boolean denoting whether the current user is the owner of the file. This key is used for the function SymEnc() to encrypt the file and create a mac, which are stored in a file struct that acts as the value in the key-value pair for datastore. The key-value pair is then stored using userlib.DatastoreSet() under a new uuid (thus no info about the filename can be leaked) by marshalling this file struct. Fileinfo is then encrypted, marshalled, and then stored at a new uuid with a unique key, and a new infowrapper struct is created containing the uuid and the unique key. This is then stored in the filesmap, which can be found through the uuid filesmap inside the userstruct, and decrypted with a hashkdf of the userkey. The filesmap is a two dimensional map, the first map keyed by filename, and then the second by username. Since the user owns this file, the second map is simply keyed with the user's username. In share, however, this 2-d map (from now on referred to as mapA and mapB make sharing easier).

User.LoadFile:
The filesmap is loaded from the datastore and decrypted using the user key. The infowrapper is then located by calling filesmap[filename][username], which allows us to load the fileinfo struct and decrypt it using the key stored in infowrapper. The fileinfo can then be used to load the file, and use the key stored in fileinfo to decrypt the file. However, we would first check the integrity of the file using HMACEqual. If the MACs match, the file will be loaded. Otherwise an error will be raised. The file is then returned.

User.AppendFile:

We first load the existing file struct in the same manner as User.Loadfile, but obtain the File struct (which contains the data and the MAC) rather than returning the file data.
In order to allow for efficient file appending, the file and mac inside the File struct are stored as 2-d arrays. When a new chunk of the file is needed to be stored, we simply append the next segment of the encrypted file in the manner of User.StoreFile, and similarly generate the MAC for just this new appended data. Thus, preexisting data does not need to be reencrypted or reauthenticated. The file is then re-encrypted in the same manner as well, and stored again on Datastore.

User.ShareFile:
We generate a copy of the fileinfo struct, with one change: the owner boolean is set to false. We store this at a new uuid, and generate a new corresponding infowrapper. We encrypt the marshalled infowrapper using the receiving user's public key using PKEEnc. A digital signature would then be generated with DSKeyGen, using our own sign key. The signkey and encrypted infowrapper would be stored in the Share struct, which is then marshalled and converted into a string, and returned. This copy of infowrapper is also stored in the user's filesmap under the receiving user's username.

User.ReceiveFile:
The receiving user converts the string back to bytes and unmarshals it to the Share struct. The user verifies the digital signature first with DSVerify. If the verification doesn't pass, an error is raised. Then, the user uses its private key (stored in its user struct) to decrypt the infowrapper. The user now has the infowrapper to access the struct which contains the UUID for their unique fileinfo and decryption key for the file shared. The infowrapper is stored in the filesmap similarly to how it is done in User.Storefile, the only difference being that the owner also has the ability to modify or delete the receiving user's fileinfo (as they created it).

User.RevokeFile:
The user deletes the contents inside the filesmap designated to the revoked user. Then, the file shared is re-encrypted using a different key and also stored with a different UUID inside datastore's key-value pair. The old key-value pair inside datastore is deleted using userlib.DatastoreDelete(). The other shared users who are not revoked would have the contents inside their fileinfo structs updated to the new key and new UUID.

Numbered Questions:
1. Please refer to our User.Storefile implementaion above.
2. Please refer to our User.Sharefile and User.Receivefile above.
3. Please refer to our User.Revokefile above.
4. Please refer to our User.Appendfile above.

Part 2: Exploits

Another possible exploit is forcing a uuid collision to successfully call api calls on data that the attacker is not supposed to have access to. Since uuid's are based on the first 16 bytes of a value (using the uuidfrombytes function), the attacker can attempt to create a collision at places that uuid's are created if the information used to create the uuid is public (such as username) or brute-forcable. For example, we are not guaranteed that filenames are high-entropy, and a malicious user could, using a precomputed dictionary of filenames, find collisions. If a uuid is generated based on a combination of filename and username, as our initial design suggested, a collision might occur and the attacker may be able to successfully use API calls to overwrite or even append to files this way. However, our implementation prevents this attack by using uuid.new() each time, which disconnects any data that the user passes in that may not have high entropy (as the only data passed in guaranteed to have high entropy is the password) with the location in the datastore that it is saved.

Another possible attack involves the attacker attempting to modify the file metadata of a file that he or she does not own. In this case, the attacker can change what the file metadata describes, and also make all other users with access to this file see these changes as valid. However, only the true owner of the file should be able to modify a file's metadata. Our implementation avoids this by creating an independent copy of the file metadata for each user, so even if the attacker attempts to modify the metadata the attacker can't change anyone else's metadata of the file, as the only person with access to every shared user's metadata will be the true owner themselves.

The attacker could attempt to modify their own userdata's username field, as well as other relevant publicly obtainable fields, in an attempt to successfully call file api functions with a "spoofed" userdata. Since we need to always keep updated information about a user's files across multiple instantiations, this can be a security hole, as in our original design, we stored info about the user's files in the user struct, which necessitated refreshing the user struct before any file-based api calls and used the username field to refresh the userdata. However, we are able to avoid this by ensuring the data within the user struct remains static, so that there is no need to refresh the user struct itself, and any dynamic data is referenced by the user struct through static pointers into the datastore.

A man in the middle attack can be mounted by the attacker between the share and receiving user. The attacker sends a magic string for a malicious file to the receiver. However, this problem is prevented as the magic string includes a digital signature. The receiver would try to verify the identity of the sender by verifying the magic string. If the magic string doesn't match the digital signature of the supposed sender, an error would be raised which prevents the attack.

We know that a malicious user can store information shared with it about a file even after a revoke. This is problematic, as it allows them to load and modify encrypted data and spoof authentication, as they can remember the key(s) needed for file encryption and authentication. We prevent this by changing the key when a revoke occurs, and then updating all file metadata for users that still have valid share access to that new key, and re-encrypting and authenticating the file data with this new key.