Wesley Hsiao and Eashaan Katiyar
Design Doc
Part 1: Function Designs
InitUser:
We create an authentication struct called auth_struct. auth_struct would include 2 salts generated by RandomBytes() used for hashing the password and the hashed password hashed by HashKDF() using salt1. We also generate a UUID by passing the username into uuid.FromBytes(). Then, we create a user struct, which stores Argon2Key(username + password + salt2) as a symmetric key. 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 both the auth_struct as well as the user struct, and we then call userlib.DataStoreSet(UUID, wrapper) to store all the info in the datastore for retrieval by the getuser function. 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 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 auth_struct and user struct. We then hash the password using salt1 contained in auth_struct and compare it to the preexisting hashed password in auth_struct to authenticate the password. If the hashes are identical, we run Argon2Key() with salt 2 and the concatenation of username and password to obtain the key. This key is then used to decrypt the user struct using symDec(). We then demarshal the json to user struct, and return its pointer.

User.StoreFile:
Inside user struct there will be a field that contains username and 3 salts. A temporary struct “temp” is created, containing username and filename. “temp” is then marshaled to json. The json is then hashed using salt1 that is stored in user struct through HashKDF(). The hash is then used as the argument for uuid.FromBytes() to generate the key for the key-value pair. The file is hashed using salt2 contained in user struct using HashKDF(). A key is generated through the concatenation of the username, salt1, salt2 hashed with salt3 using Argon2Key(). This key is used for the function SymEnc() to encrypt the file, which now acts as the value in the key-value pair. The key-value pair is then stored using userlib.DatastoreSet(). The file is stored in chunks of specified bytes stored under a struct. If the file size is greater than the size of one chunk, another chunk is created and the remainder is stored in there. The file struct also contains a HMAC key computed based on the contents of the file (HMACEval). A dictionary is included in the user struct which uses the filename as the key and the mac key as the value, so the integrity of the file can be checked.

User.LoadFile:
A temporary struct “temp” is created, marshaled to json. The json is hashed using a salt stored in user struct through HashKDF(). The hash is used as the argument for uuid.FromBytes() to generate the key to retrieve the encrypted file. Using the concatenation of username, salt1, and salt2 salted with salt3 with Argon2Key(), we generate the key 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.

User.AppendFile:
First we generate the temporary struct “temp”, marshaling it to json. The json is hashed using a salt contained in user struct through HashKDF(). The hash is used as the argument for uuid.FromBytes() to create the key to obtain the file. If there is no matching key contained in datastore we return an error. Otherwise, we authenticate the file using HMACequal. If authentication doesn’t pass, we return nil and raise error. After authenticating the file, we append data to the existing file by checking the chunks and filling the last chunk. If the last chunk is full create a new chunk and fill it.

User.ShareFile:
We will create a share struct that includes the UUID to the file struct and a key to decrypt the file. This struct will have an UUID. This UUID would be generated through the function uuid.FromBytes using marshaled file struct which is JSON. This share struct will be encrypted using the receiving user’s public key using PKEEnc. A digital signature would then be generated with DSKeyGen, storing the verifykey to the keystore. The signkey would be stored in the user struct. This ciphertext would then be signed through DSSign(). Then we will send the signed ciphertext to the shared user.
If the user plans to share a file that he or she didn’t create, the user can only share the access token through the same encryption method (public key encryption + digital signature).
When the user shares the same file to a different user, a different share struct is created. The share structs are included in the user struct.

User.ReceiveFile:
The user receives the signed ciphertext (access token). 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 access token. The user now has the access token to access the struct which contains the UUID and decryption key for the file shared. This is stored in user struct under the section “files shared”, which is different from the user’s own files.

User.RevokeFile:
The user deletes the contents inside the share struct 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 share structs updated to the new key and new UUID.

Part 2: Exploits

One possible exploit is that a malicious datastore could allow any user’s file to be overwritten with random garbage. Although the attacker would be unable to intelligently modify the unencrypted information in the file, they could corrupt important data. In our design, we deal with this issue by generating a MAC while storing files. This not only allows us to know whether another instance of the user has modified the file, but if the MAC has indeed been modified - and doesn’t match the unencrypted data underneath - then we know that a malicious user has modified our file. This implementation renders our files tamper-evident.

Another possible exploit is possible if a user shares a file with a malicious user for whatever reason, and later, realizing their mistake, revoke the malicious user’s access to the file. However, in the case that the implementation of the revoke file only updates the access key but does not relocate the file, the malicious user, by remembering the UUID of the file, the key to decrypt it and the filename, could call StoreFile or AppendFile in order to overwrite the unencrypted bytes of the file, allowing the malicious user to overwrite the file with a malicious virus. Our implementation prevents this by re-encrypting the file with a completely new key and new UUID which is broadcasted to users that still should have access, while the revoked user only has information about the old version of the file.

Another possible exploit is accessing the file of a different user through chosen username and filename. If the filename/username+filename is used to generate the UUID, the malicious datastore can identify the filename since UUID isn’t safe and can leak information of the filename. To deal with this our system hashes the plaintext used for UUID before generating the UUID. The adversary can further attack through chosen filename and username. For example: if a user has the username “a” and filename “sk” and another malicious user has the username “as” and filename “k”. The hash of the concatenation would be the same, which would still end up granting the adversary access to the encrypted file. However, we marshal the struct that includes the filename and the username and then hash that and feed it to UUID.FromBytes, which eliminates this threat.

Another possible exploit is that a user, for whatever reason, shares a file with a malicious user. What this malicious user could then do is share the file with a dummy child user. So if the malicious user has its access revoked, the dummy child still has access to the file. However, this is prevented as users can only share files that they didn’t create through sharing the UUID to the struct that contains the key for decrypting the file and the UUID to the file. When the user is revoked, the contents of the UUID and key no longer work as the file is re-encrypted and placed to a different place in datastore, with the old destination deleted. It, and all of its children can only modify this old data.