Sheerak Megerdichian
Jonah Moon

System Design

Our design is dependent on structuring our file system in the malicious datastore similarly to how operating systems address their directories in using slash notation to separate the user's name and file name. Once we create the absolute path for a given file, we hash the path then use the first 16 bytes as a parameter to the FromBytes() function to convert the hash into a universally unique identification and use the UUID as the key and a file struct as the value. To store the content we will hash the address with a /content concatenated. We will hash all keys before converting to UUID, since UUID reveals information about the argument. Everything will be encrypted with symmetric and public keys to ensure confidentiality, and sharing will be digitally signed to ensure authenticity.

```
const MAX_LENGTH = 10000
type File struct {
        Owner string
        Filename string
        Address string
}
```

InitUser(username string, password string) (userdataptr *User, err error)

- Hash name into uuid.frombytes(), map to random salt
- Hash name/pw/salt into uuid.frombytes() as key to map to encrypted hashed password/salt, using password generated Argon2Key()
- Store public key in key store, hash name/pkey as uuid
- Struct has priv key, map of symm keys (shared_user maps to symm_key)

GetUser(*username string*, *password string) (userdataptr *User*, *err error*)

- Hash name get salt then check hashed of name/pw/salt and get password generated Argon2Key() to decrypt to verify pw/salt

StoreFile(*filename string*, *data []byte)*

- Store by creating a file struct containing owner, filename, slice of shared recipients, slice of content where we limit each index 10000 bytes, marshalled shared recipients and slice of content, and use the frombytes() to create UUID for each slice.
- We will then use the hash of user/filename then converting the value with fromBytes() as the key and marshal the struct and encrypt it before storing

LoadFile(*filename string) (data []byte*, *err error*)

- Load by getting the hash of user/filename for uuid.frombytes() then getting the encrypted data from the datastore, then decrypt with our symmetric key

AppendFile(*filename string*, *data []byte) (err error*)

- To append, we will hash user/filename to get the key, then retrieve the data and unmarshal the file struct. As the last step, we traverse the slice and find the index where we will need to append our new data.

ShareFile(*filename string*, *recipient string) (access_token string*, *err error*)

- To share the file, create a file struct with the owner, shared_recipients, contents, and length filled out properly. Then marshall struct and sign the resulting byte[], then encrypt the byte[] concatentated with the signature and concatentate an integer representing where the signature index begins.
- Since we are working with pointers we can create a file for the recipient and have the content point to the original file contents

ReceiveFile(*filename string*, *sender string*, *access_token string) (error*)

- Check the digital signature of the received byte[] by retrieving the key from the keystore, then unmarshal the byte[] into the file struct

RevokeFile(*filename string*, *target_username string) (error*)

- Delete the target_username's shared file version of the file from the datastore and check the target's file struct before to recursively delete all other users the target shared the file with

1. How is a file stored on the server?

- A file is stored on the server when a user invokes StoreFile(). We will first create a file struct and initialize the member variables, owner, filename, slices of shared_recipients and content, and marshalled then encrypted keys for the slices. Then we will json.marshall() the struct into a byte[] and store the data into the datastore keyed by username/filename converted into an UUID.

2. How does a file get shared with another user?

- When we share a file, we create a file struct for the recipient with the same member variables as the original file except the shared_recipients. We will have a shared_recipients slice for each recipient so that we can handle revoking subtrees of access when the owner revokes a specific user. We share the file through the encrypted marshall of the contents slice.

3. What is the process of revoking a user's access to a file?

- The process of revoking a user's access consists of deleting file struct that we stored on the datastore for a specific user. Once that file struct is deleted, the user has no way to access the shared file.

4. How does your design support efficient file append?

- We support efficient file appending by limiting each index within the contents slice to be a length of 10000 bytes. Since we limit each index to be 10000 bytes, our worst case run time would be O(n + (10000 + m)) to append, where n is the amount of indices we traverse and 10000 + m for iterating through the available index and writing m bytes.

Security analysis

1. Since the datastore is insecure and all communication can be seen, the database is vulnerable to a Dictionary Attack if we store the passwords naively as their plaintext values. In order to prevent this attack, we will generate a random salt for each user and hash the password concatenated with the salt using a '/' as a delimiter. Lastly, we will encrypt the hashed salted password with a symmetric key. Now when a user tries to login, we will check the calculated hashed salted password and verify that it matches the value we have stored in the database.

2. When sharing a file, the access_token can be observed or modified by the malicious server or user. Therefore in our design we encrypt our access_token and digital signature with the public key of the intended user. Once the intended user receives the access_token, they can decrypt it with their private key, check the digital signature with the sender's public verification key for integrity and authentication checks.

3. If a malicious user has gained access to a shared file and once we revoke the access, the user should not be able to access or load the new version of the file. We will ensure that the user does not have access by creating a map that has the owner/path of a file as the key and maps to the hash of the datastore key for that specific file. Once the user is revoked, we delete the entry in the users map so that the user cannot access the new version of the previously shared file.

4. Filenames must be confidential. An adversary should not be able to guess the length of any filename, or even narrow down the set of possible values for its length. Lengths of file contents do not need to be hidden

5. using the same keys for different stages.