Ryan Hurst
3032757769

Project 2 Design Doc

Section 1: System Design

1. How is a file stored on the server?

The client starts by creating a new random UUID, along with random symmetric keys for storing and retrieving the encrypted file. Next, the client initializes KeyMap and FileMap, mapping the given *filename* to the generated encryption and signature keys, and the given *filename* to the randomly generated UUID, respectively. KeyMap and Filemap will later be used for mapping the filename to symmetric keys along with UUIDs for new updates from any user with access to the file. After initializing and filling FileMap and KeyMap, the client json.Marshals the raw data for the given *filename*, encrypts and signs the bytes generated, and stores the encrypted data in DataStore under the random UUID generated for the given *filename*.
To initialize our structure for sharing files with other users, we check that the given file doesn't contain any updates, then proceed by creating an *appendsUUID*, along with random encryption and signature keys for encrypting and storing file updates. We then create an *appendsUUID* and keys for updates, storing them in FileMap and Keymap under *filename* + "numappends" so we can update any file changes for any user with access to the file. For later sharing functionality, we initialize an empty *Node* struct, which creates a shared tree structure by storing the UUID of the *Shared* struct (*covered in (2.)*). Finally, the client updates, encrypts, and stores the updated User Struct in DataStore under the given users UUID.

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

The client first checks that there is a fileUUID for the given *filename.* To allow for proper sharing and revocation functionality, the client receives the UUID for the sharing *Node* from the filemap and keymap, decrypts, and generates the necessary symmetric Keys & UUID for the recipient, which gives the recipient access to the Shared struct. After encrypting and storing the new *Node* on DataStore for proper file sharing & revocation, the client then creates a Shared struct that holds the symmetric keys & UUIDs for the file, updates to the file, and any Nodes the recipient may use for later sharing. The client then generates a magicString containing Shared struct UUID & symmetric keys. To send the information needed to receive the *sharedStruct,* the client encrypts the magicString with the Recipient's RSA public encryption key, and signs with the Sender's RSA Private Signature key. The encrypted magic string and it's RSA signature are appended and sent as a string over the insecure channel to the recipient.
To receive a file, the recipient verifies the signature on the magic string with the Sender's Public RSA verification key to ensure integrity and authenticity. Once verified, the recipient then decrypts the magic string with the recipient's RSA private decryption key, receiving a byte array with the UUID, decryption, and verification keys to receive the *sharedStruct* from DataStore. After receiving and decrypting the shared struct with the keys obtained in the magicString, the

Ryan Hurst
3032757769

user initializes and stores the UUIDs & keys for the given *filename* by traversing the tree of *Node* structs.

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

The client first ensures that the requested *filename* to revoke exists for the given user, then saves the original file plus all of the appended data if the given file exists by calling LoadFile on *filename*. Next, the receives the UUIDs for the root file *Node* (FileMap[*filename* + *"_shareNode"])* and its updates (FileMap[*filename* + *"_shareNode"]*), which gives us access to the Users in the shared tree structure. To remove all access to the file for the given user, the client then traverses from the root *Node*, decrypting and checking for the *target* user's Node. Once the target user's *Node* is found, the client removes the *Node* from Datastore. This removes access to the *target* user, along with anyone that the *target* user has shared with, *without* removing file access for anyone else.

4. How does your design support efficient file append?

To support efficient file appends, I split the storage of file data for the original file from the storage of appended data in DataStore. This allows us to decrease runtime demand by only requiring the encryption and storage of the *new* appended data. The process is laid out in further detail below:

To append file data, the client first gets the fileUUID (FileMap[*filename])*, along with the UUID for Updates struct from the User's FileMap (FileMap[*filename* + "numAppends"]). The client then gets the encrypted User Struct stored in DataStore under the UUID for the updates struct, and gets the decryption & verification keys for the encrypted Updates struct from the User's KeyMap (KeyMap[*filename* + "numAppends"]). Next the client decrypts the Updates struct by receiving the Updates struct and obtains the number of appends already made to the file by the length of the number of appendUUIDs stored in the update struct. Now that the client has access to the Updates struct, which contains the UUIDs for each encrypted append, it can initialize a new set keys and random UUID for the next append. For user access, The UUID and encryption and signing keys for receiving and decrypting the new append are stored under the FileMap and KeyMap. The new data to be appended is then json.Marshalled, encrypted with the encryption and signing keys for the given append, and stored in datastore under the UUID for the appended data. To update all necessary data for the append, the User struct is updated and stored, and the updates struct is updated to store the UUID for the append (updates.UpdateUUID[numAppend]), along with the decryption and verification keys (updates.Keys[numAppend]). After updating the necessary data for numAppends, the updates struct is json.Marshalled, encrypted and signed with the keys obtained for the updates struct in the beginning, then stored under in DataStore under the UUID for the updates struct.

Ryan Hurst
3032757769

Section 2: Security analysis [maximum 1 page]

1) **Man in the Middle:** To demonstrate the Integrity and Confidentiality of our system we introduce an active eavesdropper, Eve, who listens in on Alice sharing a file to Bob. In the case where Eve is able to intercept the magic string associated with the shared file with the intent to either a) tamper with the file sent to Bob, or, b) send Bob a different Magic string to a malicious file, our system is able to defend against the attack by requiring that Eve verify the signature and decrypt the magic string with Bob's private RSA key.

2) **DataStore Vulnerability:** Since the DataStore is untrusted, we can assume Eve has access to the files stored in DataStore. To exploit this vulnerability, Eve utilizes access to the contents of DataStore, and tries to swap files on the datastore so that when the user loads a file from the server they load whatever malicious file Eve has replaced it with. However, all file data stored in DataStore is encrypted using the generated encryption key, along with the file's decryption key being stored in the *trusted* server KeyStore. So, if Eve tries to swap out files, the signature on Eve's file won't match those stored in the User's KeyMap, so the next time the user attempts to load and verify the signature on the file, the client will throw an error.

3) **File Revocation:** In this case, Eve has been listening in and saving information sent between Bob and Alice after sharing multiple files with each other. If Bob decides to revoke Alice's access to one or more files, Eve will be able to use past information about the files shared between Bob & Alice, along with information from the User struct data stored in DataStore to gain access to the revoked file(s). My design prevents against this attack by first marshalling, encrypting, and signing the User struct before storing it in DataStore, preventing against Eve gaining information about changes in the FileMap, along with encrypting the sharing struct so that any information Eve collected over time is inaccessible.
