CS161 Project 2 Design Document

System Design

There are two servers that users will be accessing:

- **Keystore**
  This is a **trusted** server where users can publish their RSA public keys. It is a key-value store of `(username->RSA public keys) pairs.

- **Datastore**
  This is an **untrusted** server that is used to store files. Files are encrypted such that IND-CPA confidentiality and integrity of file names and file contents are protected. Files are partitioned by their owner, so duplicate filenames might be possible.

`InitUser(username string, password string)`
Each user needs to be assigned a unique, deterministic UUID when they are initialized. This is done by hashing `username||password` and using the result to *deterministically* generate the user's UUID.
By hashing `username||password`, we prevent the attacker from finding out a user's UUID unless they know both the username *and* password.
Each user's (hashed and salted) password must be stored in the struct to verify passwords during login attempts.
This function will also generate a key pair for digital signatures and a key pair for encryption.

1. How files are stored on the server

Each file is represented as a `File` struct. The important fields in `File` struct are:

- `Users` (`*UserTree`): A tree data structure whose nodes represent the users that have access to the file.
- `Data` (`[]byte`): The original data of the file.
- `Appends` (`[]UUID`): A list of the UUIDs of each append to this file. Each append is encrypted and tagged on the datastore using the same symmetric encryption and HMAC keys as their file.

When a file is created:

- It is given a **random** file UUID that serves as its database "filename". Each user tracks these file UUIDs in a personal map of (filename, file UUID) pairs.
- It is given a **random** `fileKey`. The `fileKey` is used to derive the symmetric encryption and HMAC keys. Therefore, the user only needs to keep track of `fileKeys`, which are kept in a personal map of (file UUID, `fileKey`) pairs

To ensure files are stored with confidentiality and integrity, they are encrypted using symmetric encryption, and then an HMAC is generated and the encrypted file is tagged with the HMAC. Files need to be padded before they are encrypted to guarantee they are a multiple of the blocksize used in AES.

2. How files are shared with other users

A user only needs to know a file's UUID and its secret `fileKey` to be able to access find it, decrypt it, and change it.
So, when a user Alice wants to share a file `foo` with Bob, she:

1. Update the file's authorized users. This is done by adding Bob as a branch under Alice in the `UserTree`.
2. Creates an `Invitation` structure that contains the file's UUID and the `fileKey`.
3. Encrypts `Invitation` with Bob's public key (to provide confidentiality) and signs the encrypted `Invitation` (to provide integrity and authenticity).
4. Stores the encrypted and signed `Invitation` in the datastore under a generated UUID, and then sends this UUID to Bob as the `accessToken`.

3. How user access to files are revoked

When the owner of a file revokes some user's access, the user's node is removed from the tree data structure that is used to track all of the users with access to the file. A new symmetric key for the file is generated and used to encrypt the file, and this new symmetric key is stored where it's accessible by the other users with access to the file.

The `UserTree` data structure is just a basic tree where each node is a file with access to the file. The root node is the original owner of the file, and each of the root's child nodes are users that file owner shared the file with directly. Each node in the tree was shared with by their parent node.

To revoke access to a file from a user we:

1. Change the `fileKey` for the file and reencrypt/retag it.
2. Then, we perform depth first search on the `UserTree` to find the user and then remove that user from their parent's list of children.
3. Then, we update each node in the new tree with the new `fileKey`.

4. How files are appended efficiently

Each append to a file is stored as a separate record in the datastore. Each file keeps track of a list of all of its appends, so when we want to append to a file, we just create a new record in the datastore and append the UUID for the record to the list.

Security analysis

Man in the Middle attack

The access token used to give users access is communicated over an unsecure channel. Eve could try to eavesdrop or intercept and modify this code. This is circumvented in my implementation by using the recipient's public key to ensure confidentiality, and signing the message to ensure integrity and authenticity.

Editing the contents of a file

An attacker may try to edit the contents of a file. However, each file is encrypted using a shared symmetric key and the encrypted file is tagged with an HMAC. If the contents of a file are changed, message authentication will fail.

Brute force attack & rainbow tables

An attacker might try to brute force passwords, but this is not feasible because passwords have sufficient entropy. The attacker may try to consult rainbow tables to find common password hashes; to thwart this I've stored the passwords as a salted hash.

Malicious user attempting to reuse UUID & fileKey

A user who has their access from a file revoked may have recorded the file's UUID & fileKey, which is all that's needed to decrypt a file. To prevent this, users are not only removed from the list of authenticated users; the file is also reencrypted/tagged using a new fileKey and this new fileKey is passed to remaining authenticated users.

User modification

An attacker may insteady to to attack the User structs that are stored in the datastore. Changing a User struct could remove the user's access to some files by deleting entries in the file or key maps. Even worse, an attack could get a user to load a malicious payload by pointing a filename and key to a datastore record of their choice. Corruption of a User record is prevented by storing the record with an HMAC tag that is *deterministically* created using the user's username and password. Therefore, the only way an attack could modify a User is if they have access to the password.