Yifan Zhang: 3032717456
Katie Kim: 3032756196

Project 2 Final Design Doc

Overview

For our InitUser function, we create a new User struct, which contains information such as unique salt, the hash of password + salt (for password checking purposes), the master key used for key management, and more. We also create a public and private key pair for both RSA encryption and Digital Signature purposes. We stored the public keys for both in the KeyStore database with the UUID generated from the standard format “RSA” + Username or “DS” + Username. The private keys are stored in the User struct. We also add an additional Integrity variable in the User struct to ensure integrity. Our method is to simply append every field of the User struct and calculate a hash. On each round that we need to retrieve user information, we calculate a new IntegrityHash and compare it with the stored one. If a difference occurs we then know that the account has been tampered with. Below we explain more in detail how file storage and sharing functions. Note that at the beginning of each function, we first check the integrity of any data we pull from Datastore to detect tampering.

Design Questions

1. How is a file stored on the server?

To perform a file storage, we simply need to obtain the UUIDs of the corresponding fileRecord and appendRecord and set them to the new file and a brand new append record respectively. Since we don’t actually care if the record already exists (as we are gonna overwrite it anyway), we can accomplish this by using DatastoreSet function. If the file is shared with the user, we can obtain the fileUUID and appendUUID as well as the corresponding keys used for decryption from the shareRecord. If it’s not shared, we simply create the above 4 fields using the following scheme.

- fileRecordUUID: UUID from Username + "_" + filename + "_fileRecord"
- fileKey: HashKDF(masterKey, []byte(filename))
- appendRecordUUID: UUID from Username + "_" + filename + "_appendRecord"
- appendKey: HashKDF(fileKey, []byte(filename))

We then create two Hashmaps both with a data field and HMAC field used for fileRecord and appendRecord. The encrypted files (done using their respective keys) are then stored at the UUID in DataStore

Yifan Zhang: 3032717456
Katie Kim: 3032756196

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

Since the main fields we need to access the file and the append records are their UUIDs and Keys, we shall create a separate shareRecord that contains these fields. The UUIDs of the shareRecords are unique to each sender, recipient and filename combinations so we avoid an overlap. The shareRecord is then encrypted using a key generated from the sender masterkey. The receiver then needs the shareUUID and shareKey to access the shareRecord. We create a token that contains a Digital Signature (DS using sender private key), recipient name (to avoid token interception), encryptedShareKey (PKE using receiver public key), and marshalledShareUUID. The receiver can then take the token, ensure the authenticity by verifying DS (using sender public key), decrypted shareKey (PKE using receiver private key) and unmarshal UUID. The implementation also checks that the receiver in the token matches whoever has the token to avoid another user intercepting the token.

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

The revoking process is rather simple. Since we know that the shareRecord is the critical field for accessing the file and append records. We simply need to delete the shareRecord. Even if the target user still has information regarding the UUID and Key of the share record, they can no longer access it. This is also efficient for deleting a whole sharing tree as subsequent shares from the direct child will be all directed to the same shareRecord. Removing this record essentially removes the entire share tree of the target user.

4. How does your design support efficient file append?

Initially we wanted to create a single hashmap for each file that handles both data storage and append. The only problem with that is each time we call append, we need to download the entire hashmap, change the append record and then recalculate integrityHash for the entire thing. Since we don't really need the file data when performing appends, we decided to create a separate appendRecord. The append process is relatively simple: we download the record; append the existing append data with the new data; recalculate a new HMAC and reupload. When LoadFile is called, we then append the append data to the file data, reset the appendRecord and return the new appended file data.

Yifan Zhang: 3032717456
Katie Kim: 3032756196

Security Analysis

1. We can foresee that an attacker can use a brute-force approach to guess users' passwords by trying common passwords. One security feature that we added in our system is salting user structs.
2. One scenario that we can foresee is when User 1 shares a file with a malicious user, User 2, and then later revokes User 2's access. When User 2 still had access, they could've stored the access token then used their secret key to decrypt this access token. This then allows the user to decrypt the share record and as a result, give them access to the file key. Even after User 2 no longer has access, with this file key, they could still access the file. However, in our implementation, we keep track of shareRecord, which contains information about the users who have access to the file. Once a user's access to a file is revoked, we just delete the corresponding shareRecord. As a result, even if the user still has information like the file key, they will be barred from accessing the file.
3. Another scenario that we can foresee is where User 1 wants to share a file with another user, User 2, but there's a malicious user eavesdropping, who figures out the access token to the file. This malicious user can then try to access the file with the overheard access token. However, in our implementation, the access token generated is unique to each sender, recipient, and filename combination, meaning that even if an eavesdropper decided to intercept the token, they wouldn't be able to access the shareRecord.