DESIGN DOCUMENT

CS161 Project2

Zixiao Wang
ID: 3038733258
University of California, Berkeley

April 21, 2023

Contents

1 Data Structure

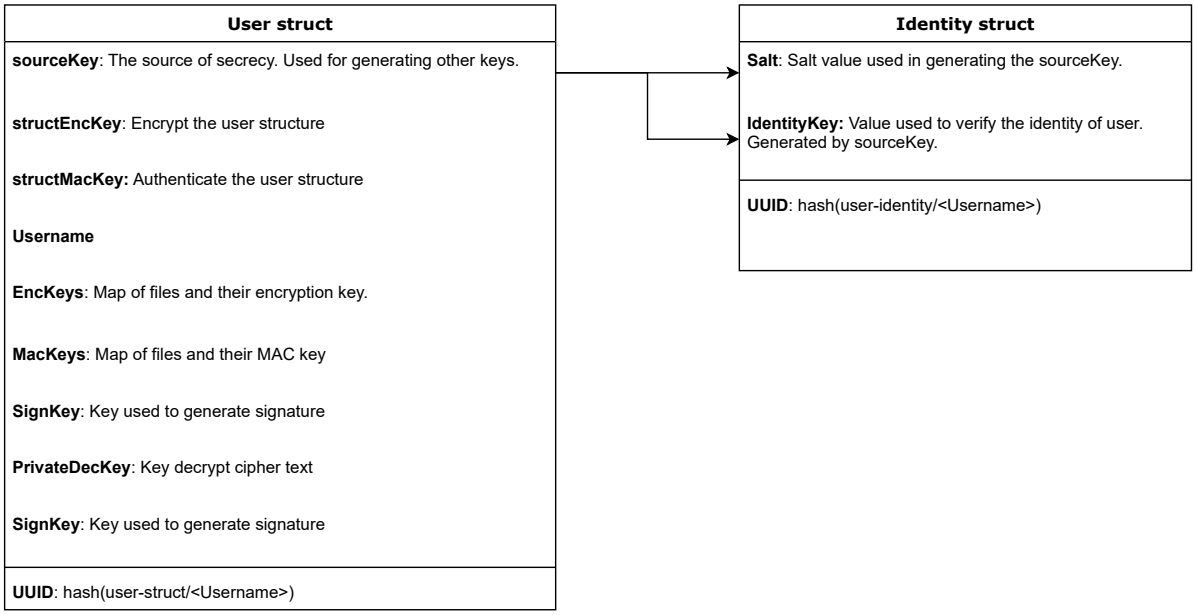

Figure 1: Data structures used for user information storage and user authentication.

(Figure 2 for file is two large to include in this page. See last page of the document for details.)

2 User Authentication

2.1 User Initialization

The **User** and **Identity** structure are created while initializing user. We first use the password to generate a *sourceKey*. Other keys are generated for future use.

- *structEncKey*: A symmetric encryption key for user structure
- *structMacKey*: A key used to generate MAC value for data structure
- *SignKey, VerifyKey*: A randomly generated key pair used to sign and verify
- *PrivateDecKey, PublicEncKey*: A randomly generated key pair used for asymmetric encryption.

The symmetric encryption keys and MAC keys are derived from *sourceKey*. The source key is generated from the password and salt. The *sourceKey* is the source of secrets and we don't store it. It can be generated as long as the user provide the correct username and the corresponding password. The two public-private key pairs are

randomly generated for each user. The public keys are stored in **Keystore** and any other keys are fields of the **User** structure.

Also an *IdentityKey* which is used to verify the user's identity is generated and store as a field of **Identity** structure. The **Identity** structure also store the salt value which is used to generate the *sourceKey*.

After all fields of **User** and **Identity** structure are generated, we store the two structures to **Datastore**. The **User** structure is encrypted with *structEncKey* and then authenticated with *structMacKey*. The **Identity** structure is not encrypted and we generate a digital signature for it to ensure its integrity.

2.2 Login

When users try to login, they should provide the username and password. The following steps are used to authenticate the user:

1. Generate UUIDs for user's **User** and **Identity** structure.
2. Find the **Identity** structure in **Datastore** with their UUIDs. Use the verification key to verify its integrity.
3. Use the salt value stored in the **Identity** structure and the provided password to generate *sourceKey*. With the source key, we can generate the *IdentityKey*.
4. If the calculated *IdentityKey* is different from the stored *IdentityKey*, the user fails to login. If they are the same, proceed to step 5.
5. Login successfully. Use the *sourceKey* to derive *structEncKey* and *structMacKey*. Authenticate then decrypt the **User** structure.

Any verification/authentication failure in above steps will cause an error.

Session Synchronicity. To ensure the synchronicity of multiple user sessions, a helper method *SyncUser* is used. **User** structure is stored again whenever there is an update of its fields (e.g. new keys generated after storing a new file). For example, Alice stores a new file with her laptop, she has a new encryption key and MAC key for the file. The keys are stored to the **User** structure in **Datastore** but her phone session doesn't have the new keys. If she try to load the new file, the *SyncUser* function will be called first to update the **User** structure in her phone's session. More details about the *SyncUser* method will be covered in section 5.

3 File Storage and Retrieval

3.1 File Storage

Every file consists of three parts:

- File content: Contents of the file

- File location: A list of UUIDs of the file contents.
- File header: Some metadata of the file.

When a user tries to store a new file:

1. Three UUIDs will be generated for content, location and header.
2. Generate symmetric encryption key and MAC key for the file (user *sourceKey*). Store the keys to *EncKeys* and *MacKeys* in the **User** structure as key-value pair {filename: key}.
3. Use the generated key to encrypt then authenticate the file content. Store the authenticated cipher text to **Datastore**.
4. Create location list. Store the UUID for the content to location list. Use the user's *SignKey* to sign the location list and store it to **Datastore**.
5. Create file header and initialize its fields. The *SourceHeader* of the header is initialized to the UUID of location list.
6. Use the user's *SignKey* to sign the file header and store it to **Datastore**.

The file header and location are not encrypted.
If the user wants to store to an existing file:

1. Generate UUIDs for content, location.
2. Find the location list by tracking back the *SourceHeader* field of the headers (details will be included in Section 5 *findContentLocations*), verify it. If it is not tampered, we get all locations of the file content.
3. Delete all the contents currently stored in **Datastore**. Delete all UUIDs in location list.
4. Store the content UUID in step 1 to the location list.
5. Use the encryption and authentication used before (the first time the file is stored) to encrypt then authenticate the content. Store the authenticated cipher text into **Datastore**.

3.2 Efficient Append

The user tries to append to an existing file:

1. Generate file header UUID and verify the integrity of the file header.
2. Find the location list of file.(Details in Section 5 *findContentLocations*).

3. Generate new content UUID according to the current number of contents. For example, if the current number of contents is 1, then the UUID is generated from hash(file-content/Username/hash(filename)/1) (The initial storage has index 0). Append new content UUID to the location list.

4. Encrypt then authenticate the new content with encryption and MAC key. Store the authenticated cipher text to **Datastore**.

By using different UUID for contents of the same file, and managing a list to track the locations of the contents, we can append to file efficiently without loading previous stored content from the **Datastore**.

3.3 File Retrieval

When the user tries to retrieve the file content:

1. Generate file header UUID and verify the integrity of file header.

2. Find the location list of file.(Details in Section 5 *findContentLocations*).

3. Find the contents of the file using the file content list. Authenticate then decrypt the contents.

4 File Sharing and Revocation

4.1 Create Invitation

When the user invite other users:

1. Create the **Invitation** structure. The invitation structure contains:
   - The symmetric key used for decryption.
   - The MAC key used for authentication.
   - The sender's file header UUID.

2. Use the recipient's public encryption key to encrypt the invitation and the sender's sign key to generate digital signature of the invitation.

3. Store the invitation to the **Datastore**.

4. Add the recipient's username to the *ShareWith* list of the file header.

Instead of symmetric encryption scheme, we use the recipient public key to encrypt the invitation so that no one else except for the recipient can decrypt the content.

4.2 Accept Invitation

When the recipient decide to accept the invitation:

1. Use sender's verification key in **Keystore** to verify the integrity of the invitation.
2. Use the private decryption key to decrypt the invitation. Store the encryption key, MAC key to *EncKeys* and *MacKeys* in the **User** structure.
3. Create a new file header with the *SourceHeader* field initialized to the *Source* field in the invitation.
4. Delete the invitation once it is accepted.

4.3 File Revocation

When the owner of the file wants to revoke access from other user:

1. Get the file header in the owner's namespace. Delete the username from the *ShareWith* list of the header.
2. Generate new encryption key and MAC key for the file. Decrypt then authenticate the file content with the new keys and store it to previous location in **Datastore**.
3. Use the *ShareWith* list to determine the user who still have the access to the file.
4. Create new invitation to the user who still have the access to the file. The new invitation contains the new keys.
5. Just as before, encrypt the invitation with recipient's public key and sender's sign key. Store the invitation to **Datastore**.

The revoked users don't have the new keys, so they cannot decrypt the file or tamper the file content without being detected. Also, to make the user who still have the access to the file get the new keys automatically, we use the *SyncUser* function to update their new key automatically before they load file or do any other operations.

5 Helper Methods

SyncUser: The function is used to ensure the user can have multiple sessions and help with the revocation of the access. The function has the following steps:

1. Check whether there is any update of keys due to the revocation of other users. If some updates available, update the corresponding keys in the *MacKeys* and *EncKeys* map in the **User** structure. Encrypt then authenticate the new **User** structure and store it.

2. Also, other sessions of the current user may store new files, but the current session does not have the keys. So we reload the *User* structure.

findContentLocations: The function is used to find the location list of the file content given the current header. The task can be completed by tracing back headers:

1. Check the *Owner* field of the current header. If it is *true*, then the location UUID is the *SourceHeader* field of the current header.

2. If the current header is not the owner of the file, set the current header to the header pointed by the *SourceHeader*. Go back to step 1 and repeat the process.

encThenMac: It is used for the symmetric encryption scheme.

1. Encrypt the message and then generate an MAC for the message.

2. Return MAC $\|$ cipher.

getSignedMsg:

1. Generate signature of the data.

2. Return the signed data Signature $\|$ data.

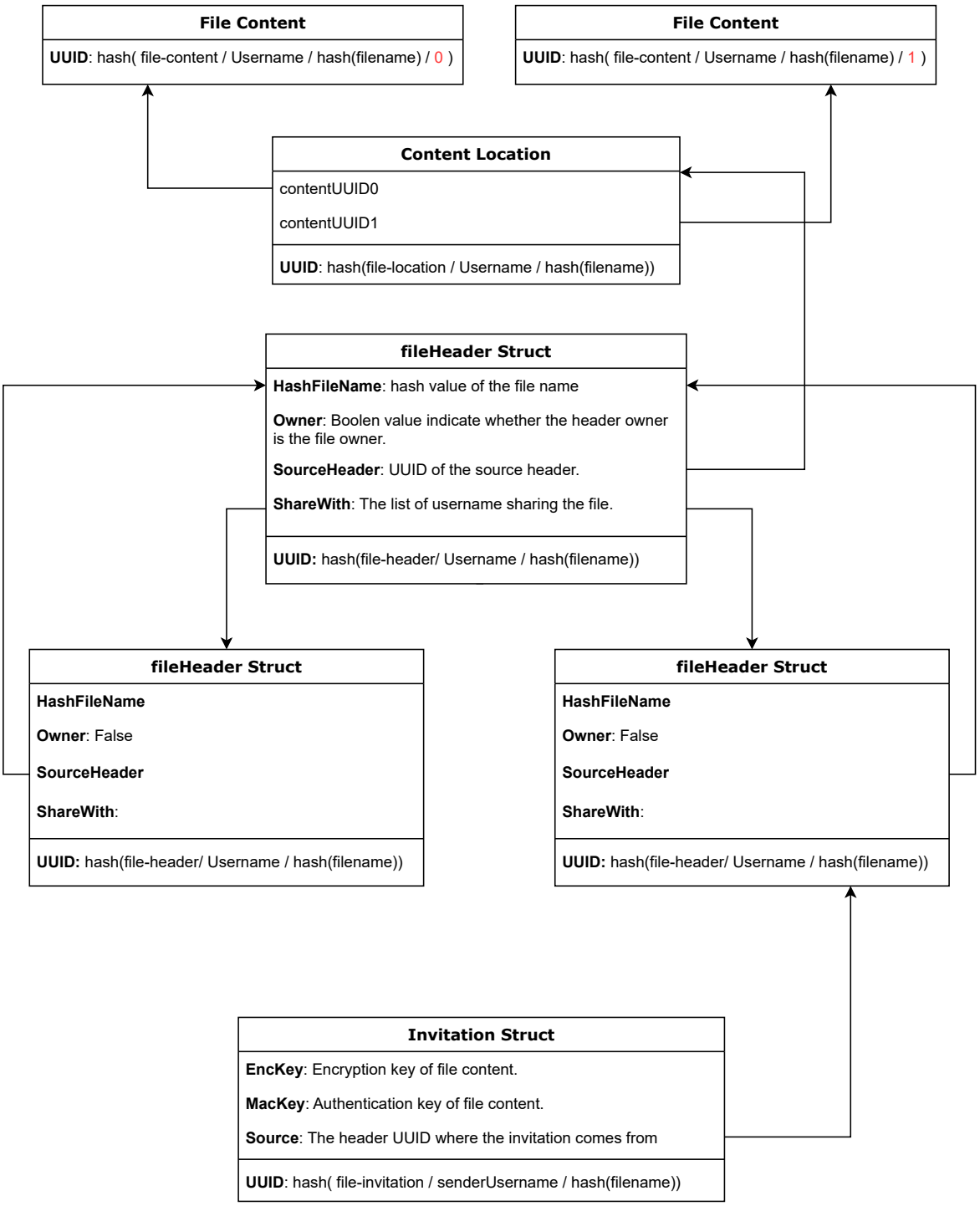

Figure 2: Data structures used for file system.