Justin Williams Design Doc for Project 2, 161

InitUser(username string, password string)

1. Check if the username is empty; return an error if so. (Empty passwords are permitted.)

2. Hash the username and derive the first 16 bytes to create a deterministic UUID. If this UUID is already in the Datastore, return an error (duplicate user).

3. Generate a 16-byte root key using Argon2 with the password and a randomly generated salt.

4. Derive two additional keys using HashKDF from the root key: one for encryption and one for HMAC.

5. Generate a public-private key pair for encryption (PKE) and another for digital signatures (DS). Store the public encryption key in the Keystore under username_enc and the public signing key under username_sign.

6. Construct a Usermetadata struct, which includes the username, an empty FileAccessMap, and an empty Files map. Serialize and store the private PKE and DS keys inside the struct.

7. Marshal the Usermetadata, encrypt it using the encryption key, and compute its HMAC using the HMAC key.

8. Store the concatenation [salt || HMAC || ciphertext] in the Datastore under the user's UUID.

9. Return a non-persistent User struct in memory that includes the username, root key, hashed password, and FileAccessMap.

GetUser(username string, password string)

1. Check if the username is empty; return an error if so.

2. Hash the username and derive the first 16 bytes to get a deterministic UUID. Fetch the corresponding record from the Datastore. If the record is missing or less than 80 bytes, return an error.

3. Parse the record into three parts: the first 16 bytes are the salt, the next 64 bytes are the stored HMAC, and the remaining bytes are the ciphertext.

4. Recompute the root key using Argon2 with the password and extracted salt.

5. Derive two additional keys from the root key using HashKDF: one for encryption and one for HMAC.

6. Recompute the HMAC over the ciphertext using the derived HMAC key. Compare it with the stored HMAC to detect tampering.

7. If the HMACs match, decrypt the ciphertext using the encryption key to retrieve the serialized Usermetadata.

8. Unmarshal the plaintext into a Usermetadata struct.

9. Return a non-persistent User struct that includes the username, root key, hashed password, and FileAccessMap.

StoreFile(filename string, content []byte)

1. Hash the filename and take the first 16 bytes to create a filename key.

2. Generate a random encryption key and a random IV. Encrypt the file content using these.

3. Generate a random HMAC key and compute the HMAC over the encrypted file content.

4. Create a FileContent struct containing the encrypted file data, the computed HMAC, and a nil NextBlockUUID (indicating the end of the file).

5. Serialize and store the FileContent struct at a new random UUID in the Datastore.

6. Create a FileMetadata struct containing metadata such as the owner's username, the file's starting block UUID, the encryption and HMAC keys, an empty SharedUsers map, the version history initialized with the first block, and block count information.

7. Serialize the FileMetadata, encrypt it with a new random encryption key, compute an HMAC over it with a new random HMAC key, and store the concatenation [HMAC || ciphertext] at a new random UUID.

8. Add an entry into the User's in-memory FileAccessMap mapping the hashed filename key to the corresponding AccessInfo (metadata UUID, encryption key, HMAC key).

9. Fetch and decrypt the latest Usermetadata from the Datastore to keep consistency. Update the FileAccessMap in the persistent Usermetadata with the new file's access info.

10. Serialize, encrypt, and HMAC the updated Usermetadata and overwrite the original Usermetadata in the Datastore.

LoadFile(filename string)

1. Hash the filename and take the first 16 bytes to create a filename key.
2. Retrieve and decrypt the latest Usermetadata from the Datastore by deriving the UUID from the username. Verify the HMAC to ensure integrity.
3. Look up the AccessInfo corresponding to the hashed filename key in the FileAccessMap. Return an error if it does not exist.
4. Retrieve and decrypt the FileMetadata from the Datastore using the MetadataUUID found in the AccessInfo. Verify the HMAC for integrity.
5. Starting from the initial FileUUID recorded in FileMetadata, traverse the linked list of file blocks:
    - For each block, retrieve and unmarshal the FileContent struct from the Datastore.
    - Verify the block's HMAC to ensure data integrity.
    - Decrypt and append the plaintext file data.
    - Follow the NextBlockUUID if it exists; otherwise, stop when nil is reached.
6. Return the assembled plaintext file data.

AppendToFile(filename string, content []byte)

1. Hash the filename and derive the filename key using the first 16 bytes.
2. Retrieve and decrypt the latest Usermetadata from the Datastore by deriving the UUID from the username. Verify the HMAC to ensure integrity.
3. Look up the AccessInfo corresponding to the hashed filename key in the FileAccessMap. Return an error if the file is not found.
4. Retrieve and decrypt the FileMetadata from the Datastore using the MetadataUUID in the AccessInfo. Verify the HMAC for integrity.
5. Encrypt the new content with the file's encryption key, and compute its HMAC with the file's HMAC key.
6. Create a new FileContent block with the encrypted data, HMAC, and no next block pointer. Marshal it and store it at a fresh UUID in the Datastore.
7. Retrieve the last block in the file's linked list. Update its NextBlockUUID to point to the newly created block, then marshal and store the updated block back to the Datastore.

8. Update the FileMetadata:
    - Set LastBlockLoc to the new block’s UUID.
    - Increment NumBlocks.
    - Append the new block’s UUID to the VersionHistory.
9. Marshal the updated FileMetadata, encrypt it with the AccessInfo encryption key, compute the new HMAC, and overwrite the old metadata in the Datastore.
10. Return successfully.

CreateInvitation(filename string, recipientUsername string)

1. Retrieve and decrypt the latest Usermetadata from the Datastore using the user’s derived UUID. Verify the HMAC to ensure integrity.
2. Hash the filename and derive the filename key using the first 16 bytes. Look up the AccessInfo for this file in the FileAccessMap. Return an error if the file is not found.
3. Retrieve the recipient's public encryption key from the Keystore using the key "recipientUsername_enc". Return an error if the key is not found.
4. Create a SharedAccessNode struct containing:
    - The owner's username.
    - The MetadataUUID, encryption key, and HMAC key from AccessInfo.
5. Marshal the SharedAccessNode to bytes, generate a random symmetric key, encrypt the SharedAccessNode bytes with the symmetric key, and then encrypt the symmetric key using the recipient's public encryption key.
6. Concatenate the encrypted symmetric key and the encrypted SharedAccessNode to form the final invitation payload.
7. Store the invitation payload in the Datastore under a fresh random UUID.
8. Retrieve and decrypt the FileMetadata for the file using the AccessInfo. Verify the HMAC for integrity.
9. Update the FileMetadata's SharedUsers map to include the recipient’s username and the new invitation UUID.

10. Marshal the updated FileMetadata, encrypt it with the AccessInfo encryption key, compute the HMAC, and overwrite the metadata in the Datastore.
11. Return the UUID where the invitation payload was stored.

AcceptInvitation(senderUsername string, invitationPtr uuid.UUID, filename string)

1. Retrieve and decrypt the latest Usermetadata from the Datastore using the user's derived UUID. Verify the HMAC to ensure integrity.
2. Update the in-memory FileAccessMap to match the latest persistent Usermetadata for consistency.
3. Hash the new filename and derive the filename key using the first 16 bytes. Check if this filename already exists in the user's namespace. If it does, return an error.
4. Retrieve the invitation payload from the Datastore using invitationPtr. Verify the payload is long enough (at least RSAKeySizeBytes).
5. Parse the invitation payload:
    - The first RSAKeySizeBytes are the encrypted symmetric key.
    - The remainder is the encrypted SharedAccessNode.
6. Unmarshal the private decryption key (SKEnc) from the stored PrivateKeys in Usermetadata.
7. Decrypt the symmetric key using the recipient’s private decryption key.
8. Decrypt the SharedAccessNode using the decrypted symmetric key.
9. Unmarshal the SharedAccessNode struct containing:
    - OwnerUsername
    - MetadataUUID
    - Encryption key
    - HMAC key
10. Validate the sender:
    - Check that OwnerUsername matches senderUsername.

- Ensure that the sender is not the same as the recipient (a user cannot accept their own invitation).

11. Add a new entry to the FileAccessMap under the new filename, pointing to the shared file's metadata and keys.
12. Marshal, encrypt, and HMAC the updated Usermetadata and overwrite the user's record in the Datastore.
13. Delete the invitation payload from the Datastore so it cannot be reused.

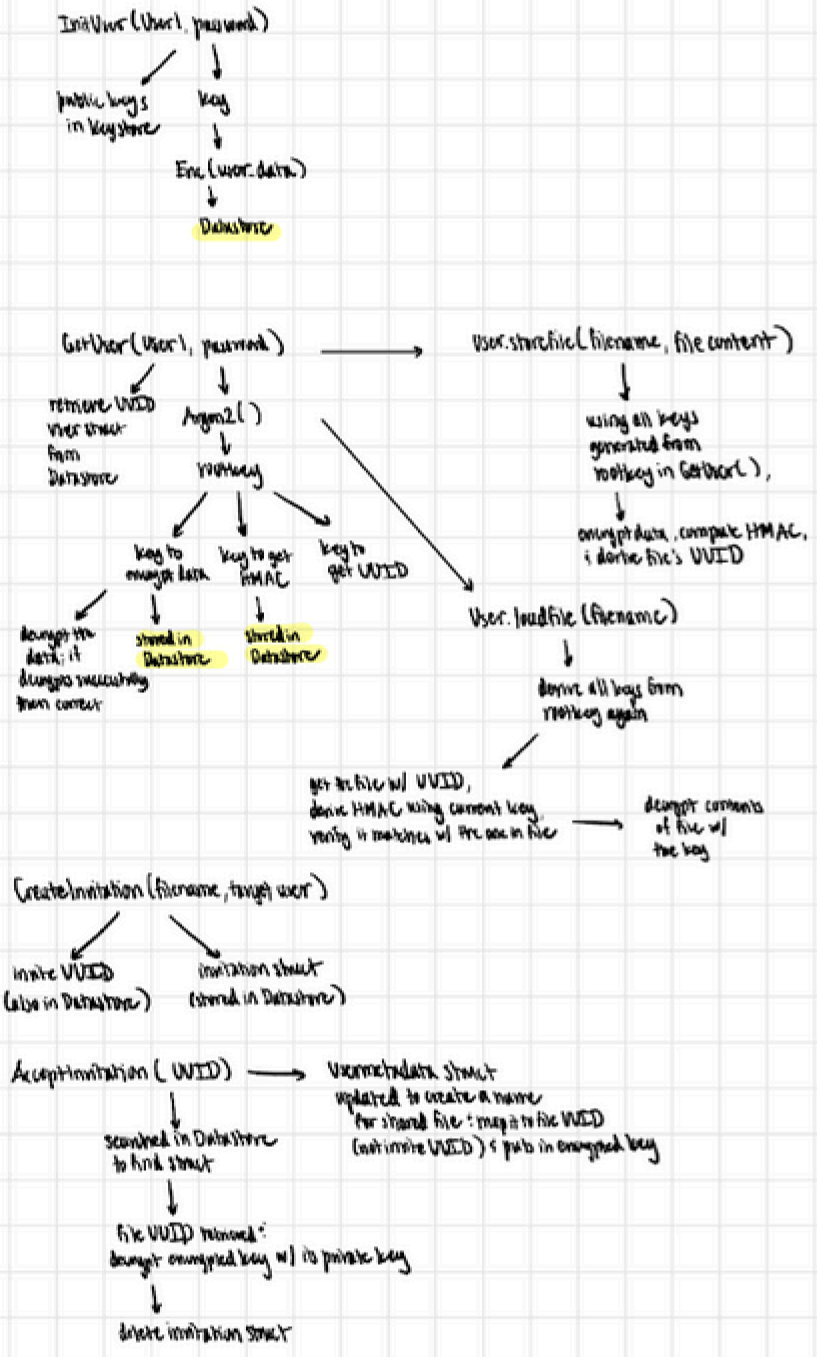
RevokeAccess(filename string, recipientUsername string)

1. Hash the filename and derive the filename key using the first 16 bytes of the hash.
2. Retrieve and decrypt the user's latest Usermetadata from the Datastore using their UUID. Verify the HMAC to ensure integrity.
3. Find the AccessInfo for the file using the derived filename key from the FileAccessMap.
4. Retrieve and decrypt the FileMetadata from Datastore using AccessInfo.MetadataUUID. Verify the HMAC to ensure integrity.
5. Verify that the caller is the owner of the file by comparing FileMetadata.OwnerID to the caller's username. If not the owner, return an error.
6. For each user in FileMetadata.SharedUsers:
    - Delete the corresponding invitation UUID from the Datastore.
7. Clear the SharedUsers map in FileMetadata to remove all prior shares.
8. Delete the old FileMetadata record from the Datastore to invalidate stale pointers.
9. Generate a fresh MetadataUUID, a new symmetric encryption key, and a new HMAC key for the file.
10. Marshal the updated FileMetadata (with cleared SharedUsers), encrypt it with the new key, and HMAC it.
11. Store the new FileMetadata into the Datastore under the new MetadataUUID.
12. Update the FileAccessMap to point the filename to the new MetadataUUID and new keys.
13. Marshal, encrypt, and HMAC the updated Usermetadata, and overwrite the user's record in Datastore.

Project 2 Diagram

Structs: (in Datastore)

```
struct User {
    String username
    Bytes password hash
    Bytes rootkey }
struct Usermetadata {
    String username
    Files
    Private key }
struct Filemetadata {
    owner_ID
    file_UUID
    encryption key
    shared users
    version history
    num blocks
    last block location }
struct FileContent {
    file data
    HMAC }
struct Invitation {
    owner username
    other username
    File UUID
    Encrypted key }
struct Append {
    parent file UUID
    encrypted chunk
    HMAC }
```



InitUser(User1, password)
public keys in keystore
key
Enc(user.data)
Datastore
GetUser(User1, password)
retrieve UUID user struct from Datastore
Argon2( )
rootkey
key to encrypt data
key to get HMAC
key to get UUID
decrypt the data; it decrypts successfully then correct
stored in Datastore
stored in Datastore
User.storefile(filename, file content)
using all keys generated from rootkey in GetUser( ),
encrypt data, compute HMAC, & derive file's UUID
User.loadfile(filename)
derive all keys from rootkey again
get the file w/ UUID, derive HMAC using current key, verify it matches w/ the one in file
decrypt contents of file w/ the key
CreateInvitation(filename, target user)
invite UUID (also in Datastore)
invitation struct (stored in Datastore)
AcceptInvitation(UUID)
Usermetadata struct updated to create a name for shared file & map it to file UUID (not invite UUID) & puts in encrypted key
scanned in Datastore to find struct
file UUID retrieved & decrypt encrypted key w/ its private key
delete invitation struct