System Design Outline

Authentication Functions

- CreateUser: CreateUser will first verify that a username is not already stored on the dataserver. If not, we will create a user with a username, password, salt, private key, and their associated User Class. The User Class will look as follows:

```
User {
      listOfFiles: [[filename1, owner], [filename2, owner],...]
      listOfFileKeys: {[filename,username]->[list of keys for datablocks]}
      // a “datablock” is a unit of appended data to the file
      messages: [sender_username, E(list of file’s keys)_{receiverpubkey},
      digitalSignature(sender), shared_filename]
}
```

  The listOfFiles and listOfFileKeys will each be encrypted with a different key, which will be generated using PasswordKDF as well as a random salt, uniquely associated with each user. Thus, at memloc(Username), there will be the following data:
    - User class
    - HMAC of password
    - Salt for listOfFiles key
    - Salt for listOfFileKeys
    - Salt for HMAC generation
    - Salt for integrity check
    - Integrity check (described in next section)
    - Merkle Tree Key, encrypted with a user’s public key
    - Merkle Tree Hash
    - Salt for Merkle Tree Hash

- AuthenticateUser: To authenticate a user, we will check if the given username and password match what is in the dataserver. If so, we should also do another check to make sure someone has not tampered with our user data. We will store another verification check which is the entire user entry in the dataserver, and then encrypt then HMAC the row. That way, if someone tampers the data, we can look up our verification check and see that it does not match.

Storage Methods

To store our files, we will have a (memloc(<filename>-<owner’s username>), file_contents) schema where a filename consists of the filename and the user’s username. We will encrypt the file_contents, which will be a list of the different blocks of data (each block is an append). Specifically, a file will be of the form:

    [E(metadata), E(d0), E(d1), …]

The metadata will include the root user, and a list of users who have access to the file. d0 is the unencrypted contents of the upload, d1 is the unencrypted contents of the first append, d2 is the unencrypted contents of the next append, and so on. Each block is symmetrically encrypted with a separate key K_i, which the user has stored in their mapping of filenames to keys.

- UploadFile: When we upload a file, we will check to see if a file of the same name is owned by the same user. Then, we will encrypt the file and store it at the proper memory location.
- DownloadFile: Unencrypt file and download.
- AppendFile: The file will be stored such that each append is symmetrically encrypted using a separate key. That way, when we append, we only have to encrypt the bytes that will be appended to the file and append that to the list of data blocks that comprise a file. Therefore, a user will have to keep a list of separate keys for each block of data. This will run in O(nu) time, where n is the length of the file block to append, and u is the number of users.

Sharing Methods

- ShareFile: To share a file with a user, we will use an efficient key sharing mechanism where a file has a key, but to share the key with each user who should have access, we will encrypt the key with their public key. That way, each user will respectively have to decrypt the file key with their own private key. The owner of the file will have access to the recipient's "messages" and add in the key encrypted with the recipient's public key.
- ReceiveFile: When a user logs in, they will call receiveFile and then check their messages to see a new key. When this happens, the user can decrypt the key, and then use the key to decrypt the file and add it to their list of files. To check if the file has been corrupted, we will check the Merkle Tree. ** detail more about the merkle tree **
- RevokeFile: When a user is revoked from a file, we need to issue new keys for the file. This requires sending out new keys to each user, which should follow the same pattern as ShareFile. That way, the user who was removed does not have access anymore, but everyone else does.

<u>Odds and Ends</u>

- File verification: There will be a global Merkle tree for File verification, and each user will store the final hash in their user class, which will be symmetrically encrypted using a salt and a PasswordKDF key. The Merkle tree will be the result of hmac-ing each file block using the Merkle tree key (not each file, since files are stored in separate blocks), and anytime a user modifies a file, they will modify each of the nodes of the Merkle tree above the block they modify. They will then store the resulting hash in their user class. Before a user accesses data, they will always recompute part Merkle tree (specifically the parts involving their files), and then compare the resulting hash to the one that they have stored. If the Merkle tree has been changed, then the user will know because hmac values do not collide.

<u>Possible Attacks & Mitigations</u>

1. UserA creates a file named file1. UserB attempts to overwrite this, by also creating a file named file1 and sharing it with UserA. This attempt to overwrite userA's file1 fails, even after userA calls receiveFile(). This is because the two files have different names; file1 is stored at memory location E(file1), and file2 is stored at E(file1), where each encryption is a different symmetric encryption with a different private key. In addition, userA has a list of available files stored as a map of [filename, owner] pairs -> keys. Thus, two files with the same name from different users can't overlap.

2. UserA shares file *file1* with UserB, and UserB receives this file. UserB then attempts to attack UserA by calling revoke_file on *file1*. This attack is prevented because the file stores metadata, which is encrypted and includes the root user. When revoke_file is called, the metadata is encrypted and the username of the calling user is checked against the username of stored on the dataserver.
3. UserA shares a file with UserB, but UserB has not yet called receive_file(). In our system, that means that UserB has a message waiting for them in their message list. A malicious user deletes this last message. This attack fails, as in our system, each message list is treated as a file, and each message is a data block that is hashed in our Merkle Tree. Thus, when UserB calls receive_file(), a util.dropboxError will be returned, because the Merkle Tree will not match the files currently on the dataserver.
4. User A has an account userA with a secret password. The hmac of this password is stored on the dataserver. A malicious user attempts to overwrite the password's hmac with the hmac of a password of their choice, so that when they attempt to login using username userA, their login request is approved. This attack fails, because when userA logs in, all of their user information (username, salts, hmac of password) is hashed using the Password_KDF(salt, password), and compared to a check-value also stored on the dataserver. Since the malicious user doesn't know A's original password, they can't compute this checksum for this attack.
5. User A owns a file, File1 on the filesystem. User B has been revoked access from File1 (therefore knowing the filename) and wants to spoof a message to themself to gain access to File1. To do this, User B appends to their own messages list. [UserA_username, E(list of File1's keys)_{UserBPubKey}, digitalSignature(UserA), File1]. Even if UserB had copied over User A's digital signature from a time when User B was still shared on the file, User B does not have access to the list of File1's new Keys, which are generated at the time File1 access was revoked from User B. Therefore, even if User B spoofs a message, they will not have access to File1 again.