Raymond Guo, 3032719978
Kush Khanolkar, 3032754935

CS161 Project 2 Design Document

Section 1: System Design

We implemented auth and deauth to encrypt and decrypt Users and Files when uploading and retrieving from the Datastore. In auth, we SymEnc with key1 and a 16 bit random IV, then HMAC with key2 and upload both the ciphertext and its HMAC to the Datastore at UUID. In deauth, we retrieve from Datastore at a given UUID, ensure that the HMAC is correct, then SymDec with key1 to get the original message.

We defined User as a struct that contains username, encryption key (public), decryption key (private), sign key (private), verification key (public), and a map of filename to file info (file UUID and its keys).

When we initialize a new User, we generate key1 using Argon2Key, passing in password (our only source of entropy) and salting with username. Key1 is used to symmetrically encrypt, ensuring confidentiality. We use HKDF to generate two additional random values, salting with password and username. These two values become key2 and UUID. All three values (key1, key2, and UUID) have sufficient entropy such that each can only be obtained if the username and password are known. We then construct a new User struct, adding its username and password, generating its private and public keys, and inserting an empty file map. We put the two public keys (encryption and verification keys) into the Keystore, using the username + “enc” / “ver” as the key, respectively.

We use key1, key2, and UUID to upload Users to the Datastore (keyed by UUID), calling auth. Whenever a User logs in, we call deauth using the given username and password. If the username and password are correct, then the User is successfully decrypted; otherwise, and error will return.

To implement Files, we adopted a nested structural approach. Namely, we have two datatypes: the File and the Segment. A File has an associated filename, key1, key2, and UUID, but it does not contain any of the file contents. Instead, it contains all of the info (UUID, key1, key2) of a list of Segments. Each Segment has a UUID, key1, key2, and also a portion of the contents of the File. By doing this, we ensure that whenever AppendFile is called, we don’t have to load the full File; instead, we can generate a new Segment, upload the Segment to the Datastore, and modify the File to contain the new Segment. This process is much more efficient, especially for large Files.

We generate key1, key2, and UUID of Files and Segments completely randomly, using RandomBytes. Thus, nothing can be revealed about filename from these values. To associate filename to its info, we made it so that every User who has access to the file has a file map that maps filename to the associated File’s UUID, key1, and key2. Similarly, each Segment’s UUID, key1, and key2 are stored in its associated File, so only users who have access to the File can determine the info of its Segments (and thus decrypt the file contents). Using this approach allows the same File to have different filenames across different Users, despite being in the Datastore only once.

Thus, the process of storing a new File is the following: Construct a new Segment and put in the file content. Generate key1, key2, and UUID of the Segment randomly. Then SymEnc the Segment (calling auth) and put it in the Datastore. Construct a new File and add the Segment’s UUID, key1, and key2 to its list of Segment info. Generate key1, key2, and UUID of the File randomly. Then SymEnc the File (calling auth again), and put it in the Datastore. Then add a new (key, value) pair of (filename, {File UUID, File key1, File key2}) to the User’s file map, and update the User to the Datastore.

When loading a file, look in the User's file map, retrieving the File UUID, key1, and key2 from its filename. Then retrieve the File from the Datastore, deauth it, and iterate through the list of Segments' info, retrieving the corresponding Segment from the Datastore. Deauth it, and append its content to a string variable. Return the string variable, which should be all of the Segment contents appended to each other.

To share a file with another User, a User simply has to take the File's UUID, key1, and key2, and encrypt it with the receiving User's public encryption key, then sign it with his/her own signing key to create the magic string. The public encryption key can be retrieved from the Keystore, which is keyed by the receiver's username + "enc". The receiver then ensures integrity and authenticity by verifying the sender's signature (once again, by getting the sender's public verification key from the Keystore), before decrypting the magic string with his/her own private decryption key and getting the File UUID, key1, and key2. Then the receiver can add this new File info to his/her own file map, with his/her own filename.

When revoking a file, the User creates a new File, a new set of keys and UUID, then retrieves the original File from the Datastore, copying over the list of Segments' UUIDs and keys from the original File to the new generated File. Then the User modifies his/her own file map with the new File's UUID and keys, deletes the original File from the Datastore, and uploads the new File to the Datastore. Because all other Users now have a wrong UUID and set of keys of the File, they cannot access the File anymore.

We tested each of these functionalities (loading, storing, sharing, revoking, and appending) by creating unit tests that tested just that function. We then combined multiple functionalities in other tests to make sure that they worked in conjunction with each other. Finally, we also created a few edge cases (such as changing the magic string, thus simulating a Mallory that tampered with the magic string, or trying to a receive a File with a filename that the receiving User already used) that would cause errors, and made sure that the program would error out correctly.

Section 2: Security Analysis
One potential attack on this system would be a man-in-the middle attacker during the file sharing step. This man in the middle could take Alice's file magic string and instead send Bob the magic string of a file that they own that may look like the original file. Thus when Bob edits the file the attacker would have access to all these changes. Our system prevents this by sending a digital signature concatenated into our file magic that has a unique RSA Digital signature key that only Alice could produce. Thus while the attacker can encrypt any file using Bob's public key, in our Receive File function a malicious file magic string would result in DSVerify throwing a authenticity error if it was not from Alice.

Both passive and attacks attacks on the Datastore do not work, because our encryption scheme ensures complete confidentiality with symmetric encryption (CTR mode with a random IV) and authenticity and integrity with HMAC. All keys are sufficiently entropic, so any tampering can be detected, and no deciphering can occur.

Another attack would be a replay attack. The attacker could try to load a revoked file by reloading the magic string. However, our encryption scheme does not allow this to happen, because when a file is revoked, its given a completely new set of UUIDs and keys, and the file at the original UUID is completely deleted from the Datastore. Thus, nobody other than the person who revoked the file can access the file.

Another attack could be a type of phishing attack where an attacker could request a non-sensitive document and try to use this to access a more sensitive document. However, because each file's UUID and keys is completely random and independent, our encryption scheme prevents this.