Darren Sun
Adithya Kumar
CS161 SU22

CS161 Securities Project 2 Design Document

Initial Proposal and Brainstorm for **Checkpoint**

Data Structures

What data structures are you going to use? List any struct definitions that you plan on including, along with the attributes these structs will contain. We'd recommend starting with a few core data structures (e.g. struct user, struct file, etc.), and adding additional attributes and structs as you need them.

The data structure we are going to use for storing and adding files is a linked list. A linked list is appendable and basically acts as a cloud for all the files ever put into the environment. The linked list will contain the hash of the PublicKey of the user, the UUID at which the file contents were stored in the datastore, and the symmetric encryption of the file contents that were uploaded to the linked list.

There will also be User and State Structs defined as follows:
Struct User:
- Username
- Password
- PublicKeys
- PermissionFrom
- PermissionTo
- UUIDsGiven
- UUIDsTaken

Struct State:
- LinkedList<E(pk), UUID, E(File contents)>

User Authentication

How will you authenticate users? What information will you store in Datastore/Keystore for each user? How will you ensure that a user can have multiple client instances (e.g. laptop, phone, etc.) running simultaneously? Relevant Client API Methods: InitUser, GetUser

Datastore is an untrusted server that is mapped with key value pairs. Each key is a unique UUID, stored with a symmetrically encrypted value of the file contents that were uploaded. This UUID needs to be updated to a specific User's instance attribute UUIDsGiven so that they can then have easier access to that file in the datastore going forward. In authenticating different users, if they want to access a specific State in the linked list and Datastore for decryption, they will have to pass themselves and the User who uploaded the file to a helper function that makes sure proper PermissionFrom and PermissionTo attributes were set. If this is true, we then search through the linked list to see which files were uploaded by that uploader's public key. The linked list is made of the encryption of the user's public key, the UUID the files is posted at in the datastore, and the symmetrically encrypted file contents. We can then search through their UUIDs given and taken to find matches within Datastore. Once they do that, the reader, who was requesting the file can use the decryption scheme to read the value in that UUID index of datastore if proper permissions were established. Anytime someone requests to read files, it can be thought of as a pull request, if the original poster of the file has given them access, they can read it, if they've never "met" before the reader is denied access and we return error. All public keys storage history is encrypted before updated to the User's possessed attribute and the keystore.

File Storage and Retrieval

How will a user store their files? How will a user retrieve their files from the server? How will your design support efficient file append? Relevant Client API Methods: LoadFile, StoreFile, AppendToFile

When a user wants to store a file, we encrypt their public key, and the contents of the file, and append this into the linked list. We then also compute a UUID for this file storage event and upload it to Datastore with the symmetric encryption of the file content. This UUID will also be put in the linked list. When a User tries to access the Datastore, we check if they have posted any files, since if they did they'd be in the linked list. If not, we proceed to check their permissionsTo and permissionsFrom attributes to see if they have been shared with the file before from the owner. If the accessing knows which UUID they want to open already, they can access it after the previous check from the Datastore directly. If they have proper permissions, but don't know which UUID they want, we look through the matching UUIDsGiven and UUIDsTaken attributes of the Uploader and Requester and then pull all of those from the DataStore to decrypt. If the Requester's public key is the Uploader's public key, or if the Requester's permissionFrom attribute is a match, they can decrypt that file.

File Sharing and Revocation

How will a user share files with another user? How does this shared user access the shared file after accepting the invitation? How will a user revoke a different user's access to a file? How will you ensure a revoked user cannot take any malicious actions on a file? Relevant Client API Methods: CreateInvitation, AcceptInvitation

If a user is given access to another user's files, they become a "child" node of the original user. When different users give access to each other, they can share the UUIDs they give access to and update the permissionsTo, permissionsFrom, UUIDsGiven, and UUIDsTaken attributes appropriately. The system relies on direct trust between sender and receiver.

Helper Methods

Are there helper methods that you'll create? There are a few in particular that may help simplify your implementation. Hint: think about authenticated encryption.

1) Decrypting the Datastore value
2) Checking permissions
3) Search through Datastore
4) Search through linked list

Tests To Be Written

1) Design Requirement: The number of public keys should not depend on the number of files stored [3.3.2]
   Define `CountKeystore()`, a method that counts the number of keys in Keystore.
   Create one user.
   `X = CountKeystore()`
   Store 100 files.
   Expect that the number of public keys didn't change (expect `CountKeystore() == X`)
2) Keystore Initiation, Traversal
3) Datastore Initiation, Traversal
4) Linked List Traversal
5) User Permissions Update Check
6) User Permissions Revocation Check

Final Project 2 DropBox Design Document

Data Structures

What data structures are you going to use? List any struct definitions that you plan on including, along with the attributes these structs will contain. We'd recommend starting with a few core data structures (e.g. struct user, struct file, etc.), and adding additional attributes and structs as you need them.

The data structure we are going to use for storing and adding credentials is a tree. Our trees map and find the proper sharing credentials of files, and file content are stored in the Datastore. User keys are remembered in the keystore.

Our structs are defined as follows:

```
User struct {
   Username string
   //Data store namespace uuid pointer
   NamespacePtr uuid.UUID
   //Key for symmetric file encryption
   PassKey []byte
   //Private PKE Key for decryption
   PKEKey userlib.PKEDecKey
   //Private signing key
   DSKey userlib.DSSignKey
   //Symmetric key to decrypt encrypt namespace
   NSKey []byte
   //Symmetric key to hmac namespace
   NSHmacKey []byte
```

```
File struct {
   FileUid        uuid.UUID
   ShareTree      Tree
   ContentPointer uuid.UUID
}
```

```
Tree struct {
   Username    string
   Children    []Tree
   InviteToken uuid.UUID
}
```

```
Invitation struct {
   Sender    string
   Recipient string
   FileUID   uuid.UUID
   SymmKey   []byte
   HMACKey   []byte
}
```

```
SigningPair struct {
   Data      []byte
   Signature []byte
}
```

```
InviteSigning struct {
   Invitation   []byte
   InvitationDS []byte
   UnlockKey    []byte
   UnlockKeyDS  []byte
   Signer       string
}
```

```
FileSigningPair struct {
   FileData           []byte
   FileHMAC           []byte
   ContentPointer     []byte
   ContentPointerHMAC []byte
}
```

User Authentication

How will you authenticate users? What information will you store in Datastore/Keystore for each user? How will you ensure that a user can have multiple client instances (e.g. laptop, phone, etc.) running simultaneously? Relevant Client API Methods: InitUser, GetUser

InitUser initializes the user struct, generates the PKE and DS public and private keys, and stores the public key to datastore at Key username + "_[ds or pke]" We then compute a passKey (see User) from the password and username as the salt. This passKey is used to derive a HKDF key which is hashed and stored in the datastore at the UUID calculated by hashing the username II password and converting to a UUID from bytes. This is the key-value pair we calculate with the password in GetUser to verify correct information is entered.

File Storage and Retrieval

How will a user store their files? How will a user retrieve their files from the server? How will your design support efficient file append? Relevant Client API Methods: LoadFile, StoreFile, AppendToFile

StoreFile first checks whether the filename currently exists in the namespace (after fetching, verifying, and decrypting). If it does not, we create a new File (see Files) and a new invitation addressed with both the current user as the user and the recipient. It saves the file content, the file struct, and the invitation to the Datastore. If it does, then it fetches the current files and overwrites the old content pointer with the new constructed file content. LoadFile fetches the invite access token from the namespace and uses the invitation after decrypting it with the user's private key to decrypt the file. It then fetches and verifies all the file content pointers, appending the decrypted data in reverse order to account for the appends. Append File loads the invitation and associated file. It creates a new file content pair using the current file pointer as the file pointer in the new struct. It then creates a new UUID updating the to the file struct with that pointer and setting the file content at the UUID in the Datastore.

File Sharing and Revocation

How will a user share files with another user? How does this shared user access the shared file after accepting the invitation? How will a user revoke a different user's access to a file? How will you ensure a revoked user cannot take any malicious actions on a file? Relevant Client API Methods: CreateInvitation, AcceptInvitation

ShareFile creates a new file invitation (see Invitations) signing each sender's and asymmetrically encrypting with the recipient's key. We then create a new node under the sender's TreeNode of the File's share tree. ReceiveFile verifies the invitation, that the sender and intended recipient are correct, and that file is not current in the namespace. It then adds the filename to the User's namespace.

RevokeFile creates a new symmetric and hmac key used to encrypt / sign the file struct and file contents. This key is then used to re-encrypt the file and file contents. We then parse the file Share Tree to remove the revoked user and revoked user's shared children from the tree, additionally deleting their invitations from the datastore. Finally, we update all the remaining invitations of shared users with the new hmac and symmetric key, signing all invite pairs with the owner's digital signature.

Helper Methods

Are there helper methods that you'll create? There are a few in particular that may help simplify your implementation. Hint: think about authenticated encryption.

1. Decrypting the Datastore value
2. Checking permissions
3. Search through Datastore
4. Finding, recognizing, and fetching public keys in keystore
5. Search through the sharetree
6. JSON marshal/unmarshal

Tests Written

1. TestSetupAndExecution
2. InitUser/GetUser on a single user
3. Single user Store/Load/Append
4. Create/Accept Invite functionality with multiple users and multiple instances
5. Revocation
6. GetUser with wrong password
7. File storage and retrieval via User.LoadFile(<<name str>>)
8. Appending and Modifying File test with Sharing and Revocation
9. Unshared user trying to modify a file that belongs to a different owner
10. Sharing and Revokes
11. Test share multiple files with one User
12. Test share and revocation, then modify file and check reflected changes
13. Test non unique filenames
14. Test empty filename