CS 161: Project 2 Design Document
William Henderson and Nico Galin

Data Structures

```
struct PublicUser {
      saltPW []byte: Random salt for the password
      Tag []byte: HMAC Tag for verifying integrity of PublicUser
      EncryptedUser []byte: Encrypted byte slice of User struct
      hashedPW []byte: Hash(password || saltPW)
}
struct User {
      Username string: User's username
      RSAPrivateKey userlib.PKEDecKey: user's rsa private key
      DSSignKey userlib.DSSignKey: user's private key for signatures
      hashedPassword []byte: Hash(password || saltPW)
}

struct File {
      Tag []byte: HMAC Tag for verifying integrity of PublicUser
      Head userlib.UUID: UUID of first node in file
      Tail userlib.UUID: UUID of last node in file
      Size int: length in nodes of File struct
}

struct Node {
      Tag []byte: HMAC Tag for verifying integrity of PublicUser
      Blob userlib.UUID: UUID Blob struct associated with this Node
      Next userlib.UUID: UUID of next Node in the file
      HasNext bool: boolean to indicate if there is another node in the file
}

struct MiddleStructWrapper {
      EncryptedMiddleStruct []: Encrypted byte slice of MiddleStruct struct
      Owner string: username of the file owner
      OwnerTag []byte: HMAC Tag for verifying integrity of Owner string
      Signature []byte: Digital Signature for verifying integrity of
                        EncryptedMiddleStruct
}
```

```
struct MiddleStruct {
      FileId userlib.UUID: UUID of File struct associated with this file
      FilePassword []byte: Root password used to encrypt all parts of File
                           struct
}

struct SignedEncData {
      EncryptedData []byte: Byte slice of data that has been encrypted
      Tag []byte: HMAC tag or digital signature verifying integrity of
                  EncryptedData
}

struct Invitation {
      FileId userlib.UUID: UUID of File struct associated with this file
      FilePassword []byte: Root password used to encrypt all parts of File
                           struct
      Tag []byte: Digital signature verifying integrity of FileId and
                  FilePassword
}
```

User Authentication

Relevant Client API Methods: InitUser, GetUser

How will you authenticate users?

- The user enters a username and password. In the datastore, the PublicUser object is stored with the key H(user/{username}), so if the key does not exist with the given username, the user does not exist. We also store the initialized password, randomly salted and hashed (with a public salt value), so the given password can be cross-checked with the stored salted & hashed password to authenticate.
- In order to change the username, the adversary must change the key of the PublicUser struct and it would no longer appear under the original key, returning an error message.
- Inside of the PublicUser, we have an encrypted User struct that stores the username, hashed password (with the salt), RSAPrivateKey, and DSSignKey for the user. This encryption is also HMAC'd so we would be able to see if an adversary tried to change it.

What information will you store in Datastore/Keystore for each user?

- In the keystore, we maintain (at least) two values for each user, (1) their RSA public key used for encrypting file access passwords, and (2) their digital signature public key to verify the authenticity of any data they sign.

- In the datastore, we maintain a PublicUser struct containing their password salt value, their hashed and salted password, an encrypted and HMAC'd User struct that has their private keys (RSA and Digital Signature), Username, and HashedPassword.
- Additionally, each user has a MiddleStructWrapper stored in the datastore for each file they can access. This is stored at H(user-filemap/{username}/{filename}), and contains the original file owners name, an HMAC over the file owner's name to ensure integrity, an Encrypted version of the MiddleStruct struct, and a Signature over the Encrypted MiddleStruct struct to verify its integrity. This MiddleStruct Struct contains the UUID of the file that is associated with it, and that File's Password that is used to encrypt all parts of the file. The MiddleStruct struct is encrypted using the user's public rsa key from the data store so that the original file owner can change it and re-encrypt it with the user still being able to decrypt it again.

How will you ensure that a user can have multiple client instances running simultaneously?

- Because there is no data stored locally and all the decryption required to access and modify files depends on static values in the user object alongside dynamic values in the file object(s), there will be no issue with race conditions, differing user instances, or any client requests.

File Storage and Retrieval

Relevant Client API Methods: LoadFile, StoreFile, AppendToFile

How will you store and retrieve files from the server?

- The datastore will store File structs under a random UUID, which is stored in the associated MiddleStruct struct of users who have access to the file. The content of files that are uploaded to the datastore are stored in blobs, where each append adds a new blob. A blob is a version of the SignedEncData struct, with encrypted data using a derivation of the FilePassword from HashKDF, and HMAC'd for integrity. Each blob has a pointer to it stored in a Node Struct, which also points to the next Node in the file if it exists. This separation between Blob and Node allows for access to Nodes to happen in constant time, regardless of any file size or blob size.
- To retrieve files, We simply loop through all of the Nodes in a File, and download their content that are stored in each blob. We then append all of the content together from each blob and return that to the user.

How will your design support efficient file append?

- In our File struct, we have a pointer to the Tail UUID of the last Node in the file struct. The size of the tail node does not depend on the size of the underlying data it points to, so downloading it does not

incur a large use of resources. We update this tail node to point to a new node that we create that points to a blob with the encrypted and HMAC'd content that we want to append, and change its HasNext bool to true. Finally we update our original File struct so that the Tail uuid now points to the new node that we have created.

File Sharing and Revocation

Relevant Client API Methods: CreateInvitation, AcceptInvitation

How will you allow files to be shared with other users?

- Each user that has access to a file has its own children map[string]uuid, that maps between its children's username and their UUIDs of an intermediary middlestruct wrapper. This intermediary middlestruct wrapper is created upon creation of an invitation, and is located at a predictable location (H(user-childrenmap/{username}/{fileId}). We use a predictable location so the user and the original file owner can both access the children map without having to store it in either the MiddleStruct struct or User struct.
- When a file is shared, we create an Invitation Struct that has the FilePassword, FileId, and original owner username stored in it. This allows the share-ee to create their own middle struct upon acceptance of the file. The share-ee then creates a MiddleStructPointer that is encrypted with the original file owner's public key, and stored at H(user-inv-map/{username}/{fileId}). This allows the owner to know the location of the share-ee's middleStruct, which is needed when updating their middle struct with a new FilePassword and FileId upon revocation.

How will you manage file revocation?

- As mentioned above, each user with access to the file has a children map, which stores all of the users they have shared the file with. When the original file owner wants to revoke access to the file, they copy the file over to a new location in the datastore, and then encrypt it with a new FilePassword. Then, they delete the child that they are revoking access to from their children map, and loop through the rest of their children. For each child, they update their middle struct to have the newFileId and newFilePassword, re-encrypt and sign it, then loop through all of their children and do the same thing. This repeats recursively to update all MiddleStruct structs of all users who have access to the file, including the original owner.
- This structure prevents the revoked user from knowing where the new file is stored at, and deletes the old struct so if they try to loadFile it would fail.

- Because the revoked child was deleted from the children map before looping, all children of the revoked child will not have their MiddleStruct structs updated with the newFilePassword and newFileId, so they will also not be able to access the file after their parent has had their access revoked.

How will you ensure a revoked user can't take any malicious actions on a file?

- Because the file password is changed upon revocation and the HMAC keys are recomputed, the revoked users have absolutely no way of reading the file data or computing the proper HMAC to modify the file data and metadata. Further, since the fildId has changed, the revoked user will not know the location of the file in the datastore and will not be able to know when the file is modified.

Helper Methods

func (userdata *User) GenerateNodeTag(curNode Node, filePassword []byte, index int) (tag []byte, err error) {}

- Generates a HMAC tag for a node to insure node integrity

func (userdata *User) VerifyNode(curNode Node, filePassword []byte, index int) error {}

- Verifies the integrity of the HMAC of a node

func (userdata *User) AddBlobToContents(curNode Node, filePassword []byte, fileContent *[]byte, index int) error {}

- Takes in a Blob UUID and adds its data to the fileContent

func (userdata *User) VerifySignature(signature []byte, owner string, encryptedBytes []byte) error {}

- Verifies the signature of the middle struct over the encryptedData

func (userdata *User) DecryptMiddleStruct(marshalledMiddleStructWrapper []byte, middleStructPtr *MiddleStruct) error {}

- Decrypts the middle struct and returns its value into the middleStructPtr

func (userdata *User) GenerateFileTag(filePtr *File, filePassword []byte) (tag []byte, err error) {}

- Generate a HMAC tag for a file

func (userdata *User) createNode(blobStorageKey userlib.UUID, filePassword []byte, index int, nodePtr *userlib.UUID) (err error) {}

- Create a node and sets the nodePtr to point at the node in the datastore

func (userdata *User) createBlob(content []byte, index int, filePassword []byte, uuidPtr *userlib.UUID) error {}

- Creates a blob and sets the uuidPtr to point at the blob in the datastore

func (userdata *User) UpdateMiddleStructs(children map[string]userlib.UUID, newFileId userlib.UUID, newFilePassword []byte, oldFileId userlib.UUID, oldFilePassword []byte) (err error) {}

- Loops through the children map, and updates their middle structs and their children maps recursively.

func (userdata *User) UpdateMiddleStruct(middleStructWrapper MiddleStructWrapper, middleStructWrapperKey userlib.UUID, newFileId userlib.UUID, newFilePassword []byte, username string) (err error) {}

- Updates the middle struct wrapper with a new encrypted middle struct that has the newFileId and newFilePassword

func (userdata *User) GetMiddleStruct(middleStructUUID userlib.UUID, middleStructPtr *MiddleStruct) (err error) {}

- Returns the middle struct that is at middleStructUUID

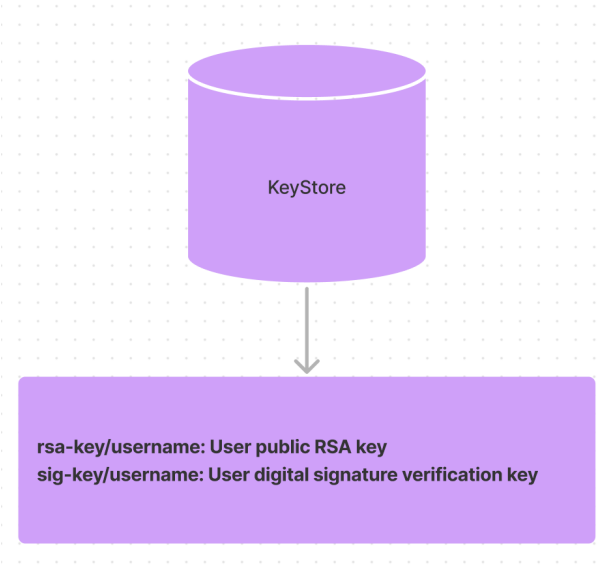
func (userdata *User) makeChildrenMap(fileId userlib.UUID, filePassword []byte) (err error) {}

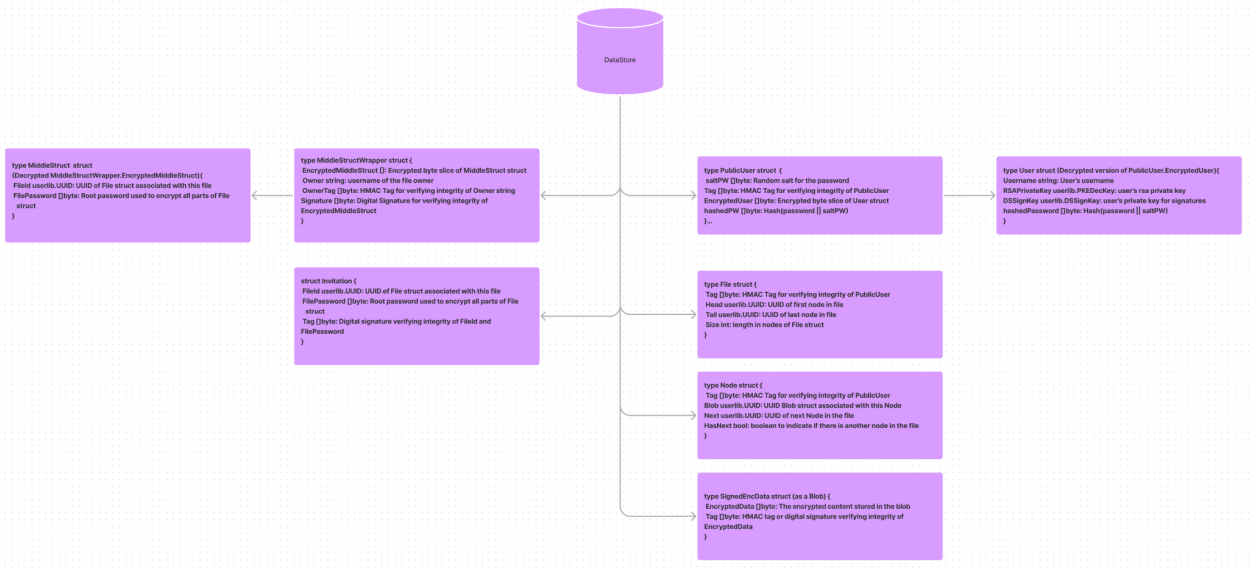
- Creates the children map for a specified user and file

func (userdata *User) overwriteFile(filename string, content []byte, existingMarshalledMiddleStructWrapper []byte) (err error) {}

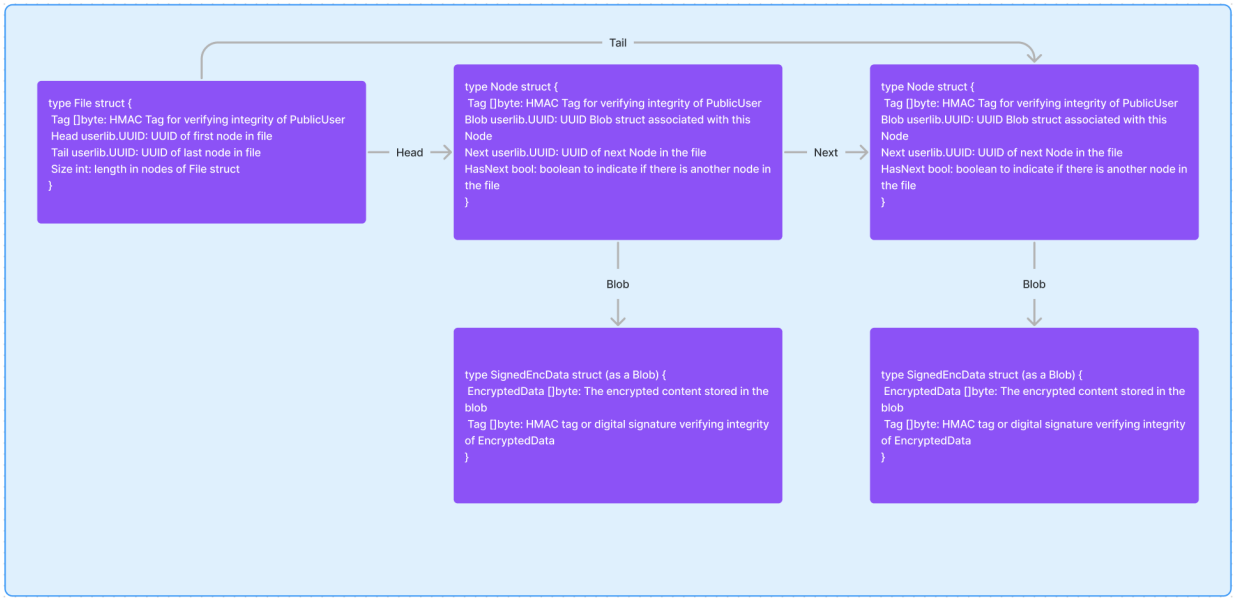
- Overwrites the specified filename with content.

Datastore, Keystore, and File Design





File Structure in Datastore



Test Proposal

1. **Design Requirement:** The client MUST NOT assume that filenames are globally unique. For example, user bob can have a file named foo.txt and user alice can have a file named foo.txt. The client MUST keep each user's file namespace independent from one another.
    a. Initialize two users Alice and Bob
    b. Alice and Bob both create a file named "foo.txt" with identical contents
    c. Bob appends random text to his "foo.txt" file
    d. Ensure that Alice's "foo.txt" is no longer the same as Bob's "foo.txt", and that the changes are only reflected in Bob's "foo.txt".
2. **Design Requirement:** The bandwidth of the AppendToFile() operation MUST scale linearly with only the size of data being appended and the number of users the file is shared with, and nothing else.
    a. Alice creates a file "foo.txt" that is several MB and a file "bar.txt" that is empty.
    b. Append 1 character to both "foo.txt" and "bar.txt" hundreds of times, timing the operation.
    c. Ensure that both append operations take roughly the same time on average across all iterations.
3. **Design Requirement:** When the owner revokes a user's access, the client MUST enforce that any other users with whom the revoked user previously shared the file also lose access.
    a. Initialize three users Alice, Bob, and Eve.
    b. Alice creates a file "foo.txt" and shares it with Bob.
    c. Bob accepts the invitation to "foo.txt" and subsequently invites Eve to share "foo.txt"
    d. Ensure that Alice, Bob, and Eve all have access to "foo.txt"
    e. Alice revokes Bob's access to "foo.txt"
    f. Ensure that Eve no longer has access to "foo.txt"
4. **Design Requirement:** The client MUST support a single user having multiple active sessions at the same time. All file changes MUST be reflected in all current user sessions immediately (i.e. without terminating the current session and re-authenticating).
    a. Alice calls GetUser on a computer, and then also calls GetUser on a separate phone.

    b. Alice creates a file “foo.txt” on the computer, and the loads that file on the phone
    c. Ensure that Alice has access to the file on the phone.
5. **Design Requirement:** The client MUST enforce authorization for all files. The only users who are authorized to access a file using the client include: (a) the owner of the file; and (b) users who have accepted an invitation to access the file and that access has not been revoked.
    a. Alice creates a file “foo.txt”.
    b. Have Bob try to access the file “foo.txt”
    c. Alice calls CreateInvitation to send an invitation to Bob
    d. Bob tries to access the file “foo.txt” and is denied
    e. Bob accepts the invitation, and then tries to access the file “foo.txt” successfully.
6. **Design Requirement:** Changes to the contents of a file MUST be accessible by all users who are authorized to access the file.
    a. Alice creates a file “foo.txt” and shares it with Bob.
    b. Alice appends a character string to “foo.txt”, and Bob loads the file.
    c. Ensure that Bob has the updated version of “foo.txt”.