Project 2 Design Doc

Edan Bash and Michael Wildfeuer

Data Structures

```
type User struct {
    Username   string
    Salt       []byte
    PKEDecKey  userlib.PrivateKeyType
    DSSignKey  userlib.PrivateKeyType
    userEncKey []byte
}
type FileNode struct {
    OwnerUsername string
    ContentPtr    uuid.UUID
    ParentNode    uuid.UUID
    Children      map[string]uuid.UUID
}
type Invitation struct {
    Sender              string
    Recipient           string
    MasterFileKey       []byte
    ParentFileNodeUUID uuid.UUID
}
type FileContentNode struct {
    ContentUUID     uuid.UUID
    NextContentUUID uuid.UUID
    LastContentUUID uuid.UUID
}
```

User Authentication

5.1. InitUser

Verify Valid Username

1. Check if username is empty string
   a. if yes → ERROR (*Username cannot be empty*)
2. Check if uuid.fromBytes(hashed(username)[:16]) exists in Datastore
   a. if yes → ERROR (*Username already exists*)

Generate and Store RSA and Signing Keys

1. Generate user's RSA keys with PKEKeyGen() → PKEEncKey, PKEDecKey
2. Generate UUID for public PKEEncKey → uuid.fromBytes(hashedUsername[16:32])
3. Store in Keystore → Key: PKEEncKeyUUID, Value: PKEEncKey
4. Generate user's Digital signature keys with DSKeyGen() → DSVerifyKey, DSSignKey
5. Generate UUID for public DSVerifyKey → hashedUsername[32:48])

Generate and Store userStruct

1. Create the user struct (username, salt, PKEDecKey, DSSignKey)
2. Serialize the user struct → json.Marshal(userStruct)
3. Generate password derived key (userEncKey) → Argon2Key(password, salt, 32)
4. Encrypt user struct → SymEnc(userEncKey[:16], userStruct)
5. Compute HMAC for user → HMACEval(userEncKey[16:], encryptedUserStruct)
6. Store in Datastore → Key: userStructUUID, Value: encryptedUserStruct + userTag
7. Generate saltUUID → hashedUsername[48:])
8. Store Salt in Datastore → Key: saltUUID, Value: salt

5.2. GetUser

1. Generate userUUID → uuid.fromBytes(hashed(username)[:16])
2. Retrieve userStruct from DataStore with userUUID
   a. If UUID is not found in Datastore → ERROR (*There is no initialized user in the database*)
3. Generate saltUUID → hashedUsername[48:])
4. Retrieve salt from Datastore
5. Generate the userEncKey → Argon2Key(password, salt, 32)

6. Decrypt userStruct → SymDec(userEncKey[:16], encryptedUserStruct)
7. Deserialize userStruct → json.unMarshal(userStruct)
8. Retrieve HMAC attached of encryptedUserStruct
9. Compute HMAC for userStruct → HMACEval(userEncKey[16:], encryptedUserStruct)
10. Compare HMAC tags → HMACEval(derivedHMACTag, retrievedHMACTag)
    a. If tags are not equal → ERROR (*Account has been compromised or Invalid credentials*)

File Storage and Retrieval

5.3. User.StoreFile

1. Generate fileNode UUID → uuid.fromBytes(username + filename)
2. Generate lockBoxUUID → uuid.fromBytes(username + fileNodeUUID)
3. If fileNodeUUID exists in DataStore:
   a. Call publicDec() to retrieve the masterFileKey
   b. Retrieve, Verify, and Dec fileNode with masterFileKey (purpose=username+"node")
   c. Retrieve, Verify, and Dec fileNode.fileContentNode with masterFileKey (purpose="list")
4. If fileNodeUUID doesn't exist in DataStore:
   a. Create a new fileNode struct with UUID → fileNode(fileNode.ContentPtr = uuid.New(), ownerUsername, children = [], parent=null)
   b. Generate new masterFileKey → RandomBytes(16)
   c. Call publicEncrypt(username, masterFileKey, lockBoxUUID) to encrypt the masterFileKey with current user's PKEEncKey and store in DataStore
   d. Securely store the new fileNode in Datastore using masterFileKey
   e. Generate uuid for fileContentNode.contentUUID → uuid.new()
5. Set the fileContentNode.next = null
6. Set fileContentNode.last = fileNode.ContentPtr
7. Securely store the updated fileContentNode in Datastore (Key: fileNode.ContentPtr)
8. Encrypt then HMAC new content using masterFileKey
9. Store secureContent in Datastore (Key: fileContentNode.contentUUID)

5.4. User.LoadFile

1. Generate fileNode UUID → uuid.fromBytes(username + filename)
   a. If fileNode UUID doesn't exist in DataStore → ERROR(*file doesn't exist*)

2. Generate lockboxUUID → uuid.fromBytes(username + fileNodeUUID)
3. Call publicDec() to retrieve the masterFileKey
4. Retrieve, Verify, then Dec fileNode with masterFileKey
5. Use fileNode.ContentPtr to retrieve first fileContentNode struct from Datastore
6. Initialize fullContent = []
7. At each iteration of the loop:
    a. Retrieve, Decrypt, Verify currContent using masterFileKey (purpose="content")
    b. Append this content resultContent
    c. Move to the next node in the fileContentList
8. return fullContent

5.5. User.AppendToFile

1. Generate fileNodeUUID → uuid.fromBytes(username||filename)
    a. If UUID doesn't exist in DataStore → ERROR(*file doesn't exist*)
2. Generate lockboxUUID → uuid.fromBytes(username + fileNodeUUID)
3. Call publicDec() to retrieve the masterFileKey
4. Retrieve, Verify, then Dec fileNode with masterFileKey
5. Use fileNode.ContentPtr to retrieve first fileContentNode struct from Datastore
6. Use firstContentNode.LastContentUUID to retrieve last fileContentNode struct from Datastore
7. Create a new fileNodeContent struct (contentUUID = uuid.New(), nextContentUUID=null)
8. Secure store new content with masterFileKey (Key: newFileContentNode.ContentUUID)
9. If only one contentNode in list
    a. Set the firstContentNode.nextContentUUID = newFileContentNodeUUID
    b. Set the firstContentNode.lastContentUUID = newFileContentNodeUUID
    c. Secure store the updated fileContentNode
10. Else there are multiple nodes in list
    a. Set the lastContentNode.NextContentUUID = newFileContentNodeUUID
    b. Set the firstContentNode.LastContentUUID = newFileContentNodeUUID
    c. Secure store the updated fileContentNodes

File Sharing and Revocation

5.6. User.CreateInvitation

1. Generate the file's fileNodeUUID and user's lockBoxUUID
2. Retrieve, Verify, Decrypt the masterFileKey with publicDec()
3. if fileNodeUUID does not exists
    a. Return error → *Cannot share file that does not exists*
4. Create invitation struct (sender=username, recipient=recipientUsername, parentFileNode=fileNodeUUID, masterFileKey = masterFileKey)
5. Serialize the invitation
6. Sign the invitation with current user's DSSignKey
7. Call pubEnc() to encrypt the invitation with recipient's publicKey
8. Store the secure invitation in Datastore

5.7. User.AcceptInvitation

1. Retrieve, Verify, and Dec invitation with publicDec()
2. Deserialize the invitation
3. Retrieve the DSVerifyKey of sender
4. Verify signature on invitation with user's DSVerifyKey
5. Generate a fileNodeUUID and lockBoxUUID
6. Securely store the retrieved masterFileKey in lockbox with publicEnc()
7. Retrieve the parentFileNode from parentFileUUID
8. Create a new FileNode struct:
    a. Set all the proper attributes using parent
9. Secure store new FileNode in Datastore
10. Add fileNode.Children[username] = fileNodeUUID to parent's attribute
11. Secure store updated parentFileNode in Datastore
12. Delete the invitation from Datastore

5.8. User.RevokeAccess

1. Retrieve the current user's fileNode
2. Assert curr user == owner file struct
3. Iterate through owner's child list
    a. If recipient is not in children → ERROR (*File not shared with user)*
4. Retrieve proper fileNode struct for the recipient
    a. If requested file is not in userStruct.fileSpace → ERROR (*File does not exist)*
5. Delete the fileNode from Datastore and recursively do this for it's children
6. Remove the recipient from the user's fileNode.Children map
7. Generate newMasterFileKey for filename
8. For current user and all children, recursively:

a. Update the lockbox with publicEnc() to overwrite their lockbox with newMasterFileKey
b. Re-encrypt and securely store the fileNode with newMasterFileKey

9. Re-encrypt each fileContentNode in the FileContentList with newMasterFileKey

Helper Methods

```
// Returns an encrypted and tagged message given a sourceKey a
nd purpose
func EncThenHMAC(sourceKey []byte, purpose string, plainText
[]byte) (secureMsg []byte, err error)


// Securely stores an object in Datastore given a sourceKey an
d purpose
func secureStore(keyUUID uuid.UUID, v interface{}, sourceKey
[]byte, purpose string) (err error)


// Retrives, verifies, and decrypts an object in Datastore giv
en a sourceKey and purpose
func retVerifyDec(keyUUID uuid.UUID, sourceKey []byte, purpose
string) (bytes []byte, err error)


// Verifies and decrypts a secureMsg given a sourceKey and pur
pose
func VerifyThenDec(sourceKey []byte, purpose string, secureMsg
[]byte) (result []byte, err error)


// Encrypts and stores an entry in DataStore entry using PKEEn
cKey for user
func publicEnc(username string, keyUUID uuid.UUID, purpose str
ing, data []byte) (ret []byte, err error)


// Decrypts a DataStore entry using PKEDecKey for user
```

```
func publicDec(userdata *User, keyUUID uuid.UUID, purpose stri
ng) (msg []byte, err error)


// Return the fileNodeUUID and masterFileKey for given user an
d filename
func getFileAndKey(userdata *User, filename string) (fileLocat
ion uuid.UUID, key []byte, err error)


// Return the unserialized fileNode for given user
func getFileNode(fileNodeUUID uuid.UUID, masterFileKey []byte,
username string) (f FileNode, e error)
```