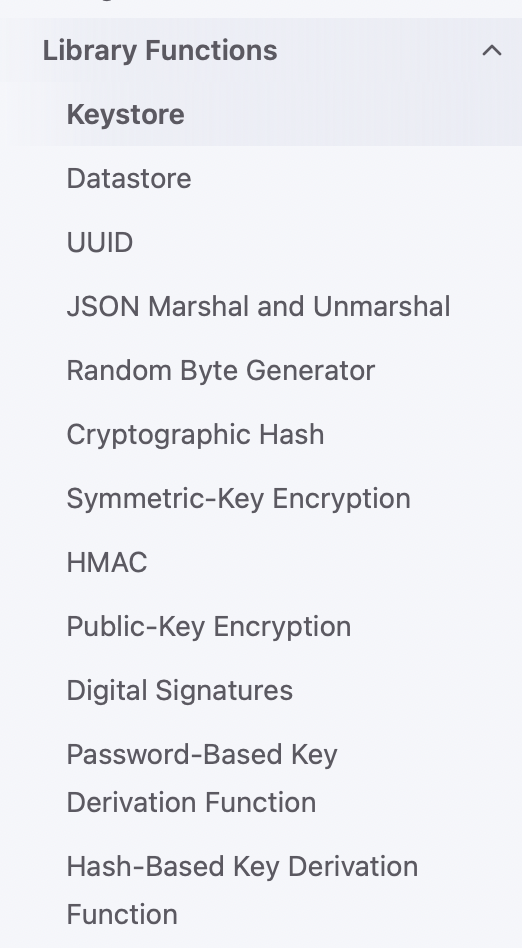
**LIBRARY FUNCTION:**

You should not write your own cryptographic functions for this project. For example, you shouldn’t write code to implement AES-CTR yourself. Instead, you should call the existing SymEnc function that we’ve provided.

As discussed in class, you should avoid any unsafe cryptographic design patterns, such as reusing the same keys in different algorithms (see the tips section for more details), or using MAC-then-encrypt.



* Keystore:

Go’s type-checking will enforce that all values stored are public keys. **You cannot store salts, hashes, structs, files, or any other data that is not a public key on Keystore.** If you want to store something that is not a public key, you must store it in Datastore, not Keystore.

You cannot store any data that is not a public key in Keystore. Keystore is immutable: A name-value pair cannot be modified or deleted after being stored in Keystore. Any attempt to modify an existing name-value pair will return an error.

* DataStore:

Stores name and value as a name-value pair into Datastore.

Datastore is mutable: If name already maps to an existing name-value pair, then the existing value will be overwritten with the provided value.

* UUID:

There are two ways to create UUIDs. You can randomly generate a new UUID from scratch. Alternatively, you can take an existing 16-byte string, and deterministically cast it into a UUID.

Creates a new UUID by copying the 16 bytes in b into a new UUID. Returns an error if the byte slice b does not have a length of 16.

Note: This function does not apply any additional security to the inputted byte slice. You can think of this function as casting a 16-byte value into a UUID. Anybody who reads the UUID will be able to determine what 16-byte value you used to generate the UUID, so you should not pass sensitive information into this function.

* JSON:

In the name-value pairs of Datastore, the value should be a byte array. If you want to store other types of data (e.g. structs) in Datastore, you will need to convert that data into a byte array before storing it. Then, you will need to convert the byte array back into the original data structure when retrieving the data.

Only struct fields that start with a capital letter will have their values restored. Struct fields that start with a lowercase letter will be initialized to their default value.

* Random Byte Generator:

You don’t need to worry about the underlying implementation (e.g. you don’t have to think about reseeding any PRNG). You can assume the returned bytes are indistinguishable from truly random bytes.

* Symmetric-Key Encryption:

SymEnc(key []byte, iv []byte, plaintext []byte) (ciphertext []byte)

If the provided ciphertext is less than the length of one cipher block, then SymDec will panic (remember, your code should always return errors, and not panic).

* Public-key Encryption (RSA):

PKEKeyGen() (PKEEncKey, PKEDecKey, err error)

PKEEnc(ek PKEEncKey, plaintext []byte) (ciphertext []byte, err error)

Note: RSA encryption does not support very long plaintext. If you need to use a public key to encrypt long plaintext, consider writing a helper function that implements hybrid encryption.

Recall the hybrid encryption process: Use the given public key to encrypt a random symmetric key. Then, use the symmetric key to encrypt the actual data. Return the symmetric key (encrypted with the public key) and the data (encrypted with the symmetric key).

Recall the decryption process for hybrid decryption schemes: Use the given private key to decrypt the symmetric key. Then, use the symmetric key to decrypt the data.

* Password-Based Key Derivation Function (PBKDF):

Argon2Key(password []byte, salt []byte, keyLen uint32) (result []byte)

Argon2Key is called a Password-Based Key Derivation Function (PBKDF) because the output (i.e. the hashed password) can be used as a symmetric key. An attacker cannot brute-force passwords to learn the key because the hash function is too slow. Also, the hash function makes the hashed password look unpredictably random, so it can be used as a symmetric key.

You can assume that the user’s chosen password has sufficient entropy for the PBKDF output to be used as a symmetric key.

If you choose to use the hash as a key, then the salt also ensures that the two users don’t use the same key.

* Hash-Based Key Derivation Function (HBKDF):

HashKDF(sourceKey []byte, purpose []byte) (derivedKey []byte, err error)

HashKDF is a fast hash function, similar to HMAC, that essentially hashes the source key and the purpose together. Changing either the source key, or the purpose, or both, will cause the output of HashKDF to be unpredictably different. One way you can use HashKDF is by calling it multiple times with the same source key but different, hard-coded purposes every time. This will generate multiple keys, one per call to HashKDF.

**User Authentication:**

There is no log out operation. If EvanBot is done running file operations, they can simply quit the program, which will destroy the User object (and its instance variables). This should not cause any data to be lost.

Note that different users could choose the same password.

* InitUser:

Recall that when the program quits, the User struct and all its data will be lost. Only data on Datastore and Keystore persists after the program quits.

Returns an error if (1) A user with the same username exists. (2) An empty username is provided.

* GetUser:

If you stored the data of the User struct to Datastore in InitUser, then you could download that data from Datastore and create a new User object in local memory using that data.

Returns an error if: (1) There is no initialized user for the given username. (2) The user credentials are invalid. (3) The User struct cannot be obtained due to malicious action, or the integrity of the user struct has been compromised.

* Passwords:

The passwords provided by users have sufficient entropy for the PBKDF slow hash function to output an unpredictable string that an attacker cannot guess by brute force.

The passwords provided by users do not have sufficient entropy to resist brute-force attacks on any of the other fast hash functions (Hash, HashKDF, or HMAC).

Passwords can be any string with 0 or more characters (not necessarily alphanumeric, and could be the empty string).

* Multiple devices:

User should be able to call GetUser multiple times, with the same username and password, to obtain multiple different copies of the User struct on multiple different devices.

All changes to files made from one device must be reflected on all other devices immediately (i.e. a user should not have to call GetUser again to see the changes).

**File Operation:**

* NameSpace:

Note that different users can have files with the same name. A user’s namespace is defined as all of the filenames they are using. One user’s namespace could contain a filename that another user is also using. In that other user’s namespace, that same filename could refer to a different file (or the same file, if it was shared - more details about sharing later).

* StoreFile:

has never stored to foods.txt before, this creates a new file called foods.txt in EvanBot’s personal namespace.

Because foods.txt is an existing file, this call should overwrite the entire file with the new contents.

overwrites the entire file (including appends) with the new contents

creates a new file in CodaBot’s personal namespace named foods.txt. This should not interfere with the foods.txt file in EvanBot’s namespace, which is a different file.

The client must allow content to be any arbitrary sequence of bytes, including the empty sequence.

Note that calling StoreFile after malicious tampering has occurred is undefined behavior, and will not be tested.

Note that calling StoreFile on a file whose access has been revoked is undefined behavior, and will not be tested.

**Returns an error if** the write cannot occur due to malicious action.

* LoadFILE:

sees an error, because there is no file named drinks.txt in EvanBot’s personal namespace.

Note that, in the case of sharing files, the corresponding file may or may not be owned by the caller.

**Returns an error if: (1)** The given filename does not exist in the personal file namespace of the caller. (2) The integrity of the downloaded content cannot be verified (indicating there have been unauthorized modifications to the file). (3) Loading the file cannot succeed due to any other malicious action.

* AppendToFile:

sees an error, because there is no file named drinks.txt in EvanBot’s personal namespace.

content can be any arbitrary sequence of 0 or more bytes.

Note that, in the case of sharing files, the corresponding file may or may not be owned by the caller. You are **not** required to check the integrity of the existing file before appending the new content (integrity verification is allowed, but not required).

**Returns an error if: (1)** The given filename does not exist in the personal file namespace of the caller. (2)Appending the file cannot succeed due to any other malicious action.

* FILE:

You must ensure that no information is leaked about these 3 pieces of data:

* 1. File contents for all files. (2) Filenames for all files. (3)The length of the filenames for all files.

**You must also ensure that no information is leaked that could be directly or indirectly used to learn these 3 pieces of data.**

For example, if you have a secret key that you’re using to encrypt some file contents, you’ll need to ensure that secret key is not leaked either.

You may leak information about any other values besides the ones listed above.

For example: It’s okay if an adversary learns usernames, length of a file, how many files a user has, etc.

**Sharing & Revocation:**

EvanBot created the new file with a call to StoreFile, EvanBot is the owner of this file.

* CreateInvitation

EvanBot calls CreateInvitation("foods.txt", "codabot").

This function returns a UUID, which we’ll call an invitation Datastore pointer.

The invitation UUID can be any UUID you like. For example, you could collect/compute any values that you want to send to the recipient user for them to access the file. Then, you could securely store these values on Datastore at some UUID, and return that UUID.

EvanBot uses a secure communication channel (outside of your system) to deliver the invitation UUID to CodaBot. Using this secure channel, CodaBot receives the identity of the sender (EvanBot) and the invitation UUID generated by EvanBot.

**Returns an error if: (1)**The given filename does not exist in the personal file namespace of the caller. (2) The given recipientUsername does not exist. (3) Sharing cannot be completed due to any malicious action.

* AcceptInvitation:

CodaBot calls AcceptInvitation("evanbot", invitationPtr, "snacks.txt").

CodaBot passes in the identity of the sender and the invitation UUID generated by EvanBot.

CodaBot also passes in a filename (snacks.txt here). Note that CodaBot (the recipient user) can choose to give the file a different name while accepting the invitation.

Note that different users can refer to the same file using different filenames.

**Note that all users should be able to see modifications to the file.**

**You can assume that after the sender generates an invitationPtr UUID, the sender uses a secure communication channel (outside of your system) to deliver that UUID to the recipient.** Using this secure channel, the recipient receives the UUID and the sender’s username, and can input them into AcceptInvitation.

**Returns an error if: (1)** The user already has a file with the chosen filename in their personal file namespace. (2) **Something about the invitationPtr is wrong (e.g. the value at that UUID on Datastore is corrupt or missing, or the user cannot verify that invitationPtr was provided by senderUsername).** (3) The invitation is no longer valid due to revocation.

All users who have accepted an invitation to access the file (and who have not been revoked) must also be able to access the file. These users must be able to:

Read the file contents with LoadFile,Overwrite the file contents with StoreFile, Append to the file with AppendToFile, Share the file with CreateInvitation.

**If a user changes the file contents, all users with access must immediately see the changes. The next time they try to access the file, all users with access should see the latest version.**

* RevokeAccess

RevokeAccess(filename string, recipientUsername string) (err error)

Revokes access to filename from the target user recipientUsername, and **all the users that recipientUsername shared the file with (either directly or indirectly).**

Note: This function could be called either before or after the target user calls AcceptInvitation. Your code should be able to revoke access either way.

After revocation, the revoked users should not be able to access the file. They should get an error if they try to access the file using your system (e.g. by calling LoadFile, AppendToFile, etc.). Note that calling StoreFile on a revoked file is undefined behavior and will not be tested.

**Returns an error if: (1)** The given filename does not exist in the caller’s personal file namespace. (2) The given filename is not currently shared with recipientUsername. (3) Revocation cannot be completed due to malicious action.

Users B, D, E, and F should all lose access to the file.

If any of these users try to access the file (load, store, append, create invitation), the function should error.

These users can now be malicious: they can use values they’ve previously written down, and access Datastore (without listing out all UUIDs). However, they should still be unable to read the file, modify the file, or deduce when future updates are happening.

* Revoke User Adversary

The Revoked User Adversary’s goal is to re-obtain access to the file. The revoked user will not perform malicious actions on other files that they still have access to. Their only goal is to re-obtain access to the file that they lost access to.

The Revoked User Adversary might attempt to re-obtain access by calling functions with different arguments (e.g. calling AcceptInvitationagain).

The Revoked User Adversary may also try to re-obtain access by calling DatastoreGet and DatastoreSet and maliciously affecting Datastore. However, unlike the Datastore Adversary, they do not have a global view of Datastore (i.e. they cannot list all UUIDs that have been in use).

Prior to having their access revoked, the Revoked User Adversary could have written down any values that they have previously seen. The Revoked User Adversary has a copy of your code running on their local computer, so they could inspect the code and learn the values of any variables that you computed.

Your code should ensure that the Revoked User Adversary is unable to learn anything about any future writes or appends to the file (learning about the file before they got revoked is okay). For example, they cannot know what the latest contents of the file are, and they should be unable to make modifications to the file without being detected. Also, they cannot know when future updates are happening (e.g. they should not be able to deduce how many times the file has been updated in the past day).

Revoked User Adversary

* They will only try to regain access to the file. They won’t tamper with files they still have access to.
* They could try to call functions with different arguments.
* They could directly read/modify Datastore, but cannot list all UUIDs being used.
* They can write down any values (i.e. local variables your code computed) before getting access revoked.
* They should be unable to read the latest file, modify the file without being detected, or deduce when future updates are happening.

Notes on Key Management

In order to simplify your key management scheme, it may be useful to store a small number of root keys, and re-derive many keys for specific purposes “on-the-fly” (a.k.a. when you need them) using the HashKDF function.