Design Overview

In this section, we give a high-level overview of what you'll be designing. We also describe general requirements that apply to the entire design. (Requirements specific to individual functions are described later.)

Functionality Overview

In this project, you will be designing a system that allows users to securely store and share files in the presence of attackers. In particular, you will be implementing the following 8 functions:

- `InitUser`: Given a new username and password, create a new user.
- `GetUser`: Given a username and password, let the user log in if the password is correct.
- `User.StoreFile`: For a logged-in user, given a filename and file contents, create a new file or overwrite an existing file.
- `User.LoadFile`: For a logged-in user, given a filename, fetch the corresponding file contents.
- `User.AppendToFile`: For a logged-in user, given a filename and additional file contents, append the additional file contents at the end of the existing file contents, while following some efficiency requirements.
- `User.CreateInvitation`: For a logged-in user, given a filename and target user, generate an invitation UUID that the target user can use to gain access to the file.
- `User.AcceptInvitation`: For a logged-in user, given an invitation UUID, obtain access to a file shared by a different user. Allow the recipient user to access the file using a (possibly different) filename of their own choosing.
- `User.RevokeAccess`: For a logged-in user, given a filename and target user, revoke the target user's access so that they are no longer able to access a shared file.

More details about these functions later in the spec. You don't need to implement a frontend for this project; all users will interact with your system by running a copy of your code and calling these 8 functions.

The User Class

You will implement the 8 functions above as part of the `User` class. The `User` class has:

- Two constructors, `InitUser` and `GetUser`, which create and return new `User` objects. A `User` object is represented as a reference to a `User` struct containing instance variables. A constructor is called every time a user logs in.
- Six instance methods (the other functions listed above), which can be called on a `User` object. The user calls these functions to perform file and sharing operations.
- Instance variables, specific to each `User` object, which can be used to store information about each user. The instance variables are stored in the `User` struct returned by a constructor, and the instance variables are accessible to all the instance methods.

If a user calls a constructor multiple times, there will be multiple `User` objects that all represent the same user. You will need to make sure that these `User` objects do not use outdated data (more details later).

Atomic Operations

You do not need to worry about parallel function calls or any concurrency issues for this project. You can assume that at any given time, only one function is being executed. Another function call will only begin after the current function call has completed.

You can also assume that we will not quit the program in the middle of a function call. As long as your code doesn't crash, every function call will run to completion.

You can also assume that malicious action will only happen in between function calls, and will not happen in the middle of a function call.

Stateless Design

Multiple users may use the system by calling your functions. Each user may have multiple devices (e.g. Alice may have a laptop and a phone). Every device runs a separate identical copy of your code.

As a consequence of this, your code cannot have global variables (except for basic constants). This is because these global variables will not be synced across devices.

If a user's copy of the code crashes after running a function, or if the user quits running the code, they will lose all data stored in the code's local memory (e.g. global variables, instance variables in `User` objects, etc). Therefore, you cannot rely on storing persistent information in the code's local memory.

All devices running your code are able to send and receive data from two shared remote databases called Datastore and Keystore (more details later). All persistent data must be stored on Datastore or Keystore.

Keystore

Keystore is a remote database where you can store **public** keys.

Keystore is organized as a set of name-value pairs, similar to a dictionary in Python or a HashMap in Java. (Note: Sometimes these are called key-value pairs, but we will call them name-value pairs to avoid confusion with cryptographic keys.)

The name in each name-value pair must be string. The value in each name-value pair must be a public key (either a public encryption key of type `PKEEncKey`, or a public verification key of type `DSVerifyKey`).

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.

Each user may only store a small, constant number of public keys on Keystore. In other words, the total number of keys on Keystore should only scale with the number of users. In other words, you cannot create a new public key per file, per share, etc.

Once a name-value pair is written to Keystore, it cannot be modified or deleted. Everybody (including attackers) can read all values, but cannot modify any values, on Keystore.

Keystore is already implemented for you. You can write new name-value pairs using `KeystoreSet`, and you can read the value corresponding to a name using `KeystoreGet` (more details later).

Datastore

Datastore is an insecure remote database where you can store data. The Datastore Adversary is an attacker who can read and modify any data on Datastore (more details later). Therefore, you must protect the confidentiality and integrity of any sensitive data you store on Datastore.

Datastore is organized as a set of name-value pairs, just like Keystore. The name in each name-value pair must be a UUID, a unique 16-byte string (more details later). The value in each name-value pair can be any byte array of data.

Given a specific name (UUID), there is one and only one corresponding value, which can be read and modified by anybody who knows the name (UUID).

Datastore is already implemented for you. You can write new name-value pairs using `DatastoreSet`, you can read the value corresponding to a name using `DatastoreGet`, and you can delete a name-value pair using `DatastoreDelete` (more details later).

DESIGN QUESTION

Design Question: Data structures: What data structures are you going to use to organize data on Datastore?

List any struct definitions that you plan on using, along with the fields that each struct will contain.

We recommend starting with a few core data structures (e.g. `struct user`, `struct file`, `struct invitation`, etc.) and adding additional fields and structs as you need them.

For every subsequent design question, think about what data structures are used, and what is being stored and where. How will you generate the UUID that you are storing the information at? How will you ensure confidentiality and integrity? Which keys will you use, and how are you generating and accessing them?

Threat Model: Datastore Adversary

The Datastore Adversary is an attacker who can read and modify all name-value pairs, and add new name-value pairs, on Datastore. They can modify Datastore at any time (but not in the middle of another function executing).

The Datastore Adversary has a global view of Datastore; in other words, they can list out all name-value pairs that currently exist on Datastore.

The Datastore Adversary can take snapshots of Datastore at any time. For example, they could write down all existing name-value pairs before a user calls `StoreFile`. Then, they could write down all existing name-value pairs after a user calls `StoreFile` and compare the difference to see which name-value pairs changed as a result of the function call.

The Datastore Adversary can see when a user calls a function (e.g. if a user calls `StoreFile`, they know which user called it and when).

The Datastore Adversary can view and record the content and metadata of all requests to the Datastore API. This means that they will know what the inputs and outputs to the functions are.

The Datastore Adversary is not a user in the system, and will not collude with other users. However, the Datastore Adversary has a copy of your source code (Kerckhoff's principle), and they could execute lines of your code on their own in order to modify Datastore in a way that mimics your code's behavior.

The Datastore adversary will not perform any rollback attacks: Given a specific UUID, they will not read the value at that UUID, and then later replace the value at that UUID with the older value they read. (Deleting a value at a UUID is not a rollback attack.)

They will also not perform any rollback attacks on multiple UUIDs. For example, they will not revert the entire contents of Datastore to some previous snapshot of Datastore they took.

There is one other adversary besides the Datastore Adversary, called the Revoked User Adversary. The two adversaries do not collude. This additional adversary will be described later.

Error Handling

All 8 functions have `err` as one of their return values. If the function successfully executes without an error, you should return nil (the null value in Go) for the `err` return value.

If a function is unable to execute correctly, all you need to do is return an error that is not nil. The function could fail due to functionality issues (e.g. a user supplies an invalid argument), or security issues (e.g. an attacker has tampered with data that prevents your function from executing correctly). The error message can be anything you want (as long as the error is not nil), though we recommend using informative error messages for easier debugging.

You only need to detect when errors occur in this project; you do not need to recover from errors. For example, suppose an attacker has tampered with a file stored in Datastore, and the user calls `LoadFile` to try and read the file contents. Your code only needs to detect that tampering has occured and return any non-nil error. You do not need to recover from the error (i.e. you don't need to try and recover the original file contents).

After an adversary performs malicious action, your function must either return an error, or execute correctly as if the adversary had not performed malicious action.

As soon as a non-nil error is returned, all subsequent function calls can have undefined behavior. Undefined behavior means that your code can do anything (execute without an error, return an error, crash, etc.), as long as you do not violate any security requirements.

EXAMPLE

A user stores two files, FileA.txt and FileB.txt. Your code stores the contents of these files in Datastore. Then, an attacker modifies some values on Datastore, including the contents of FileB.txt.

If the user tries to load FileB.txt, you should return a non-nil error value to detect that tampering has occurred. You do not need to try and recover the original unmodified contents of FileB.txt.

If the user instead tries to load FileA.txt, you have two options. If your code can successfully load the original unmodified contents of this file, you can return a nil error. Alternatively, if your code is unable to load the original unmodified contents of this file, you can return a non-nil error.

If the user's call to the load function returned a non-nil error, then all subsequent function calls have undefined behavior, as long as you do not violate any security requirements (e.g. subsequent function calls still cannot leak the contents of a confidential file).

Your code should never `panic` (the Go term for crashing the program). You should always return a non-nil error that can be safely processed by other code (e.g. the autograder). If your code panics, then the autograder might crash and be unable to give you a score.

Terminology Note: Pointers

In this project, the term "pointer" can refer to two different concepts. In this spec, we will always clarify which concept we're referring to.

A *Go memory pointer* is a variable that contains a memory address of some other object in the Go runtime's local memory. Go memory pointers are similar to pointers in C. In this project, because you cannot store persistent data in the program's local memory, you should rarely need to use Go memory pointers.

The only scenario where Go memory pointers are required for this project are in the `InitUser` and `GetUser` constructors. The constructors create a new `User` object by creating a new `User` struct in local memory, and returning a Go memory pointer that contains the address of this `User` struct in local memory.

EXAMPLE

Here's a code snippet showing how the User class works.

```
var alice *client.User
var err error
```

```
alice, err = client.InitUser("Alice", "password")
alice.StoreFile("FileA.txt", []byte("Some text."))
```

The variable `alice` is a Go memory pointer, containing the address of a User struct that was created by the `InitUser` constructor.

`StoreFile` is an instance method, so you cannot call it by itself. Instead, you need to call it on an existing User object (`alice` in this example). This function call has access to all the instance variables in the User struct that `alice` points to.

Recall that in Datastore, you can store any data you want at a given UUID. If the data you choose to store is another UUID, then you've created a *Datastore pointer*. If you fetch the data at the given UUID, you will receive another UUID which references another location in Datastore.
