CS 161 Project 2

An End-to-End Encrypted File Sharing System

In this project, you will apply the cryptographic primitives introduced in class to design and implement the client application for a secure file sharing system. Imagine something similar to Dropbox, but secured with cryptography so that the server cannot view or tamper with your data.

The client will be written in Golang and will allow users to take the following actions:

- Authenticate with a username and password;
- Save files to the server;
- Load saved files from the server;
- Overwrite saved files on the server;
- Append to saved files on the server;
- Share saved files with other users; and
- Revoke access to previously shared files.

We provide several resources to get you started:

1. We provide two servers that you can utilize in the design of your client application: the Keystore, and the Datastore.
2. We provide implementations of several cryptographic algorithms and a number of functions that you can use to interact with Keystore and Datastore. These utilities are defined in userlib, which is already imported into client.go.
3. We define eight API functions in the starter code (see client.go) that you must implement (see Grading and Deliverables).

Using these resources and your knowledge of computer security, you will design a secure client application that satisfies all of the Design Requirements.

As always, if you have questions about this documentation (or find errors), please make a post on Piazza!

Staff Advice

Design a solution before starting the implementation. Students consistently agree that design is harder than implementation across multiple iterations of this project. A faithful implementation of a faulty design will not earn you many points.

To approach the design process, read through the Design Requirements and the function definitions that you are required to implement in client.go. Think about how you can design your client to provide the required functionality. Here are some useful questions to get you started:

- What data do you need to track for each user?
- What data do you need to track for each file?
- What will you store in the datastore? What will you store in the keystore?

If you are stuck, try ignoring the file sharing functionality and instead focus on how to provide just the store/load file functionality. While you might need to later change your design to support secure sharing, this project is much easier to grasp when sharing is not involved!

1. Grading and Deliverables

For this project, you may either work alone, or in a team of two. We recommend working in teams of two, since it helps to talk through many of the more challenging components of this project with a partner.

Project 2 is worth a total of 150 points, broken down as follows:

Points	Description
20 points	Checkpoint Submission
10 points	Final Design Doc
20 points	Final Test Case Coverage
100 points	Final Autograded Code

View how this project is weighted relative to other components of this class on the course website.

Checkpoint (due Friday, March 11)

The checkpoint submission consists of:

1. A draft design document, and;
2. A draft test proposal, consisting of six proposed test cases you'd like to implement

Draft Design Document

An outline of what we expect in your design document is here.

The draft design document will be graded on correctness, which is defined by the ability for your design to satisfy the design requirements. It should be a maximum of four pages, with an optional fifth page containing a diagram. There is no minimum length, as long as you cover all of the required portions.

Draft Test Proposal

The draft test proposal should contain six proposed test cases. Each test case should contain (a) one or more design requirements that the test is assessing, and (b) pseudocode to indicate what the test will do.

Here's an example of a test:

> **Design Requirement: The number of public keys should not depend on the number of files stored [3.3.2]**
>
> 1. Define `CountKeystore()`, a method that counts the number of keys in Keystore.
> 2. Create one user.
> 3. `X = CountKeystore()`
> 4. Store 100 files.
> 5. Expect that the number of public keys didn't change (expect `CountKeystore() == X`)

For inspiration of other test possible test cases, take a look at the Design Requirements and the provided tests in the starter code. Your test cases cannot include the test cases provided in the starter code, or the specific test case described above.

The draft test proposal will be graded on completion. It should be a maximum of two pages.

Design Reviews (Monday, March 14 - Friday, March 18)

After the checkpoint, we'll offer **optional** design reviews for you to chat with a TA about your proposed design.

If you do not recieve full credit on your checkpoint submission, you'll have a chance to **receive points back if you sufficiently address the feedback on your design doc during your design review.**

If you do recieve full credit on your checkpoint submission, this does not indicate that your design is foolproof! The autograder is the true measure of design correctness, as outlined by the specification. The checkpoint is just an indicator of whether or not you're on the right track. Note that even if you do recieve full credit for your design, you can still sign up for a design review to chat with a TA.

Note that we will aim to grade and return your design documents **with feedback** over the weekend after they are due. Due to the tight turnaround, it's in your best interest to turn in your design documents on time!

Final Submission (due Friday, April 8)

Your final submission consists of:

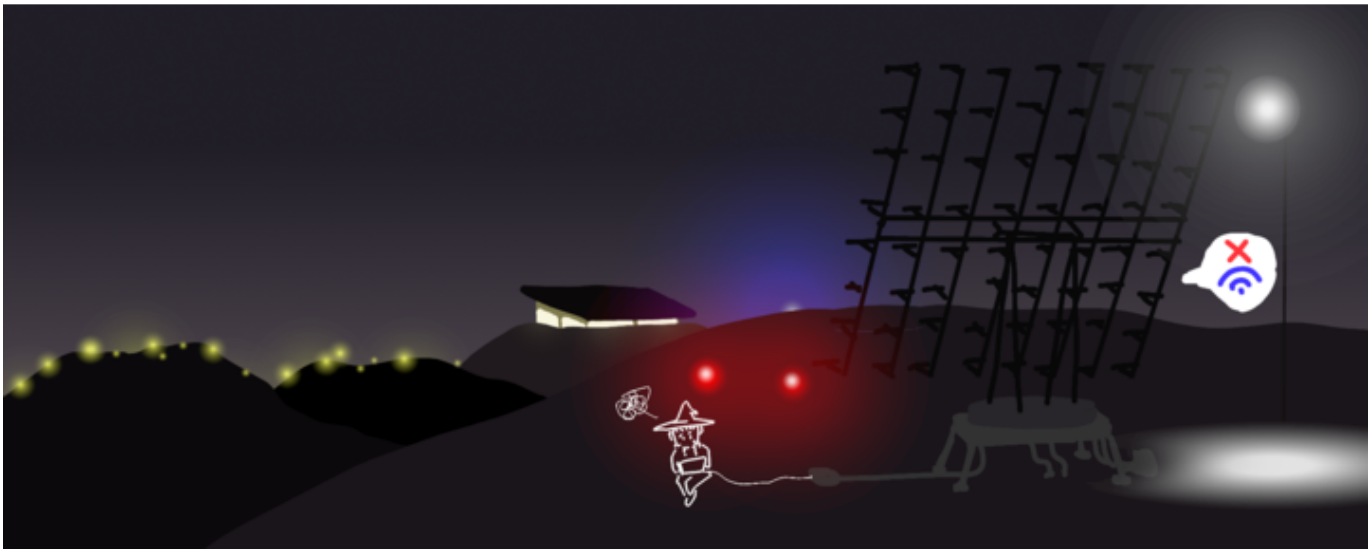
1. A final design document, and;
2. A final code submission, consisting of...
    a. A client implementation
    b. A comprehensive test suite

Project Schedule

Week	Description
Week of 2/28	Start Design Doc
Week of 3/7	Finish Design Doc
Week of 3/14	Design Reviews / Begin Implementation
Week of 3/21	Spring Break
Week of 3/28	Continue Implementation
Week of 4/4	Finish Implementation

Staff advice: **start early!** This project is like a take-home midterm, and is challenging! We're here to support you, so take advantages of the resources we offer: walkthrough videos, office hours, and project parties.

Just for fun - no project-relevant content in this box.



REGULUS guarantees availability when line of sight can be established between you and the moon. Requests to REGULUS may be processed when line of sight is unavailable depending on the availability of uplink proxies around the globe and conditions of the ionosphere.

We guarantee 0-60% uptime depending on the time of month, cloud cover and occurance of solar flares.

TABLE OF CONTENTS

2. Threat Model

We consider two, independent adversaries: the **datastore adversary**, and the **revoked user adversary**. These two adversaries do not collaborate.

> Remember Shannon's Maxim and Kerckhoffs' principle: You should assume that all adversaries know the design of your client application and have access to your source code.

2.1 Datastore Adversary

The Datastore is an **untrusted** service hosted on a server and network controlled by an adversary. The adversary can view and record the content and metadata of all requests (set/get/delete) to the Datastore API. This allows the adversary to know who stored which key-value entry, when, and what the contents are.

The adversary can add new key-value entries at any time and can modify any existing key-value entry. However, the Datastore will never execute a rollback attack (full or partial).

The Datastore will not launch any Denial of Service (DoS) attacks. However, assume that it implements a rate-limiting scheme which prevents a user from enumerating the key/value pairs in the Datastore.

2.2 Revoked User Adversary

Assume that each user records all of the requests that their client makes to Datastore and the corresponding responses.

A user who is granted access to a file is considered trusted and will only use the system through the Client API functions. However, **after** a user has their access to a shared file revoked, that user may become malicious, ignore your client implementation, and use the Datastore API directly.

Malicious users may try to perform operations on arbitrary files by utilizing the request/response information that they recorded before their access was revoked. All writes to Datastore made by a user in an attempt to modify file content or re-acquire access to file are malicious actions.

3. Design Requirements

> *The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.*

3.1 Usernames and Passwords

1. Usernames
   a. The client SHOULD assume that each user has a unique username.
   b. Usernames are case-sensitive: `Bob` and `bob` are different users.
   c. The client SHOULD support usernames of any length greater than zero.
2. Passwords
   a. The client MUST NOT assume each user has a unique password. Like the real world, users may happen to choose the same password.
   b. The client MAY assume each user's password generally is a good source of entropy. However, the attackers possess a precomputed lookup table containing hashes of common passwords downloaded from the internet.
   c. The client SHOULD support passwords length greater than or equal to zero.

3.2 User Sessions

1. The client application MUST allow many different users to use the application at the same time. For example, Bob and Alice can each run the client application on their own devices at the same time.

2. 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).

   For example:

   - Bob runs the client application on his laptop and calls `InitUser()` to create session `bobLaptop`.

- Bob wants to run the client application on his tablet, so he calls `GetUser` on his tablet to get `bobTablet`.
- Using `bobLaptop`, Bob stores a file `file1.txt`. Session `bobTablet` must be able to download `file1.txt`.
- Using `bobTablet`, Bob appends to `file1.txt`. Session `bobLaptop` must be able to download the updated version.
- Using `bobLaptop`, Bob accepts an invitation to access a file and calls the file `file2.txt` in his personal namespace. Bob must be able to load the corresponding `file2.txt` using `bobTablet`.

3 The client DOES NOT need to support concurrency. Globally, across all users and user-sessions, all operations in the client application will be done serially.

3.3 Cryptography and Keys

1 Each public key SHOULD be used for a single purpose, which means that each user is likely to have more than one public key.

2 A single user MAY have multiple entries in Keystore. However, the number of keys in Keystore per user MUST be a small constant; it MUST NOT depend on the number of files stored or length of any file, how many users a file has been shared with, or the number of users already in the system.

3 The following SHOULD be avoided because they are dangerous design patterns that often lead to subtle vulnerabilities.

a Reusing the same key for multiple purposes.

b Authenticate-then-encrypt.

c Decrypt-then-verify.

3.4 No Persistent Local State

1 The client MUST NOT save any data to the local file system. If the client is restarted, it must be able to pick up where it left off given only a username and password. Any data requiring persistent storage MUST be stored in either Keystore or Datastore.

3.5 Files

Any breach of IND-CPA security constitutes loss of confidentiality.

1 The client MUST ensure confidentiality and integrity of file contents and file sharing invitations.

2 The client MUST ensure the integrity of filenames.

3 The client MUST prevent adversaries from learning filenames and the length of filenames. The client MAY use and store filenames in a deterministic manner.

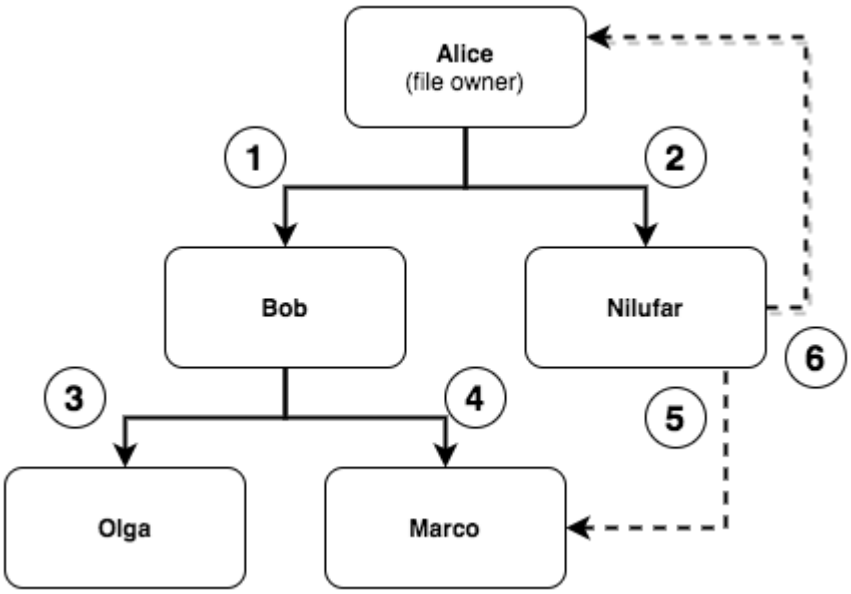
4 The client MUST prevent the revoked user adversary from learning anything about future writes or appends to the file after their access has been revoked.

5 The client MAY leak any information except filenames, lengths of filenames, file contents, and file sharing invitations. For example, the client design MAY leak the size of file contents or the number of files associated with a user.

6 Filenames MAY be any length, including zero (empty string).

7 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.

3.6 Sharing and Revocation



1 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.

2 The client MUST allow any user who is authorized to access the file to take the following actions on the file:

- Read file contents (`LoadFile()`).
- Overwrite file contents (`StoreFile()`).
- Append additional contents to the file (`AppendToFile()`).

- Share the file with other users (`CreateInvitation()`).

For example, all of the users listed in Figure 1 are authorized to take the listed actions on the file.

3 Changes to the contents of a file MUST be accessible by all users who are authorized to access the file.

4 The client MUST enforce that there is only a single copy of a file. Sharing the file MAY NOT create a copy of the file.

5 The client MUST ensure the confidentiality and integrity of the secure file share invitations created by `CreateInvitation()`.

6 The client MAY assume that `CreateInvitation()` will never be called on recipients who are already authorized to access the file.

7 The client MAY assume that `CreateInvitation()` will never be called in a situation where the sharing action will result in an ill-formed tree structure. In a **well-formed** tree structure, each non-root node has a single parent and there are no cycles. For example, in Figure 1, `Nilufar`'s attempts to share with `Marco` or `Alice` in steps 5 and 6 would be undefined behavior, since both users are already authorized to access the file, and the sharing actions would create an ill-formed tree structure.

8 The client MUST enforce that the file owner is able to revoke access from users who they directly shared the file with.

Any other revocation (i.e. owners revoking users who they did not directly share with, or revocations by non-owners) is undefined behavior and will not be tested.

For example, in Figure 1, `Alice` is the only user who MUST be able to revoke access, and she MUST be able to revoke access from `Bob` and `Nilufar`. If any user other than `Alice` attempts to revoke access, or `Alice` attempts to revoke access from any user other than `Bob` or `Nilufar`, this is undefined behavior and will not be tested.

9 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.

For example, in Figure 1, if `Alice` revokes access from `Bob`, then all of the following users MUST lose access: `Bob`, `Olga`, and `Marco`. As the file owner, `Alice` always maintains access. `Nilufar` maintains access because `Bob` did not grant `Nilufar` access to the file (`Alice` did).

10 The client MUST prevent any revoked user from using the client API to take any action on the file from which their access was revoked. However, recall from Threat Model that a revoked user may become malicious and use the Datastore API directly.

11 Re-sharing a file with a revoked user is undefined behavior and will not be tested.

3.7. Efficiency

1 The client MUST allow users to **efficiently** append new content to previously stored files.

We measure efficiency in terms of the bandwidth used by the `AppendToFile()` operation (the total size of the data uploaded and downloaded via calls to `DatastoreGet()` and `DatastoreSet()`).

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. Logarithmic and constant scaling in other terms is fine.

For example, appending 100 B to a 10 TB file should not use 10 TB of bandwidth. The 1,000th and 10,000th append operations to the same file should not use significantly more bandwidth than the 1st append operation. This is not an exhaustive list of restrictions.

2 `AppendToFile()` is the only function that has an explicit requirement for efficiency. However, the client MUST implement other functions such that they do not time out the autograder (e.g. they cannot be grossly inefficient).

You can submit to Gradescope as often as you like to verify whether your tests finish within the allotted time (usually about ~10 minutes max).

3.8. Golang

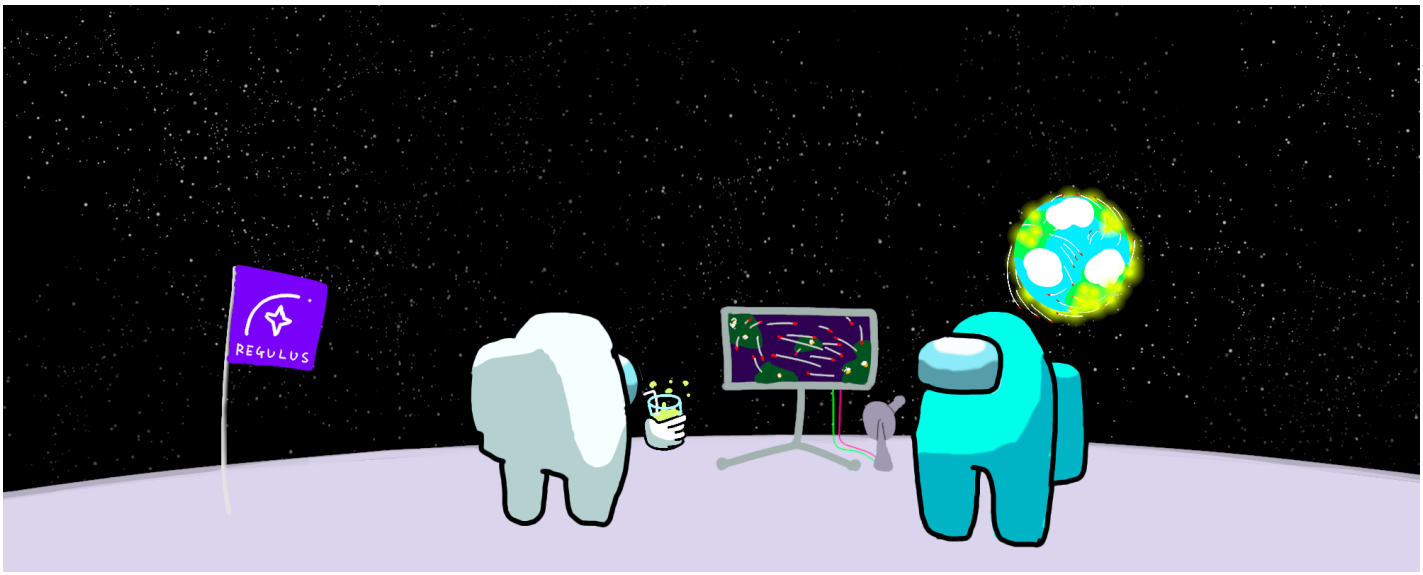
1 The client application code MUST NOT use global variables (except for basic constants).

2 Your Go functions MUST return an error if malicious action prevents them from functioning properly. Do not panic or segfault; return an error.

3 Your Go functions MUST return `nil` as the error if and only if an operation succeeds, and a value other than `nil` as the error if and only if an operation fails.

4. Server APIs

Just for fun. No project-relevant content in this box.



We guarantee 100% data durability. REGULUS is able to retain your data even in natural and artificial disasters up to and including global thermonuclear war. However, data durability may be compromised in event of a second space race.

TABLE OF CONTENTS

5. Client Application API

> **Staff Advice:** Design a solution before starting the implementation. Students consistently agree that design is harder than implementation across multiple iterations of this project. A faithful implementation of a faulty design will not earn you many points.

The API for each of the eight functions that you are required to implement in client.go are documented here.

To see an example of this API in use, see the client_test.go file.

TABLE OF CONTENTS

5.1 InitUser

```
InitUser(username string, password string) (userdataptr *User, err error)
```

Creates a new User struct and returns a pointer to it. The User struct should include all data required by the client to operate on behalf of the user.

Returns an error if:

1. A user with the same username exists.
2. An empty username is provided.

Parameters: **username** (*string*), **password** (*string*)

Return: **userdataptr** (**User*), **err** (*error*)

Do not forget that your design must satisfy all requirements, including:

- The client application must not use the local filesystem.
- Users can have multiple active user sessions at once.

5.1 GetUser

```
GetUser(username string, password string) (userdataptr *User, err error)
```

Obtains the User struct of a user who has already been initialized and returns a pointer to it.

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.

Parameters: **username** (*string*), **password** (*string*)

Return: **userdataptr** (**User*), **err** (*error*)

Do not forget that your design must satisfy all requirements, including:

- The client application must not use the local filesystem.
- Users can have multiple active user sessions at once.
- You cannot use global variables to store user information.

5.3 User.StoreFile

```
StoreFile(filename string, content []byte) (err error)
```

Given a `filename` in the personal namespace of the caller, this function persistently stores the given `content` for future retrieval using the same `filename`.

If the given `filename` already exists in the personal namespace of the caller, then the content of the corresponding file is overwritten. Note that, in the case of sharing files, the corresponding file may or may not be owned by the caller.

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.

Returns an error if:

1. The write cannot occur due to malicious action.

Parameters: **filename** (*string*) - the name of the file
content (*[]byte*) - the contents of the file

Return: **err** (*error*)

Do not forget that your design must satisfy all requirements, including:

- Different users can store files using the same filename, because each user must have a separate personal file namespace.
- Overwriting the contents of a file does not change who the file is shared with.
- Users can have multiple active user sessions at once.

5.4 User.LoadFile

```
LoadFile(filename string) (content []byte, err error)
```

Given a `filename` in the personal namespace of the caller, this function downloads and returns the content of the corresponding file.

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.

Parameters: **filename** (*string*) - the name of the file

Return: **content** (*[]byte*), **err** (*error*)

Do not forget that your design must satisfy all requirements, including:

- Users can have multiple active user sessions at once.

5.5 User.AppendToFile

```
AppendToFile(filename string, content []byte) (err error)
```

Given a `filename` in the personal namespace of the caller, this function appends the given `content` to the end of the corresponding file.

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

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.

Parameters: **filename** (*string*) - the name of the file
content (*[]byte*) - the data to be appended

Return: **err** (*error*)

Do not forget that your design must satisfy all requirements, including:

- The implementation of this method must satisfy the append efficiency requirement.

- Users can have multiple active user sessions at once.

5.6 User.CreateInvitation

```
CreateInvitation(filename string, recipientUsername string) (invitationPtr UUID, err error)
```

Given a `filename` in the personal namespace of the caller, this function creates a secure file share invitation that contains all of the information required for `recipientUsername` to take the actions detailed in Sharing and Revoking on the corresponding file.

The returned `invitationPtr` must be the UUID storage key at which the secure file share invitation is stored in the Datastore.

You should assume that, after this function is called, the recipient receives a notification via a secure communication channel that is separate from your client. This notification includes the `invitationPtr` and the username of the caller who created the invitation.

Note that the first parameter to the `StoreFile()`, `LoadFile()`, and `AppendToFile()` functions in the client API is a filename in the **caller's personal namespace**. The recipient **will not** have a name for the shared file in their personal namespace until they accept the invitation by calling `AcceptInvitation()`.

You may assume this function will not be called on a recipient who is already currently authorized to access the file (see Sharing and Revoking).

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 complete due to any malicious action.

Parameters: **filename** (*string*) - the name of the file to share with the recipient
recipient (*string*) - the username of the user with whom the file should be

shared

Return: **invitationPtr** (*UUID*), **err** (*error*)

Do not forget that your design must satisfy all requirements, including:

- All Sharing and Revocation requirements.

5.7 User.AcceptInvitation

```
AcceptInvitation(senderUsername string, invitationPtr UUID, filename string) (err error)
```

Accepts the secure file share invitation created by `senderUsername` and located at `invitationPtr` in Datastore by giving the corresponding file a name of `filename` in the caller's personal namespace.

After this function returns successfully, the caller should be able to take the actions detailed in Sharing and Revoking. Note that the first parameter to the StoreFile(), LoadFile(), and AppendToFile() functions in the client API is a filename in the **caller's personal namespace**; accepting the invitation allows the caller to choose a name for the shared file in their personal namespace.

After this function returns successfully, the given invitation is considered accepted. Calling this function on an invitation that is already accepted is undefined behavior and will not be tested, which means your client can handle that scenario in any way that you feel makes sense.

Returns an error if:

1. The caller already has a file with the given `filename` in their personal file namespace.
2. The caller is unable to verify that the secure file share invitation pointed to by the given `invitationPtr` was created by `senderUsername`.
3. The invitation is no longer valid due to revocation.
4. The caller is unable to verify the integrity of the secure file share invitation pointed to by the given `invitationPtr`.

Parameters: **senderUsername** (*string*) - the username of the sender
invitationPtr (*UUID*) - the UUID storage key at which the sender's secure

file share invitation is stored in the datastore
filename (*string*) - the filename that the recipient would like to assign to the shared file in their personal file namespace

Return: **err** (*error*)

Do not forget that your design must satisfy all requirements, including:

- There must only be a single copy of the file.
- Users can have multiple active user sessions at once.

5.8 User.RevokeAccess

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

Given a `filename` in the personal namespace of the caller, this function revokes access to the corresponding file from `recipientUsername` and any other users with whom `recipientUsername` has shared the file.

A revoked user must lose access to the corresponding file regardless of whether their invitation state is created or accepted.

The client MUST prevent any revoked user from using the client API to take any action on the file. However, recall from Threat Model that a revoked user may become malicious and use the Datastore API directly (see Sharing and Revoking).

After revocation, the client MUST return an error if the revoked user attempts to take action through the Client API on the file, with one exception: the case in which a user calls StoreFile on a file that has been revoked is undefined behavior and will not be tested.

You may assume this function will only be called by the file owner on recipients with whom they directly shared the file (see Sharing and Revoking).

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 complete due to malicious action.

Parameters: **filename** (*string*) - the name of the file in the caller's personal file namespace

recipientUsername (*string*) - username of the user to revoke access from

Return: **err** (*error*)

Do not forget that your design must satisfy all requirements, including:

- All Sharing and Revoking requirements.