Dropbox Project

Design (Part 1) due: 11:59 pm, Monday, April 8
Implementation (Part 2) due: 11:59 pm, Thursday, May 2
Pentesting (Part 3) due: 11:59 pm, Thursday, May 9

Contents

1 Introduction

In this project, you will implement a simple "dropbox" service that allows users to upload, access, modify, and delete files. Most importantly, you will implement it *securely*. This project will give you experience not only with writing secure software, but equally importantly, with *designing* secure software. The key to creating secure software is careful thinking and design before you write a single line of code.

1.1 Partners

You are *required* to work with a partner on this project. When you have found a partner, *one* partner from each team should submit this form to register the partner that they will be working with: `https://forms.gle/Ysb8Awei8hKJafHz9`

Everyone, regardless of whether or not they have a specific partner they want to work with, must fill out this form by **Thursday, April 4 @ 5:00 PM**. If you don't already have a partner in mind, you can use Piazza's "Search for Teammates" feature to find a partner! If you are unable to find a partner (or want to be assigned randomly), you should still fill out the "Partner 1" fields and leave the "Partner 2" fields blank and we will assign you randomly.

Important: CS162 students have additional project requirements that must be completed (see the handout for more details). We recommend that teams should only be comprised of only CS166 students (or only CS162 students), as teams comprised of both CS166 and CS162 students must complete all of the CS162 requirements for the project. (Since the CS162 requirements are extra credit for CS166 students, in this latter case, the CS166 student would receive extra credit for the additional features implemented while the CS162 student would be graded normally according to the CS162 guidelines described in this handout.)

1.2 Project Overview

This project is divided into three phases:

- **Design** (20%): In the Design phase, you will write a *detailed* design document that details your plan for your Dropbox service.

- **Implementation** (60%): In the Implementation phase, you will implement your Dropbox service in Go. We will provide support code that sets up a basic (but very insecure) implementation of the Dropbox service—it's up to you to extend the code to provide a secure service for your users.
- **Pentesting** (20%): In the Pentesting phase, you will be penetration testing other Dropbox implementations created by your TAs when they took the course. During this phase, you will work individually (that is, you will not work with your partner from the Design and Implementation phases).

More information about the latter phases of the project will be released after the previous phase's due date. Please refer to the front page of the project handout for more information on due dates for each of these projects.

1.3 Late Days

No late days may be used on any of the deadlines for the final project. Please consult the syllabus for more information about late handins.

If there are extenuating circumstances preventing you from completing an assignment on time (e.g., illness), please contact the instructor before the assignment is due via the "Extension Requests" form on the `http://cs.brown.edu/courses/cs166/resources/` page.

2 Specification

In this section, we outline the base specification for your Dropbox service.

You will implement your Dropbox service in two parts: a client and a server (these are described more in depth later in the handout). You will be given a lot of leeway on how to design and implement your service. However, there are some basic requirements of what your service must do—it is not sufficient to implement a service that does nothing and proclaim that you've succeeded in creating a secure service. What's difficult about security is not simply creating secure systems—it's creating systems that *do interesting things* while still remaining secure.

2.1 Requirements

Below we describe the minimum requirements for your service. Where things are unspecified, it is up to you to make reasonable design decisions that are in the interest of the overall security of your service.[1]

- Accounts and support for multiple users
  - You must support an arbitrary number of users; while you might be limited by practical limitations such as disk space, you can't, for example, create ten users ahead of time and not let new users sign up.
  - Users must be able to sign up for new accounts.
  - Users must be able to login and logout.
  - Users must be able to delete their own account.
  - You may choose how you handle users, signing up, and so on. For example, you may decide that each user is identified by a username that they choose, or a randomly assigned user ID, or an interpretive dance that they perform every time they want to access or modify a file. You can also choose how you authenticate that users are themselves (perhaps the interpretive dance approach would work well here).
  - There must be some conception of "sessions." In particular, your authentication mechanism must *not* simply work by providing the user's login credentials with every API request. Further, it must be the case that if any session information kept by the client is stolen, after a certain time period, the session will expire and this information will no longer be sufficient to authenticate an attacker (if it ever was sufficient in the first place).
  - Any authentication data stored on the server must be stored in a form such that, if it were to be stolen, it would not immediately allow an attacker to impersonate a legitimate user (for example, if passwords are used, they must be hashed and salted).
- Allow users to upload, download, modify, and delete files
  - Users must be able to upload new files.
  - Users must be able to access the contents of the files that they have uploaded.
  - Users must be able to list the contents of their own directories.
  - Users must be able to modify the contents of files that they have uploaded. (*Note*: For modification, it is sufficient to upload a new file with the same name as the old one and replace it. You do *not* have to implement a text editing feature.)
  - Users must be able to `cat` files.
  - Users must be able to delete files that they have uploaded.

[1] Ask the TAs what constitutes "reasonable" if you're unsure about a particular idea.

  - Users must be able to refer to files by a path from their root directory, just as in Unix systems (root directories are discussed below).
  - Users must be able to choose what to name files when they are created (that is, it's not acceptable to give files random IDs and only allow the user to access, modify, or delete the file if they can guess the random ID; it's also not acceptable to let a user upload a file but then inform them that the file has been named "dinosaurs.txt" and that they can't change it).
  - You may, if you wish, impose reasonable limits on these names or paths (for example, you could require that references contain only certain characters, or that they aren't longer than some reasonable character length limit).
  - You may, if you wish, impose a reasonable cap on file sizes.
  - You may, if you wish, impose a reasonable cap on the total storage used by a user.
    * You may want to consider whether both of these size limits make sense together, or if one alone is sufficient.
  - You may *not* impose a limit on the number of files that a user may upload, except for limits that are implied by your other restrictions (for example, if you have a total storage limit of 100MB, and you count file storage usage such that each file takes up a minimum of 1 byte, users would be effectively prevented from uploading more than 100,000,000 files).
- Directories
  - Each user has an associated "root" directory in which all of their files and directories are stored (just like on Dropbox or Google Drive).
  - All of the user's files are stored either in this root directory or in subdirectories under the root.
  - All of the user's directories are stored either in this root directory or in subdirectories under the root.
  - Users must be able to create directories.
  - Users must be able to delete directories.
    * It's up to you to decide how to handle users who request to delete a non-empty directory.
  - Users must be able to, when they upload a file or create a directory, specify what directory it will be stored in.
  - Users must be able to choose what names to give to directories, just as with files.
  - You may, if you wish, impose reasonable limits on these names, just as with files.
  - You may *not* impose a limit on the number of directories that a user can have, except for limits that are implied by your other restrictions (for example, if you use traditional Linux-style file paths, and limit the total length of file paths, there's an upper bound on the number of possible file paths).
  - A user who has authenticated has a "current working directory" analogous to that on a Unix system.
  - A user may give a reference to a file or directory that is relative to this current working directory.

2.1.1 Extra Credit

For extra credit, you may implement additional features. We will allow extra credit for the following features, and may allow extra credit for other features—please confirm with us if you want to implement an extra credit feature not on this list:

- Sharing—see the CS162 section below for details.
- Deduplication—see the CS162 section below for details.
- File/folder integrity—the client keeps some small amount of data stored locally that allows it to verify the integrity of the contents of files and folders returned from the server. Thus, the server is unable to act maliciously and convince the client of false file names, false contents of files, or false directory contents.

You do *not* need to include extra credit features you will implement in your design doc. (Since sharing and deduplication are required features for CS162 students, CS162 should include design details for sharing and deduplication in their design documents.)

2.1.2 CS162

CS162 students must additionally implement sharing and deduplication, defined as follows:

- Sharing for files
  - Users must be able to share files with other users (you do *not* need to be able to share directories).
  - Users must have some way of referring to files that are shared with them.
  - It must be that all users have access to a live version of the file such that edits by any user on a given file affect all users' copies of the file (this is how sharing works, for example, on Google Drive).
  - When a user shares a file that they own with other users, they must be able to specify either read-only or read-write mode. In read-only mode, the user with whom the file is shared can see the file (including any future updates to it), but is not able to make modifications of their own.

    Note that there are concurrency issues here (for example, what happens if user A downloads a copy of a file, and then user B downloads a copy of the same file, and then user A makes an edit to their local copy and uploads the new file resulting from the changes?). You are not responsible for handling these in any clever manner—doing the naive thing and simply accepting upload requests is fine.
  - Users must be able to modify sharing on a file.
    - Users must be able to modify the permissions that a specific other user has on a given file (changing read-only to read-write, or vice-versa).
    - Users must be able to remove permissions that a specific other user has on a given file (that is, un-sharing the file with them).
  - Note that there are some subtleties and edge cases that we have left unspecified. In these cases, it is up to you to do something reasonable, and document your choice.
- Server-side deduplication
  - The server must be designed so that if multiple copies of the same file are uploaded (where two files are "the same" if they have the same contents), even if they are uploaded by different users,the server only stores the contents of the file once. It is acceptable for some amount of metadata to be stored for each copy of the file. This is known as "deduplication," and it should have no effect on how users experience the server - the behavior of all API calls should be unaffected, as should the security properties.

2.2 Client/Server Architecture

You will implement your service as two components: a client and a server. The bulk of the work will be in designing and implementing the server, as it is the software which is more complex and which must be secure—the security of the client software is out of scope for this project.

Both your server and your client will run as compiled Go binaries. We will supply support code that handles the majority of the necessary networking code between the server and the client (discussed more below).

You should consider the following technical requirements when you're designing your Dropbox service:

- It must be possible to start your server simply by running the server binary (though you may specify any command-line flags you deem necessary).
- There must be a way to cause the server to exit cleanly so that all necessary state is saved. Most importantly, it must be the case that if the server is shut down and then started up again, no data has been lost. However, you may assume that all shutdowns are clean—you don't have to deal with the case in which the server binary is stopped before it has a chance to clean up, or the server crashes, etc.

2.2.1 Support Code

The support code we will provide handles the majority of all necessary networking code needed for the client to communicate with the server. It allows the client to invoke functions on the server as if they were running in the same process on the same machine (the server cannot invoke functions on the client). This is known as a *Remote Procedure Call (RPC)*. It is guaranteed that no two of these functions will ever be running at the same time, so the functions called on the server may be written with the assumption that no other code will be running at the same time. Additionally, the server must provide a finalizer function, which will be run once when the server shuts down (when a user types `CTRL+C` on the command line).

More information about the support code (and the support code itself) will be released after the Design phase due date.

2.2.2 Standard Library and Third-Party Code

In general, use of standard library or third-party code is allowed so long as that the third party code:

- Doesn't implement any high-level security functionality (for example, cryptographic primitives like hash functions or encryption are fine, while entire authentication frameworks are disallowed).
- Doesn't implement any significant feature of Dropbox for you (for example, for CS162 students, it would not be acceptable to implement deduplication by using a deduplicating database).

Please ask the course staff if you are unsure if a given standard library or third-party package is allowed for the Dropbox project.

2.2.3 Dependencies for the Server

Whatever state your service stores persistently (in files, in databases, etc) must be stored within a single folder on the machine it runs on (though this can have sub-folders). This especially means that you must not rely on system-wide services such as MySQL/PostgreSQL/etc. Databases that are entirely local (e.g., SQLite) are fine. This requirement is intended to make your service more portable and less tied to the machines they are installed on. This will also make it both easier for you to develop your service and also easier for the TAs to run your service alongside others. It will also mean that if you ever mess things up beyond repair, you can easily wipe away your state and start over.

To be clear, your server binary must be able to be invoked in the following manner:

`./server <path-to-server-dir> <server-addr>`

where `path-to-server-dir` is a path to a directory on the machine your server is running on that will store all of the necessary files, subdirectories, etc. that are necessary for your server to run properly, and `server-addr` is the address for the server to listen for client connections on.

Additionally, you will have to provide a `--reset` option to your server binary that will reset the file structure/data storage to its original blank version. That is, running the following command:

`./server --reset <path-to-server-dir>`

will generate a directory at `path-to-server-dir` that should contain all of the necessary files to start your server for the first time.

You do not need to worry about the command-line arguments to your server during the Design portion of the project, though we provide this information now so you have an idea of what you'll need to be able to do when you start implementing the project.

2.3 Security Assumptions

You will not be required to protect against the following types of attacks:

- Denial-of-service attacks that involve connecting large numbers of clients to the server (or other similar attacks that involve clogging the network connection).
- Attacks that involve eavesdropping on unencrypted connections.
- Attacks that exploit vulnerabilities in the support code. Any vulnerabilities in the support code will neither count against your project nor will be given any credit during the penetration testing phase (though if you find any, please report them!).

Part I

Design

3 Assignment

In the first phase, you will write a detailed design document for your Dropbox service. A critical part of creating secure software is careful, thoughtful design. We expect you to spend a good amount of time and effort on this—that's why this document constitutes 20% of the final grade for the project.

Note that some elements of the design are left intentionally open and are not constrained to a particular design choice by the requirements presented above. This is intentional. This project is designed to give you a chance to do some critical thinking about security design in a broad sense. We're not just trying to test your ability to pick a strong hash function or avoid path escaping vulnerabilities. We encourage you to do lots of brainstorming, and consider many possible designs.

Relatedly, you should not pick a particular design, and then try to figure out how to secure it. Instead, you should consider different designs, and for each, how much it will naturally lend itself to being secure. Often you will find that an entire class of vulnerability may go away entirely when the right design is chosen (though likely not without introducing a new set of vulnerabilities to contend with). Finding and choosing a design that leaves you with a manageable set of potential vulnerabilities is very subjective, and again we emphasize that spending some time brainstorming with your partner will be very beneficial here.

3.1 Guidelines

Your design document should include the following sections:

- *High-level security goals.* Describe in layman's terms what the high-level security goals of your service are for authentication, access control, and file storage (for CS162 students, sharing and deduplication as well). For example, "users should not be able to read each others' files".
- *Authentication.* This section should, at a minimum, address the following questions:
  - How will users be identified (e.g., username, user ID, etc)?
  - How will users prove their authenticity to the server?
  - How will an authenticated user prove that they have already been authenticated (sessions)?
  - How will authentication be verified by the server?
  - How will authentication information be stored on the server? What about when the server isn't running (that is, how will authentication information be saved when the server is stopped and then started again)?
  - How will sessions be implemented, including creation, validation, and expiration?
- *Access control.* This section should, at a minimum, address the following questions:
  - Given a request from a client which you have determined as being from a particular user (that is, the request includes the requisite authentication information), how will you determine whether or not the request should be allowed? Keep in mind that this will likely depend on other aspects of your design.
- *File storage.* This section should, at a minimum, address the following questions:
  - How will file data be stored on the server?
  - Given a file path supplied by the client, how will you determine:

    - Whether this identifies a file, a directory, or does not exist
    - If it's a file, where this file's data is stored
    - If it's a directory, what files/directories are inside it
  - How will you ensure that users do not have access to one another's' file trees?
  - [CS162 Students Only] Be sure to cover deduplication here as appropriate.
- [CS162 Students Only] *Sharing.* This section should, at a minimum, address the following questions:
  - How will sharing data be stored?
  - How will shared files be referred to by users with whom they are shared? That is, will they live somewhere inside the user's root? If so, where? If not, how will they be referred to?
  - How will you handle making updates to sharing information?
- [CS162 Students Only] *Deduplication.* This section should, at a minimum, address the following questions:
  - How will it be detected when two duplicate files are uploaded to the system?
  - How will it be detected when a given file no longer exists on the system? In other words, if a file is deleted or a file is replaced by a new file with different contents, how will it be determined whether the old file had any duplicates, and thus whether to delete the storage or not?
  - How will deduplication interact with sharing?
- *Persistence.* How will you ensure that data persists between runs of your server program? This may overlap somewhat with other sections, which is fine.
- *API.* What methods will your server expose to be callable by the client? For each, specify:
  - What are the arguments to this method?
  - What are the return values from this method?
  - What are the semantics of this method (what is its purpose / what does it do with arguments)?
  - What, if any, authentication will be performed on this method?
  - What, if any, access control will be performed on this method?
- *Client computation.* What work will be done only on the client, that the server may assume has already been performed?
- *Vulnerabilities and mitigating their risk.* For authentication, access control, and file storage (for CS162 students, sharing and deduplication as well):
  - What vulnerabilities might you expect out of a system employing your design?
  - How will you ensure that your implementation does not have these vulnerabilities? Explain how you will reduce the risk of introducing vulnerabilities during the implementation of your design. This must include concrete actions or approaches (for example, saying "we will be very careful" is not sufficient).
- *Verification.* Explain how, after you have implemented parts and all of your service, you will analyze it to verify that it meets your stated security requirements, and does not contain any vulnerabilities. During implementation, this should occur both after large independent components are implemented and additionally after the entire service is completed.

You do not need to describe how you will implement any of the networking components handled by our support code (for example, you don't need to describe how file data is transferred from the server from the client); see the "Support Code" section earlier in the handout for more information.

3.2 Advice

In designing your service, you may find the following pieces of advice helpful:

- Security should be thought of as a property of a design, not as a feature. If you approach the problem by first designing a service, and then afterwards adding security, your design will be complicated, and likely quite insecure (this is essentially how the Web was designed, and we've seen how that turned out). Instead, you should aim to have security be a goal that drives what design choices you make throughout the process.
- Complexity is the enemy of security. The simpler your design is, the easier it will be for you to reason about its behavior, and the less likely it will be to behave in surprising, insecure ways. Again, the Web is a good example of what can happen when this design principle isn't followed.
- One way to simplify your implementation is to push logic to the client. In particular, you may want to make it so that the client performs any kind of input transformations or parsing. This way, your server can take these input values in an already-parsed form.
- This doesn't mean, of course, that you should trust the client. From the server's perspective, the client is completely untrustworthy, and all inputs from it should be treated as hostile (and in fact, graders *will* send your server hostile input, bypassing any sorts of validation done by your client).
- Any kinds of security decisions should be made after inputs are fully parsed. You want to make sure you have parsed input from the client and interpreted it fully before proceeding to execute that input. For example, in access control, you want to make sure you have parsed a path before accessing that path. If the server tries to access a path before parsing that path, it may end up allowing a user to access something they should not have been able to access.

3.3 Handing In

Your design document should be formatted as a PDF. It does not need to be a formal document (that is, you may use bullet points when describing your service's design).

You should *clearly mark in your document whether or not the team includes at least CS162 student* (which means you must complete the CS162 requirements for the design document).

Please hand in your design document on Gradescope. Only **one** partner should hand in the PDF—you must use the team selection dropdown in Gradescope to select your partner's name when you hand in. (A deduction may be applied if more than one partner from a team hands in or the team is not correctly selected on Gradescope.)

3.4 Design Discussions

There will be "design discussions" after the Design phase deadline in which you and your partner will meet with a TA and have the opportunity to discuss your design for the Implementation phase for 30 minutes.

These are not interactive gradings—when you meet with your TA, your design document will have already been graded (you will receive your Design grade at the beginning of this meeting). These meetings are designed to help your team translate your design ideas into practice as you start implementing your Dropbox service as well as give you an opportunity to have dedicated time with a TA for improving your project's design before you start implementing it.

We will send out information on how to sign up for design meetings after the Design phase deadline.

Part II
Implementation

4 Assignment

In the second phase, you will implement your Dropbox service.

In the "Specification" section of the handout, we have already discussed some of the technical details regarding what we expect for your service's implementation—we won't reiterate those details here, so you should revisit that section before continuing with the second phase of the project. Below, we discuss how to get started with the support code for your implementation as well as some requirements to keep in mind when you're implementing the project.

4.1 Github Classroom

The support code for this assignment will be distributed on a Github repository via Github Classroom. To access the code, **one** partner from each team should follow these steps:

- Register a new team for the Dropbox assignment at `https://classroom.github.com/g/s4aWDiaB`. Your team name should be comprised of the CS logins in the partnership separated by hyphens. (For example, the group name for the team of `zespirit` and `zkirsche` should be "`zespirit-zkirsche`".) The order of the CS logins does not matter.
- When you register your team, this will take you to a page that will automatically set up your team's repository and import the Dropbox stencil code. This may take a few minutes. Once it's done, you will have access to a Github repository containing the support code for the assignment.

All other team members should do the following:

- Find the team that your partner created for the Dropbox assignment at `https://classroom.github.com/g/s4aWDiaB`, then click on "Join" next to the team name.
- This will add you to your team's Github repository. You should now have access to the same Github repository as your partner.

TAs will be able to assist helping teams set up their Github Classroom assignment during the "design discussions" regarding the Design phase of the project.

You are required to use Git and Github for this project. This will help facilitate collaboration between you and your partner as well as allow us to better evaluate your work in the end. If you're looking for a Git tutorial, we recommend starting with the tutorials at `https://try.github.io`.

4.2 Support Code, In More Detail

Your Github repository will be pre-populated with the support code. This support code consists of an example Dropbox "implementation" client and server that are meant to illustrate how to use the *Remote Procedure Call (RPC)* framework provided for calling server functions from the client. The support code we provide establishes a *very insecure* implementation of the Dropbox service—you should only use the code as an example of how to use the remote procedure call framework described in the "Client/Server Architecture" section above.

Comments in the code go into more detail about the purpose of each of the files we provide, but here we provide a general overview of the code.

The support code is split up into the following directories:

- `internal` - code common to both the client and the server
- `lib/support` - our support code
- `client` - the source code of the client
- `server` - the source code of the server

The key piece of code for you to look at is the `Client` interface defined in `lib/support/client`. In the Design phase, we asked you to come up with the necessary API methods for your Dropbox system in order to get you to start concretely thinking about the project—now that we're in the Implementation phase, we provide this interface as a recommendation for what functions should be included in your API and the inputs and outputs for each of those functions.

We *recommend* (but do not require) that your client implementation implement the provided `Client` interface. This will allow you to use the `RunCLI` function in the `lib/support/client` package, which will automatically set up a command-line REPL for your client. However, if you modify the `Client` interface, you're welcome to change the `RunCLI` implementation to fit your new `Client` implementation.

The idea behind the `Client` interface is that it represents an authenticated client. Remember that "authentication" is design-dependent, so you will need to write all code relating to authentication yourself (including the command-line interface that the user interacts with for authenticating themselves). After the client has successfully authenticated, you should construct a value of a type that satisfies this interface, and pass it to `RunCLI` in order to run the CLI. Once `RunCLI` returns, you should perform any post-logic such as logging out of the server (again, this is dependent on your design).

5 Requirements

Your second handin will be the completed service. It must properly and securely implement all functionality outlined in the "Specification" section of the handout. Additionally, it must satisfy the requirements listed in the following sections.

5.1 Client Command Specification

In order to provide consistency between Dropbox implementations for grading purposes, each of the standard commands that clients in your Dropbox implementation will rely on (`cd`, `cat`, `ls`, etc.) has a particular input and output format that you are required to follow.

The client implementation in the support code[2], by default, prints out the output of each of the commands in the required format already, so you can use the support code to check that the result of each of your commands is printed out in the correct format. Additionally, the test cases that we provide for our system tester (discussed below in Section 5.2.1) provide a few examples of the expected output for a given set of inputs, so you can reference those test cases to determine what output you should print out for a given scenario.

Our command output requirements are listed below. Note that all command descriptions assume access is permitted by the calling user—it is up to you to handle any edge cases in a reasonable manner. You are welcome to add any commands to supplement these in your implementation if you think they are necessary (for example, you will probably need to implement sharing-related commands if you are implementing sharing), but at a minimum your client implementation must support the commands below.

[2]As a reminder, the support code we provide already provides a very basic implementation of the Dropbox service as a whole (without users or authentication).

5.1.1 List of Commands

- `upload <localpath> <remotepath>`

  Semantics. Uploads the file at `<localpath>` (a path to a file on the client's filesystem) to `<remotepath>` (a path in the Dropbox server).

  Output. No output should be printed on success.

- `download <remotepath> <localpath>`

  Semantics. Downloads the file at `<remotepath>` from the current user's dropbox account to client's local filesystem at `<localpath>`

  Output. No output should be printed on success.

- `cat <file-path>`

  Semantics. Consumes a `<file-path>` to a file on the server..

  Output. Prints the contents of the file if it exists.

- `ls [<path>]`

  Semantics. If no `<path>` is specified, prints out all of the files and the directories in the current working directory. Otherwise, prints out all of the files and the directories in the directory specified by `<path>`.

  Output. See *Semantics*. Each directory is denoted with a leading character "`d`"; each file is denoted with a leading character "`-`". Directories and files are printed out in alphabetical order.

  As an example—assume we have a file named `test.pdf`, a directory named `tmp`, and a directory named `secret` in the current working directory:

  ```
  $ ls
  d secret
  - test.pdf
  d tmp
  ```

- `mkdir <path-to-newdir>`

  Semantics. Makes a new directory at `<path-to-new-dir>`.

  Output. No output should be printed on success.

- `rm <path>`

  Semantics. Removes the file or directory at `<path>`.

  Output. No output should be printed on success.

- `pwd`

  Semantics. Prints out the current working directory.

  Output. See *Semantics*.

- `cd <path>`

  Semantics. Changes the currently logged-in user's current working directory to `<path>`.

  Output. No output should be printed on success.

- `quit` or `exit`

  Semantics. Closes the client program.

  Output. No output should be printed on success.

5.2 Testing Mode

While the contents of your client's REPL prompt and the details of your system's authentication are up to you, you must make sure your Dropbox client implements a particular feature in order to make your system easier to grade.

At the top of the `main` function provided in the support code for the client in `client.go`, you will see the following block of code:

```
// Check if we're in testing mode. (See the handout for more information.)
if compileMode == "test" {
    isTestingMode = true
}
```

This automatically sets the `isTestingMode` variable (which is, by default, set to `false`) to `true` if your client binary is compiled with a special Go flag that specifies that the client should be compiled in "testing mode". Your client binary can be compiled in "testing mode" by running the `make test_client` command (as opposed to the `make client` command).[3]

If `isTestingMode` is `true`, your client binary is in "testing mode" and **must do the following**:

- The client needs to authenticate itself on start up. For example, you might register a new account, then log that account in (if your implementation does not immediately log in an account after registering it). The reason for this is that authentication processes are implementation-dependent, and as a result will not work with our automated test suite.
- The command-line REPL prompt should not be printed. Note that our support code already handles this for you—the `RunCLI` consumes a boolean flag that states whether or not to print the REPL prompt.
- Any user-defined writes to standard output (for example, output from the authentication process, debugging printouts) from your client should not print when the client is in "testing mode". Otherwise, the test suite will not run correctly with your client.

5.2.1 System Tester

To allow you to check that you've correctly set up the "testing mode" version of your client, check that your client implementation prints the expected output for each of your commands, and verify the correctness of your implementation as a whole, we've provided a basic test suite and testing program that will run system tests against your Dropbox server and client.

You can run the tester using the following command on a department machine:

```
cs166_dropbox_tester [--client <path-to-client>] [--server <path-to-server>]
                     [--test-folder <path-to-test-folder>]
```

The `cs166_dropbox_tester` will start a server instance of the server specified by the `--server` flag and connect the "testing mode" client binary specified by the `--client` flag to the server. It will then run each of the tests in the specified `--test-folder`, each of which pass input to the client and compare the output of your client to some expected output.

To get you started, we've placed a basic set of tests in `/course/cs1660/pub/dropbox/tests`. Each test is a directory containing an `input` file and an `output` file, where the `output` file is the expected output to stdout after all of the commands in the `input` file have been entered on the client. To run these tests against a client binary located at `./test_client` and a server binary located at `./server`, you could run:

```
cs166_dropbox_tester --client ./test_client --server ./server
                     --test-folder /course/cs1660/pub/dropbox/tests
```

[3]The details of how this special compilation mode works are not important to the project and are outside the scope of this course, but if you're curious, you can look at the contents of the provided `Makefile` for more information.

We *strongly recommend* writing your own set of system tests, as the tests provided do not cover all cases or commands. See the contents of each of the tests in the `/course/cs1660/pub/dropbox/tests` folder for examples of how to write your own test cases.

Make sure to run the tester program with the "testing mode" version of your client binary.

The tester program comes with some other helpful options that you can use when running it—you can see a full list of options by running `cs166_dropbox_tester --help`.

5.3 Documentation

5.3.1 Commenting

Above each function, you must document:

- The expected/allowed input values and return values.
- Any pre-conditions and post-conditions, including any security assumptions (for example, "the caller of this function has already verified that this action is allowed").
- The exact behavior of the function.

We expect extensive, descriptive, and readable documentation explaining any potentially confusing pieces of code. You will be graded on this, so when in doubt, comment.

5.3.2 README

Along with your code, you should submit a `README`, which should contain:

- Any changes that you have made to your original design during the Implementation phase, including justification for those changes.
- All testing/verification that you have done to verify that your service is secure. This includes any additional tests you may have written for the `cs166_dropbox_tester` program.
- Any vulnerabilities that you discovered during this testing/verification, and how you fixed them.
- Any third-party code or libraries that you used and how to install them.

6 Logistics

6.1 Handing In

The last commit pushed to the `master` branch on your Github Classroom repository prior to the deadline will be considered your team's final submission for the Implementation phase. (You *do not* need to—and should not—run the `cs166_handin` command.)

If you've written any additional tests for the `cs166_dropbox_tester` program, you should also include those in a folder in your repository. *Extra credit may be provided for particularly good and thoughtful tests* that are included in implementation submissions.

6.2 Grading

Out of the 60% allocated for the Implementation phase in the Dropbox project, the grading breakdown for this portion is as follows:

- 20% Functionality
- 40% Security

If any extra credit features are implemented, these will increase the overall value of the Implementation portion of the project. Functionality and Security scores will maintain the same ratio (1:2), but these will now be out of this increased total score.

Finally, you can only get Security credit for what you implement. If you were to implement only half of the required functionality, you would not be able to get more than 20% on the Security score.

Part III

Pentesting

7 Assignment

In the third phase of the project, you will be penetration testing Dropbox implementations written by your TAs when they were students in the course. Dropbox teams will not work together during this phase; instead, **all students must work individually**.

Each student will be given several Dropbox implementations to "pentest". You can find these in `/course/cs1660/pub/dropbox/pentesting`. (Please *do not* redistribute these project files outside of the course—it is a violation of the Collaboration Policy to do so.) For each implementation, you will be given each team's implementation code.

We have provided six implementations—you must pentest **three** of these and meet the target scores defined below for each. Additionally, at least one of the implementations you choose must be a CS162 implementation—these implementations are clearly marked in the folders provided. (This rule applies to all students, not just those enrolled in CS162.)

7.1 Restrictions

Please note that while you are given full access to the server, you may only use that access to set up the service, verify that your exploits have worked, and reset it if necessary. Additionally, you may not take advantage of any vulnerabilities that rely on the client and server to run on the same machine. In essence, any vulnerabilities you report must not rely on having direct access to the server in order for them to work—it must be possible to exploit them solely by making RPC calls against the server.

Please also see the prior Dropbox specification in this handout for other vulnerabilities which are outside the scope of this project.

7.2 Changes from Year-to-Year

The project specifications are updated from year-to-year based on course feedback and pedagogical changes. As a result, the implementations created by your TAs may not fully match the specification presented in this year's handout.

In particular, the implementations provided for pentesting do not necessarily follow the same client command specification as prescribed in the this year's project handout, and therefore the `cs166_dropbox_tester` program may not work with these implementations.

Recall from your experience with hacking unfamiliar systems that it is incredibly important for you to first learn how the system that you're penetration testing works before actually trying any exploits. Read the method comments fully until you understand it, and then read the implementation and make sure you understand how it and each of the components work. The better you understand how the service works, the better the chance you'll have of actually finding vulnerabilities once you start looking for them.

Remember that you have the distinct advantage of having had implemented a similar system yourself, so think about the errors and edge cases you found while implementing your implementation and use that as a starting point to attack the TAs' implementations!

8 Finding Vulnerabilities and Exploits

As in previous projects, your task will be to identify vulnerabilities and develop exploits for those vulnerabilities. Two submissions will count as distinct from one another if the vulnerabilities they leverage are distinct. Two vulnerabilities are considered distinct if they would require separate updates to the code in order to fix—ask on Piazza what constitutes "separate updates" before proceeding with two separate exploits in order to avoid losing points.

Point values for each submission will be assigned based on the severity of the exploit that you are able to accomplish. They will be evaluated on the following scale:

Exploit	**Description**	**Points**
Remote Code Execution	Execute arbitrary code on the server. This is similar to the Arbitrary Code Execution exploit from the Handin project (but you do not need code to execute with any particular permissions).	10
Account Takeover	Take control of another user's account, whether directly (by being able to impersonate their account or steal the credentials needed to login) or indirectly (arbitrarily manipulate a given user's file system as another user).	8
File Modification / Exfiltration	Being able to read or change a file owned by another user. (If you're able to arbitarily manipulate all files owned by a given user, this would be considered Account Takeover.)	6
File Deletion	Deleting a user's file that you should not have access to delete.	5
Password Hash Exfiltration	Recovering a password's hash for another user.	5
Denial of Service	Anything that crashes the server or prevents other users from using the service. (Note that attacks that involve connecting large numbers of clients to the server, or other similar attacks that involve clogging the network connection, are not permitted as per the Dropbox specification.)	4
Metadata Exfiltration	Being able to discover *sensitive* metadata such as the names of files or directories that other users have created. (This does *not* include finding out whether or not a username already is in use or not. Many websites, like Reddit and Gmail, will tell you if a username is already taken, so this isn't considered a vulnerability in the real world.)	4
Metadata Deletion	Deleting metadata.	2

An exploit will be associated with the highest-scoring severity category (but not more than one category). For example, if an exploit allows for metadata exfiltration, and the metadata stolen (such as a password) allows for account takeover, the exploit will be given 8 points, as the Account Takeover category (8 points) is worth more than the Metadata Exfiltration category (2 points).

Finally, please note that the TAs are human too and some of the *functionality* might be a little buggy. Your goal is not to find functionality errors, but to find security vulnerabilities (given the above table). If you are unsure if something that you have found is a functionality issue or a security issue, please ask the TAs on Piazza.

8.1 Target Scores

For each project, we have computed a target score based on what vulnerabilities were found during last year's grading. Your grade for each project will be the ratio of the number of points that you got out of the target score. (The target score is also a cap on the number of points you can score on a given implementation—there is no extra credit in this phase of Dropbox.)

The target scores are as follows:

- **Balthazar**: 1 point
- **Kevin**: 6 points
- **BoogieBots**: 14 points
- **Margo** (CS162): 1 point
- **LucyWilde** (CS162): 4 points
- **Vector** (CS162): 9 points

Note that implementations with a target score of 1 point only need one valid exploit to get full credit. However, the target scores are determined from known exploits on each of the systems, so implementations with lower scores may have less apparent (or, possibly, no) vulnerabilities when compared with the higher point implementations.

8.2 Exploits

The requirements for each exploit you submit are as follows:

- You must include a detailed `README` that:
  - Reports what vulnerability exists in the system that your exploit takes advantage of.
  - Explains how your exploit takes advantage of this vulnerability. This should include instructions sufficient to reproduce the exploit that would allow even a person completely unfamiliar with the system to reproduce the exploit on their own. *There should be no question how the vulnerability or the exploit works after reading your explanations.*
  - Documents the severity of the vulnerability by choosing one of the provided exploit severity categories. If you feel there is no category that adequately suits your vulnerability, please include a detailed explanation of your suggested class of vulnerability.
- Any files (such as scripts, payloads, etc.) that are needed to run your vulnerability. Not all exploits need to be scripted—if you're unsure if something should be scripted, ask the TAs on Piazza.

You do not have to describe fixes for each vulnerability.

9 Logistics

9.1 Handing In

Each project has a separate handin command—you only have to hand in pentesting results for **three** implementations. *Note that if you hand in exploits for more than three implementations, we will randomly pick three of those handins to grade*—so do not hand in pentesting exploits for a given implementation unless you're certain you want to commit to pentesting that implementation.

To hand in pentesting exploits for a given implementation, run the corresponding command in a directory containing all of your exploits for that implementation (each exploit should be in its own subdirectory, each with its own separate `README` and exploit code, payloads, etc):

- **Balthazar**: `cs166_handin dropbox_pentesting_balthazar`
- **Kevin**: `cs166_handin dropbox_pentesting_kevin`
- **BoogieBots**: `cs166_handin dropbox_pentesting_boogiebots`

- **Margo** (CS162): `cs166_handin dropbox_pentesting_margo`
- **LucyWilde** (CS162): `cs166_handin dropbox_pentesting_lucywilde`
- **Vector** (CS162): `cs166_handin dropbox_pentesting_vector`

9.2 Grading

The three implementations that you pentest will each count for $\frac{1}{3}$ of the grade for Pentesting, which counts for 20% of the final Dropbox grade.