Dropbox Project

First Due Date: 11:59 pm, Thursday April 12
Second Due Date: 11:59 pm, Thursday May 3
Third Due Date: 11:59 pm, Thursday May 10

Contents

1 Introduction

In this project, you will implement a simple dropbox service that allows users to upload, access, modify, and delete files. But 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.

2 Partners

You will do this project in partners. When you have found a partner, one of you should fill out this form.[1] You will both need GitHub[2] accounts, so sign up for one if you don't have one already.

3 Requirements

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.

- 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 - whatever. You can also choose how you authenticate that users are themselves (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.

[1] `https://goo.gl/forms/sYhk87piFyFLUMUp1`
[2] `https://github.com/`

  - 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.
  - 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). *NOTE*: During Pentesting, large file uploads WILL count as attacks if they hinder the ability of other clients to use your Dropbox.
  - 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).

- Support 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.

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.

3.1 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.

4 Architecture

You will implement your service in 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.

You will be given a VM to work in. Your service will run on this VM. You are free to make any configuration changes to this VM that you want, but you must document any in your design handin or in your final README.

The primary entry point to your service (that is, the program which clients interact with directly over the network) will be an executable. It must be possible to start your service simply by running this executable with whatever command-line flags you deem necessary. There must be a way to cause the executable to exit cleanly so that all necessary state is saved. Most importantly, it must be the case that if the service 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 binary is stopped before it has a chance to clean up, or the server crashes, etc.

4.1 Security

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

- Volume-based DoS
- Attacks that exploit vulnerabilities in the support code

4.2 Support Code

You are required to write your service in Go, and to use the provided support code (for both the client implementation and for RPCs - this will be explained below). This is for a few reasons. First, it allows us to standardize the interface so that, during the penetration testing phase, and for our own testing (for grading purposes), others will know how to use your Dropbox, and will be able to write standardized tests or attacks that will work against anyone's Dropbox. Second, it allows us to provide you with implementations of a few key components that you would otherwise need to implement yourself. Third, it allows us to provide you with an autotest functionality that will run a basic set of functionality tests to ensure that your Dropbox works properly. This is *not* a replacement for your own testing, of course, but it means that when penetration testing others' projects, you can be guaranteed a base level of functionality.

The support code we provide 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).

Also note that the security of the support code is out of scope for this project. 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!).

4.3 Standard Library and Third-Party Code

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

4.4 Dependencies

Whatever state your service stores persistently (in files, in databases, etc) must be stored within a single folder (obviously 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. It will make it both easier for you to develop your service and also easier for TAs and penetration testers 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.

Additionally, you must provide a `--reset` option to your server binary that will reset the file structure/data storage to its original blank version.

4.5 Helpful Packages

You may find the following packages in Gos standard library helpful.

- crypto/rand
- io/ioutil
- os
- path/filepath

4.6 Github

For the implementation portion of the project, you will be given your own GitHub repository for collaboration with your partner. While you are not required to use these, they *will* be required once we enter the penetration testing phase of the project, so it will be easier if you just start using them up front (plus, they should make collaboration much simpler). If you are not comfortable with Git, you can check out this introduction.[3]

Your GitHub repository will be located at `https://github.com/brown-csci1660/s18-userA-userB`, where `userA` and `userB` are your two CS department logins in ascending alphabetical order.

Your GitHub repository will be pre-populated with the support code and an example client and server that are meant to illustrate how to use the support code. Briefly, there are 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

[3] `https://try.github.io`

The key piece of code for you to look at is the Client interface defined in `lib/support/client`. Your client implementation must implement this interface, and must use the `RunCLI` function in the same package to run your command-line interface. The idea behind this interface is that it represents an authenticated client (whatever authentication means, which will be dependent on your design). This means that you will need to write all code relating to authentication yourself (including the command-line interface that the user interacts with). 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).

Additionally, we provide a function which runs tests against your `Client`, `TestClient`. Passing these tests is necessary (but not sufficient) to get a full functionality grade for your implementation.

All code that is meant only as an example is marked with a comment: `// EXAMPLE CODE`

5 First Handin - Design

Due: 11:59 pm, Thursday April 12

Your first handin will be a detailed design document. A critical part of creating secure software is careful, thoughtful design. We expect you to spend a lot of time and effort on this - that's why we're giving you a week to do it. Additionally, it will constitute 25% of your final grade for the project.

Note that some elements of the design are left intentionally open, 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 brainstorming will be very beneficial here.

Your design document should include the following sections:

- *High-level security goals.* Describe in layman's terms what the security goals of your service are (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?
  - How will sessions be implemented, including creation, validation, and expiration?
  - What vulnerabilities might you expect out of a system employing this design?
  - How will you ensure that your implementation does not have these vulnerabilities?
- *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.
  - What vulnerabilities might you expect out of a system employing this design?
  - How will you ensure that your implementation does not have these vulnerabilities?

- *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?
  - What vulnerabilities might you expect out of a system employing this design?
  - How will you ensure that your implementation does not have these vulnerabilities?
  - [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?
  - What vulnerabilities might you expect out of a system employing this design?
  - How will you ensure that your implementation does not have these vulnerabilities?

- [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, 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?

- *Mitigating risk of 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 part or all of your service, you will analyze it to verify that it meets your stated security requirements, and does not contain any vulnerabilities. Preferably this should occur both after large independent components are implemented and additionally after the entire service is completed.

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, and thus you don't have to worry about security vulnerabilities introduced during these steps.
- 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 and penetration testers will likely send your server hostile input, bypassing any sorts of validation done by your client).
- Any kinds of security decisions (such as access control) should be made after inputs are parsed. Using raw, unparsed input to make security decisions is likely to lead to a situation in which your security code makes different assumptions about the format of the input than other code, which can in turn lead to vulnerabilities where, for example, an action is determined to be legitimate because of one interpretation of the input, but then a different interpretation is used to actually carry out the action, and the action is one that should have been disallowed.

Your design document must be a PDF, and be named `DESIGN.pdf`. Only one partner needs to hand in the PDF. Please do *not* include your logins on your PDF.

Your design doc need not be a formal document. *i.e.* You may use bullet points.

Please hand in your design doc on gradescope as a PDF file.

6 Second Handin - Implementation

Due: 11:59 pm, Thursday May 3

Your second handin will be the completed service. It must properly implement all functionality, and be secure. Additionally, we require:

- 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.

Along with your code, you will be submitting a `README` which will document any changes that you have made to your original design during the implementation. Additionally, it must document:

- All testing/verification that you have done to verify that your service is secure
- Any vulnerabilities that you discovered during this testing/verification, and how you fixed them

While you are working on GitHub, we still require that you hand in a copy of your code and `README` as normal by running `cs166_handin dropbox_cs166_implementation` or `cs166_handin dropbox_cs162_implementation` depending on which class you're in.

7 Third Handin - Penetration testing

11:59 pm, Thursday May 10 (subject to change)

In the third phase of the project, you will be penetration testing other Dropbox implementations (these will be implementations from past years, not this year). During this phase, you will work individually, not with your implementation partner.

Each student will be assigned three implementations to penetration test - one by a team in CS162, and two by teams in CS166. You will be given each teams design document and implementation, and a VM in which to run their server. You can find these in `/course/cs166/student/<your-login>/dropbox-pentesting`.

Please note that while you are given full access to this VM, you may only use that access to set up the service, verify that your exploits have worked, and reset it if necessary. Any vulnerabilities you report must not rely on being SSHd into the server in order for them to work - it must be possible to exploit them solely by making RPC calls against the server.

7.1 Details

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.

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:

- 10 pts - Remote code execution
- 7 pts - Account takeover
- 6 pts - File modification/exfiltration
- 5 pts - File deletion

- 5 pts - Password hash exfiltration
- 5 pts - Denial of service
- 4 pts - Metadata exfiltration
- 2 pts - Metadata deletion

(Data means file contents; metadata means usernames/file names/directory structure/etc.)

A vulnerability will be given the highest available number of points. For example, if stolen metadata such as passwords allows for account takeover, the vulnerability will be given 7 points. Note that these point values are estimates and we will make adjustments based on the details of the vulnerability.

7.2 Submitting Vulnerabilities

You can use this google form to submit vulnerabilities: `https://docs.google.com/forms/d/e/1FAIpQLSczPlF_nrbuGomeBOfaLyWk1zR1-sBKxbgITMphrGNiM8d78A/viewform?usp=sf_link`.

7.3 Pentesting Grades

For each project, we will compute a target score based on what vulnerabilities students last year found, and your grade will be the ratio of the number of points that you got out of the target score. You start off with 0 points per project you're pentesting.

See the Dropbox Pentesting instructions for more details: `https://cs.brown.edu/courses/cs166/files/assignments/Pentesting.pdf`

7.4 Advice

As was noted in the penetration testing lecture, 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 design document and method comments fully until you understand it, and then read the implementation and make sure you understand how it corresponds (or differs from) the design document. 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. The design documents are located at: /course/cs1660/pub.

8 Grading

- Design - 25%
- Implementation - 60%
  - Functionality - 20%
  - Security - 40%[4]
- Penetration testing of others' services - 15%

If any extra credit features are implemented, then the implementation score will increase. Functionality and Security scores will maintain the same ratio (1:2), but will be out of this increased total score.

[4]Note that you can only get security credit for what you implement. If you were to implement only half of the functionality, you would not be able to get more than 20% on the security score.

8.1 Late Days

Please note that no late days can be used on Dropbox.