Weaver
Fall 2020

CS 161
Computer Security

Project 2

An End-to-End Encrypted File Sharing System

In this project, you will design and implement a secure file sharing system in Go. Imagine something like Dropbox, but secured with cryptography so that the server cannot see or tamper your data.

Logistics:

- **Team size:** Up to two students.
- **Design document checkpoint due date:** Wednesday, October 28, 11:59 pm.
- **Implementation and revised design doc due date:** Friday, November 13, 11:59 pm.

In order to aid in immersion, this project has a story. It is just for fun and contains no relevant information about the project.

> You decide to infiltrate Kaltupia and build a network of insiders to change the company from within. Luckily for you, EvanBot's clever wits and outgoing personality already gained some friends while it was in captivity. The first thing you need to do is to establish a secure channel to communicate with your colleagues. Then you will be able to listen in on every devious plan Kaltupia concocts, always staying a step ahead.
>
> Secrecy is of the essence. You cannot afford to endanger your allies behind the lines by letting Kaltupia catch even a whiff of this plan. Yet keeping a line of communication without tipping off Kaltupia is near impossible when Kaltupia maintains a presence in all forms of technology. Heck, even using something as simple as HAM Radio is thwarted by Kaltupia's radio wave scanners.
>
> Without a place to stay out of Kaltupia's sight, EvanBot suggests a different approach: Outwit Kaltupia by flying right under their radar, hiding communication in plain sight. Protect the secret messages from Kaltupian attempts to understand it or tamper it.
>
> **REGULUS** is an unstructured cloud data storage service offered by Kaltupia Cloud Services. It provides a key-value store that can be accessed by clients using a set of provided API. By carefully protecting data stored on this storage, you believe you can upload and share sensitive data with others without alerting Kaltupian surveillance.
>
> Be warned; Kaltupia is fully expected to monitor any data stored in its servers, and if need be, strategically tamper them to protect their corporate interests. Kaltupia owns a big portion of the internet infrastructure, too, so expect them to sniff and tamper network data.

1 Project Overview

You will build the client for a file sharing system. The client will allow users to store and load files, share files with other users, and revoke access to a shared file from other users.

Users of your application will launch your client and provide their username and password. Once authenticated, they will use your client to upload and download files to/from the server. Your client will be the interface through which users can interact with the files stored on the server.

You will implement 8 functions on the client: `InitUser`, `GetUser`, `StoreFile`, `AppendFile`, `LoadFile`, `ShareFile`, `ReceiveFile`, and `RevokeFile`. In addition, you will write tests to ensure the robustness of your client. We include some basic functionality tests, but it is up to you to create a thorough test suite to cover different attacks.

We provide 2 servers that you can use:

- The first server, Keystore, provides a public key store for everyone. It is **trusted**.
- The second server, Datastore, provides key-value storage for everyone. It is **untrusted**.

Using just these two servers and your knowledge of computer security, you will implement the 8 functions above to provide a secure application. Please note that your design will have to take into account how to ensure confidentiality and integrity of files in addition to the basic file-sharing functionality. Your client must also be stateless; if the client is restarted, it must be able to pick up where it left off given only a username and password.

2 Environment Setup

Please complete the following four steps to get started:

- Install Golang (link).
- Complete the online Golang Tutorial (link).
- Download the skeleton code (§2.1).
- Read the functionality specifications (§3).

2.1 Skeleton code and user library

Skeleton code. You will use the following template for this project:

https://cs161.org/assets/projects/2/proj2.zip

All of your code should go to `proj2.go` and `proj2_test.go`:

- `proj2.go`: Your implementation goes here.
- `proj2_test.go`: Your tests go here.

User library. We have provided some useful functions in `userlib.go`. To fetch this library, run:

```
go get -u github.com/cs161-staff/userlib/
```

You can familiarize yourself with these functions by reading the online documentation (includes both userlib API and implementation requirements for individual functions):

`https://cs161.org/assets/projects/2/docs/index`

We also provide the source code for this library:

`https://github.com/cs161-staff/userlib/`

3 Functionality specifications

We provide detailed specifications for the functionality your client must implement, by specifying the desired behavior of each of the 8 client-side functions you must implement. We provide details on the functionalities you must implement in Sections 5–6 of the following site (not in this document):

`https://cs161.org/assets/projects/2/docs/index`

4 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 the functionality specs, and think about how you will provide the required functionality. It might be helpful to think about where you will store data, what data will be stored on which server, and what data structure you will use to store the data.
- If you are stuck, it is fine to start by figuring out how to provide the store/load functionality without worrying about sharing yet. 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.
- Make sure your implementation does not panic on the basic functionality tests provided in the skeleton. An implementation that panics on those tests will get a **zero** in the code section.
- Submit to the autograder once in a while. The autograder will warn you if your implementation panics in any of the hidden tests.

5 Threat model

We provide a trusted keystore server. The keystore server is trusted and will behave honestly. No adversary will be able to maliciously overwrite or tamper with a key record stored on the keystore server.

We provide an untrusted datastore server. The adversary is assumed to control the datastore server and the network. The adversary can view, record, modify, and/or add any key-value pair stored in the datastore server. In addition, the adversary can observe which users store each key-value pair and the order in which key-value pairs are stored, viewed, or deleted. The adversary can also view and modify any `access_token`(s) generated by `ShareFile` when users share files with each other. You should assume that the adversary knows the design of the system and all specifics of your implementation.

We also assume that the adversary controls one or more malicious users. It is possible that a legitimate user might share a file with a malicious user. If so, the adversary can record all relevant information about the file (including any cryptographic material and any `access_token`(s) generated by `ShareFile`) and remember it, even if access to the file is later revoked.

Finally, the adversary can read the contents of the public keystore and take advantage of automation and brute-forcing, constrained by practical limits.

6 Clarifications

- You may assume each user has a unique username, which can be of any length but not empty.
- Do not assume each user has a unique password. Two users might happen to choose the same password.
- You may assume each user's password generally is a good source of entropy, except the attackers may possess a precomputed lookup table containing hashes of common passwords downloaded from the internet.
- Do not assume every file has a unique name. User `A` can have a file named `foo`, but that should not prevent user `B` from naming one of their file `foo`.
- Files and filenames can be any length, including length of zero (empty string).
- The number of files associated with a user does not need to be hidden.
- An adversary will not conduct a rollback attack. However, a malicious datastore server should not be able to fool `LoadFile` into returning a version that mixes old and new data.
- You do not have to implement the passing of `access_token` from one user to another. You just have to implement `ShareFile` and `ReceiveFile`, and passing of `access_token` will be done for you.
- Please do not use the keystore to store anything other than the public encryption/signing keys for a user; do not abuse the keystore to store any other information. Specifically, each user is only allowed to publish a fixed amount of public keys; the number of public keys published by a user should 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.

7 General Implementation Requirements

- No global variables. Global constants are fine.
  - All the functions you write in `proj2.go` must be stateless; any data requiring permanent storage should be stored in the Keystore or Datastore.
- Return `nil` as the `err` iff an operation succeeds. Return a value other than `nil` as an error iff an operation fails.
- Your functions must return an error if malicious action prevents them from functioning properly. Do not panic, do not segfault; **return an error**.
- You must support multiple active instantiations of a given user.
  - For example, there can be two instances of a user `U`, `U1` and `U2` (one running on a laptop and one running on a phone), created by two calls to `GetUser()` with the same credentials.
  - Example: After `U1` stores a file `F1`, `U2` must be able to download it (immediately).
  - Example: After `U1` appends to `F1`, `U2` must be able to download the updated version.
  - Example: After `U1` received a file `F2` via `ReceiveFile`, `U2` must be able to download it.
  - You do not have to support concurrency; assume individual operations will be done serially.
- Your solution must protect the confidentiality and integrity of the contents of all files and `access_token`(s) generated by `ShareFile`.
  - Only authorized users—those who have been shared the file, and aren't revoked—should be able to read the contents of files.
  - Any breach of IND-CPA security constitutes loss of confidentiality.
  - Avoid reusing the same key for multiple purposes (e.g. encryption, authentication, key-derivation, etc), authenticate-then-encrypt, and decrypt-then-verify. These are dangerous design patterns that often lead to subtle vulnerabilities. We test for them and treat any instance of them as a breach of confidentiality/integrity.
- Filenames must be confidential. An adversary should not be able to guess the length of any filename, or even narrow down the set of possible values for its length. Lengths of file contents do not need to be hidden.

This is not a complete list of properties that must be guaranteed by your client implementation. Refer to the documentation for specific requirements of individual functions (§3).

8 Deliverables

8.1 Client Implementation

You must implement the following functions in `proj2.go`.

- `InitUser(username string, password string) (userdataptr *User, err error)`
- `GetUser(username string, password string) (userdataptr *User, err error)`
- `StoreFile(filename string, data []byte)`
- `LoadFile(filename string) (data []byte, err error)`
- `AppendFile(filename string, data []byte) (err error)`
- `ShareFile(filename string, recipient string) (access_token string, err error)`
- `ReceiveFile(filename string, sender string, access_token string) error`
- `RevokeFile(filename string, target_username string) error`

You are allowed and encouraged to write any structs or helper functions to aid your implementation, but all of your code must be contained in `proj2.go`.

The functionality, efficiency and security properties that must be satisfied by each function are provided in the relevant section of the online documentation. (§3)

You are not allowed to import any library other than those already listed in the template code.

8.2 Tests

Write tests for your client. Your tests should test for correct functionality of the client, correct handling of erroneous inputs, and any security problems. Add your tests to `proj2_test.go`. Each test should go in a new function. We have provided several basic functionality tests as reference. To test your code, run the following command on the terminal:

```
go test -v
```

Your tests will be graded. (§9)

8.3 Design Document

Write a clear, concise design document of your implementation.

Section 1: System Design [maximum 2 pages, plus an optional page of diagram]

Summarize the design of your system. Explain the major design choices you made, written in a manner such that an average 161 student could take it, re-implement your client, and achieve a grade similar to yours.

As **a part** of your design document, answer the following questions about your design in **number-list format**:

1. How is a file stored on the server?
2. How does a file get shared with another user?
3. What is the process of revoking a user's access to a file?
4. How does your design support efficient file append?

Section 2: Security analysis [maximum 1 page]

Present **at least three** and **at most five** concrete attacks that an attacker may conduct and explain how your design protects against each attack. You should not need more than one paragraph per analysis. You will be graded on the three analyses which provide the most credit.

You must make sure your attacks:

- Cover different aspects of the design of your system. That is, don't provide three attacks all involving file storage but nothing involving sharing or revocation.
- Do not include any of the following three kinds of attacks:
  - Breach of confidentiality of unencrypted data.
  - Breach of integrity of unauthenticated data.
  - An attack that results in leaking the length of a filename.

  An example of a more sophisticated analysis is provided here:

  "After user A shares a file with user B and later revokes access, user B may try to call ReceiveFile with the original access token to regain access to the file. To prevent a revoked user from regaining access this way, our design..."

  This analysis is satisfactory because it describes a concrete attack that can be derived from the provided threat model (§5) and the function description as specified in the online doc (§3). You may not discuss this specific attack in your part 2 writeup.

  The attack you describe must have a security consequence: It cannot be a simple bug. The attack must result in an attacker discovering forbidden information or executing an unprivileged action.

9 Grading

You will receive four grades for this project.

Autograded code [100 pt]

We will test your implementation of the client with a series of functionality and security tests to determine your code score. Due to the nature of computer security, most of the tests are hidden. Some test results are available before the deadline, and others will only be available after the deadline.

Each failed code test will incur a multiplicative penalty on your score. This means each subsequent test you fail has less impact on your grade. This is to emulate an environment where the existence of a vulnerability is more important than the exact scope of the vulnerability.

Your implementation must *at the very least* not panic on the tests provided in the skeleton. An implementation that panics on the provided tests will not receive a code score of 0.

The autograder will run multiple fixed random seeds for each test. Because your code must be robust to failures, we will take the lowest score across these multiple runs as the score for your code. Thus, any nondeterminism introduced by your code will be penalized.

Test Coverage [20 pt]

We will also test the staff implementation of the client with the tests you wrote. We check the coverage of your tests; many lines in the staff implementation are instrumented so that its execution will be a point towards your test score. A comprehensive test suite will have better coverage and increases your score. Your test score will be visible on Gradescope before the deadline.

Design document score [20 pt final draft, 10 pt first draft]

Your design and security analysis will be graded on clarity and quality in accordance with the requirements outlined in the relevant section of the design doc (§8.3).

The initial draft of the design document will be graded on completion and must be submitted by the checkpoint deadline (October 23). The final design document will be graded against a rubric and must be submitted by the implementation deadline (November 13).

The late penalty is calculated individually for the two deadlines. Late submission for the design doc, for example, will not penalize an on-time code submission.

9.1 Autograder rules

No adversarial behavior. Your implementation will be autograded against a series of hidden tests. The autograder flags and rejects any submission that imports new libraries or attempts to:

- Spawn other processes.
- Read or write to the file system.
- Create any network connections.
- Attack the autograder by returning long, long output.

Code will run in an isolated sandbox. Adversarial behavior will be dealt as cheating.

10 Submission and Deliverables Summary

There are three submissions on Gradescope, as follows:

- Implementation and testing: submit

```
proj2.go
proj2_test.go
```

- Design document checkpoint: submit

```
design.pdf
```

- Revised design document: submit

```
design.pdf
```

> silent stars and restless nights.