Weaver
Fall 2019

CS 161
Computer Security

Project 2

An End-to-End Encrypted File Sharing System

Abstract: We want to design and implement a file sharing system (like Dropbox) that protects user privacy. In particular, user files are always *encrypted* and *authenticated* on the server. In addition, users can *share* files with each other.

Logistics:

- **Team size:** Up to two students.
- **Due date:** October 30 2019, 11:59 pm.

---

1 Environment Setup

The setup consists of five steps.

- Install Golang (link).
- Complete the online Golang Tutorial (link).
- Complete the Golang Quizzes (§1.1).
- Download the skeleton code (§1.2).
- Review the autograder rules (§1.3).

1.1 Golang quizzes

The following five questions will help refresh the Golang Tutorial.

Quiz one: What does the `:=` mean in `x := 5`?

Quiz two: If an identifier in Golang starts with a non-capitalized letter, will this identifier be exported *i.e.*, accessible from outside this package?

Quiz three: How is a string converted into a byte slice?

Quiz four: How is the user structure converted into a byte slice? How is it recovered back from the byte slice?

Quiz five: Which website provides detailed documents about many Golang libraries?

1.2 Skeleton code

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

`https://cs161.org/assets/projects/2/proj2.tar`

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

- `proj2.go`: Where you will write your implementation.
- `proj2_test.go`: Where you will write the tests. You must add tests to this file. The autograder will check the completeness of your test suite (§3).

User library. You can access 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 §2, or by reading the source code:

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

1.3 Autograder rules

Rule 1: No adversarial behavior. Your submission will be executed by the autograder on a series of tests. The autograder rejects any submission that imports new libraries or attempts anything similar to these:

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

Code will be run in an isolated sandbox. Any adversarial behavior will be seen as cheating.

Rule 2: No global variables. Do not use global variables in `proj2.go`.

All the functions you write in `proj2.go` must be stateless. That is to say, put data that needs permanent storage in the server stores, Keystore (§2.3) or Datastore (§2.2). Note that there can be multiple instantiations of the user structure for a given user.

Rule 3: Return `nil` as the error code if an operation succeeds; return a value different from `nil` as the error code if an operation fails. Many functions in this library have an output `error` for the error code.

Please return `nil` *if and only if there is no error*. The autograder may decide whether the function is successful depending on whether the function returns an `nil` as the error code.

2 API

In this section we introduce some useful functions, including Google UUID (§2.1), Datastore (§2.2), Keystore (§2.3), and a few cryptographic functions (§2.4).

2.1 Universally unique identifier (UUID)

UUIDs are like unique names. As we describe in §2.2, Datastore requires us to use UUIDs as the key.

You will be using the Google UUID library.[1] Below we outline how to randomly generate a UUID and how to deterministically generate a UUID from a byte slice.

Random sampling a UUID. To sample a random UUID, we write:

```
new_UUID := uuid.New()
```

Deterministic encoding to a UUID. To convert a byte slice with ≥ 16 bytes into a UUID, we write:

```
(new_UUID, _) := uuid.FromBytes(byte_slice[:16])
```

Warning: The encoding does not hide the information in the byte slice. That is to say, you should not pass a confidential secret key as the byte slice here and use the resulting UUID publicly; an attacker may figure out information about the secret key from that UUID.

2.2 Datastore: A store of encrypted data

We should place encrypted file data (and other metadata, as required by your design) on this server.

This server is **untrusted**, as discussed in §4; thus, we need to guarantee the **confidentiality** and **integrity** of any sensitive data or metadata you store on it.

We can use the following three API functions in `userlib`.

- `DatastoreSet(key UUID, value []byte)`
  - Store `value` at `key`.
- `DatastoreGet(key UUID) (value []byte, ok bool)`
  - Return the value stored at `key`.
  - If the key does not exist, `ok` will be false.
- `DatastoreDelete(key UUID)`

[1] https://godoc.org/github.com/google/uuid

  - Delete that item from the data store.

Note that the storage server has one namespace and does not perform access control, so anything written by one user can be read or overwritten by any other user who knows the `key`. The client must ensure that their own files are not overwritten by other clients by using secret UUIDs that only the client knows.

2.3 Keystore: A store of public keys

You place your keys on a **trusted** public key server, which allows you to post and get public keys. You may need to post more than one key to the Keystore. There are two functions:

- `KeystoreSet(key string, value PKEEncKey/DSVerifyKey) error`
  - Set the entry for `key` to be `value`.
  - Cannot modify an existing key-value entry, which will trigger an error.
- `KeystoreGet(key string) (value PKEEncKey/DSVerifyKey, ok bool)`
  - Return the public key under the entry indexed by `key`.
  - If the key does not exist, `ok` will be false.

You can assume no attacker will overwrite any entry you add to the Keystore.

2.4 Cryptography functions

There are some cryptographic functions you can use. Note that if you use external cryptography libraries (or any other libraries), the autograder will refuse that and you will **get a zero**.

- You cannot use external cryptography libraries, or any other libraries.
- The autograder will refuse external libraries except those given in the skeleton.

2.4.1 Public key encryption (PKE)

Data types:

- `PKEEncKey`: The encryption key (or the public key) for public-key encryption.
- `PKEDecKey`: The decryption key (or the private key) for public-key encryption.

Functions:

- `PKEKeyGen() (PKEEncKey, PKEDecKey, error)`
  - Generate a RSA key pair for public-key encryption.
- `PKEEnc(ek PKEEncKey, plaintext []byte) ([]byte, error)`
  - Use the RSA public key to encrypt a message.
  - *Warning:* It does not support very long plaintext. See here for a discussion.
- `PKEDec(dk PKEDecKey, ciphertext []byte) ([]byte, error)`
  - Use the RSA private key to decrypt a message.

2.4.2 Digital signatures (DS)

Data types:

- `DSSignKey`: The signing key (or the private key) for a digital signature scheme.
- `DSVerifyKey`: The verifying key (or the public key) for a digital signature scheme.

Functions:

- `DSKeyGen() (DSSignKey, DSVerifyKey, error)`.
  - Generate a RSA key pair for digital signatures.
- `DSSign(sk DSSignKey, msg []byte) ([]byte, error)`.
  - Use the RSA private key to create a signature.
- `DSVerify(vk DSVerifyKey, msg []byte, sig []byte) error`.
  - Use the RSA public key to verify a signature.

2.4.3 Hash-based message authentication code (HMAC)

HMAC generates a MAC given a 128-bit symmetric key and the byte slice.

Functions:

- `HMACEval(key []byte, msg []byte) ([]byte, error)`.
  - Compute a SHA-512 HMAC of the message.
- `HMACEqual(a []byte, b []byte) bool`.
  - Compare whether two MACs are the same in a constant-time manner.

2.4.4 Hash-based key derivation function (HKDF)

HMAC can also be used as a simple hash-based key derivation function. HKDF allows us to derive a 128-bit symmetric key from a previous 128-bit symmetric key. If used properly, it can simplify key management.

Function:

- `HMACEval(key []byte, msg []byte) ([]byte, error)`.
  - You can use `HMACEval` and you **additionally** do a postprocessing step to take the first 16 bytes of the output as the symmetric key.

Warning: One key per purpose. If we use a key for symmetric encryption or HMAC, we should not use the same key for HKDF.

2.4.5 Password hashing function

Password hashing functions, also known as password key derivation functions, are commonly used to generate keys from passwords with some sort of entropy.

Function:

- `Argon2Key(password []byte, salt []byte, keyLen uint32) []byte`.
  - Output some bytes that can be used for symmetric keys. The size of the output equals `keyLen`.

As the name implies, the password hashing function is appropriate to derive a key from a password, where the password may only have some *middle* level of entropies. We should not (and will not be able to) use HKDF to derive a key from the password.

2.4.6 Symmetric encryption

Functions:

- `SymEnc(key []byte, iv []byte, plaintext []byte) []byte`.
  - Encrypt using the CTR mode the plaintext using the key and an IV and output the ciphertext. The ciphertext will contain the IV, so you don't need to store the IV separately.
- `SymDec(key []byte, ciphertext []byte) []byte`.
  - Decrypt the ciphertext using the key.

Warning: One key per purpose. If we use a key for HKDF or HMAC, we should not use this key for symmetric encryption.

2.4.7 Random byte generator

Function:

- `RandomBytes(bytes int) (data []byte)`.
  - Given a length of random bytes needed, return the random bytes.
  - Can be used for IV or random keys.

3 Design Security in From the Start

Plan your design in the design document.

- Create a Google document in Google Drive, shared with your teammate.
- Section 1 of the design document has been specified in §6. Copy those questions to the Google document.
- As you are working on the project, try to write down the design choices you make in the design document.
- Use the comment/assignment feature of Google document to nudge your teammate to do something.

Writing and running small tests for the project.

- Open `proj2_test.go` which already contains some basic tests.

- When you finish some code, run the following command in the terminal:

```
go test -v
```

- Add more tests for the things you are not sure about.
- It is normal that you failed some tests because you haven't yet implemented some functions. Don't be frustrated. We hope such error messages can encourage you to finish this project asap.

You should be adding more tests to `proj2_test.go` because the project will be graded in part based on the code coverage of your tests on the *staff solution*. 80% of code coverage will suffice for full credit for code coverage. You can submit to the autograder to see your coverage. Alternatively, you can use `go cover` to test coverage locally, which should correlate well with the autograder. Please find how to test the code coverage in Golang here.

4 Part 1: Single-User File Storage

System architecture. The file sharing system consists of many users and two servers.

- Many users connect to the two servers via a client program.
- The first server, Datastore, provides key-value storage for everyone.
- The second server, Keystore, provides a public key store for everyone.

You will be implementing the stateless client program for the users.

Security guarantees:

- Any data placed on the servers should be available only to the owner and people that the owner explicitly shared the file with, not the server.
  - Unless the file has been shared with them, other users and the server should not learn the file content, the filename, the length of the filename, and who owns the file.
- The server is allowed to know the length of the file content you store.
- If the server modified the data, you should be able to detect that and trigger an error. It is okay if the server returns a previous version of the file, which the client does not need to detect.

4.1 InitUser: Create a user account.

Implement this function:

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

This function should initialize all of the necessary state for a user to effectively use the server. It should return a pointer to the initialized user data structure and the relevant error code (`nil` if function executes successfully).

We can assume that user passwords have sufficient entropy. Thus, it is infeasible for an attacker to guess a password. However, it is possible that two honest users choose the same password.

Warning: The client is stateless, meaning that the client forgets everything after system reboot. So, you need to ensure that given the same username and the same password, the user can later run `GetUser` to obtain the user's state again.

4.2 GetUser: Log in using username and password.

Implement this function:

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

This function should obtain the associated user data structure. It should return a pointer to it and the relevant error code (`nil` if function executes successfully). Note that for a given user, this can function can be called multiple times.

Warning: You cannot use global variables to store user information, which will make the autograder unhappy. Like what was said in the `InitUser`, you need to store the user structure somewhere persistently (see §2 for some options).

4.3 User.StoreFile: Store user files

Implement this method:

```
StoreFile(filename string, data []byte)
```

This function should store the data for the file persistently on Datastore, with confidentiality and integrity guarantees. Note that the filename does not have sufficient entropy, and different users should be allowed to use the same filename without interfering with each other. There is no need to return an error code for this function.

Warning: If you change something in the user structure, don't forget to upload the new user structure – the user structure does not automatically synchronize with the remote version.

4.4 User.LoadFile: Load user files

Implement this method:

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

This function should return the latest version of the file data if it exists. In the case that the file doesn't exist, or if it appears to have been tampered with, return `nil` as the data and trigger an error.

4.5 User.AppendFile: Efficiently append data to an existing file

Implement this method:

```
AppendFile(filename string, data []byte) (err error)
```

This function should append the data to the end of the file. If the file does not exist, return `nil` as the data and trigger an error. You do not need to check the integrity of the existing file; however, if the file is badly broken, return `nil` as the data and trigger an error. This is not required. Do not forget to update the user structure if you change it.

Importantly, this append must be **efficient**, meaning that if before the append, the file has a size 1000TB, and you just want to add one byte, you should not need to download or decrypt the whole file.

You need to write your own tests for AppendFile.

Warning: Recall that it is okay for the server to return *a previous version* of the file, but **not** a version that likely mixes the old and new data, as the security guarantees (§4) said. Double check whether your new design achieves this guarantee.

4.6 Before reaching the next stage

Congratulations!

Before you proceed to the next part, please do the following:

- Run `go test -v` in case there are some issues for the first two tests.
- Write some sentences in the design document (§3 and §6).
- Write some additional tests, if there is any need to increase the code coverage (see §3 for how to measure the code coverage).

5 Part 2: Sharing and Revocation

System architecture, additional information: The file sharing system allows users to share files.

- Suppose there are two users U and V. They should have already posted their public keys somewhere so each one of them can find each other's public keys.
- If U wants to share file F with V, U assembles something we call *a magic string* and sends this magic string to V.
- Using this magic string, V can obtain full permission to F, allowing V to read, write, and share the file.
- U can revoke the permission for F from all other users.

Security guarantee, additional information:

- The channel that U and V use to talk to each other is insecure. Thus, that magic string needs both confidentiality and integrity. You can look for some options in §2.4.

API example: To help understand the functionality we want, here we show an example code:

```
u1, _ := GetUser("user_alice", "pw1")
u2, _ := InitUser("user_bob", "pw2")

v1, _ := u1.LoadFile("the_file_that_I_want_to_share_with_Bob")
```

```
magic_string, err := u1.ShareFile("the_file_that_I_want_to_share_with_Bob", "↩
    user_bob")

u2.ReceiveFile("the_file_from_alice", "user_alice", magic_string)
v2, _ = u2.LoadFile("the_file_from_alice")

// v1 should be the same as v2
```

As you can see here, after user Alice gave the magic string, user Bob can access the file (under a different filename, but actually the same file).

Importantly, if later Bob changes the file, Alice's file **WILL be UPDATED**. That is to say, Alice and Bob are actually sharing the same file, not just Alice sending a copy to Bob.

5.1 User.ShareFile: Securely sharing a file

Implement this method:

```
ShareFile(filename string, recipient string) (magic_string string, err error)
```

This function should:

- Generate a magic string with some confidentiality, integrity and authenticity.
  - That is to say, only the recipient can use this magic string to obtain permission (confidentiality), and the recipient can verify whether the magic string is from the correct sender (authenticity) and has not been tampered with (integrity).
- If the file does not exist, trigger an error and return with an empty string.
- If you find it difficult to convert something into a string, see here: link.
- The recipient should later be able to do all the following:
  - Read data in this file.
  - Write data to this file, which the sender will see the update.
  - Share this file with others.
  - Revoke this file, discussed later in §5.3
- The recipient should only have permission about the file being shared, not other files from the sender. For example, sending the sender's password is not a valid solution.
- Do not assume the sender and the recipient are online at the same time. The permission must be passed in one shot, via the magic string.

5.2 User.ReceiveFile: Adding file permission

Implement this method:

```
ReceiveFile(filename string, sender string, magic_string string) error
```

This function should:

- Create a file with the filename. Note that the filename does not need to be the same with the filename that the sender uses to call that file.
- If the filename has already been used in the recipient side, trigger an error and return.
- Do not forget to upload the user structure, if you change it.
- The recipient should be able to read, modify, and share this file as if they own this file.
- The sender should see all the changes of the file. That is to say, there is only one file, and the file is really being shared.
- *Requirement:* Do not store the file multiple times. For example, if user Alice shares a 1000TB file with Bob, Bob should not create another file that takes another 1000TB storage space.

5.3 User.RevokeFile: Burn it with fire

Read the title carefully, and implement this method:

```
RevokeFile(filename string, target_username string) error
```

Suppose a user U owns file F and has shared this file with A and B. You may assume that only the original owner of the file, U, may revoke access. If U calls this method on file F and user A, then:

- A should not be able to update F.
- A should not be able to read the updated contents of F (for any updates that happen after A's access was revoked).
- If A shared the file with C, C should also not be able to read or update F.
- A should not be able to regain access to F by calling `receive_share()` with U's previous `msg`.
- U and B should still be able to read and update F.
- Return an error if F does not exist.

It is okay for your implementation to:

- Allow revoked user A to still have access to a version of F before A's access was revoked, as long as A is not able to observe new updates.
- Allow revoked user A to mount a denial of service (DoS) attack on U by overwriting the file contents. However, U should never accept these changes as valid. In this case, U should either raise an `IntegrityError`, or return `None` (if A deleted U's files).
- Have undefined behavior if anyone except the original owner U calls this method. This case will not be tested.
- Have undefined behavior if U calls this method on anyone except U's direct child. For example, if U shared F with A and A shared this file with C, U calling `RevokeFile()` on (F, C) results in an undefined behavior.

Revocation must not require any communication between clients.

Example Here is an example workflow of `RevokeFile`, as follows.

If user Alice owns "`file`" and shared the file to Bob.

```
u1, _ := GetUser("user_alice", "pw1")
u2, _ := InitUser("user_bob", "pw2")

magic_string, err := u1.ShareFile("file", "user_bob")
u2.ReceiveFile("alice_file", "user_alice", magic_string)
```

Then Bob will have access to Alice's file, under the filename "`alice_file`".

Next, Alice revokes Bob's permission to this file.

```
u1, _ := GetUser("user_alice", "pw1")
u1.RevokeFile("file", "user_bob")
```

Now, Alice can still access "`file`", but Bob cannot.

```
file_data, _ := u1.LoadFile("file")
```

Alice can update the file:

```
u1, _ := GetUser("user_alice", "pw1")
u1.StoreFile("file", new_file_data)
```

Alice no longer shares this file with Bob. You must ensure two things:

- Bob does not have access to the new file content.
- Bob does not even realize that the file is recently updated by Alice.

This property might require us change the implementation of key management, including `StoreFile` (§4.3), `LoadFile` (§4.4), and `AppendFile` (§5.1), which is expected if you did not follow the recommendation in §4.3. Basically, file location and file keys should not depend on the filename.

6 Design document

Write a clear, concise design document to go along with your code. Your design document should be split into two sections. The first contains the design of your system, and the choices you made; the second contains a security analysis.

A well-written design document receiving full points could be even less than two pages! If a design document is excessively verbose, this design document might lose points.

You do not need to draw fancy figures in the design document because it is not worthwhile. A full-point design document only needs to explain the ideas clearly.

If you followed the recommendation in §3, the Google document should be a good start for your design document.

Section 1: System Design

In the first section, 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. It should also describe your testing methodology in your `proj2_test.go` files.

If you're looking for somewhere to get started, you can begin by asking yourselves six questions:

- How is each client initialized?
- How is a file stored on the server?
- How are file names on the server determined?
- What is the process of sharing a file with a user?
- What is the process of revoking a user's access to a file?
- How were each of these components tested?

Section 2: Security Analysis

The second part of your design document is a security analysis.

⋄ Present **at least three** and **at most five** concrete attacks that you have come up with and how your design protects against each attack.

Only three are needed for full credit; in the event that more than three are provided your score will be determined by the three which provide us the most credit.

You should not need more than one paragraph each to explain how your implementation defends against the attacks you present. Make sure that 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.

7 Submission and Grading

Your final score on this part of the project will be the minimum of the functionality score and security score. Each failed security test will lower the security score, weighted by the impact of the vulnerability.

All tests will be run in a sandbox, and if your code is in any way malicious, we will notice, as described in §1.3.

8 Submission Summary

There will be two submissions on Gradescope, as follows:

- Implementation and testing

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

- Design Document

  ```
  design.pdf
  ```