Dropbox Gearup

Goals

Idea: design an end-to-end encrypted file sharing service

Learn how to design a secure system using the cryptography and security tools we've learned so far!

- Thinking about how to design a system securely
- Iterate on your design after receiving feedback
- Think about attacking your design based on a threat model

Goals

Goal: client for end-to-end encrypted file sharing service

What you have

- Crypto library
- Some insecure data storage
- Threat model (what kinds of attacks to defend against)

What you'll build

- Client API for storing data securely on insecure data storage

You get to figure out how to use the provided crypto operations to accomplish this goal!

How you'll do this

- Now: Design document
  - Think carefully about how you'll implement the requirements
  - How you'll store data, how you'll use crypto to secure it
  - ~4 pages + diagrams => See handout for specific details

=> Meet with TA afterward for direct feedback => use this time wisely!

How you'll do this

- Now: Design document
  - Think carefully about how you'll implement the requirements
  - How you'll store data, how you'll use crypto to secure it
  - ~4 pages + diagrams => See handout for specific details

=> Meet with TA afterward for direct feedback => use this time wisely!

- Implementation (Due Wednesday, May 1)
  - Submit your code + final design document

Remember: the big part is about your design!

What the client looks like

```
# Make a user
client.create_user("usr", "pswd")

#            . . .

# Log in
u = client.authenticate_user("usr", "pswd")  # Returns a User object
```

CLIENT API

What the client looks like

```
# Make a user
client.create_user("usr", "pswd")

#      . . .

# Log in
u = client.authenticate_user("usr", "pswd")  # Returns a User object

# Make some data to upload
data_to_upload = b'testing data'

# Upload it
u.upload_file("file1", data_to_be_uploaded)

# Download it again
downloaded_data = u.download_file("file1")
assert downloaded_data == data_to_be_uploaded
```

The Client API: what you'll implement

LOGIN

1. User operations: `create_user`, `authenticate_user`

2. File operations: `upload_file`, `download_file`, `append_file`

3. Sharing operations: `share_file`, `receive_file`, `revoke_file`

The Client API: what you'll implement

1. User operations: `create_user`, `authenticate_user`
2. File operations: `upload_file`, `download_file`, `append_file`
3. Sharing operations: `share_file`, `receive_file`, `revoke_file`

Your goal: implement client while preserving *confidentiality* and *integrity* ***in an insecure environment***

So what's the environment?

The Wiki

The definitive source for everything all specifications
https://brown-csci1660.github.io/dropbox-wiki/

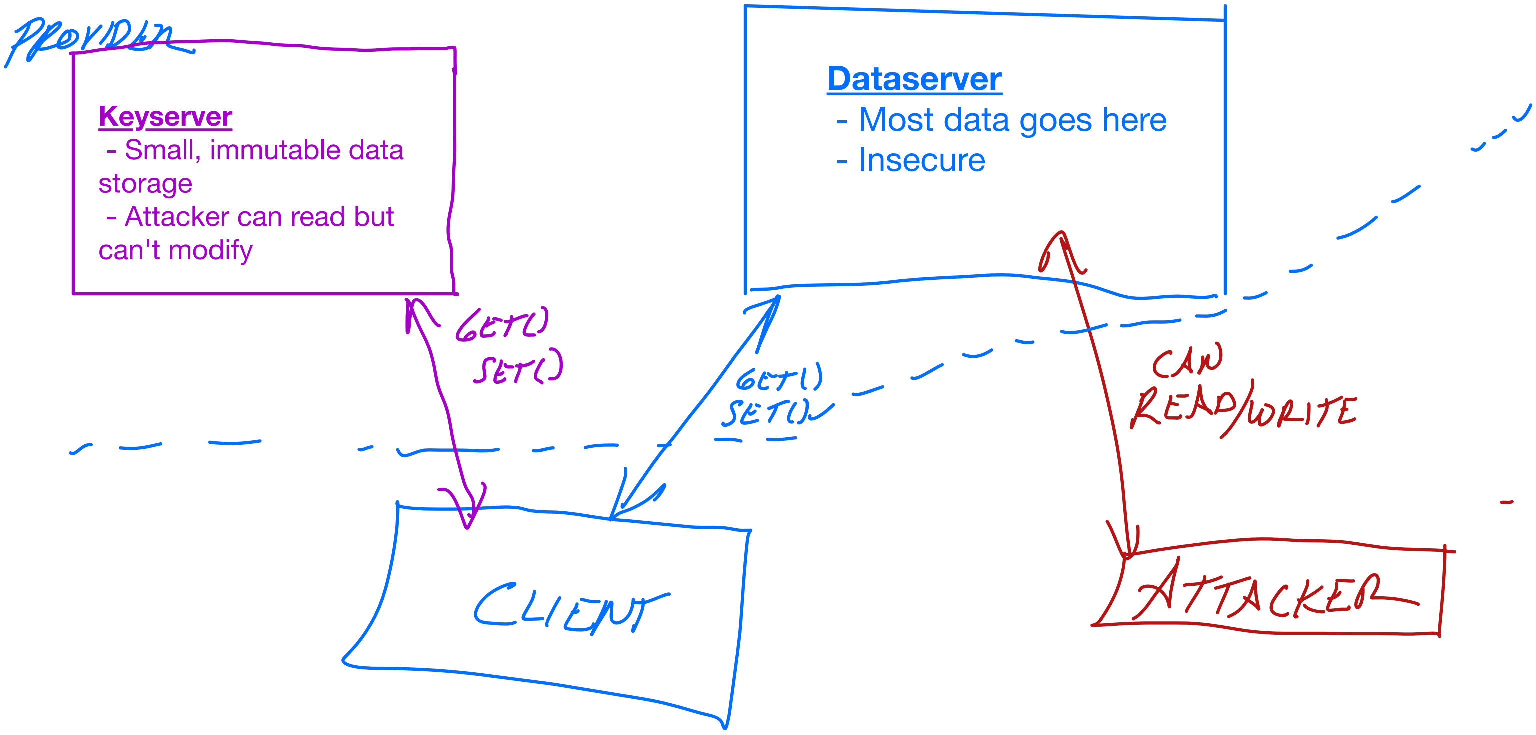
Look here for:

- Descriptions of each API function and requirements
- Detailed specifications for threat model/environment (what you can ignore)
- Documentation for all support code

Also: for implementation notes and container setup, see the setup guide:
https://hackmd.io/@cs1660/dropbox-setup-guide

THE SETUP (INITIAL VIEW)

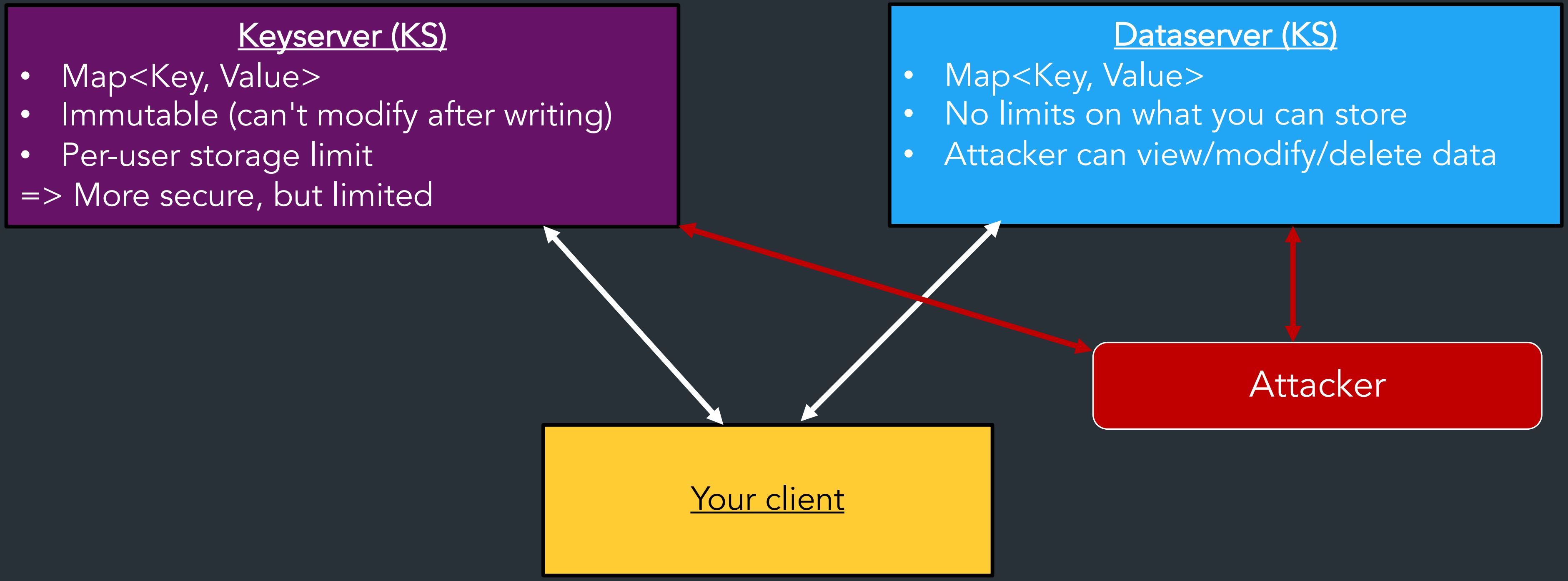
ALL STATE MUST BE IN ONE OF THESE.



Keyserver
- Small, immutable data storage
- Attacker can read but can't modify

Dataserver
- Most data goes here
- Insecure

System Overview



Keyserver (KS)

- Map<Key, Value>
- Immutable (can't modify after writing)
- Per-user storage limit

=> More secure, but limited

Dataserver (KS)

- Map<Key, Value>
- No limits on what you can store
- Attacker can view/modify/delete data

Attacker

Your client

Dataserver

- `Map<memloc, Data>`
  - memloc: 16 byte identifier
  - Data: `bytes`
- Operations: `Set(), Get()`

- Most data will be stored here
- Attacker has full access
  - What could an attacker read? => Threat to confidentiality
  - What happens if an attacker changes something? => Threat to Integrity

Remember for later: see later slides, setup guide for examples and tips on memlocs and how to serialize objects

Dataserver: how to store stuff

Memloc: arbitrary 16-byte identifier for any object

- Could be random: `crypto.SecureRandom(16)`
- Could be deterministic, eg. last 16 bytes of `Hash("alice@somefile")`

↳ EG. "SOME FILE OWNED BY ALICE"

What data can you store? Anything that you can convert to bytes()

- We provide some helpers (see Serialization API on Wiki)
- … and some code examples (see Setup Guide for links)

Can store any data structure, as long as you can serialize it to bytes

How to store stuff

SEE SETUP GUIDE / SERIALIZATION EXAMPLES, + VIDEO FOR EXAMPLES!

Keyserver

- Public, immutable key-value store
- `Map<key_name, pubkey>`
  - `key_name`: any string ("key-alice")
  - `pubkey`: Any public key (for encryption or signing)
- Operations: `Get(key_name), Set(key_name, pubkey)`

Keyserver

- Public, immutable key-value store
- `Map<key_name, pubkey>`
  - `key_name`: any string ("key-alice")
  - `pubkey`: Any public key (for encryption or signing)
- Operations: `Get(key_name)`, `Set(key_name, pubkey)`

- Designed for storing public keys
- Immutable: upload once, can't modify again (but neither can attacker)
- Number of keys per user must be constant

  => Can't grow with number of files, operations, etc.

Threat model: What the attacker can do

- Read/write/modify anything on Dataserver
- Read on the Keyserver (but not modify)
- Can create users/use client API, just like any normal user
- Knows how your client works
  - Can see your code (imagine it's public!) → DON'T RELY ON OBSCURE FILENAMES, ETC.
  - Knows what format in which you'll store data

=> For full details, see the wiki ("Threat model" section)

API Overview

API: User functions

- `create_user(user, pass) -> User`
- `authenticate_user(user, pass) -> User`

Creates/Authenticates user in your system

- Generates or fetches any keys you'll need to implement other operations
- User object: you get to decide what goes in here
- All keys for encryption/integrity/etc will depend on this password (more on this later)
  - Don't worry about the user picking a bad password

```
# Log in
u = client.authenticate_user("usr", "pswd")  # Returns a User object

data_to_upload = b'testing data’





# Upload it (using state from user object)
u.upload_file("file1", data_to_be_uploaded)

# Download it again
downloaded_data = u.download_file("file1")
assert downloaded_data == data_to_be_uploaded
```

API: File operations

BYTES()

- `User.upload_file(filename, data)`
- `User.download_file(filename, data)`
- `User.append_file(filename, data)`

STRING.

- Upload/download a file securely
- Append to an existing file
  - Performance requirement: data sent must scale only with data being appended (ie, can't download and re-encrypt entire file)

API: File operations

- `User.upload_file(filename, data)` *
- `User.download_file(filename, data)`
- `User.append_file(filename, data)`

- Upload/download a file securely
- Append to an existing file
  - Performance requirement: data sent must scale only with data being appended (ie, can't download and re-encrypt entire file)

CS1620/2660 students: Can implement "efficient updates" (more notes at end)
=> Make **`upload_file`** more efficient when file has changed
(implement this or "delegated sharing" (next)

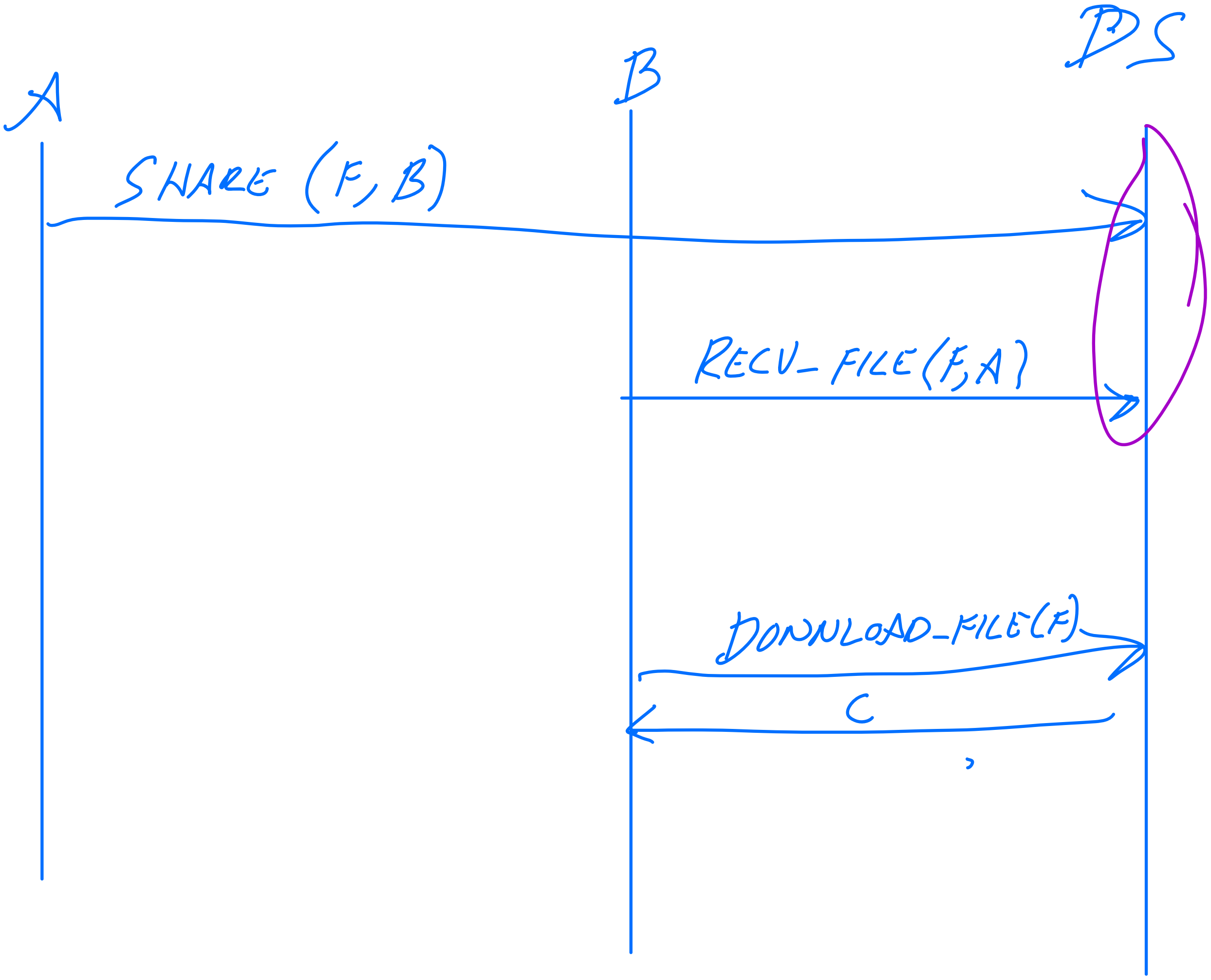
API: Sharing

- `User.share_file(filename, user_to_add)`
- `User.receive_file(filename, file_owner)`
- `User.revoke_file(filename, user)`

- Owner can *share* file with any number of users ← ONLY OWNER
- Users can do any file operations on file (upload, download, append)
  - All users see same copy of file ← ANYONE.
- Owner can *revoke* a user's access, after which user can't do any more operations on that file

CS1620/2660 students: Can extend with "delegated sharing" (implement this or "efficient updates")

HOW TO THINK ABOUT SHARING



This is what the flow of API calls would look like for Alice to share a file with Bob, and for Bob to download it. There are many possible implementations for sharing--you can decide on what information Alice needs to store when sharing such that Bob can download the file.

What you WON'T implement

- Networking (it's all local)
- Writing actual files to disk
- Crypto (we provide a library)

⇒ You can think of the actual implementation as a secure, in-memory key value store

Note: All client state must be on the dataserver/keyserver

```
# Make a user
client.create_user("usr", "pswd")

#            . . .

# Log in
u = client.authenticate_user("usr", "

# Make some data to upload
data_to_upload = b'testing data'

# Upload it
u.upload_file("file1", data_to_be_upl

# Download it again
downloaded_data = u.download_file("fi
assert downloaded_data == data_to_be_
```

Crypto primitives

The crypto library

The support code contains a crypto library for you use

- No external crypto libraries

<u>What you have</u>

- Asymmetric crypto (Encryption, digital signatures)
- Symmetric crypto (Encryption, HMACs)
- Hashing
- Key derivation functions
- Secure randomness

=> A big part of your design is about <u>how</u> you use these!

GOALS
- CONFIDENTIALITY
- INTEGRITY

Asymmetric Crypto

Encryption

- Gen() -> K_pub, K_priv
- Encrypt(k_pub, data)
- Decrypt(k_priv, data)

CONFIDENTIALITY

Signing

- Gen() -> K_pub, K_priv
- Sign(k_priv, data)
- Verify(k_pub, data)

INTEGRITY

~~A~~symmetric Crypto

Encryption

- Enc(k, m)
- Dec(k, c)

Authentication with symmetric crypto ↙ INTEGRITY

- Message authentication codes: computed based on hash of message, can verify if you have key
- HMAC(k, m) -> t
- HMACEqual(t1, t2) => {0, 1}

K, M

$$A \xrightarrow{M,\ t} B \quad K$$

$t = HMAC(K, M)$

$t' = HMAC(M, K)$

$t \overset{?}{=} t'$

=> Think of an HMAC like a keyed hash function: requires both the original message and key to compute the same output

Asymmetric vs. Symmetric crypto

Asymmetric crypto
- Public and private key
  - Super slow
    - Limit on the size of the message

=> Maybe useful for sharing

Symmetric crypto
- One key
- Key distribution is a challenge
- Multiple people could hold this key
- Much faster than asymmetric crypto (>1000x faster)
- Can encrypt any size message (eg. CBC mode, etc.)

=> Good for large data

=> You will have **many** symmetric keys

Key functions for working with keys (pun intended lol)

PasswordKDF(salt, password) => symmetric key

=> Under the hood, uses:

PBKDF2(password, salt, key_length) => symmetric key of length L

- Secure way to generate a key based on a password, involves computing a large number (>100000) iterations of Hash(salt || password)

HashKDF(key, "purpose") => another symmetric key

=> Given one key, generate another key ✨**deterministically**✨

=> Can use to compute the same key from different sessions

Example: deriving keys in different sessions with HashKDF

SESSION 1

LOGIN("A", "PASS")

$\Rightarrow K_0$

HASHKDF(K_0, PURPOSE)

$= K_P$

SOME NAME FOR KEY'S PURPOSE (PUBLIC)

SESSION 2

LOGIN("A", "PASS'

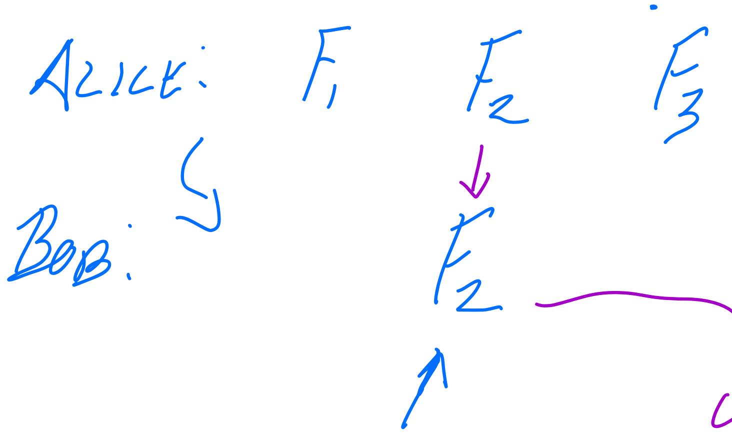
$\Rightarrow \boxed{K_0}$

HASHKDF(K_0, PURPOSE)

$= K_P$

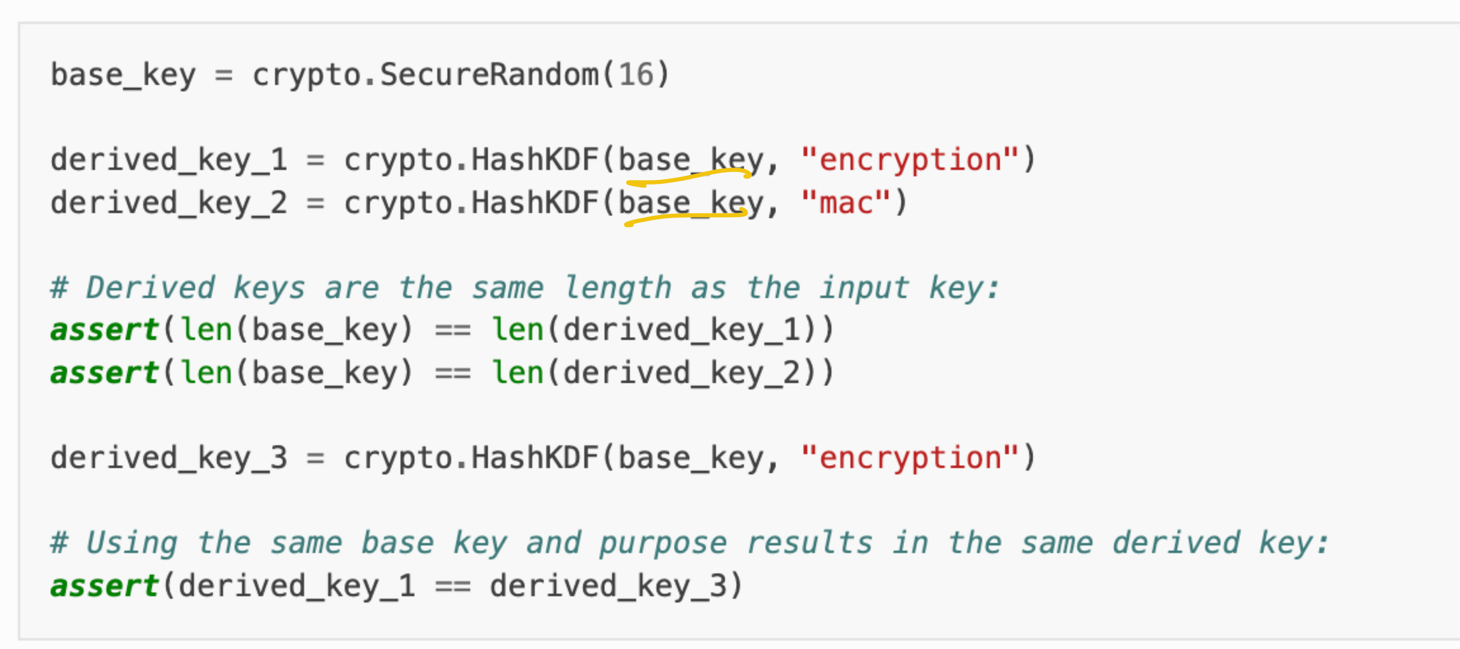
FAQ: "Why can't we just use one key to encrypt files?"

Example: Alice has 3 files, wants to share one with Bob



WHAT IF ALICE WANTS TO SHARE ONLY F_2 WITH BOB?

HashKDF example



```
base_key = crypto.SecureRandom(16)

derived_key_1 = crypto.HashKDF(base_key, "encryption")
derived_key_2 = crypto.HashKDF(base_key, "mac")

# Derived keys are the same length as the input key:
assert(len(base_key) == len(derived_key_1))
assert(len(base_key) == len(derived_key_2))

derived_key_3 = crypto.HashKDF(base_key, "encryption")

# Using the same base key and purpose results in the same derived key:
assert(derived_key_1 == derived_key_3)
```

Authenticated encryption

Your goal for most things is confidentiality AND integrity

NEED BOTH

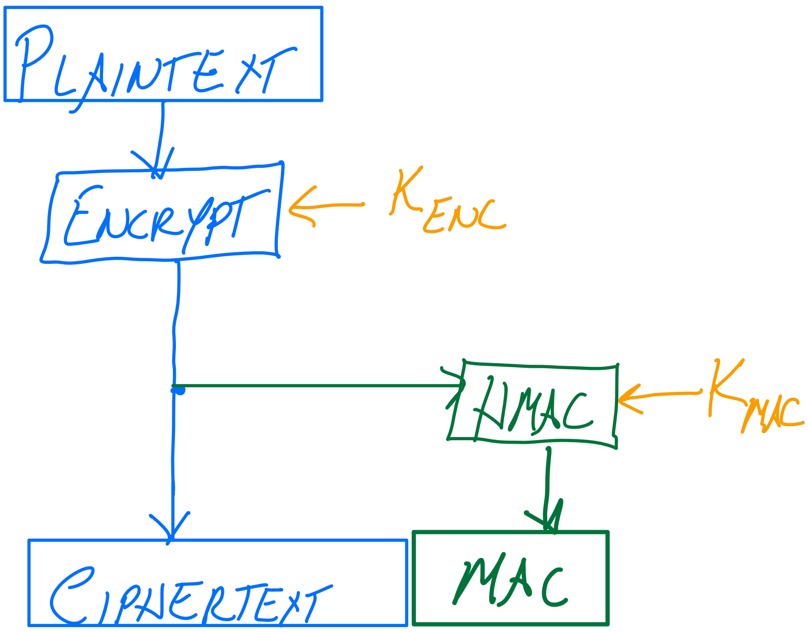
TO STORE MOST OBJECTS SECURELY

Two operations:

- Encrypt: Confidentiality => Encrypt(k, m)
- MAC: Authentication => HMAC(k, m)

- How to do this is well-studied and has common pitfalls
  - Which do you do first? (Encrypt then MAC, MAC then encrypt, Encrypt THEN MAC, …)
  - See cryptography lectures for more)
- You should use: Encrypt then MAC

ENCRYPT – THEN – MAC



These are great operations to implement as helper functions:

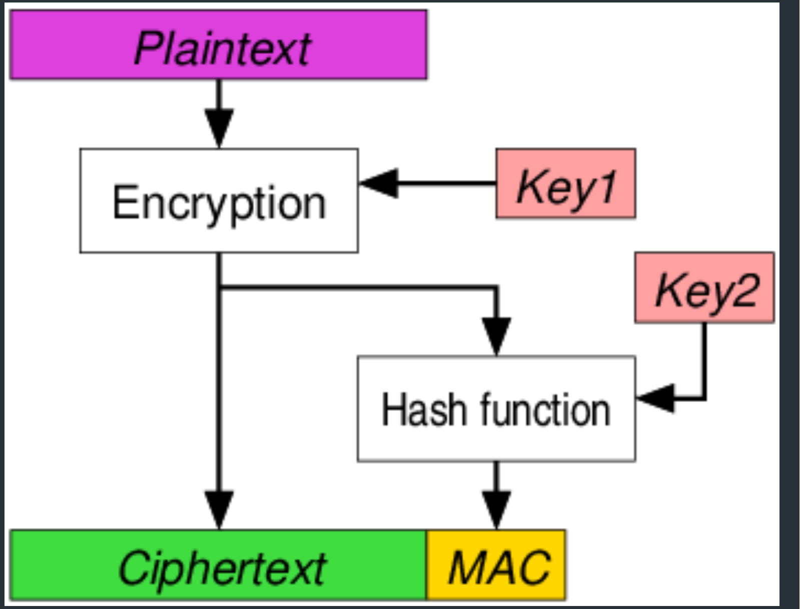
EncryptAndMAC(k, m) => returns (c, mac) => store both
DecryptAndVerify(k, c, mac) => m (or error if MAC verify fails

This is also a great place to employ HashKDF, since we need to use different keys for the encryption and the HMAC step:

k => input key
    HashKDF(k, "enc") => k_enc
    HashKDF(k, "mac") => k_mac

Authenticated encryption

- You should use: Encrypt then MAC
- Proven to give us the security properties we want, when different keys used for encryption and hashing



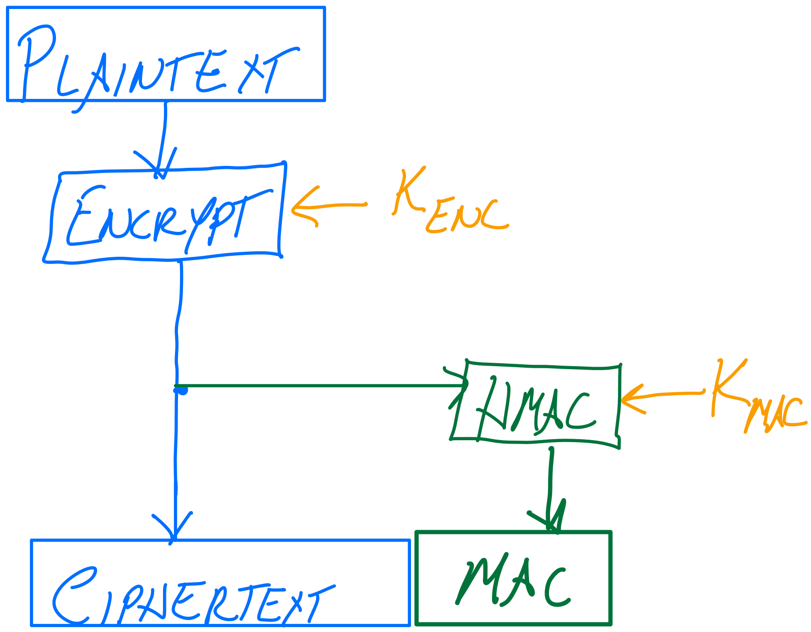
GCM

CS1515 students: We don't have AES-~~GCM~~, sorry. ☹

Design: In general

- In general, use one key per purpose
  - Think about how sharing keys between operations can affect security
  - HashKDF is your friend
- A bit of software engineering can help you!
  - Consider making some helper functions for common operations
- Setup guide: examples on how to serialize stuff

ENCRYPT – THEN – MAC



Setup and Stencil

Container setup & Environment

For this project, we'll use the "Development container" (same as project 1)

- Some slight updates—see setup guide for instructions

- Stencil uses a Python virtual environment
  - See setup guide for instructions
  - Like VSCode? You can use it with the container!

CS1620/CS2660: Efficient updates

"Efficient" updates

- Broadly, When uploading a new file, bandwidth should scale based on amount of data that was changed
- How you do this is up to you, here's one way...

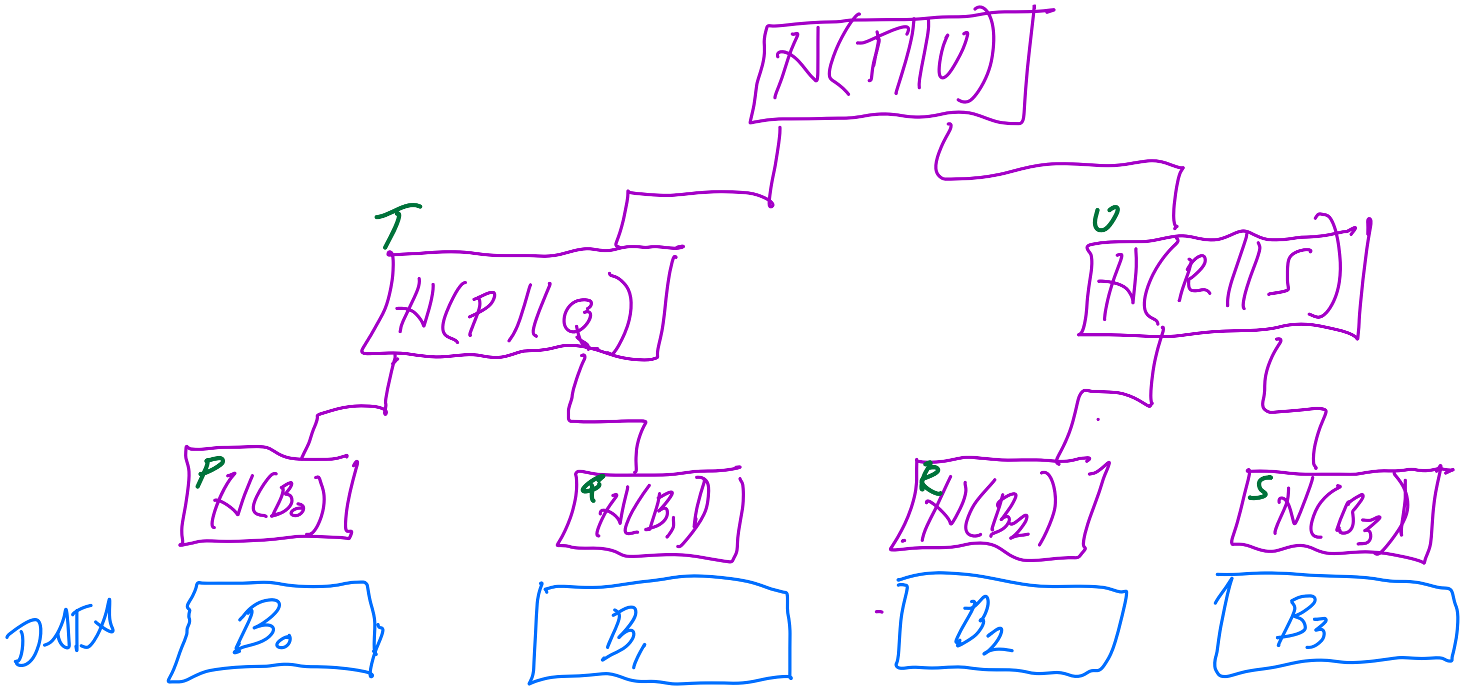
UPLOAD(F, DATA1)
UPLOAD(F, DATA2)

Basically: if re-uploading the same file, you should not be downloading and reuploading the whole file

=> Think about dividing up the file into blocks, then deal with each block
=> How you do this is up to you--there are multiple possible implementations!

How to think about integrity when the file is stored in multiple blocks?

One way: Merkle tree (hash tree)



For more notes on this, see the “Cloud Security” notes from lecture 17, starting on page 34
(Was extra reading from lecture)

"Efficient" updates

- Broadly, When uploading a new file, bandwidth should scale based on amount of data that was changed
- How you do this is up to you, here's one way...