Dropbox Gearup

Project Overview

- Goals
  - Design an end-to-end encrypted file sharing service.
  - Designing a secure system in an insecure file system.
    - Think about how to design a system securely.
    - Make design modifications after receiving feedback from design check-ins.
  - Threat modelling and potential attacks to your design/implementation.

What do you have access to

- We provide you with a crypto library (support/crypto.py)
  - This will provide you with all of the crypto functions you need for the project. E.g. AES key generation (symmetric), RSA key generation (asymmetric)
- There is also a very helpful utils library (support/utils.py)
  - Bytearray conversion functions
  - Serialization/Deserialization functions

Deadlines

- Design Document (Due April 11th)
  - **No Late Days Allowed**
  - See handout for specific details
- Final Implementation (Due April 29th)
- Final Design Document (Due April 29th)

What the client looks like

```
(env) cs1660-user@5bab06dbb672:~/dropbox-andrew$ python3 test_reference.py
Python 3.10.12 (main, Nov  6 2024, 20:22:13) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> u = c.create_user("a", "hello")
>>> u = c.authenticate_user("a", "hello")
>>> some_data = "string encoded as bytes".encode("utf-8")
>>> u.upload_file("filename", some_data)
>>> recvd_data = u.download_file("filename")
>>> some_data == recvd_data
True
>>>
```

What you need to implement

- User operations
  - create_user
  - authenticate_user
- File operations
  - upload_file
  - download_file
  - append_file
- Sharing operations
  - share_file
  - receive_file
  - revoke_file

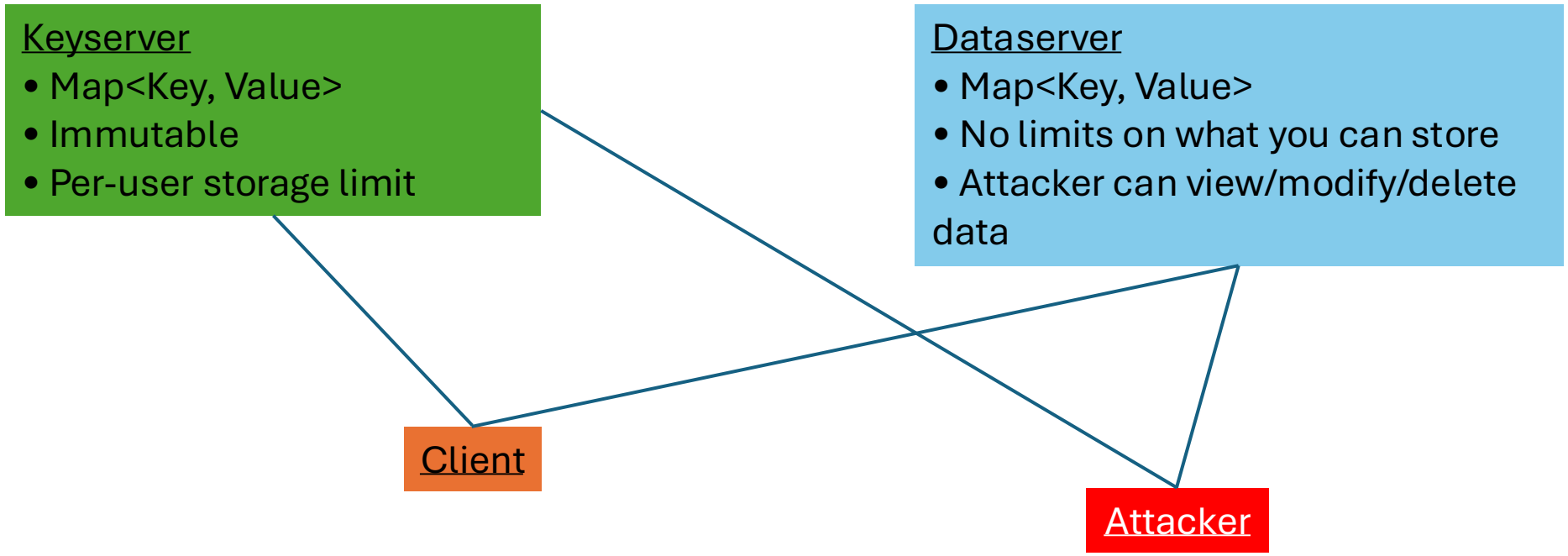
- The goal is to maintain confidentiality in an environment where data is visible to anyone.

- The dropbox wiki is a great resource when working through implementation of each of these functions.

The Wiki

- The wiki contains detailed explanations for each API function and its requirements
- Detailed specifications for threat model/environment. I.e. which threats are in or out of scope
- Documentation for all support code

Overview


Keyserver
• Map<Key, Value>
• Immutable
• Per-user storage limit
Dataserver
• Map<Key, Value>
• No limits on what you can store
• Attacker can view/modify/delete data
Client
Attacker

Dataserver

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

- Most data will be stored here
- Attacker has full access
- Think about:
  - What could an attacker read?
  - What happens if an attacker changes something?

Dataserver: how to store files

- Memloc: arbitrary 16-byte identifier for any object
- Approaches:
  - Could be random: crypto.SecureRandom(16)
  - Could be deterministic, eg. last 16 bytes of Hash("alice@somefile")

- Remember that the client needs to be able to retrieve files easily, so this may affect your approach.

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 the data
- Number of keys per user must be constant
  - Make sure all of the keys a user will need are established during creation of the user

What can the attacker do?

- Read/write/modify anything on Dataserver
- Read the Keyserver
- Can create users/use client API, just like any normal user
- Knows how your client works
- Can see your code (imagine it’s public!)
- Knows what format in which you'll store data
  - i.e obscure filenames are not a countermeasure
- See the **threat model** section of the wiki for more details

User functions

- 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
- Don't worry about the user picking a bad password

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 (1620/2660 only)

Sharing

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

- The owner alone can share file with any number of users
- Users can do any file operations on file (upload, download, append)
  - All users see same copy of file
- Owner can revoke access, after which the user is unable to do any more operations on that file

Useful functions

- 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
  - Think about how this could be used and why we can’t just use one key to encrypt files

Authentication

- Your goal for most things is confidentiality AND integrity
- 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, or …)
- You should use: Encrypt then MAC
- Similarly, when decrypting, you should verify the HMAC and then decrypt the ciphertext

Lab Component

- Efficient Updates
  - When uploading a new file, bandwidth should scale based on amount of data that was changed
  - 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
- Delegated Sharing
  - The owner needs to grant access to a file to a user
  - The user may then grant access to another user without the owner specifically granting them access
  - Owner → user → next user
  - When revoking access, need to regenerate all access tokens
  - How you structure and verify this chain is up to you