Computer Security Project

SFTP over SSH: Access Control (DAC, MAC, RBAC), and Secure File Operations

Overview

In this initial stage of the project, you will build a minimal yet interoperable Secure Shell File Transfer Protocol (SFTP) *client and server*. You will *reuse the SSH transport and connection protocol from `asyncssh`* (Python), and implement the SFTP v3 layer **yourself**. The focus is on **authentication (AuthN)**, **authorization (AuthZ)**, and **access control models**: *Discretionary Access Control (DAC)*, *Mandatory Access Control (MAC)*, and *Role-Based Access Control (RBAC)*.

This stage has to be finalized in **two weeks** by **18th November**.

Security Disclaimer & Starter Boilerplate. We provide a minimal `server.py` as a *starter* to help you begin experimenting with the transport and SFTP framing. It is intentionally lean and may contain issues or vulnerabilities. The whole point of this project is to analyze, harden, and evolve this baseline into a robust and *secure* solution. You are expected to review, test, and improve its design, including authentication, authorization, path handling, error mapping, handle management, and auditing.

Your SFTP must support directory listing, directory creation, file upload/download, and basic stat operations. You will also add a small **Capture-the-Flag (CTF)**: a hidden flag and a write-up explaining secure design choices and exploitation attempts.

Learning outcomes

1. Explain and implement the separation of concerns between SSH (transport/session) and SFTP (file protocol).
2. Implement password-based authentication and enforce authorization with DAC, MAC, and RBAC.
3. Build an SFTP client/server pair which can interoperate with standard SFTP tools.
4. Apply a default-deny policy and audit decisions.
5. Design and document a CTF-style hidden flag and its protections.

Constraints and resources

- **Compute:** laptops only; no heavy training or external servers required.
- **Language/Libraries:** Python 3.x; `asyncssh` for SSH transport; Python stdlib for everything else unless stated.
- **Data:** Publicly available or generated locally (no sensitive data).
- **OS:** Windows/macOS/Linux (ensure cross-platform paths by treating SFTP paths as POSIX).

What you will deliver

- `server/` — SFTP server implementation (reusing SSH transport via `asyncssh`).
  - `server.py` — entrypoint; password auth; SFTP v3 packet handling.
  - `auth.py` — password verification (e.g., `hashlib.scrypt` or `argon2-cffi` if allowed).
  - `policy.py` — unified `authorize(user, op, path)` implementing DAC, MAC and RBAC.
  - `audit.jsonl` — line-delimited audit records (generated at runtime).
- `client/` — SFTP client implementation.
  - `client.py` — connects via `asyncssh`, opens subsystem `"sftp"`, sends SFTP v3 packets.
  - Supports: `pwd, ls, mkdir, get, put, stat` (see Command list below).
- `tests/` — automated tests for AuthN/AuthZ and SFTP operations (e.g., `pytest`).
- `CTF_writeup.pdf` — CTF description: flag design, attack surface, exploitation attempts, and fixes.
- `README.md` — how to run (on Windows/macOS/Linux), expected outputs, known limitations.

Project Requirements

RFC-compliance

To interoperate with real SSH/SFTP implementations, your design must follow the SSH/SFTP specifications. Because SFTP is an SSH subsystem, we follow the SSH architecture and protocols, and the SFTP draft:

- **RFC 4251** — SSH Protocol Architecture
- **RFC 4253** — SSH Transport Layer Protocol
- **RFC 4252** — SSH Authentication Protocol
- **RFC 4254** — SSH Connection Protocol
- **IETF Draft** — SSH File Transfer Protocol (SFTP)

You do *not* reimplement SSH itself: reuse `asyncssh` for transport, authentication exchange, and channels; implement the SFTP message layer correctly on top.

1) SSH transport via `asyncssh` (reuse; do not reimplement SSH)

- Use `asyncssh` to provide SSH transport (encryption, key exchange, channel management).
- Use a persistent **host key** (Ed25519). Clients should verify the host key (TOFU or pre-provisioned `known_hosts`).
- Implement **password-based** user authentication (no public key login required).
- SFTP runs as an SSH **subsystem** on a session channel (`subsystem='sftp'` by default).

2) Authorization and Access Control (DAC, MAC, RBAC)

- Implement a single gate: `authorize(user, op, full_path)` returning `(allowed, reason)`.
- **DAC:** owner/group/other with Unix-style modes; apply to `read`/`list`/`stat` and `write`/`create`/`remove`.

- **MAC:** clearance levels for users and labels for resources. Enforce "no read up" and "no write down".
- **RBAC:** roles grant permissions on operations/resources; union of roles; optional per-user allow/deny (deny overrides).
- Apply the gate **after** canonicalization (jail) and **before** any filesystem call.
- **Audit** each decision to `audit.jsonl` (user, op, path, allowed, reason, timestamp).

2a) Applying DAC, MAC, RBAC & Self-Tests (required)

Documentation requirement. Add a section *Access Control Design* in your `README.md` (or a separate `policy.md`) which **explains exactly** how each model is applied and tested:

- **Operations covered:** map SFTP requests to AuthZ operations: `realpath`, `stat`, `list` (OPENDIR/READDIR), `mkdir`, `read` (OPEN/READ), `write` (OPEN/WRITE), `remove/rename` (if implemented). State the default-deny behavior.
- **Enforcement point:** authorization is checked *after* jail canonicalization via `safe_join` and *before* any filesystem call.
- **DAC:** specify ownership model, group mapping, mode bits interpretation (r/w/x) for *directories vs files*, and inheritance rules for new files/dirs. Include examples showing allow/deny for owner, same-group, other.
- **MAC:** define label taxonomy (e.g., `public < internal < confidential`) and user clearances. Prove that you enforce *no read up* and *no write down.* Show how labels are stored (metadata DB, sidecar file, or mapping).
- **RBAC:** define roles, the permission matrix (role $\rightarrow$ {op on resource/prefix}), union-of-roles semantics, and conflict resolution (e.g., *deny overrides*). Explain any resource scoping (path prefixes).
- **Composition rule:** state clearly whether final decision is $DAC \wedge MAC \wedge RBAC$ (recommended) and how conflicts are resolved; include a 2x2 table of example outcomes.
- **Auditing:** what fields are logged per decision (user, op, path, allowed, reason, timestamp) and where (`audit.jsonl`). Provide a short sample.

Self-tests (must be automated). Provide `pytest` tests in `tests/` with **clear objectives**. At minimum:

- **DAC tests:** owner can read/write their file; other cannot write without permission; directory execute bit affects `ls/stat` as specified by your design.
- **MAC tests:** a user with clearance `internal` can read `public/internal` but not `confidential`; a `confidential` user cannot *write down* into `public`.
- **RBAC tests:** role `analyst` can `read/write` under `/projects` but cannot `mkdir` under `/admin`; adding role `admin` enables it. If you implement per-user deny, verify *deny overrides.*
- **Composite tests:** when DAC allows but MAC denies (or vice versa), show final decision matches your composition rule; include at least one traversal attempt that must be denied.
- **Audit assertions:** tests that an allow/deny decision writes a corresponding audit record with the correct fields.

Each test should include a one-line *objective* (docstring) and set up its own fixtures (temporary jail paths, users/roles/labels) without requiring manual steps.

2b) User & Role Management (static datasets)

There is **no runtime creation/deletion of users or roles**. Use a *predefined, static* set of users with different permissions/clearances to exercise DAC, MAC, and RBAC. Describe these in files (mirroring Assignment 1 style) and load them at startup:

- `data/users.json` *(authentication)*: list of objects with fields `username`, `salt` (base64), `password_hash` (base64), and hashing parameters (e.g., `n,r,p,dklen` for scrypt). Example keys mirror Assignment 1.
- `data/user_roles.json` *(RBAC)*: mapping `"user": ["role1","role2",...]`.
- `data/role_perms.csv` *(RBAC)*: matrix `role,resource/prefix,read,write,delete,...` where present tokens in action columns grant that permission.
- `data/mac_labels.json` *(MAC)*: optional mapping of `"path-prefix" : "label"` (e.g., `"/projects/confidential":"confidential"`) and `"users": { "alice":"confidential", ... }` for clearances. You may also embed labels in a single file if preferred.
- `data/dac_owners.csv` *(DAC)*: optional owner/group defaults and mode templates for path prefixes (or derive from OS metadata if you choose).

Rules:

- Users/roles/labels are immutable at runtime; all changes occur by editing the files and restarting the server.
- Your `README.md` must list each predefined user (e.g., `alice`, `bob`, `eve`) with *intended* roles/clearances and a brief description of their expected powers/limits (do not print real password plaintexts; keep only salted hashes in files).
- Tests must exercise multiple users with different roles/clearances to demonstrate both allowed and denied actions.

Acceptance criteria: on startup, the server logs that it successfully loaded each file; malformed files cause a clean startup failure with a readable error; missing files are treated as fatal unless documented otherwise.

3) SFTP v3 server commands (subset)

Your server must correctly parse request IDs and reply with appropriate messages. **Note: real clients (e.g., OpenSSH `sftp`) expect STAT-family operations; implement them to ensure interoperability.**

```
INIT     -> VERSION(3)
REALPATH -> NAME
STAT     -> ATTRS | STATUS(NO_SUCH_FILE)
LSTAT    -> ATTRS | STATUS(NO_SUCH_FILE)
FSTAT    -> ATTRS
OPENDIR  -> HANDLE
READDIR  -> NAME (repeat until EOF via STATUS(code=1))
MKDIR    -> STATUS
OPEN     -> HANDLE
READ     -> DATA | STATUS(EOF)
WRITE    -> STATUS
CLOSE    -> STATUS
```

Notes:

- Path canonicalization: SFTP paths are POSIX-like ("/"), rooted at the jail; map to OS paths internally.

- Return `STATUS` codes: OK(0), EOF(1), NO_SUCH_FILE(2), PERMISSION_DENIED(3), FAILURE(4).
- Unknown messages → `STATUS(FAILURE)` or `OP_UNSUPPORTED` (optional).
- Maintain a handle table for directories and files; `FSTAT` operates on a valid file handle.

4) Client-side commands (CLI)

Implement a small CLI which uses `asyncssh` to open the SFTP subsystem and then exchanges SFTP packets:

```
sftp> pwd                # prints POSIX path (e.g., "/")
sftp> ls [path]          # lists directory (OPENDIR/READDIR)
sftp> mkdir <path>       # creates directory (MKDIR)
sftp> stat <path>        # prints attrs (STAT/LSTAT)
sftp> get <rpath> [lpath] # download (OPEN read + READ + CLOSE)
sftp> put <lpath> <rpath> # upload (OPEN write + WRITE + CLOSE)
sftp> quit
```

Client should verify the host key (TOFU or provided `known_hosts`) and handle SFTP errors gracefully.

5) Capture-the-Flag (CTF) and write-up

- Hide a **flag** (e.g., `FLAG{groupXX_sftp_wins}`) inside the server's jail under conditions that require correct authorization to access.
- Provide at least **two** mitigation layers (e.g., MAC label + DAC mode; RBAC permission + path canonicalization).
- In `CTF_writeup.pdf`, document:
  - Where the flag is, intended protections, and how a naive design would expose it.
  - At least one realistic attack attempt (directory traversal, symlink trick, handle reuse, race) and how your checks stop it.
  - Evidence (logs/screenshots) and how auditing helped you detect attempts.

Repository structure (required)

```
.
|-- server/
|   |-- server.py        # AsyncSSH transport + SFTP v3 loop (starter; harden it!)
|   |-- auth.py          # password hashing/verification
|   |-- policy.py        # DAC    MAC    RBAC gate + audit
|   `-- sftp_root/       # jail root (created at runtime; initially empty)
|-- client/
|   `-- client.py        # SFTP client CLI (pwd/ls/mkdir/get/put/stat)
|-- tests/
|   `-- test_sftp.py     # automated tests (optional but recommended)
|-- CTF_writeup.pdf
`-- README.md
```

Grading rubric (100 points)

- **Transport & Protocol correctness (25)**
  INIT/VERSION, framing, request IDs, correct replies (`STATUS/HANDLE/NAME/ATTRS/DATA`), path canonicalization, jail enforcement.
- **AuthN & password policy (10)**
  Secure password verify (salted hash), lockout/rate-limit, no secrets in logs.
- **Authorization models (35)**
  DAC (10), MAC (10), RBAC (10), unified gate with default deny + auditing (5).
- **Client CLI & UX (10)**
  Commands work as specified; clear error messages; host key verification.
- **Security quality (10)**
  Handles not leaked, robust to traversal/symlink/race basics, proper STATUS mapping.
- **CTF flag & write-up (10)**
  Flag design, protections, attack attempt(s), evidence, clarity.

Running & testing

Server:

```
# One-time: generate unencrypted host key
ssh-keygen -t ed25519 -N '' -f server/ssh_host_ed25519_key

python3 server/server.py
# Jail root will be created if missing
```

Client (your implementation):

```
python3 client/client.py --host 127.0.0.1 --port 2222 --username bob
sftp> pwd
sftp> ls
sftp> mkdir demo
sftp> stat /
sftp> put README.md /demo/README.md
sftp> get /demo/README.md ./README.copy.md
```

Interoperability check (OpenSSH):

```
sftp -P 2222 bob@127.0.0.1
# On first connect, verify/accept the host key fingerprint
sftp> ls
sftp> stat .
```

Academic integrity

You may use **any** tool you see fit to help you in implementing the project (**except** for a simple reusing ready 3rd party libraries for access control, authentication, and sftp client/server file protocol implementation) but you must understand all logic and code you submit.