- So far: isolation
- Upcoming: case studies about how systems have used isolation

OpenSSH

- Non privilege separation version:
  - On a host computer, sshd runs on port 22
    - Runs with root privileges
  - When it receives a tcp connection, it forks and creates a pseudoterminal with the user id of the corresponding user
    - The connection talks with this to do bash stuff
    - This also has to run with root permissions
      - Has to access the host key to sign messages and use user password
      - Does all of the compression / decoding / cryptography

---

- Principle of Least Privilege
  - Every component has least privileges to do its job
- Challenges:
  - How to split / isolate / share?
  - How to maintain performance?
  - How hard is it to change the code?

- OpenSSH privsep:
  - sshd still runs as root, but when it forks it creates a per connection monitor
    - This spawns a worker that actually maintains the TCP connection
    - After the connection is made and authentication is made, another worker is spawned which only has user permissions and handles making the pseudoterminal
  - Slight technicality:
    - Three process plan: the TCP connection worker stays around and just proxies to the other work
    - Two process plan: kill the TCP connection worker and reroute the TCP connection to the worker
      - OpenSSH actually does this b/c it has performance improvement
  - The monitor process provides certain things such as encrypting messages / authenticating user
    - The root monitor process is now no longer exposed directly to the user
    - The monitor also has an FSM that limits the interface provided to the user process to further increase security
- Security of various components:
  - sshd listener: root
  - monitor: root
  - pre-auth worker: anonymous UID, has access to sign message as host computer and send stuff to computers on the network
    - However, you can't use this for new connections because those would require using a new host key
    - You can only use this for existing connections that used that same host key
  - post-auth worker: could act as user, do DOS attack
- Attack surfaces:
  - sshd listener: creating new TCP connections, not even reading the TCP connections
  - monitor: interface to worker
  - pre-auth worker: reading / writing to TCP connections (most likely to be exploited)
    - Not leaking any user data / root access
  - post-auth worker: pre-auth worker state / TCP data, but this only occurs if you can log in
- All of the bugs found in the past fell in the pre-auth / post-auth worker

---

- Challenge-response protocol:

    - A process where the server sends a challenge c to a client and the client has to sign it with their secret key to prove their identification
    - The server then uses the client's public key to verify that this is in fact the client they expect
        - Client's secret key: stored in ~/.ssh/id_rsa
        - Server's authorized public key: ~/.ssh/authorized_key
    - You cannot have the worker do this process because we assume the worker could be bad
        - The monitor has to generate challenges and send it to the client
        - Otherwise the worker could pretend that it satisfied the challenge or use a replay attack with an old challenge

---

- Spawning post-auth worker
    - Need to send all of the low-level state to the post-auth worker to make the client seem like they are talking to the same progress
    - Most tricky part of the implementation