Sandbox mechanics: namespaces, cgroups & seccomp

Key takeaways

1. Namespaces narrow what a job can see; cgroups bound what it can consume; seccomp narrows what it may ask the kernel.
2. Every mount and socket you expose is a capability grant — replace it, don’t hand it through.

Four leaks, four mechanisms

Route what the job can still do to the mechanism that denies it

A sandbox is deny-by-default: each kernel mechanism below closes one way an untrusted job can act. The cooperative layers that come first — $PATH, conda — redirect names but deny nothing.

the job can still…	denied by	knob
see host processes, mounts, interfaces	namespaces	unshare / clone flags
consume unbounded memory, CPU, PIDs	cgroups v2	memory.max, cpu.max, pids.max
ask the kernel for risky operations	seccomp	syscall allowlist
touch whatever you handed through	your grant list	mounts, sockets, network

Your first virtualization layer was an environment variable

Virtualization swaps the implementation behind an unchanged interface

$PATH virtualizes an executable name

```
odachi:~> which mpirun
mpirun is /usr/bin/mpirun

epee:~> which mpirun
mpirun is /opt/amazon/openmpi/bin/mpirun
epee:~> echo $PATH
...:/opt/amazon/openmpi/bin:/usr/bin:...
```

conda virtualizes an install tree

```
$ conda activate hw3
(hw3) $ which python
/home/me/miniconda3/envs/hw3/bin/python

(hw3) $ python -c "import numpy"
# hw3's numpy, not the system's
```

- Both interpose on a name lookup: an interface with a swappable implementation.
- Both are cooperative — a program can hardcode /usr/bin, reset PATH, or write outside the env. Convenience, not confinement.

Namespaces: if you can’t name a resource, you can’t control it

The first enforced rung: deny what the job can see

A namespace wraps one global kernel resource so the processes inside see a private instance of it.

unshare: enter fresh namespaces and look around

```
host$ sudo unshare --fork --pid --mount-proc --net bash
ns$ ps -ef                      # private process table
UID    PID  PPID  CMD
root     1     0  bash          # this shell is init
ns$ ip link                     # private network stack
1: lo: <LOOPBACK> state DOWN    # no host interfaces
ns$ kill -9 4242                # a host PID
bash: kill: (4242) - No such process
```

- Seven namespaces cover the kernel’s name tables — PID, mount, net, user, UTS, IPC, cgroup; mounts and capabilities still decide what the job may do.

Cgroups put a hard ceiling on damage

Seeing less is not consuming less: cap the resources next

Limits and receipts belong together

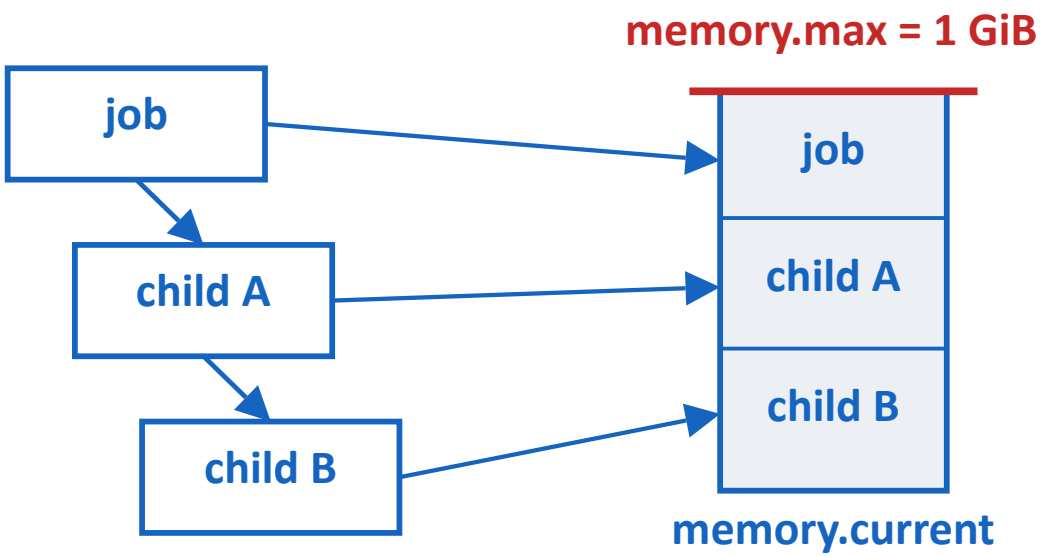
```
# cgroup v2 controls for one grading job
echo "1073741824" > memory.max   # 1 GiB, not a monitoring alert
echo "100000 100000" > cpu.max  # at most one CPU
echo "128" > pids.max            # fork bombs stop here

# Observe the outcome; do not infer it from missing output.
read memory.events               # oom / oom_kill counters
read cpu.stat                    # throttled_usec
read pids.events                 # max counter
```

- A timeout alone cannot prevent memory exhaustion or a fork bomb — limit each resource at the kernel boundary.

Cgroups meter use, not elapsed time

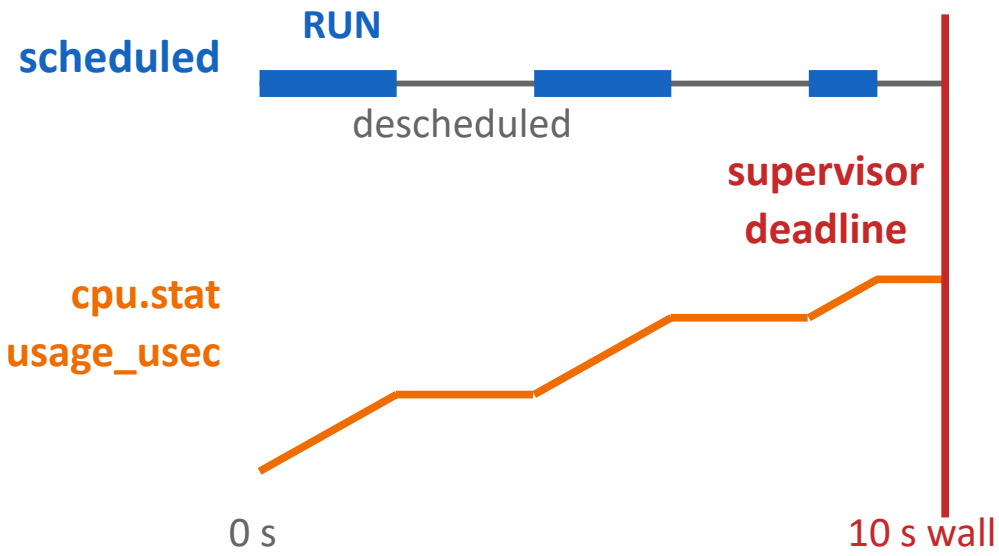
Why a wall-clock deadline cannot replace the resource caps

MEMORY one total for the process tree



memory.max limits the entire process tree.

CPU + WALL two clocks advance differently



10 s elapsed; only 5 s charged to CPU usage.

source: Linux kernel cgroup v2 — memory.current, memory.max, cpu.stat, cpu.max

Syscall filters reduce kernel attack surface

See and consume are bounded; now bound what the job may ask

Workload needs

```
openat(read-only repo)
read / write(scratch)
mmap / futex
clone(thread flags)
wait4 / exit
```

Workload does not need

```
mount / pivot_root
ptrace(host process)
bpf / perf_event_open
kexec_load
unshare(new privileges)
```

- Seccomp is an allowlist for operations, not data; filesystem and network boundaries constrain the arguments.
- The residual risk is the shared kernel — when that is too much exposure, move up a rung to a VM.

Treat every mount and socket as a capability

After the denials, your own grants are the attack surface that remains

Exposed object	Capability granted	Safer replacement
$HOME	read tokens, SSH keys, configs	empty synthetic home
Docker socket	create host-level workloads	no socket; fixed supervisor API
repo read-write	rewrite evidence under test	read-only source + scratch build
internet	exfiltrate or download mutable code	deny or allowlist proxy

Compose the mechanisms into one launch recipe

The grant list is the security property

```
# One line per capability class; anything not granted is denied.
namespaces   unshare --pid --mount --net       # sees only its own world
cgroup v2    memory.max, cpu.max, pids.max     # consumes only its ceilings
seccomp      allowlist(openat, read, mmap...)  # asks only what it needs
mounts       ./submission:ro + scratch rw      # touches only your grants
network      none                              # connects to nothing

# Afterward, read the receipts; do not infer from missing output.
memory.events    cpu.stat    pids.events
```

- Anything not granted does not cross the boundary; every grant is one reviewable line.
- The in-class OS-isolation lesson carries the threat model and the enforcement ladder; this deck is its keyboard half.