Containers 3/3: Under the Hood and Production

Linux kernel primitives; rootless containers as systemd services

Walter Rudametkin

Walter.Rudametkin@univ-rennes.fr

https://rudametw.github.io/teaching

ESIR 2 SI — Architectures Logicielles

Université de Rennes













Lecture 3: Road Map

1. DevOps and Why Containers
lab Module 0

2. Containers in Practice
lab Modules 2–4

3. Under the Hood and Production
today · lab Modules 0–1 and 5



Goal: understand what `podman run` *really* does, and run a stack that survives logout and reboot.

Previously: images, volumes, networks, builds, Compose (lecture 2).

appendix: deep dives **field guide**: recipes, troubleshooting

There Is No “Container” in the Linux Kernel

No “container” object in the kernel: the engine (Podman, Docker) only *configures* **independent kernel features**:

Primitive	Responsibility	Kernel interface
Namespaces	what it *sees*: PIDs, mounts, net, users	`clone`, `unshare`, `setns` (2)
cgroups v2	what it *uses*: CPU, RAM, I/O, PIDs	files in `/sys/fs/cgroup`
Security	what it *may do*: capabilities, seccomp, LSM	`capset`, `seccomp` (2), SELinux
OverlayFS	how its files are *stored*: layers, copy-on-write	`mount -t overlay`

A container is just an ordinary Linux process
started with these features configured.

Lab: Module 1, introduction
Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

A Container Is Just a Process on the Shared Kernel

```
$ podman run -d --name sleeper docker.io/library/alpine:3.20 sleep 600
$ ps -o user=,pid=,cmd= -C sleep        # seen from the host
alice    24817 sleep 600
$ readlink /proc/24817/ns/uts /proc/self/ns/uts
uts:[4026534010]
uts:[4026531838]
$ uname -r; podman exec sleeper uname -r
6.12.4-200.fc41.x86_64
6.12.4-200.fc41.x86_64
```

Just a process

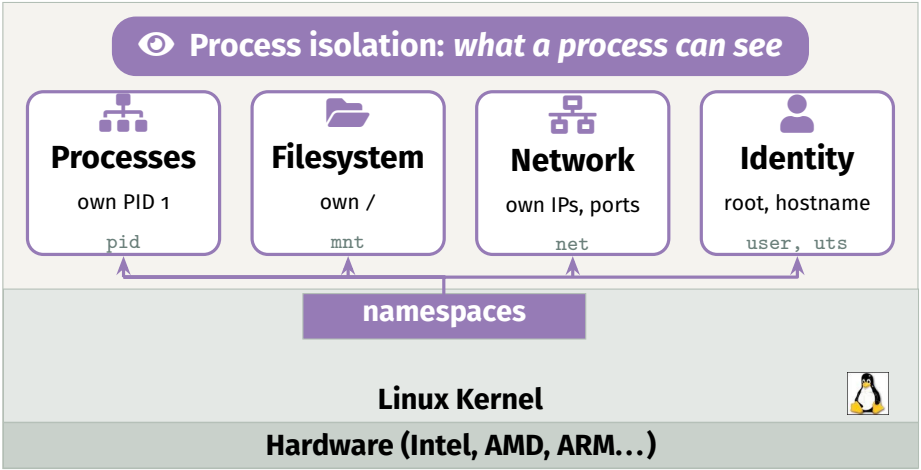
a host process, owned by you, in *other namespaces* (other inode numbers)

One shared kernel

image = *userland files* only: nothing to boot (ms), but a kernel bug hits all containers

Lab: Module 0; Module 1, Ex. 1.1 (step 6)

Linux Container Architecture: Namespaces



also: `ipc`, `cgroup` and `time` namespaces

Why Namespaces? What Each One Isolates

Problems: two apps on port 80 · an app kills all processes · "root" without being root
A namespace = a process's own view of one kind of resource:

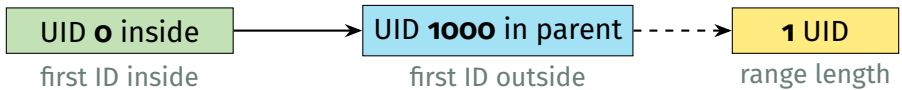
Namespace	Isolates	In a container
`mnt`	mounts, root filesystem	own `/` (image), via `pivot_root(2)`
`uts`	hostname	`podman run --hostname web`
`ipc`	System V IPC, POSIX queues	no shared memory between containers
`pid`	process IDs	entrypoint = PID 1 (own `/proc`)
`net`	interfaces, routes, firewall	own `lo`; each can listen on `:80`
`user`	UIDs/GIDs, capabilities	"root" inside = your UID outside
`cgroup`	view of the cgroup tree	`/sys/fs/cgroup` = its own node
`time`	monotonic, boot-time clocks	checkpoint / restore (CRIU)

One namespace of each type per process: `ls -l /proc/self/ns` (appendix)

User Namespace and `uid_map`

- Maps UIDs/GIDs inside ↔ parent namespace
- UID 0 inside: privileged *on the namespace's own resources* (mounts, interfaces, hostname); outside: an unprivileged user
- Linux 3.8 (2013): any user can create one → rootless containers

```
$ unshare --user --map-root-user bash
# cat /proc/self/uid_map
         0       1000          1
```



Unmapped UIDs show as `nobody` (65534); files of "root" inside are *yours* on the host.

Lab: A Hand-Made Sandbox with `unshare`

```
$ id -u; echo $$                          # 1000, and e.g. 4242
$ unshare --user --map-root-user --pid --mount --uts --fork bash
```

`--user`	new user namespace (`CLONE_NEWUSER`), created first
`--map-root-user`	write `0 1000 1` in `uid_map`: you are root inside
`--pid`	new PID namespace
`--mount`	new mount namespace (private mounts)
`--uts`	new hostname
`--fork`	fork before `exec`, so that `bash` is PID 1

No `sudo`, no Podman, no Docker: only the kernel.

Lab: Module 1, Ex. 1.1

Lab: Inside the Sandbox

```
# id
uid=0(root) gid=0(root) ...
# hostname esir-sandbox
# hostname
esir-sandbox
# echo $$
1
# ps aux | wc -l
312
```

Root, PID 1, own hostname…
but every host process?!

```
# mount -t proc proc /proc
# ps aux
USER  PID ... COMMAND
root    1 ... bash
root    2 ... ps aux
```

- `ps` reads `/proc`: still the parent's mount
- New `proc` mount = *our* PID namespace
- Allowed by `--mount` + `--map-root-user`

Lab: Module 1, Ex. 1.1 (steps 3–4)

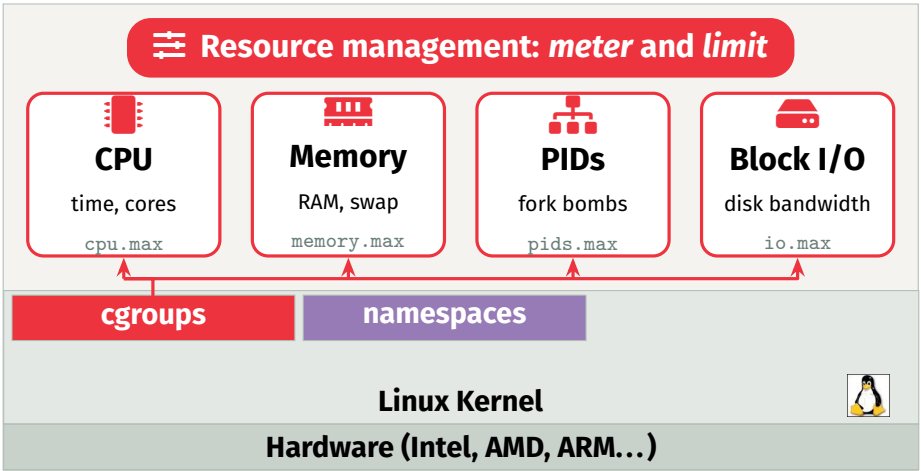
Rootless: `/etc/subuid`, `newuidmap`, Sanity Check

- Images hold files of many users: `alice:100000:65536` reserves host UIDs 100000–165535; setuid `newuidmap` / `newgidmap` write the map
- UID 0 → 1000 (you); UID $n \geq 1 \rightarrow 100000 + n - 1$ (example: lecture 2, appendix)

```
$ podman info --format \
    '{{.Host.Security.Rootless}} | {{.Host.CgroupsVersion}}'
true | v2                        note the s (Podman 5)
$ grep $(whoami) /etc/subuid
alice:100000:65536               missing?  usermod --add-subuids
$ podman unshare cat /proc/self/uid_map
         0       1000          1
         1     100000      65536
```

Lab: Module 0, sanity check; Module 3

Linux Container Architecture: cgroups



also: `cpuset`, `hugetlb`, devices (eBPF filter); no network controller in v2

Why cgroups? What Can They Control?

- Namespaces *hide*, they do not *limit*: one memory leak, busy loop or fork bomb can starve the whole host
- cgroups (Linux 2.6.24, 2008): group processes, then meter and limit their memory, CPU, block I/O, PIDs, devices…
- **Crowd control**: freeze / thaw a group (`cgroup.freeze`, `podman pause`), kill it with no fork escape (`cgroup.kill`)
- systemd: one cgroup per service and session; engines: one per container

Lab: Module 1, Ex. 1.3
Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

The cgroup v2 Filesystem

```
$ stat -fc %T /sys/fs/cgroup          # tmpfs would mean v1
cgroup2fs
$ cat /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids rdma misc
$ cat /proc/self/cgroup                # which node am I in?
0::/user.slice/user-1000.slice/user@1000.service/app.slice/...
$ ls /sys/fs/cgroup/user.slice/
cgroup.procs  cpu.max  io.max  memory.current  memory.max  pids.max ...
```

- A **tree**: `mkdir` = new node; files = knobs (`memory.max`), counters, members (`cgroup.procs`); ancestors' limits apply too
- Rootless containers sit in your `user-1000.slice`: they die with your session unless "lingering"

v2 (Linux 4.5, 2016): one unified tree, *delegated* to users (rootless); v1: appendix

Lab: Module 1, Ex. 1.3 (step 1)

Lab: A 16 MiB Container

```
$ podman run --rm -it --name mem-test --memory=16m \
      --memory-swap=16m docker.io/library/alpine:3.20 sh
/ # cat /sys/fs/cgroup/memory.max
16777216                                  = 16 x 1024 x 1024 bytes
/ # cat /sys/fs/cgroup/memory.swap.max
0                                         --memory-swap = RAM + swap
/ # tail /dev/zero
Killed
/ # echo $?
137
/ # grep oom_kill /sys/fs/cgroup/memory.events
oom_kill 1
```

Inside the container, `/sys/fs/cgroup` *is* the container's cgroup (cgroup namespace).

Lab: Module 1, Ex. 1.3

Memory Limits, the OOM Killer and Exit 137

`memory.current`	bytes used now
`memory.high`	throttle, *no* kill
`memory.max`	hard limit
`memory.events`	`oom`, `oom_kill`…

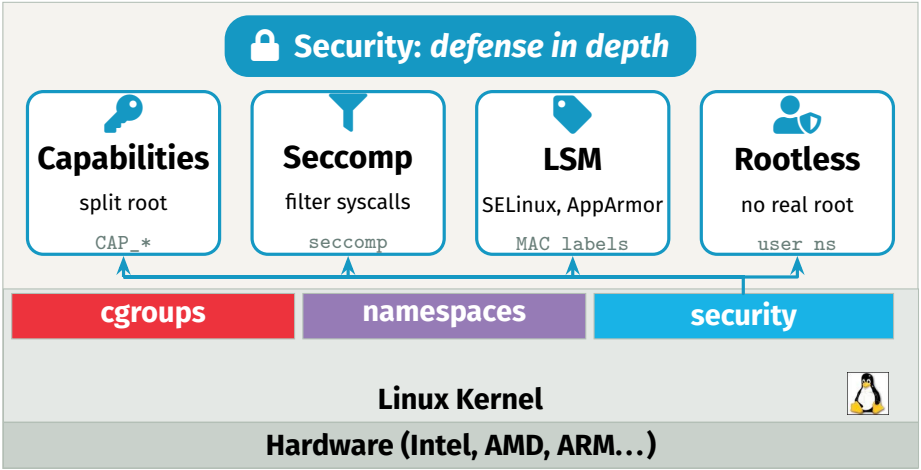
Why "Killed"? `tail` keeps the last *lines*; `/dev/zero` has no newline → one endless line → `memory.max`, no swap: nothing to reclaim

- The OOM killer, *scoped to this cgroup*, `SIGKILL`s its biggest process: `sh` survives, the host does not notice
- Killed by signal $N \Rightarrow$ exit status $128 + N$: $128 + 9 = 137$ (143: `SIGTERM`)
- Main process killed: `Exited (137)`, `OOMKilled: true`

No limit: the *global* OOM killer may pick any host process. Engine flags → cgroup files: appendix

Lab: Module 1, Ex. 1.3

Linux Container Architecture: Security



also: `no_new_privs`, read-only root filesystem, cgroup limits

Containers Do Not Contain

Containers do NOT contain!!!

A container is a process on a **shared kernel**.
One kernel bug, one misconfiguration, or one dangerous flag and the isolation is gone.

- Not everything is namespaced: kernel modules, keyrings, parts of `/proc` and `/sys`, the clock, devices…
- One shared kernel = one shared attack surface → engines stack several layers (defense in depth)


RENA
MONROVIA

Three Locks on Every Container

	What it does	Key flags
Capabilities	root's power split into ≈ 40 flags; Podman keeps 11 (not `CAP_SYS_ADMIN`)	`--cap-drop=all`, then `--cap-add` what is needed
Seccomp	BPF filter on system calls: risky ones fail (`kexec_load...`); `no_new_privs`: setuid gains nothing	`--security-opt` + `seccomp=prof.json` or `no-new-privileges`
LSM	mandatory access control, even for root: SELinux labels (one per container), AppArmor	`:z` / `:Z` on bind mounts (lecture 2)

```
$ podman run --rm --cap-drop=all --security-opt=no-new-privileges \
      docker.io/library/alpine:3.20 chown nobody /tmp
chown: /tmp: Operation not permitted    root, but without CAP_CHOWN
```

⚠ On by default. Never `--privileged` (drops all three), never disable SELinux

Rootless, and What Breaks Isolation

Rootless

- Engine and containers run as *your* user, no root daemon
- Container root = your UID: an escape lands on an unprivileged account
- Default: Podman rootless, Docker rootful
- Costs: no ports < 1024 (default), user-space networking (pasta), some devices and sysctls unavailable

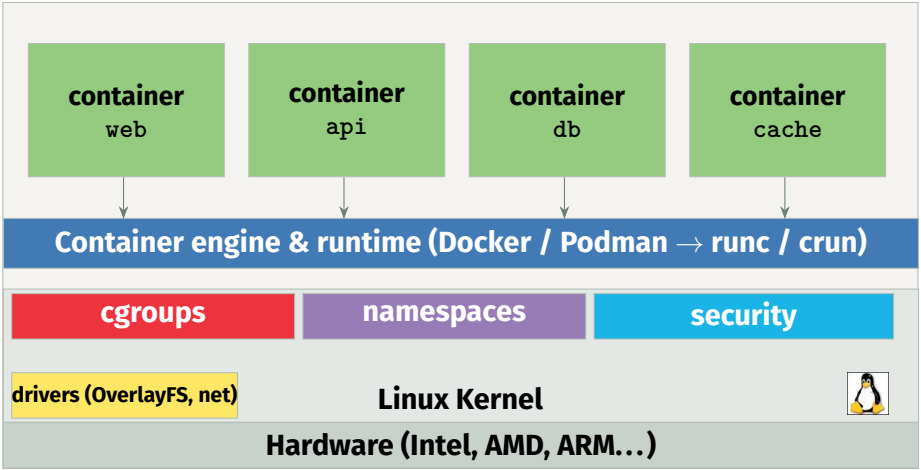
What breaks isolation

- Docker socket access = root on the host:

```
docker run --privileged \
  -v /:/hostfs -ti \
  docker.io/library/alpine:3.20
```

- `--privileged`, `-v /:/host`, `--device`, `--network=host`, `--pid=host`
- Kernel vulnerabilities
- Copy-pasted scripts and Compose files!

Linux Container Architecture: Drivers and Engine



OverlayFS: Union Mount + Copy-on-Write

- **Copy-on-write**: looks like a copy, is a reference; real copy only on *modification* (`fork()`, snapshots, VMs, containers)
- **Union mount**: several directories seen as one: read-only image layers + one writable container layer
- OverlayFS (Linux 3.18+): the default everywhere (AUFS is history; Btrfs, ZFS: snapshot drivers, not union filesystems)

`lowerdir`	image layers, read-only; `:`-separated, leftmost on top
`upperdir`	container layer, read-write: all changes land here
`workdir`	empty scratch directory, same filesystem as `upperdir`
`merged`	the unified view: the container's `/`

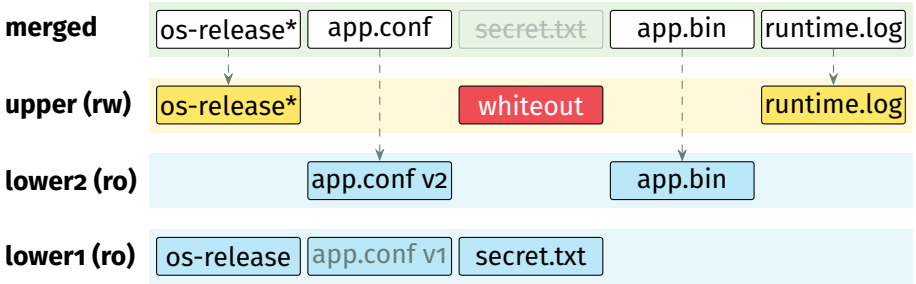
Lab: Reads, Copy-Up and Whiteouts

In the Ex. 1.1 sandbox, `lower1` = base OS, `lower2` = app layer (setup, steps 1–3: appendix):

```
# mount -t overlay overlay -o lowerdir=./lower2:./lower1,\
    upperdir=./upper,workdir=./work ./merged
# cat merged/app.conf
Config v2 (overridden by layer 2)          top layer wins
# ls -A upper                                empty
# echo "Modified at runtime" >> merged/os-release.txt
# rm merged/secret.txt
# echo "New runtime log" > merged/runtime.log
# cat lower1/os-release.txt
Ubuntu 24.04 Base Layer                    untouched
# ls -l upper
-rw-r--r-- 1 root root   57 ... os-release.txt   copy-up
-rw-r--r-- 1 root root   16 ... runtime.log      new file
c--------- 2 root root 0, 0 ... secret.txt       whiteout
```

Lab: Module 1, Ex. 1.2 (steps 4–6)

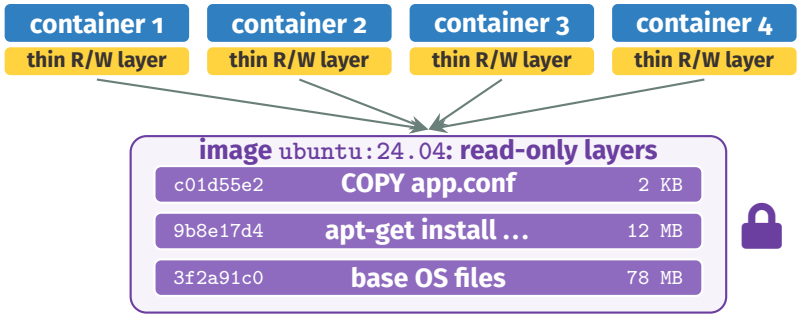
How OverlayFS Resolves Files



- **Read**: the top-most layer that has the file wins (reads fall through)
- At start, `upperdir` is empty: a new container costs nothing
- **Write**: copy-up of the *whole* file to `upperdir`, then modify
- **Delete**: a whiteout (char device 0,0) hides the lower file

Lab: Module 1, Ex. 1.2

Containers Share the Image Layers



per container: its changes only (`upperdir`), empty at start

shared: once on disk, once in RAM cache (`lowerdir`)

⚠ First write copies the *whole* file → write-heavy data in volumes
⚠ Deleted in a later layer ≠ gone: still shipped (secrets!)

Lab: Module 1, Ex. 1.2; Module 4

Putting It Together: What `podman run` Does

Podman and the OCI runtime (`crun`, `runc`): only **ordinary system calls**

1. **Podman** mounts the image layers + an empty layer	→ OverlayFS
2. **crun** `clone(2)`s a process in new namespaces	→ Namespaces
3. writes its `uid_map` (`newuidmap`, `/etc/subuid`)	→ user ns
4. puts it in a new cgroup, writes `memory.max`…	→ cgroups
5. mounts `/proc`, `/sys`, `/dev`, volumes; `pivot_root(2)`	→ mnt ns
6. drops capabilities, loads seccomp, sets the SELinux label	→ Security
7. `execve(2)`s the entrypoint → PID 1	→ pid ns

`podman exec`: `setns(2)` joins the *existing* namespaces.

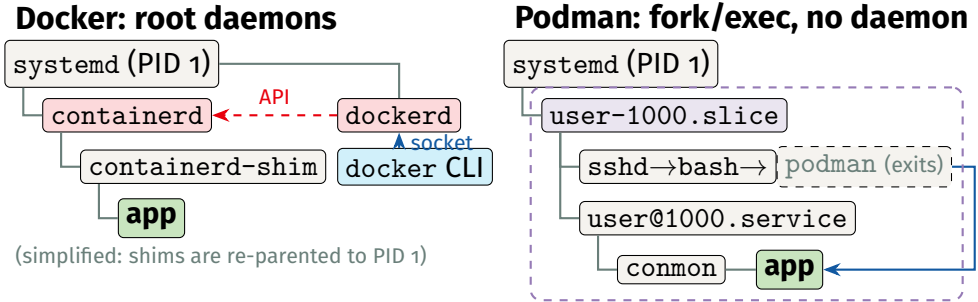
Lab: Module 1, introduction

Recap: Primitive, Flag, Where to Look

Primitive	Podman flag	Where to look
Namespaces: what it *sees*	`--hostname`, `--network`, `--pid`	`ls -l /proc/<pid>/ns/`, `lsns`
User ns: who it *is*	`--userns=keep-id`, `/etc/subuid`	`/proc/<pid>/uid_map`, `podman unshare`
cgroups v2: what it *uses*	`--memory`, `--cpus`, `--pids-limit`	`.../libpod-<id>.scope/memory.max`; exit 137, `OOMKilled: true`
Capabilities, seccomp: what "root" *may do*	`--cap-drop=all`, `--security-opt`	`grep -E 'Cap|Seccomp' /proc/<pid>/status`
SELinux, AppArmor: what it may *touch*	`:Z` / `:z` on bind mounts	`ps -eZ | grep container_t`, `ls -Z`
OverlayFS: how its `/` is *stored*	`--read-only`; volumes bypass it	`podman inspect -f '{{.GraphDriver.Data.UpperDir}}'`

`<pid>` = the container's PID 1 on the host: `podman inspect -f '{{.State.Pid}}' web`

Who Keeps Your Containers Alive?



- Docker: root daemons supervise, and **restart containers at boot**
- Podman: `conmon` supervises inside your user slice. Last SSH logout ⇒ `logind` stops `user@1000.service` ⇒ even `-d` containers die; and at boot, nobody starts them

Lab: Module 5

User Lingering: `loginctl enable-linger`

```
loginctl show-user $(whoami) -p Linger     # Linger=no
loginctl enable-linger $(whoami)
ls /var/lib/systemd/linger/                # marker file: <username>
# Compose stacks: start their "restart: always" containers at boot
systemctl --user enable podman-restart.service
```

- `user@1000.service` now **starts at boot** and **survives logout**: containers keep running (undo: `disable-linger`)
- `podman-restart.service`: no ordering, no health, no journal per service ⇒ **Quadlets** (next)

⚠ Refused over SSH (polkit)? `sudo loginctl enable-linger <user>` or ask the admin

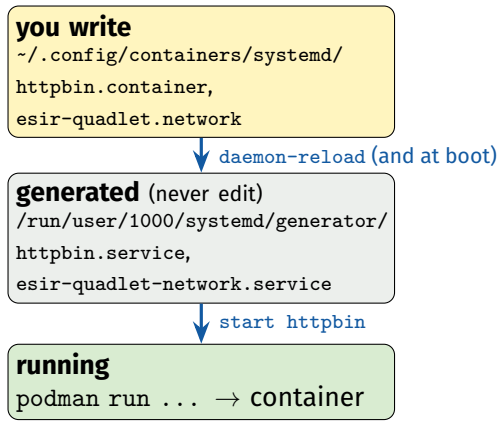
WSL2: enable systemd (`/etc/wsl.conf`: `[boot] systemd=true`).

Lab: Module 5, Ex. 5.1

Quadlets: Containers as systemd Units

- Podman $\geq$ 4.4: containers **declared** in systemd INI files
- Types: `.container` `.volume` `.network` `.image` `.build` `.kube` `.pod`
  pod = containers sharing one network namespace (appendix)
- systemd gives: boot start, ordering, restarts, cgroup limits, journal

~~`podman generate systemd`~~: deprecated



Lab: Module 5, Ex. 5.2–5.3

`httpbin.container` (1/2), Network and Volume Units

`httpbin.container`

```
[Unit]
Description=Hardened go-httpbin (Quadlet)
Wants=network-online.target     # (*)
After=network-online.target     # (*)
[Container]
Image=docker.io/mccutchen/go-httpbin:v2.15.0
ContainerName=quadlet-httpbin
Network=esir-quadlet.network
PublishPort=127.0.0.1:8080:8080  # loopback only
PidsLimit=128           # pids.max: stops fork bombs
ReadOnly=true           # read-only /: no webshell
NoNewPrivileges=true    # setuid gains nothing
AutoUpdate=registry     # podman auto-update
```

Refers to the **file** (`Network=`, `Volume=redis-quadlet.volume:/data`) ⇒ **dependency**, started first

Lab: Module 5, Ex. 5.2

`esir-quadlet.network`

```
[Network]
NetworkName=esir-quadlet-net
Subnet=10.89.50.0/24
```

`redis-quadlet.volume`

```
[Volume]
VolumeName=esir-quadlet-data
```

Same as `podman network|volume create`, **managed by systemd** (`NetworkName=`: name in Podman).

(*) not needed in user units: Quadlet adds its own network wait

`httpbin.container` (2/2): `[Service]` and `[Install]`

```
[Service]
Restart=always
RestartSec=5s
MemoryMax=64M
CPUQuota=50%
[Install]
WantedBy=default.target
```

- `[Service]`: copied into the generated unit
- `Restart=always`: restart whatever the exit status, after `RestartSec`
- `MemoryMax`, `CPUQuota` ⇒ cgroup v2 `memory.max`, `cpu.max`

`[Install] WantedBy=default.target`

- replaces `systemctl enable`: start with the user manager (at boot, with linger)
- generated units cannot be enabled: `systemctl enable` fails on them

Lab: Module 5, Ex. 5.2

Compile, Start, Audit, Crash

```
/usr/libexec/podman/quadlet -dryrun -user   # check the units
systemctl --user daemon-reload              # run the generator
systemctl --user start httpbin.service      # network unit too
curl 127.0.0.1:8080/ip
journalctl --user -u httpbin.service -n 20 --no-pager
# Crash it: SIGKILL the container's PID 1
kill -9 $(podman inspect -f '{{.State.Pid}}' quadlet-httpbin)
sleep 6; systemctl --user status httpbin.service
```

Crash recovery: PID 1 killed ⇒ exit 137 (128 + 9) ⇒ unit fails ⇒ `Restart=always`: a **fresh** container after `RestartSec=5s` (`ReadOnly`: nothing lost)

Debian/Ubuntu: `/usr/lib/podman/quadlet`. Podman ≥ 5.6: `podman quadlet install|list|print|rm`.

Lab: Module 5, Ex. 5.3

From `compose.yaml` to Quadlets

`compose.yaml`	**Quadlet** (`[Container]` unless noted)	**Note**
`image:`	`Image=`	pinned FQIN
`ports:`	`PublishPort=127.0.0.1:8080:8080`	front-end only
`volumes:`	`Volume=pgdata.volume:` `/var/lib/postgresql/data`	+ a `pgdata.volume` file
`environment:`	`Environment=`, `EnvironmentFile=`, `Secret=`	`Secret=`: `podman secret`
`healthcheck:`	`HealthCmd=` + `Notify=healthy`	"started" = healthy
`depends_on:`	`[Unit] Requires=postgres.service` `After=postgres.service`	≡ `service_healthy`
`networks:`	`Network=backend.network`	+ a `backend.network` file
`restart:`, `mem_limit:`	`[Service] Restart=always`, `MemoryMax=`	applied by systemd

`podlet compose` drafts the units from a `compose.yaml` (review them); full capstone units: **the field guide**

Lab: Module 4 stack, Module 5

Compose or Quadlets?

	`podman compose`	**Quadlets**
Description	one `compose.yaml`	one file per object
Supervised by	`conmon`, in your user slice	**systemd**, one unit each
After logout	killed, unless linger	runs (linger)
At boot	`podman-restart.service`	`WantedBy=default.target`
Restarts	`restart:`	`Restart=`
Limits	`mem_limit:`	`MemoryMax=...`
Logs	`compose logs`	`journalctl`
Updates	manual `pull` + `up`	`podman auto-update`
Best for	dev, quick stacks	**single-host prod.**

Same concepts, same kernel features: only the *supervisor* changes. For production, Quadlets are the **robust** option.

Lecture 3: What to Remember

- **No "container" in the kernel**: namespaces (what it *sees*), cgroups (what it *uses*), security (what it *may do*), OverlayFS (its files)
- **User namespace**: container root = your UID; other UIDs from `/etc/subuid`
- **cgroups v2**: over `memory.max` → OOM kill → exit 137 (128 + SIGKILL)
- **Containers do not contain**: shared kernel; drop capabilities, keep seccomp + SELinux, never `--privileged`
- **Production**: linger: containers survive logout; Quadlets = systemd units (restart, boot, journal)

→ Next: the course conclusion lectures 1–3

what to remember from the whole course; what was not covered; going further

Not Covered Here

Supply chain

- signing: cosign, sigstore
- SBOM: `syft`, `build --sbom`
- scanning: Trivy, Grype
- multi-arch images: `--platform`, manifest lists

Other worlds

- dev containers, Toolbx
- Kubernetes, multi-host networking (details: appendix)
- Windows containers
- WebAssembly (Wasm)

Same building blocks: OCI images, registries, namespaces, cgroups.

The Whole Course: What to Remember

M0 Environment	a process on the host's Linux kernel (Windows/macOS: a Linux VM); runs on *any Linux kernel of the same CPU architecture*
M1 Kernel	namespaces: what it sees · cgroups: what it uses · OverlayFS: its layers · exit 137 = OOM kill ($128 + 9$)
M2 Images	immutable OCI layers; full names, pinned tags or digests; a container lives as long as its PID 1
M3 Storage, net	rootless = UID mapping (`:Z`, `:U`); named volumes; DNS networks; publish only the front, on `127.0.0.1`
M4 Build, Compose	cache-friendly order, multi-stage, non-root `USER`, no secrets in layers; health checks + `service_healthy`; segmented networks
M5 Production	systemd supervises: linger + Quadlets (`Restart=`, limits, journal, auto-update)

Trade-off: lighter and faster than VMs, but a shared kernel = weaker isolation.

Going Further

Online

Container training (J. Petazzoni et al.)	`https://container.training/`
Podman (rootless, Quadlets, man pages)	`https://docs.podman.io/`
Docker	`https://docs.docker.com/`
OCI image, runtime, distribution specs	`https://opencontainers.org/`
Kubernetes	`https://kubernetes.io/docs/`

Books

Liz Rice, *Container Security*, O'Reilly, 2020
Daniel J. Walsh, *Podman in Action*, Manning, 2023

Man pages

`namespaces(7)`, `user_namespaces(7)`, `cgroups(7)`, `unshare(1)`, `podman-systemd.unit(5)`

This course: the appendix (deep dives) and the field guide (troubleshooting, HTTPS with Caddy, backups, the capstone as Quadlets)