Container Security

Thematic CERN School of Computing, 2026

Daniel Kouřil
kouril@cesnet.cz

MUNI

cesnet

OS-Level virtualization

Container = process isolation + limitation of resources

How is it done?

OS-Level virtualization

```
head$ id
uid=1000(kouril) gid=1000(kouril) groups=1000(kouril),24(cdrom),25(floppy),29(au
dio),30(dip),44(video),46(plugdev),100(users),106(netdev),111(bluetooth),113(lpa
dmin),116(scanner),121(tss),127(vboxusers)
head$ make_container ubuntu /bin/bash
root@01c324816467:/# ps ax
    PID TTY      STAT   TIME COMMAND
      1 pts/0    Ss     0:00 /bin/bash
      4 pts/0    R+     0:00 ps ax
root@01c324816467:/# id
uid=0(root) gid=0(root) groups=0(root)
root@01c324816467:/# exit
exit
head$
```

Container = process isolation + limitation of resources

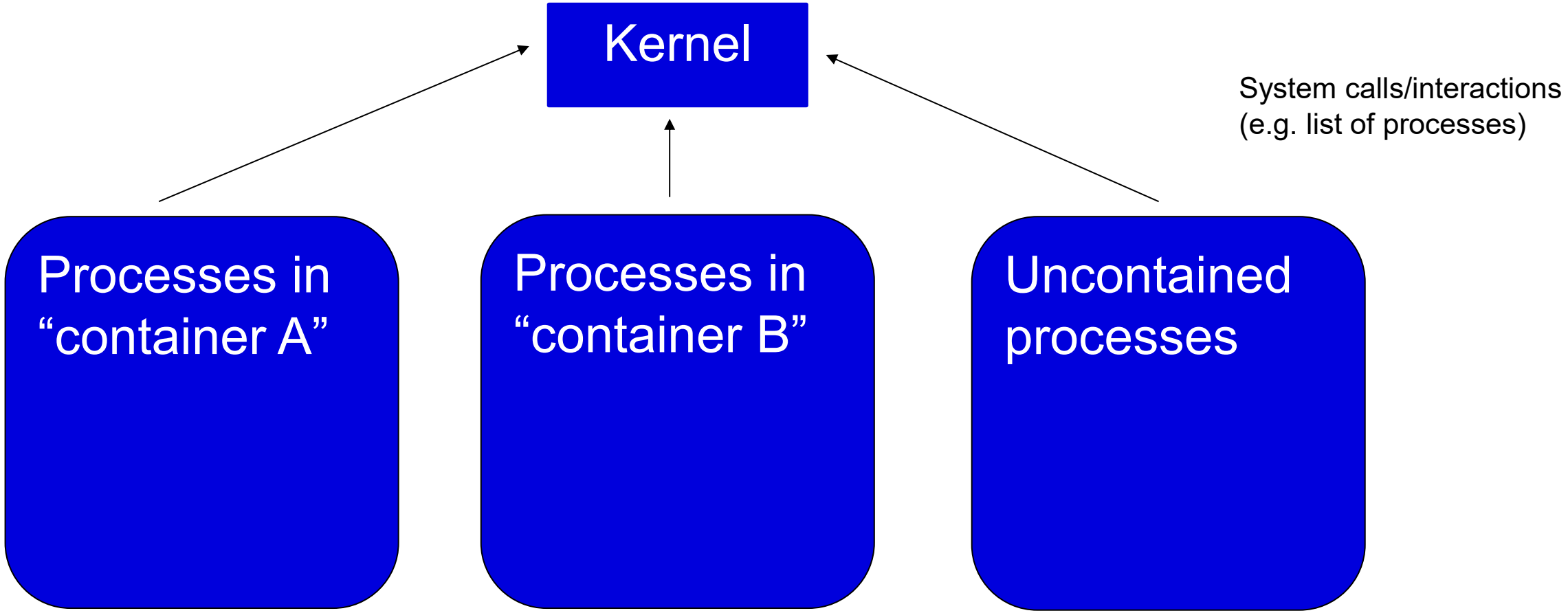
How is it done?

Linux Containers

- **Two crucial technologies in Linux kernel**
- **Namespaces**
  - Containers are *contained* in “compartments”, can’t see outside
- **Control groups (cgroups)**
  - Processes are limited in access to resources (memory, CPU, I/O)
- **Linux container =**

  process(es) assigned a set of namespaces + cgroups
  - Everything is controlled by the same OS level (kernel) (no host/guest OS)
  - The processes have “just” limited view on the whole system (or illusion)

Containment using namespaces



Common namespaces

PID namespace	• processes isolation
NET namespace	• managing/separating network interfaces
IPC namespace	• separating inter-process communication
MNT namespace	• managing/separating filesystem mount points
UTS namespace	• mainly to set the hostname and domain name visible to the process
User ID namespace	• privilege isolation

Isolating process IDs

- **Process ID namespace allows the container to establish separate process trees**

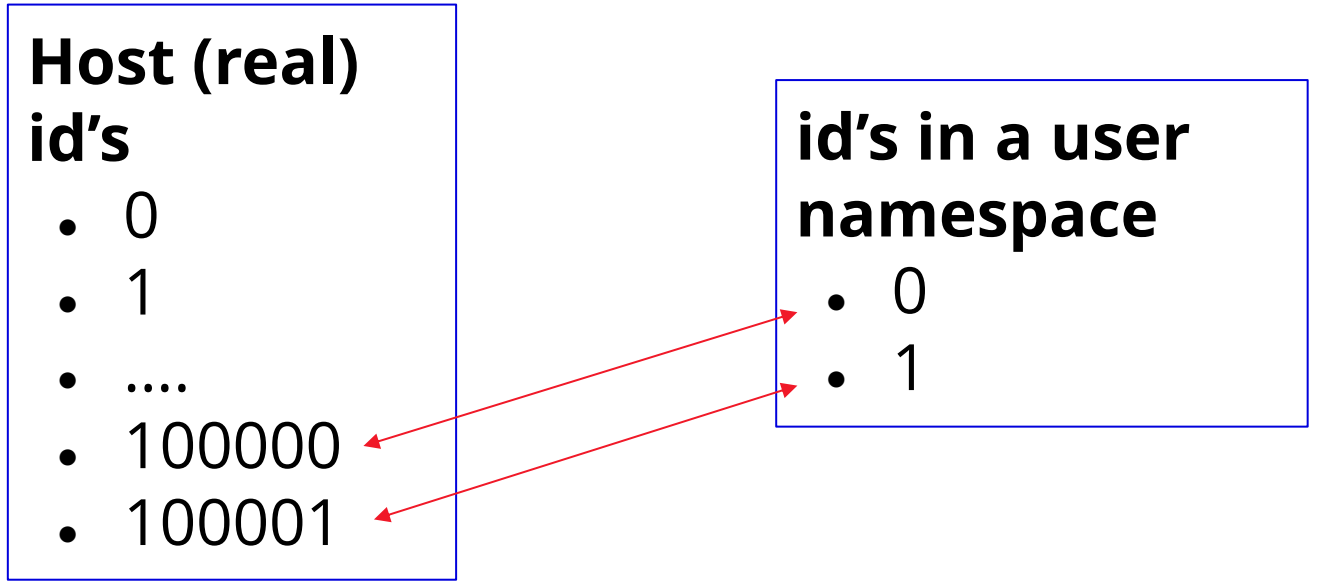
```
host# docker run -it debian bash
root@3146c2faec9b:/# dash
# ps af
  PID   TTY      STAT   TIME     COMMAND
    1   pts/0    Ss      0:00     bash
    6   pts/0    S       0:00     dash
    7   pts/0    R+      0:00      \_ ps af
```

the complete picture is still visible from the host

```
.......
1029  ?          Ssl      7:48        /usr/bin/containerd
28834 ?          Sl       0:00        \_ containerd-shim -namespace moby  .........
28851 pts/0      Ss       0:00        \_ bash
28899 pts/0      S+       0:00             \_ dash
.......
```

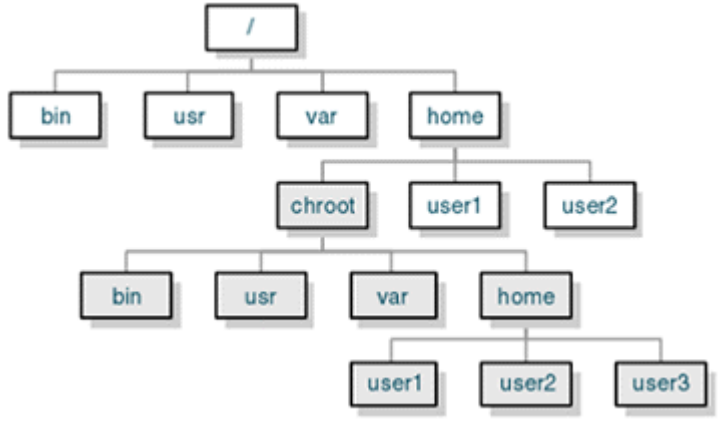
Isolating users and groups

- **Separated uid/gid allocations, decoupled from real identifiers**
- **A user process in a namespace is assigned a ‘local’ identifier that is recognized only inside the namespace**
  - a mapping need to be maintained between the namespace and “global” (real) uids/gids



Isolating container filesystem

- **Complete container filesystem is part of the host filesystem hierarchy**
- **chroot() is used to change the root directory**
  - /home/chroot
- **Processes can't "escape" the chroot'ed directory**
  - /home/chroot/bin/bash (full/global path to a shell in chroot)
  - /bin/bash (path used from within the container



Source: http://www.jmcresearch.com/src/articlehelper.php?action=print&id=3

Linux kernel capabilities

- **Capabilities turn the binary “root/non-root” dichotomy into a fine-grained access control system**
- **Capabilities can be assigned to a thread to determine whether that thread can perform certain actions.**
  - CAP_NET_BIND_SERVICE capability is needed for a process to bind to a low-numbered (below 1024) port.
- **Containers processes can be assigned different capabilities that limit or extend the rights to perform actions**

Resource restrictions using cgroups

- **Linux *control groups* limit resources processes can use**
  - CPU, I/O, memory
- **Processes can see only fraction of all available resources**
  - A container can’t consume all resources to starve other containers
- **Cgroups are maintained by the kernel**
  - Settings mediated via kernel pseudo-filesystems

```
machine$ cat /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids rdma misc
machine$
```

Limiting memory using cgroups

 # create a specific cgroup:

```
mkdir /sys/fs/cgroup/memory/memory_eaters
```

 # limit the memory usage to ca. 10MB

```
echo 10000000 > \
     /sys/fs/cgroup/memory/memory_eaters/memory.limit_in_bytes
```

 # enter the new cgroup with the current shell to apply to limit:

```
echo $$ > /sys/fs/cgroup/memory/memory_eaters/cgroup.procs
```

Security assessment

- **Is *it* secure?**
- **Threat model & landscape**
- **Containers introduced a new eco-system**
  - Containers
  - Images
  - New “middleware” components (OS support, services)

Breaking container isolation

- May be done deliberately to provide full access to the system
- Unwanted container escape compromises the security
- Escaping container refers to different possible actions
  - data access (according to C.I.A):
    - reading (violation of C.)
    - modifying (violation of I.)
  - executing code outside the container

Potential vectors to break isolation

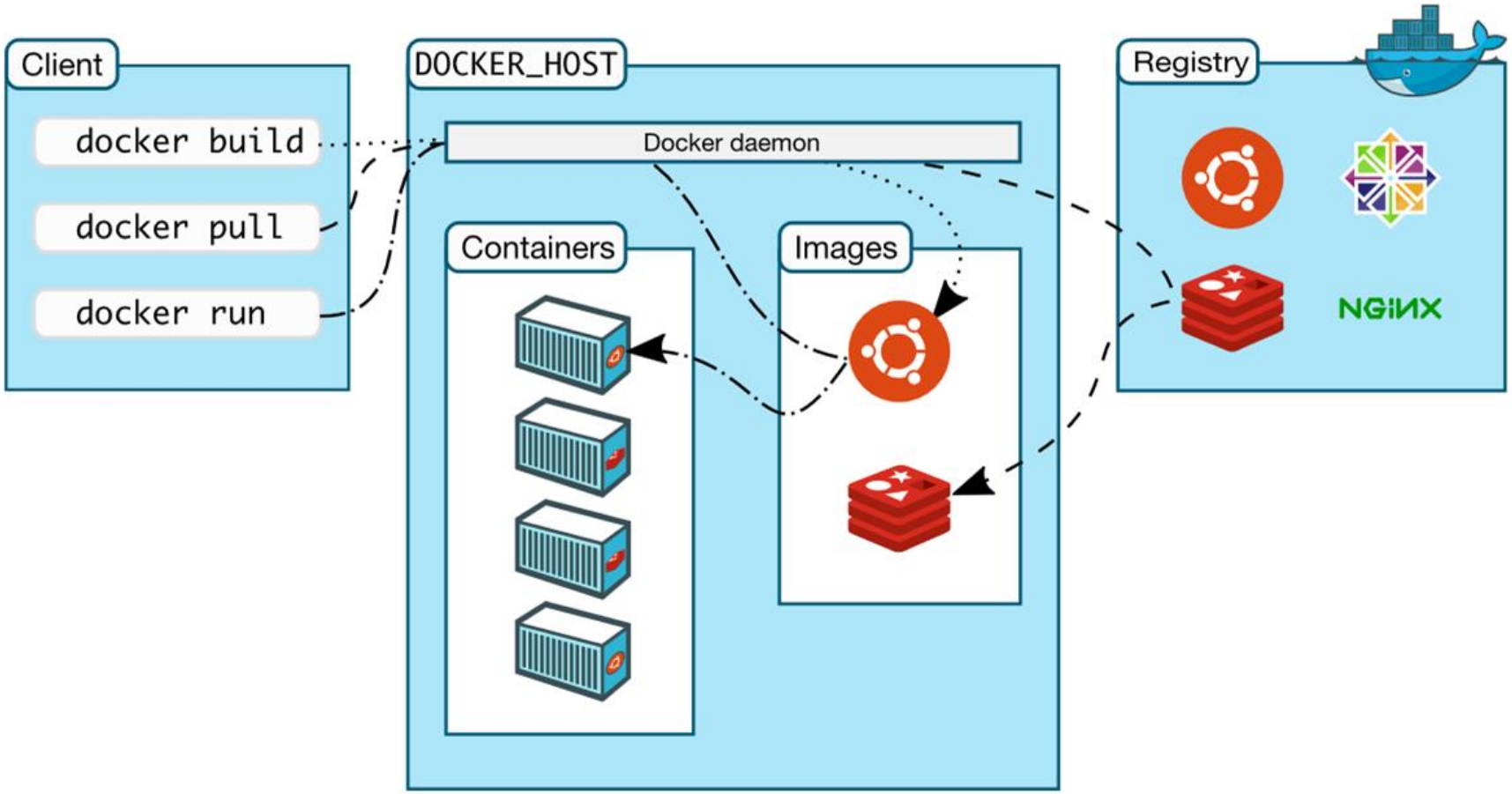
- Misconfiguration
  - Depends on the container attributes (capabilities) and setting
  - Some not obvious
    - allowing mount effectively gives access to uncontained code execution
  - inject a “hook” that is invoked by a trusted component in the system
    - a crontab rule or a kernel “notifier” running command on certain events
    - must run outside the container - APIs (e.g. inotify) won’t help
- Vulnerabilities in kernel or container components
  - Proper patch management is important
  - Updates to kernel requires reboot of the machine

Docker

Docker

- **Ecosystem to work with containerized applications**
  - Management of containers
  - Support for data management, networking, image management, …
- **Docker image as an application "package"**
  - Snapshot of filesystem (usually the application and dependencies)
    - Nothing more though – no/limited OS components, specific libraries, etc.
  - Images can be made available from repositories (the Docker Hub)
  - Can be instantiated to create the container
    - The same relation as class – object, program - process
- **`docker run -it ubuntu /bin/bash`**
  - Download the image
  - "mount" the image filesystem
  - Start the entrypoint contained in a namespace

Docker Architecture

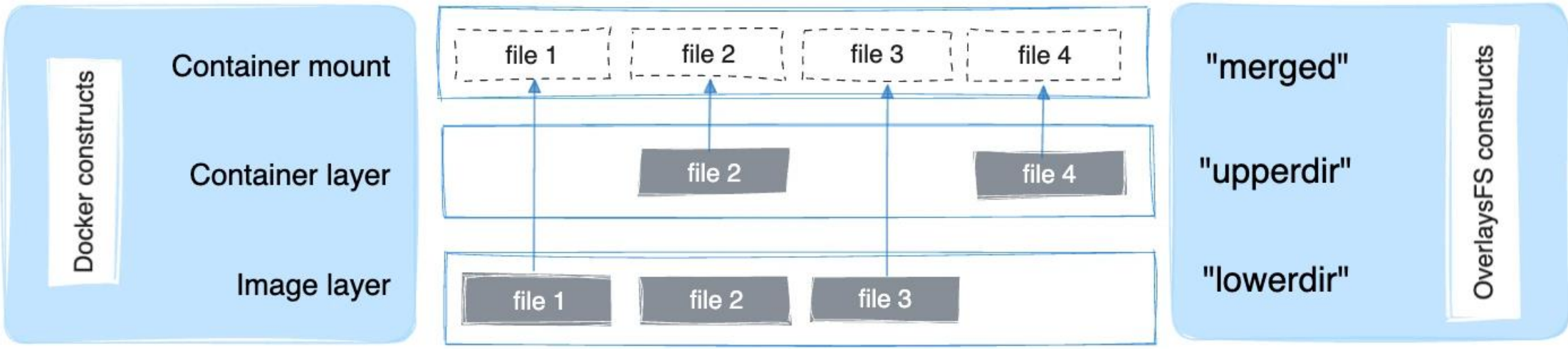


https://docs.docker.com/get-started/overview/

File arrangements for Docker

- **How to structure an image and container**
  - Is a subdirectory on a filesystem enough?
- **Applications often share similar needs**
  - Essential libraries, system configuration (timezone, ..)
- **How to arrange file system efficiently?**

Union file system



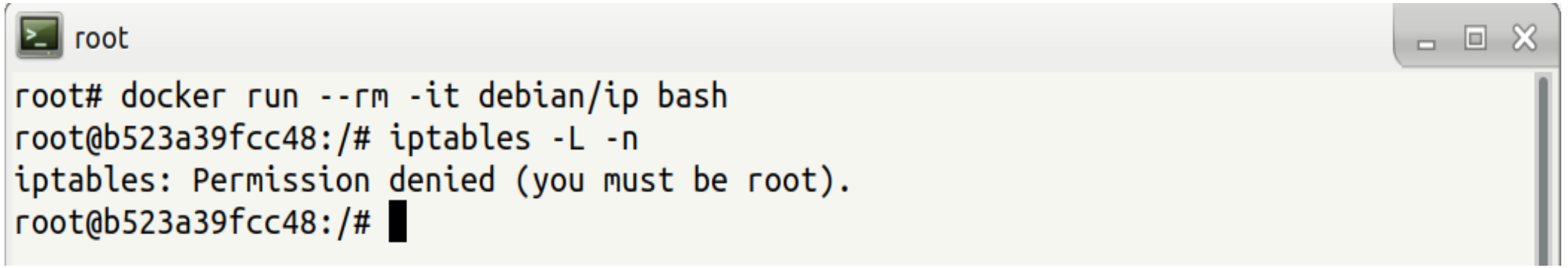
- Read-only layers
- “virtual” file system view
- Usually multiple image layers

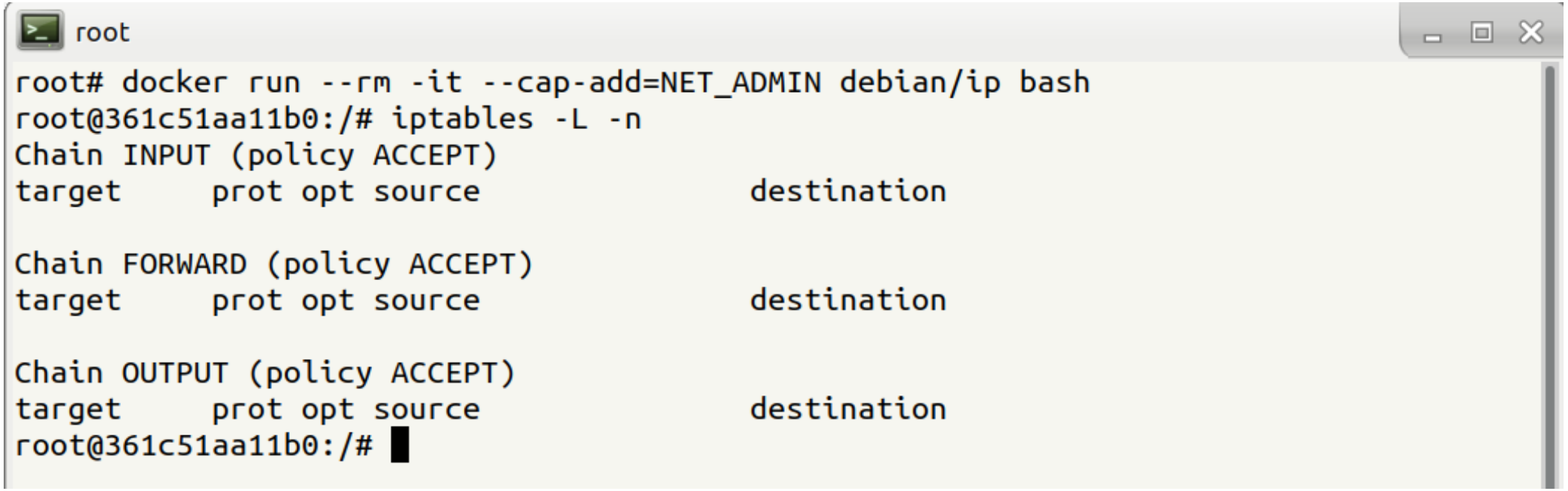
```
host# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED         STATUS         PORTS     NAMES
3461d90995cb   ubuntu    "bash"    7 minutes ago   Up 7 minutes             sad_jennings
host#
host# docker exec sad_jennings mount | grep ' / '
overlay on / type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay2/l/LB6E57RHGZ4SAIGB62KWX7LKZZ:/var/lib/docker/overlay
2/l/MF25G5RXOLBZ7ULECZYO2KRJD3,upperdir=/var/lib/docker/overlay2/e1b6562eef2ebea4125420f4faad408c5b4210f2cb8beffe3652bc08d962
7968/diff,workdir=/var/lib/docker/overlay2/e1b6562eef2ebea4125420f4faad408c5b4210f2cb8beffe3652bc08d9627968/work)
host# docker exec sad_jennings ls /var/lib/docker/overlay2/e1b6562eef2ebea4125420f4faad408c5b4210f2cb8beffe3652bc08d9627968
ls: cannot access '/var/lib/docker/overlay2/e1b6562eef2ebea4125420f4faad408c5b4210f2cb8beffe3652bc08d9627968': No such file o
r directory
host#
host# mount | grep overlay
overlay on /var/lib/docker/overlay2/e1b6562eef2ebea4125420f4faad408c5b4210f2cb8beffe3652bc08d9627968/merged type overlay (rw,
relatime,lowerdir=/var/lib/docker/overlay2/l/LB6E57RHGZ4SAIGB62KWX7LKZZ:/var/lib/docker/overlay2/l/MF25G5RXOLBZ7ULECZYO2KRJD3
,upperdir=/var/lib/docker/overlay2/e1b6562eef2ebea4125420f4faad408c5b4210f2cb8beffe3652bc08d9627968/diff,workdir=/var/lib/doc
ker/overlay2/e1b6562eef2ebea4125420f4faad408c5b4210f2cb8beffe3652bc08d9627968/work)
host# ls /var/lib/docker/overlay2/e1b6562eef2ebea4125420f4faad408c5b4210f2cb8beffe3652bc08d9627968/merged
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  this_is_my_file  tmp  usr  var
host#
host# docker exec sad_jennings touch /this_is_my_file
host# docker exec sad_jennings ls /
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  this_is_my_file  tmp  usr  var
host#
host# ls /var/lib/docker/overlay2/e1b6562eef2ebea4125420f4faad408c5b4210f2cb8beffe3652bc08d9627968/merged
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  this_is_my_file  tmp  usr  var
host#
```

I am the root. Or not?

- **multiple levels of elevated privileges, from an unprivileged user to full root rights:**
  - if user namespace is used, the root inside a container has no root privileges outside in the host system
    - Not enabled by default in Docker deployments
  - the root in a container has some elevated privileges but restricted by a set of capabilities
    - The Docker default behavior
  - we can explicitly add extra capabilities to a container on start
  - with the --privileged flag, we have full root rights granted

Root may be limited





Root(ful) Docker daemon

- Containers and images are maintained by Docker Daemon
- Docker daemon needs full access (administrative) to the system
  - Containers can be given all features they need, including controlled access to any data on the host filesystem
  - Data is easy to share, process can bound arbitrary ports, …
- Docker containers can be used to access and/or modify any part of the whole host OS!
  - e.g., full access to host files from within the Docker container

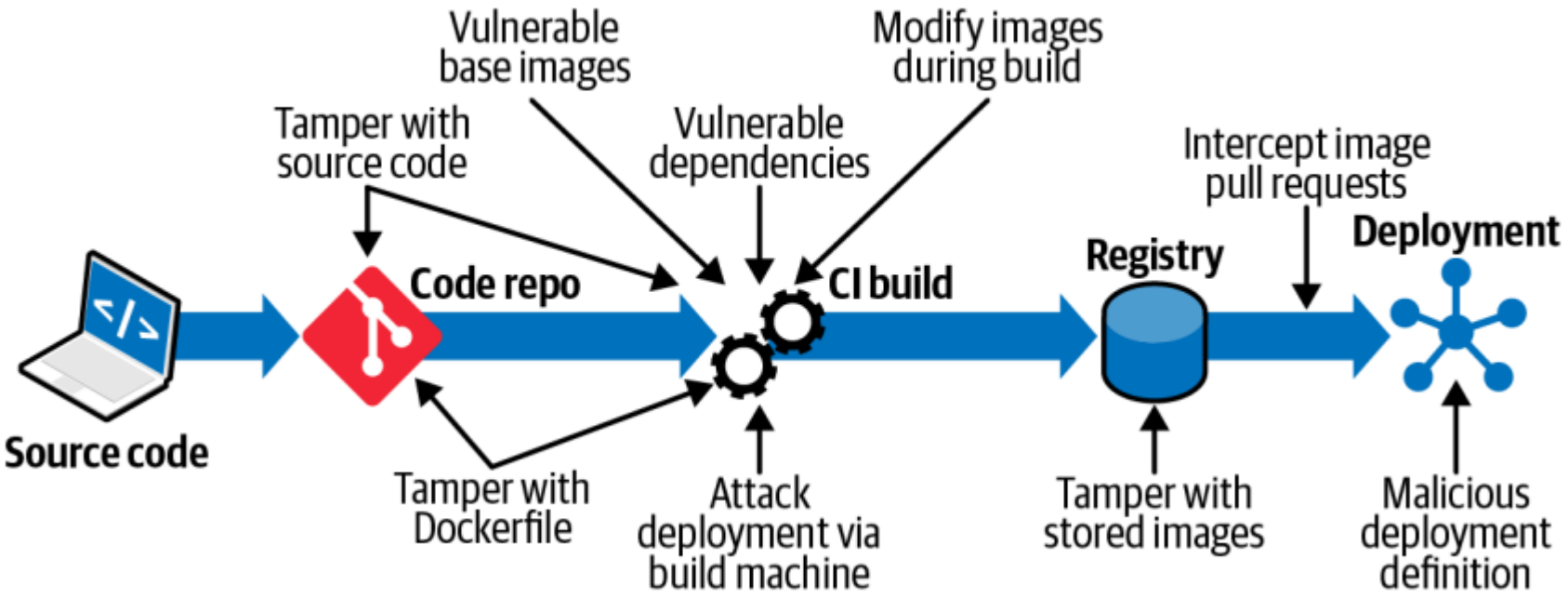
Docker containers are almighty

```
host$ cat ~root/.ssh/authorized_keys
cat: /root/.ssh/authorized_keys: Permission denied
host$ echo ssh-ed25519 AAAAC3NzaC1l.... |
>  docker run -it --rm --volume ~root/.ssh:/data ubuntu 'cat >> /data/authorized_keys'
host$
```

Docker security

- **Only trusted users should be allowed to control your Docker daemon**
  - Docker daemon listens for requests at a unix domain socket created at /var/run/docker.sock
  - it is possible to make the Docker Daemon listen on a network interface
- **Docker runs as privileged daemon**
  - Proper patch management is crucial

Image threat landscape



Container Security, by Liz Rice, https://www.oreilly.com/library/view/container-security/9781492056690/

Secure container images

- **Providing authentication and integrity verification**
  - Digital signatures vs. trusted repository
- **Image maintenance and development process**
  - Addressing vulnerabilities in the image
  - Controlled code development
- **Secure environment for building and releasing**
- **Compare to other SW distribution channels**

It’s not just Docker

- **Other technologies provide other ways for containers**
  - Podman
  - LXD
  - Systemd, firejail, …
  - Flatpak, Snap
  - Singularity
  - Containerd/runC
- **Regardless the technology it’s always namespaces + cgroups**
  - Some technologies add additional specifics

Rootless Containers

Rootless containers

- **Instantiating contained process without root privileges**
  - Enabled by *unprivileged* user namespaces
  - All steps outlined before are performed by the user who is starting the container
- **Podman**
  - Leading rootless technology
  - Drop-in replacement for Docker
    - Very good compatibility with CLI
    - `alias docker=podman`
  - No daemon or root-level access is needed though

Rootless containers

- **Useful for several scenarios**
  - Easy to deploy on user level
  - Usable for multi-user machines (HPC, grids)
- **Can only do what the user can do, which may be limiting**
  - Binding to privileged ports (e.g., 80, 443)
  - Full access to the host filesystem
  - Sharing data with other users

Cheat Sheets

start a new container
docker run IMAGE
docker run --rm IMAGE

start a new container in interactive mode (e.g. with a shell)
docker run -it IMAGE bash

start a new container from an image with a command
docker run IMAGE command

start a new container and map a local directory into the container
docker run -v HOSTDIR:TARGETDIR IMAGE

show a list of running containers

docker ps

show a list of all containers

docker ps -a

delete a container

docker rm CONTAINER

start a shell inside a running container

docker exec -it CONTAINER bash

stop a running container

docker stop CONTAINER

resume a stopped container

docker start CONTAINER

download an image from a repository

docker pull IMAGE

List local images

docker image ls

Delete a local image

docker image rm IMAGE