The Great Escape: Why Containers Won't Save You

(and How MicroVMs Will)

Part 1: Above Water Level

Laurent Bobelin

laurent.bobelin@insa-cvl.fr

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE-VAL DE LOIRE

Escaping Containers?

Container

- Lightweight, isolated runtime environment
- Packages an application with its dependencies
- Lower resources used compared to full VM

Usage:

- Microservices architectures
- Application deployment
- ...

Why Escaping from a Container?

Kernel shared with the host!

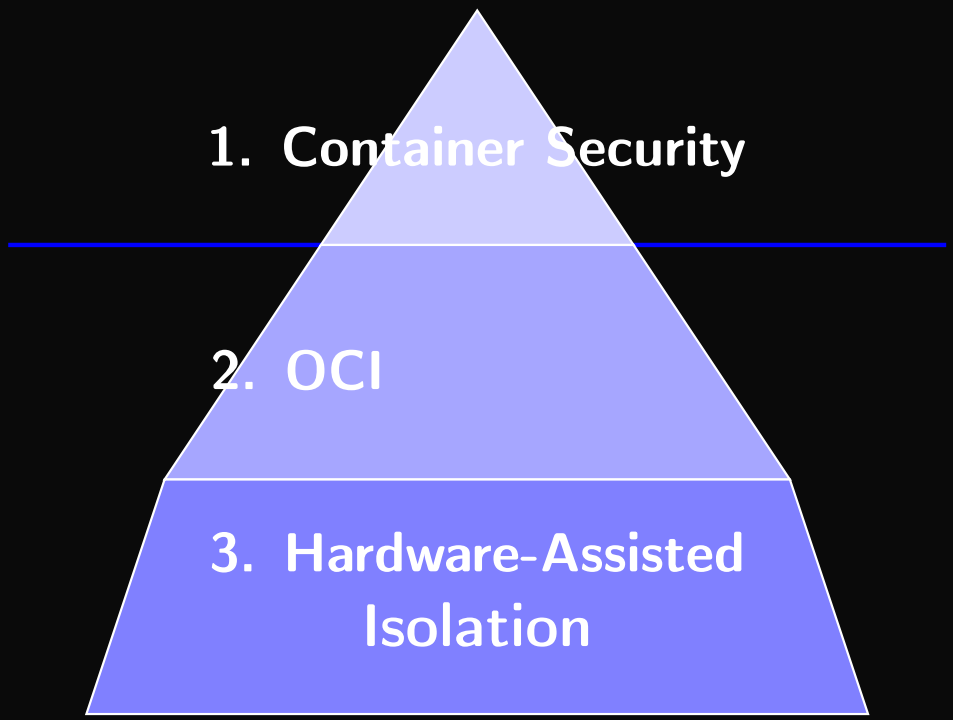
- Lateral movement
- Compromise other containers
- Use of host resources
- ...

Escaping Containers: How To

- **Misconfigurations**
  - **Privileged Mode:** Disables isolation; allows direct host device access.
  - **Excessive capabilities** granted to containers
- **Sensitive Mounts**
  - `/var/run/docker.sock` allows a container to control the host's Docker engine.
  - `/proc` or `/etc` from the host.
- **Kernel Exploits**
  - *Dirty Pipe*
  - *Dirty COW*
- **Runtime Vulnerabilities**
  - Some flaws in runtime (e.g., CVE-2019-5736) allow overwriting host binaries.

These Talks Objectives

1. Understand how (Docker) container runtime security enforcement works
2. Explore possible runtime environment enforcements
3. Mitigate escape risks (how?)



Iceberg of Container Runtime Isolation

These Talks Focus

Cybersecurity: a definition

- Protecting systems, networks, and programs from digital attacks
- Cyberattacks: accessing, changing, or destroying sensitive information; extorting money from users; interrupting normal business processes.

Ensuring a resilient posture relies on three core domains:

BUILD

Security by Design, vulnerability identification, and robust software and environment construction.

OPERATION

Active defense, intrusion detection, and real-time threat intelligence.

GOVERNANCE

Risk management, regulatory compliance, and crisis response strategies.

These Talks' Focus on a Cybersecurity Tasks Map

Building/configuring a secure runtime environment

These Talks Overview

- Part #1: Container Security Introduction:
  - Overview
  - Virtualization 101
  - Container security 1O1
  - Quiz 1
- Part #2: Going deeper into container runtime security
  - The real Docker architecture
  - OCI standard
  - Overview of a container runtime run
  - Quiz 2
- Break
- Part #3: Container Runtime Hardware-Assisted Isolation
  - Overview
  - Hardware-assisted isolation
  - OCI runtimes implementations
  - A focus on gVisor & Firecracker
  - Quiz 3

ToC

ToC

Virtualization Definition

Virtualization

- **Encapsulate** resources and processes
- **Abstract** resources, i.e. hardware details are hidden.
- **Consolidate**, i.e. many virtual units per physical unit.

COMPUTE

(Server) Decoupling the OS and applications from physical hardware.

STORAGE

(Data) Pooling physical storage devices into a single, manageable virtual unit (e.g., SAN/NAS).

NETWORK

(Connectivity) Software-Defined Networking (SDN) that abstracts hardware into virtual switches and firewalls.

Why Virtualization?

- **Resource Optimization**
  - Eliminate "Server Sprawl" by running multiple OS instances on one physical CPU.
  - Higher utilization rates (moving from 15% to 80%+ efficiency).
- **Flexibility & Agility**
  - **Abstraction:** Decouples software from physical hardware constraints.
  - **Rapid Provisioning:** Spin up a new drive or server in seconds, not weeks.
- **Isolation & Security**
  - **Segmentation:** Create VPNs, VPCs on shared physical infra.
  - **Recovery:** Easily snapshot, move, or replicate environments across data centers.

The Bottom Line: Cost & Control

Virtualization reduces costs of buying physical boxes) and energy (power, cooling, and manual management).

ToC

Types of Virtualization for Computation

- VM (1964):
  - complete instruction set
  - isolated resources (1 VM = 1 OS)
  - Hypervisor: VMWare, Virtualbox, Hyper-V
- Container (1979):
  - Reduced capabilities
  - isolated and limited resources
  - shared kernel between containers (not always)
  - ex: Docker
- Runtime Sandbox (1995)
  - Isolate execution by simulating (part of) the hardware and ensure constraints on resource use.
  - ex: JVM

Why Not Everything is a VM?

The use of VMs:

- Minimizes the costs of physical isolation
- Facilitates redeployment and resizing

However:

- Implementation complexity
- ... and maintenance complexity
- Overhead, long init process

Need to evolve:

- Need for rapid provisioning $\Rightarrow$ containers
- Need for a little bit more security $\Rightarrow$ sandboxes

ToC

ToC

Containers

Definition

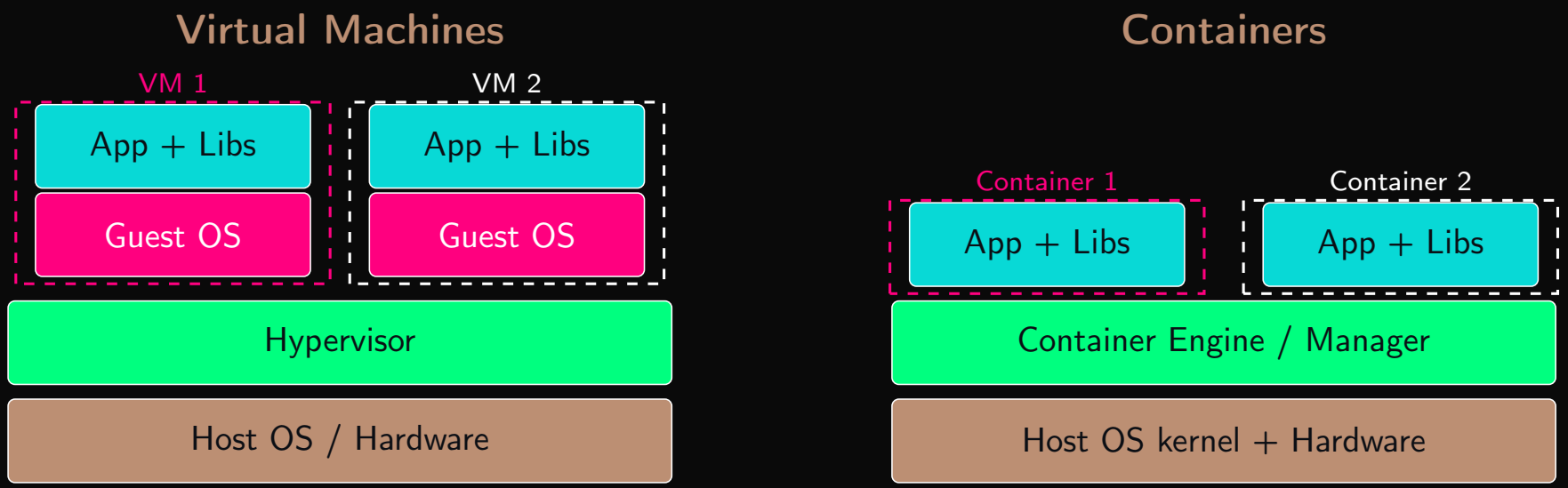
A Container is a set of isolated resources made available to a set of processes:

- System is shared → kernel common to all containers
- Mechanisms used → the ones present on the host system
- Analogy with physical containers holds for isolation

Differences with VM in terms of performance and hosting:

- Normal isolated process
- No "real" virtualization for computation ... unless micro-VMs are used.

VMs vs Containers



- VMs each have their own kernel ⇒ heavy
- Containers are isolated/managed by their container engine/manager ⇒ light

ToC

Tools for Isolation and Control (Linux)

A lot of existing mechanisms are used:

- Protection of privileged processes,
- Disk quotas,
- nice levels,
- ionice: peripheral access priority for a process
- cpu affinity: Forces a process to use defined cores
- ...

And of course LSM, namespaces, cgroups, capabilities, and seccomp

Linux Security Modules (LSM) and Docker

Linux Security Modules (LSM)

- Kernel framework allowing security modules to enforce **Mandatory Access Control (MAC)** policies.
- Security hooks in sensitive kernel operations (file access, process actions, networking).
- Enables pluggable security policies implemented by modules such as SELinux, AppArmor, Smack, TOMOYO, Landlock and integrity/hardening offered by IMA, EVM, Yama, LoadPin, ...

LSM usage in Docker

- **SELinux**: label-based access control isolating containers from the host and each other.
- **AppArmor**: profile-based restrictions limiting container capabilities and file access.

These policies complement other isolation mechanisms (namespaces, cgroups, capabilities, seccomp).

namespace

Namespaces - lightweight process virtualization

- Provides isolation: allows a process (or a few related processes) to have a view of the system different from others
- No hypervisor layer

There are 6 defined namespaces:

- mnt (mount points, filesystems): isolates file systems
- pid (processes): isolates processes
- net (network stack): isolates communications between machines
- ipc (System V IPC): isolates inter-process communication
- uts (hostname): isolates modifications made to the host or domain name
- user (UIDs): isolates user-related manipulations

Namespace: Internals

```
$ sudo ls -al /proc/4098/ns/
...
lrwxrwxrwx 1 root    root    0 oct.
13 00:03 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root    root    0 oct.
13 00:04 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root    root    0 oct.
13 00:03 net -> net:[4026532366]
lrwxrwxrwx 1 root    root    0 oct.
13 00:04 pid -> pid:[4026532516]
lrwxrwxrwx 1 root    root    0 oct.
13 00:03 user -> user:[4026532483]
lrwxrwxrwx 1 root    root    0 oct.
13 00:04 uts -> uts:[4026531838]
```

The inodes of these files are in fact symbolic links to the data structures representing the namespaces.

Namespaces

- Isolates objects created in a namespace from elements of the parent namespace
- When all processes in a namespace are terminated, the remaining objects are destroyed or returned to the parent namespace.
- Added as FLAGS to the clone system call: it then creates one or more namespaces for the child of the current process
- flag: CLONE_NEWNET, CLONE_NEWNS, CLONE_NEWPID, CLONE_NEWUTS, CLONE_NEWIPC, CLONE_NEWUSER.

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *
, /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ ) ;
```

Linux Control Groups (cgroups)

Allow kernel to:

- limit resource usage
- account resource consumption
- isolate groups of processes

Key concepts:

- Processes belong to **cgroups**
- cgroups form a **tree**
- Limits are enforced by **controllers**
- Controllers examples: CPU, Memory, Block I/O, Network

```
root
|-- system.slice
|   |-- process A
|   `-- process B
`-- container.slice
    |-- docker/ctr1
    |   `-- proc
    `-- docker/ctr2
        `-- proc
```

Example of hierarchy

Capabilities

Traditionally, Unix-like systems evaluate permissions according to 2 categories:

- privileged processes (whose UID is 0)
- non-privileged processes (whose UID is greater than 0).

Since version 2.2, Linux includes capabilities:

- Each capability corresponds to a subset of the rights usually given to privileged processes
- Each capability is allocated or not to a thread.
- they allow finer management of permissions

Capabilities

There are about forty capabilities:

- CAP_CHOWN for example allows changing the UIDs and GIDs of files.
- CAP_NET_ADMIN for configuring the network
- CAP_SYS_CHROOT for using chroot.
- ...

3 sets of capabilities are associated with each thread:

- Permitted: automatically permitted
- Effective: the set used by the kernel to check the thread's rights (active).
- Inheritable: which capabilities can be passed on during an execve.

seccomp

seccomp == Secure Computing Mode
== Linux kernel restrictions on **syscalls** of a process.

How it works

- A process loads a **seccomp filter**.
- The filter defines which syscalls are allowed or denied (cause the process to terminate)
- Modern implementations rely on eBPF.

Use in containers

- Container runtimes (e.g., Docker) apply default seccomp profiles.
- Dangerous syscalls (e.g., `ptrace`, `kexec_load`, `mount`) are blocked.

ToC

Docker in the Container Ecosystem

Docker

- Open platform for building, shipping and running containers
- Packages an application with its dependencies
- Ensures consistent execution across environments

Docker vs other Container technologies:

- **Docker** — most widely used container platform
- **Podman** — daemonless container engine
- **CRI-O** — Kubernetes-focused runtime
- **LXC/LXD** — system containers (OS-level virtualization)

Docker Typical stack

- Linux kernel features: **namespaces** + **cgroups** + ...
- High level runtime **containerd**
- Low-level OCI runtime **runc**

Docker vs Kubernetes

Different roles

- **Docker** manages containers on a single host
- **Docker** manage workload on a few hosts
- **Kubernetes** orchestrates containers across clusters

Relationship

- Kubernetes uses container runtimes
- Docker images are commonly used as workloads

Comparison Docker vs Kubernetes

Aspect	Docker	Kubernetes
Scope	Single host	Cluster orchestration
Purpose	Build/run containers	Manage distributed apps
Scaling	Manual / Docker Compose	Automatic scaling
Networking	Simple container networks	Advanced service networking
Deployment	Containers	Pods, Deployments, Services

Main Features of Docker

Containerization

- Lightweight virtualization
- Fast startup compared to VMs

Image-based deployment

- Immutable images
- Layered filesystem
- Reproducible environments

Isolation

- Process isolation (namespaces)
- Resource control (cgroups)

Developer productivity

- Dockerfile for building images
- Docker Compose for multi-container apps
- Large ecosystem and registries

Common Docker Commands

Image management

```
docker pull image
docker build -t myimage .
docker images
docker rmi image
```

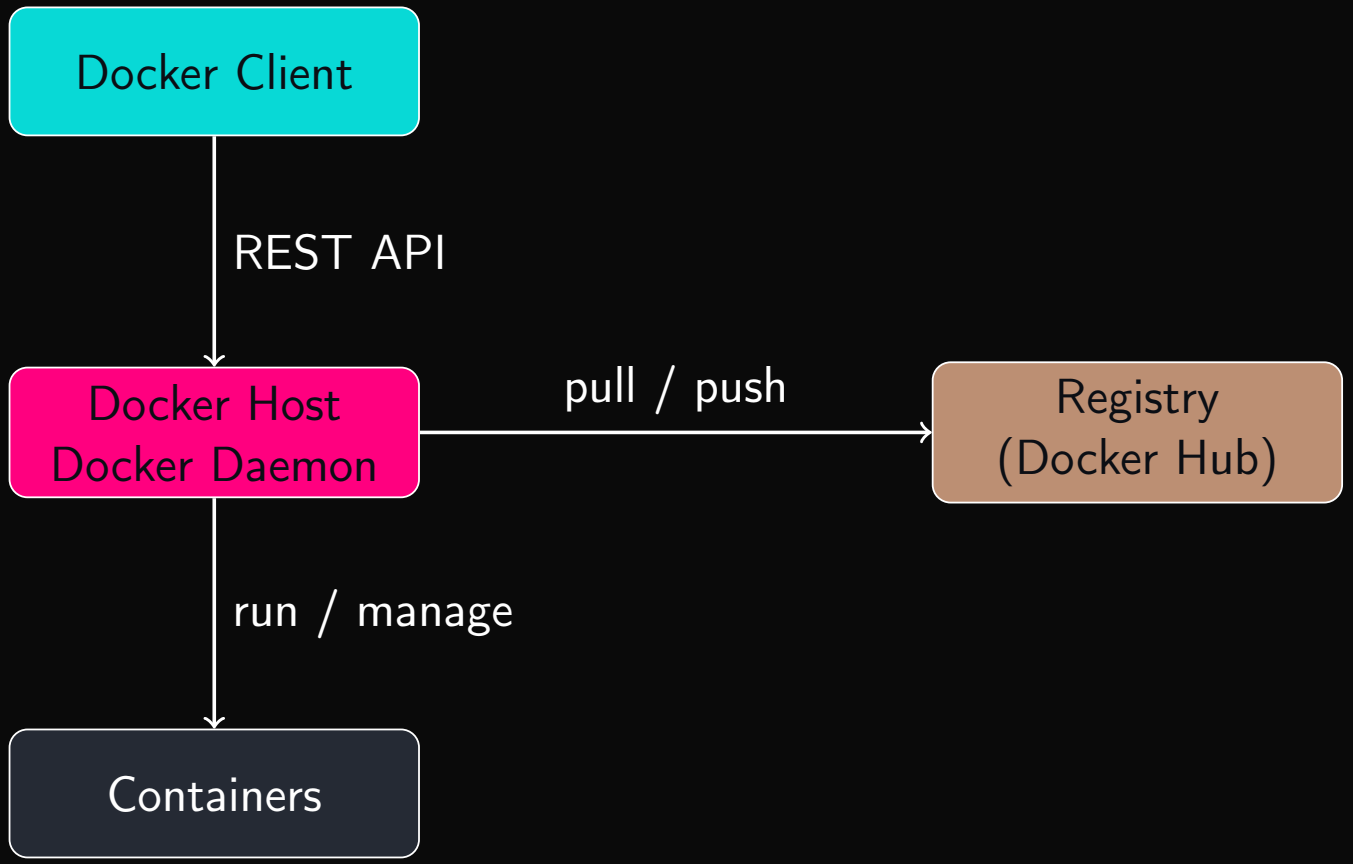
Container management

```
docker run image
docker ps
docker stop container
docker rm container
```

Inspection

```
docker logs container
docker exec -it container bash
docker inspect container
```

Docker Architecture



Components:

- **Docker Client**: command-line interface used by the user
- **Docker Daemon**: builds images and manages containers
- **Registry**: stores and distributes container images

Docker Image vs Container

Docker Image

- Read-only template
- Contains application code, libraries and metadata
- Built once, reused many times
- Versioned and shareable through registries

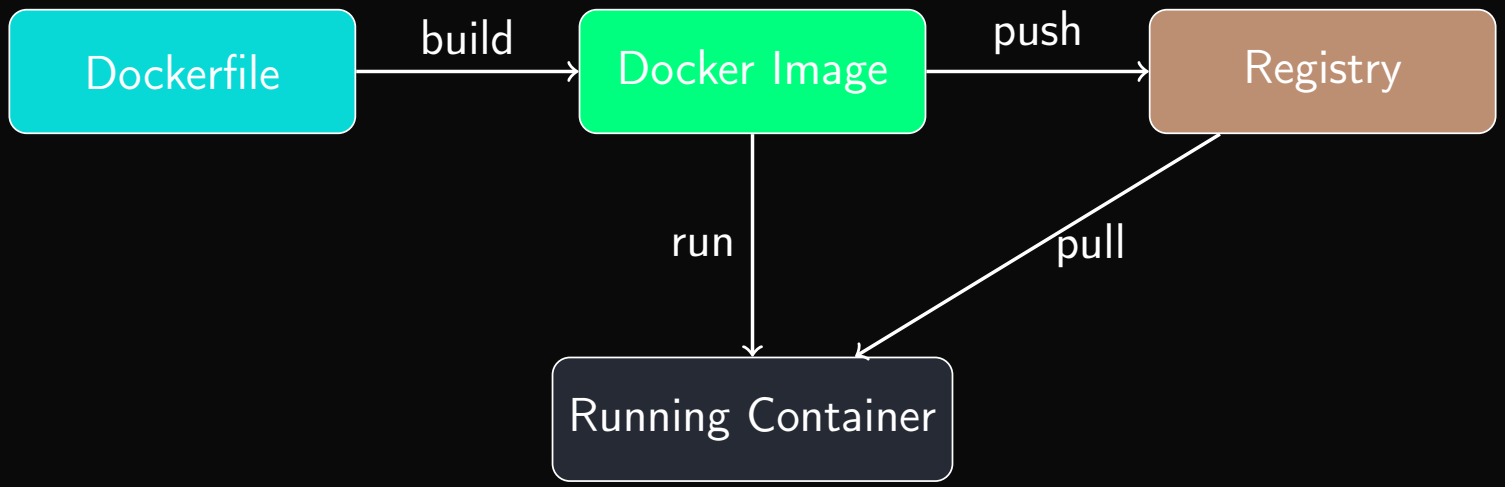
Docker Container

- Instance of an image
- Adds a writable layer on top of the image
- Has its own process space, network and filesystem view
- Can be started, stopped, restarted and removed

Examples:

- Images: `ubuntu:24.04`, `nginx:latest`, `python:3.12`
- Container: image + name + port mapping + state + ...

Docker Workflow



Typical sequence

- Write a **Dockerfile**
- Build an **image**
- Store or distribute it through a **registry**
- Run one or more **containers** from the image

ToC

Conclusion for this first Part

Containers are not a security boundary:

- Containers share the host kernel
- Misconfigurations increase attack surface
- Kernel and runtime vulnerabilities enable container escapes

There are some common escape vectors you can handle:

- Privileged containers
- Dangerous mounts (docker.sock, /proc)
- Excessive capabilities

Some you may watch:

- Kernel exploits
- Runtime vulnerabilities

Some you may prevent by:

- Reduce privileges and capabilities
- Use SELinux/AppArmor to refine policies
- Harden container runtimes