Containers and Orchestration: a Security Perspective

Mathieu Bacou
mathieu.bacou@telecom-sudparis.eu

2022 — 2023

NET5039 — SYSTEMS, VIRTUALIZATION AND SECURITY

Cloud applications

- Traditional applications are monolithic
  - Everything tightly coupled
  - On full servers, managed from OS to deployment
    - This is a constraint
- In the cloud, you don’t manage real servers
  - Shared servers with virtualization
  - Get new resources (“server”) on-the-fly
- Let’s go further!

Cloud native applications

- No OS management by the user
- Component-level application scalability

Introducing: containers

- Cloud users don’t want to run OSes
  - They want to run their applications
- How to share cloud resources closer to the applications?
  - Virtualization layer just between the OS and the application
- Virtualize the OS for multiple applications at the same time!
  - In other words, containers are OS-level virtualization
- An OS executes a container engine that runs containers
  - Docker, LXC, OpenVZ...

Actors of OS-level virtualization

I. Container engine

II. Container

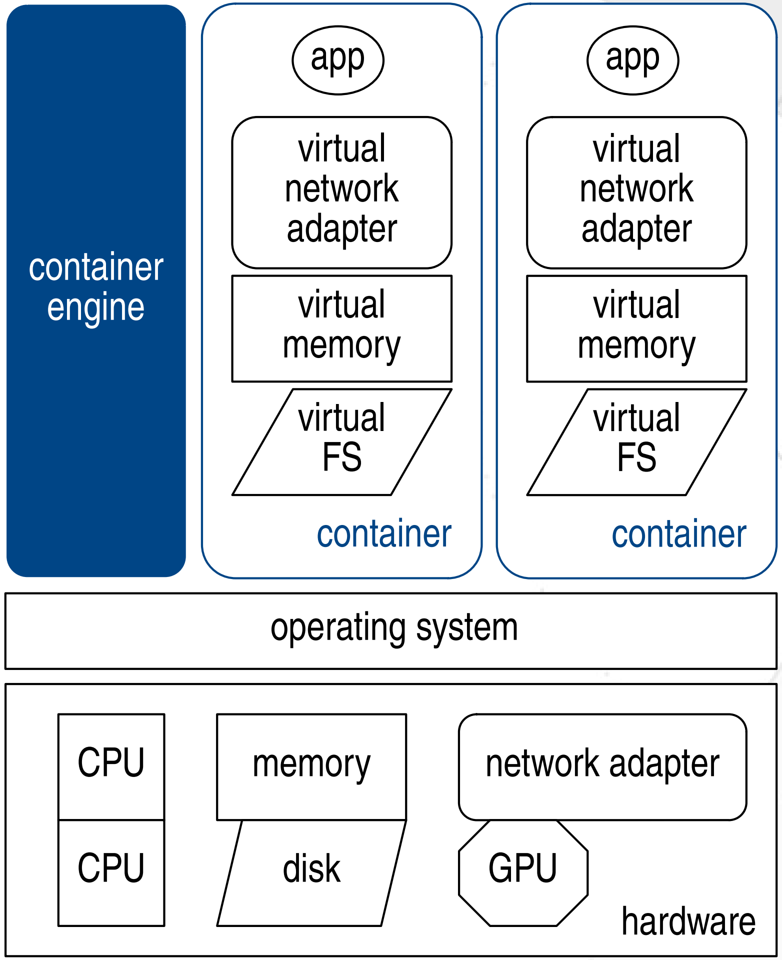
Containers and engines

- Engine:
  - Manage container lifecycle: create container from image, start and stop containers...
  - Handle out-of-container tasks: virtual networking…
  - Many engines for many uses: generic, HPC, scientific…
    - With interchangeable underlying container engine cores
- A container image packages an application and its runtime
  - Business core, dependencies, semi-static configuration
  - Registries of reusable images (DockerHub, local…)
    - Typically written in a portable, constant manner

Containers

- Container: isolated and limited virtual copy of the host OS
  - Deploys the image to “fill” the virtual copy
- Isolation: users, devices, processes…
  - Virtual filesystem: built from container image
- Limits: CPU, memory, I/O…
  - Also monitoring

Components



Stack for OS-level virtualization

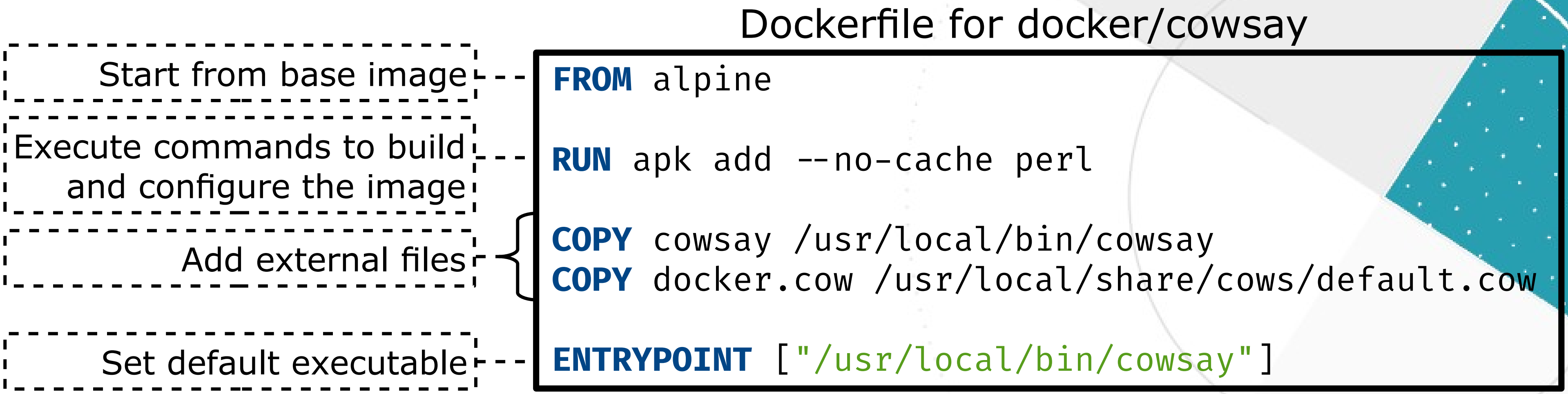
Demo: Docker

- Creation and usage of a Docker container:
  - Run an interactive image
  - Pull and run a daemon service
  - List images, monitor containers
- Docker is a bit low-level for applications: docker-compose for multi-component apps

Build containers: two ways

1) Interactively
   - From a base distribution image (Ubuntu, Alpine…)
   - Use package manager
   - `docker commit` to tag the current state of the container as an image
   - Testing and experimenting
2) Dockerfile
   - DSL to describe how to install and configure app
   - Proper method: clean, reusable, reproducible

Build containers: Dockerfile



Dockerfile for docker/cowsay

Start from base image

Execute commands to build and configure the image

Add external files

Set default executable

```
FROM alpine

RUN apk add --no-cache perl

COPY cowsay /usr/local/bin/cowsay
COPY docker.cow /usr/local/share/cows/default.cow

ENTRYPOINT ["/usr/local/bin/cowsay"]
```

- And then: `docker build -t namespace/name:tag .`
- Can start from empty image: `FROM scratch`
  - Used by distribution base images: build from archive
- Also declare users, volumes, network ports

Internals of Docker

I. Isolation

II. Limit

III. Operation control

IV. Isolation of the virtual filesystem

Limits by cgroups

docker

Isolation by namespaces

Operation control by capabilities

Isolation: namespaces

- Provide an isolated view of the OS
  - `chroot` on steroids (CHange ROOT of a process)
- 8 dimensions:

1) `mnt`: mount points
   - I.e. filesystem
2) `pid`: PID hierarchy
   - First process in the container is PID 1
3) `net`: network facilities
   - Interfaces, ports, protocol stack…
4) `ipc`: interprocess communication
   - Semaphore, message queue, shared mem
5) `user`: users, groups and privileges
   - Mappings of UIDs/GIDs between host and container
     - UID 0 is root, available in container: if you escape the container, you are root!
6) `uts`: hostname
   - Stands for "UNIX TimeSharing", or said otherwise: multi-user in UNIX
7) `time`: clock
8) `cgroup`: control groups (next slide)

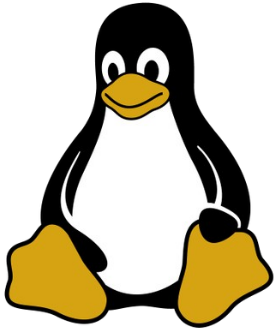
Limit: control groups (cgroups)

- Constrain resource usage
  - Also monitoring facilities
- 12 dimensions:

1) `cpu`: CPU time
2) `cpuacct`: CPU accounting
3) `cpuset`: CPU pinning
4) `memory`: memory and swap
5) `devices`: access rights to devices
6) `freeze`: freeze, suspend processes
7) `net_cls`: network packets classes
8) `net_prio`: network packets priority
9) `blkio`: block devices (disk) I/O
10) `perf_event`: performance mon.
11) `hugetlb`: huge pages usage
12) `pids`: number of processes

Demo: namespaces & cgroups

- Spawn new process in namespaces
- Put process in control groups
  - Set limit and monitor resource usage
- Demonstrated filesystem interface
  - Also a programmatic interface with syscalls



Operation control: caps and MAC

- Capabilities: selectively drop root privileges
  - Remove privileges from a “root” container
- Mandatory Access Control (MAC): system-level operational policies
  - Linux Security Modules (LSM): SELinux, AppArmor...
- 40 capabilities (CAP_XXX):

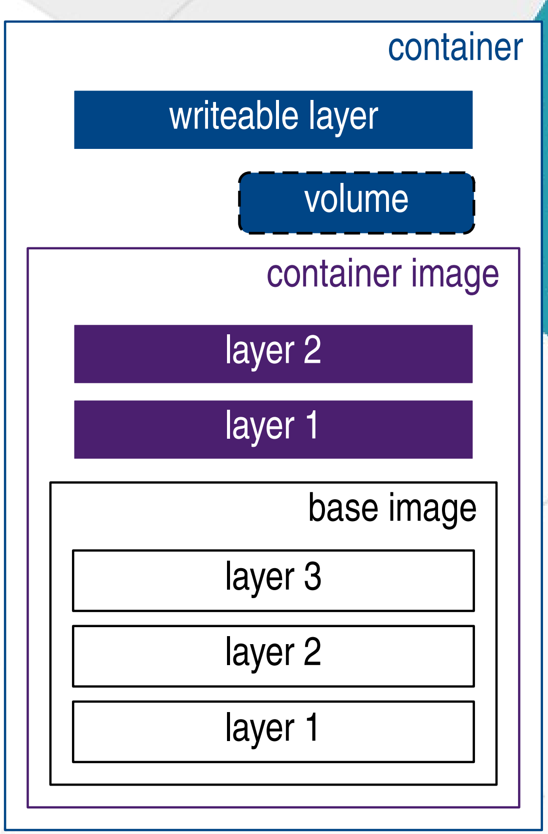
1) `CHOWN`: change owner
2) `SETGID/SETUID`: change process GIDs/UIDs
3) `KILL`: send signals
4) `NET_ADMIN`: network admin
5) `NET_RAW`: use RAW sockets
6) `SYS_ADMIN`: system admin (mount…)
7) `SYS_CHROOT`: change root path of process
8) `SYS_MODULE`: (un)load kernel modules
9) `SYS_NICE`: change process niceness
10) `SYS_TIME`: change system clock

Virtual filesystem

- Isolated filesystem: `mnt` namespace
  - Also with `chroot`
- Two parts:
  - Container image: basis for virtual filesystem
    - Docker specifics, see next
  - Volumes: external data storage
    - Mounted into the virtual FS of the container

Container image with Docker

- An image has layers
  - Like git commits
  - Reusable by other images, caching
  - `docker image history IMAGE_NAME`
- Layers from Dockerfile are read-only
  - For execution, add a writeable layer
  - Use copy-on-write to modify files from lower layers
- Union file system: virtual FS driver for layers
  - Many drivers: AUFS, OverlayFS, devicemapper...



Container image layers and volumes

Union FS and copy-on-write

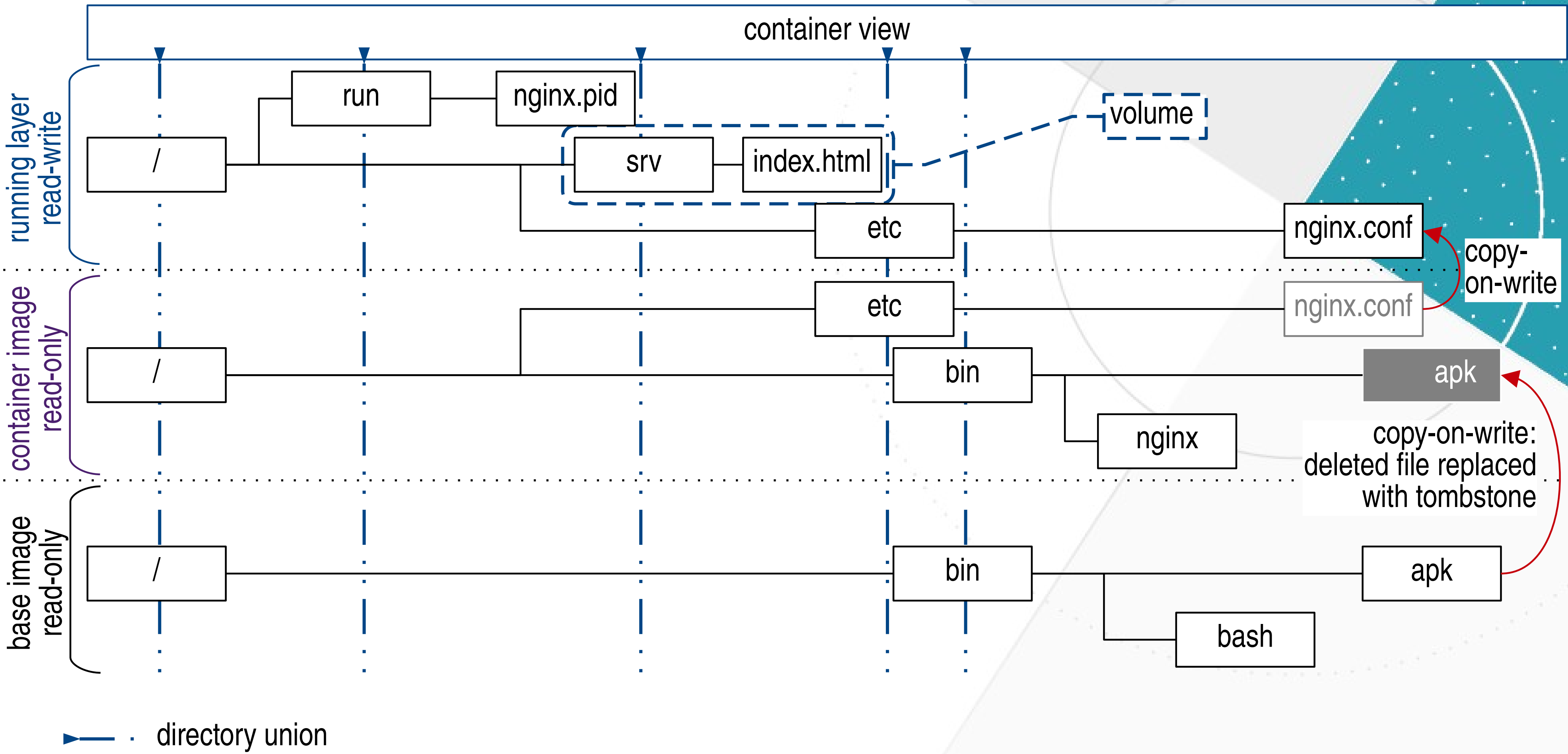
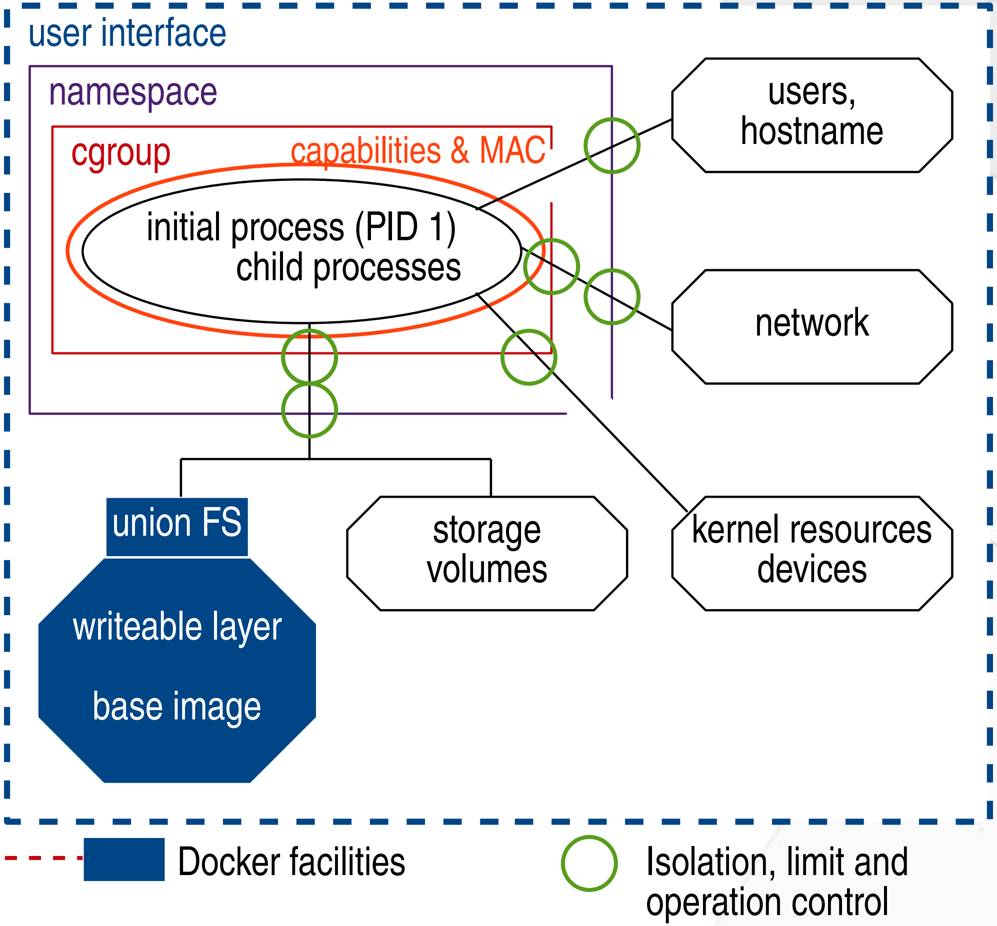


Illustration of union filesystem and copy-on-write

Docker container engine



Everything else
are features from
Linux kernel!

Low-level facilities of Docker container engine

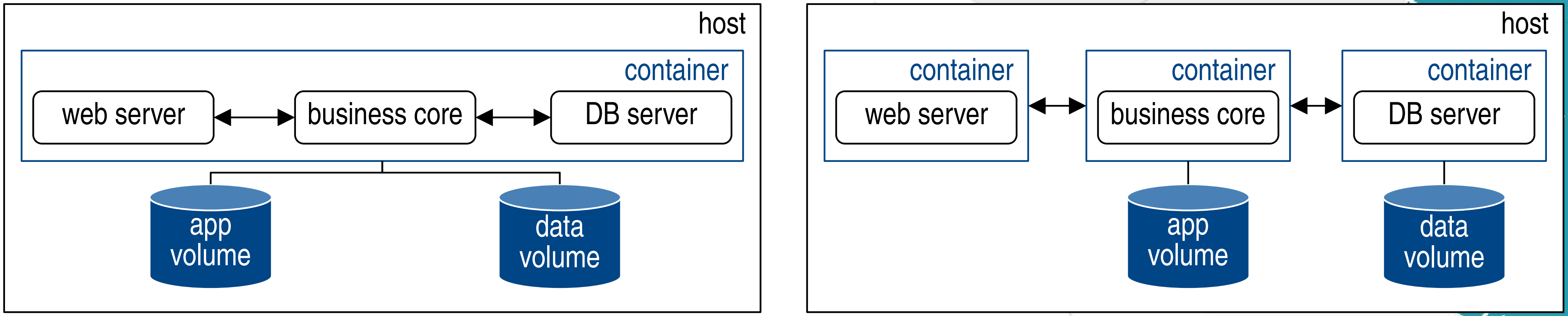
Containers for the cloud

I. Application architecture in the cloud
II. Micro-services
III. Orchestration

Cloud application architecture

- Historic pattern: monolithic application
  - All components are ad-hoc, tightly coupled
- Unfit for the cloud
  - Must manage all components at once for scalability, deployment, service quality
  - Hard to reconfigure
- New paradigm enabled by container: micro-services

Micro-services



✗ Monolithic container

✓ Composition of containers: micro-services

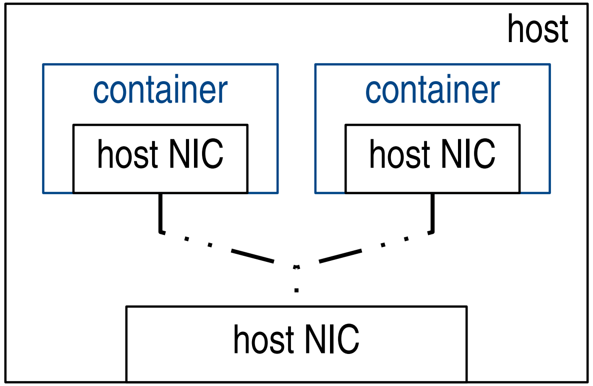
- Components as processes
  - Manual interfacing
    - Need in-container PID 1 to run multiple processes
  - Cons of monolithic apps

- Components as containers
  - Max reuse of images
  - High flexibility, easy configuration
  - Fine-grained scalability

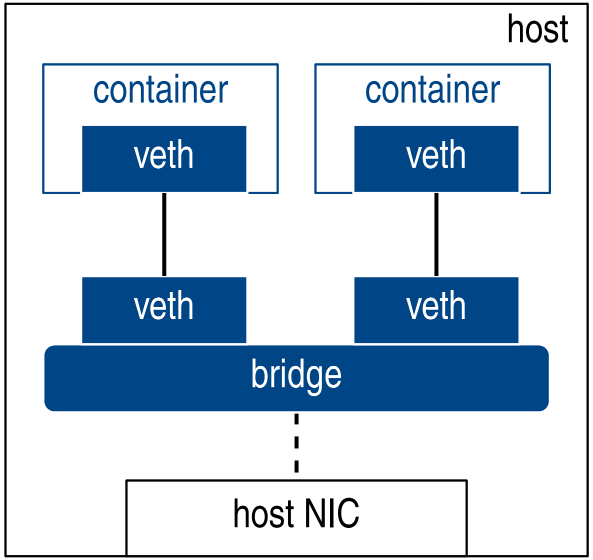
Network for micro-services

- Configuration of network by Docker
  - Dedicated links between component containers
  - Controlled link to the Internet
- Network drivers:
  - Host: expose host network devices to the container
  - Bridge: local virtual network
    - Can be exposed to the Internet
  - Overlay: inter-host inter-container network
  - None: no networking

Network for micro-services

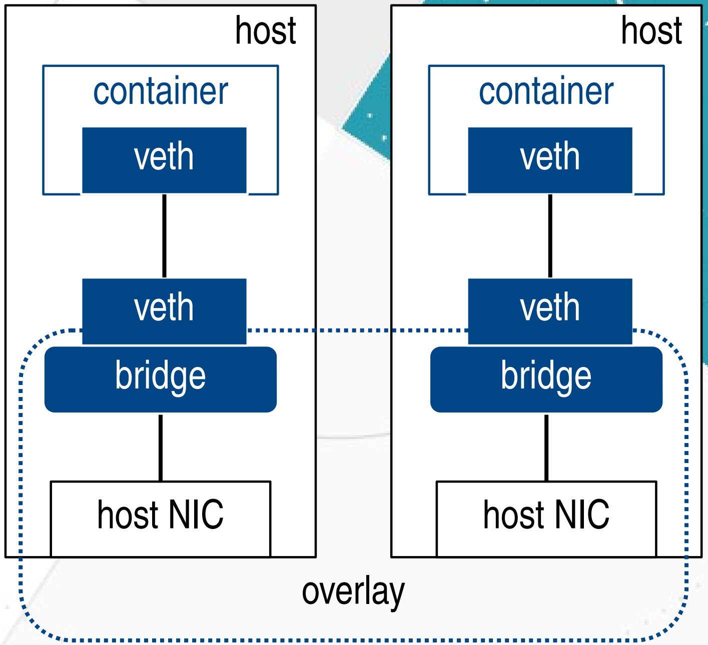


Host



veth: Virtual ETHernet

Bridge



Overlay

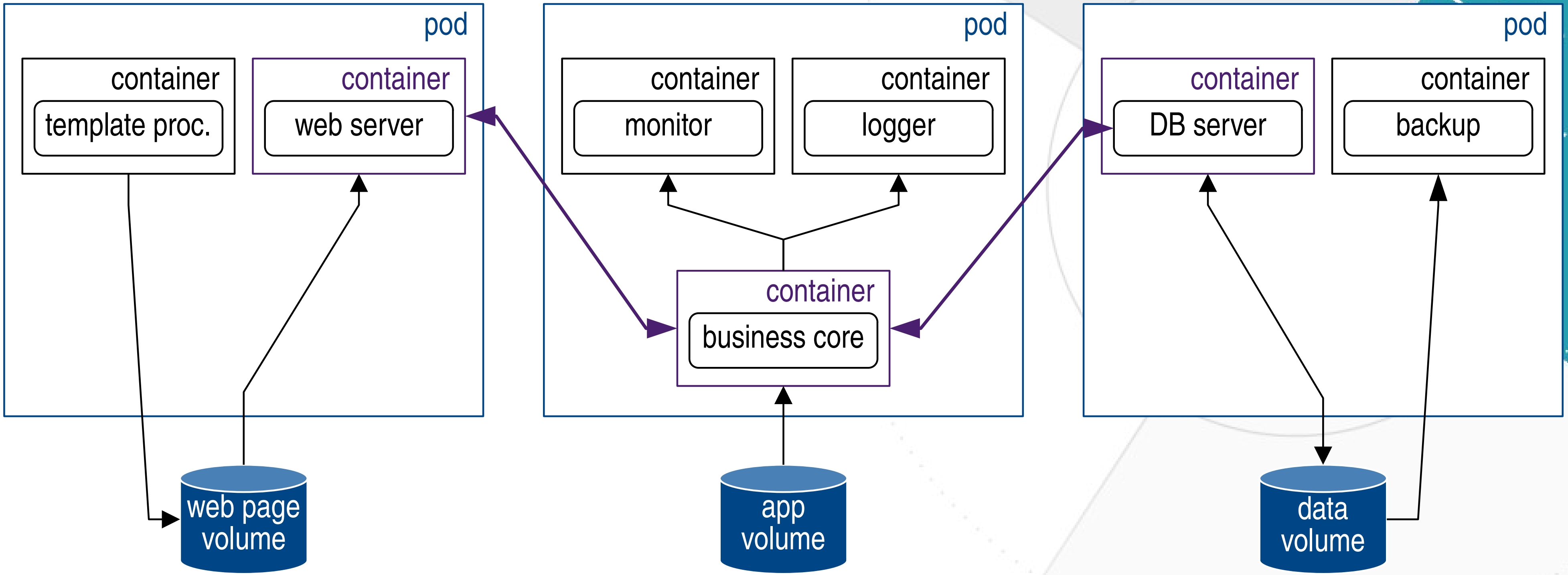
Orchestration

- Composition: build application as micro-services
  - Example: docker-compose
- Orchestration: manage micro-services
  - Distribution
  - Replication
  - Load-balancing
  - Availability
  - Higher-level interfaces to composition features
    - Acts as the user front-end
  - Examples: Kubernetes, Docker Swarm
- Abstraction of management unit: the pod

Orchestration: scheduling

- Manual criteria: filters
  - Handle host heterogeneity
    - Settings of Docker engine, host OS…
  - Container affinity: force placement for resource access
    - Image availability, volume placement, other container...
- Strategies for deployment on physical hosts
  - Spread: balance load over hosts
  - Binpack: colocate as much as possible
- Handle colocation of tightly-coupled containers: pods
  - Containers in a pod share the same network namespace and same volumes
  - Pod = service container + helper containers (logging, interfacing...)

Orchestration of pods



Application architecture with pods

Demo: Kubernetes

- Create and use a pod
- Create and use a deployment
  - Scalability
  - Roll-out

Kubernetes app.: deployment.yml

Scalability: set number of replicas

Pod composition (containers)

```yaml
kind: Deployment
# [ ... ]
spec:
  replicas: 3
  selector:
    matchLabels:
      app: simpleserver
  template:
    metadata:
      labels:
        app: simpleserver
    spec:
      containers:
      - name: pythonserver
        image: python:simpleserver
        resources:
          requests:
            cpu: 0.5
        ports:
          - containerPort: 8080
```

Security of containers

I. Isolation
II. Threat models
III. Good practices

Isolation

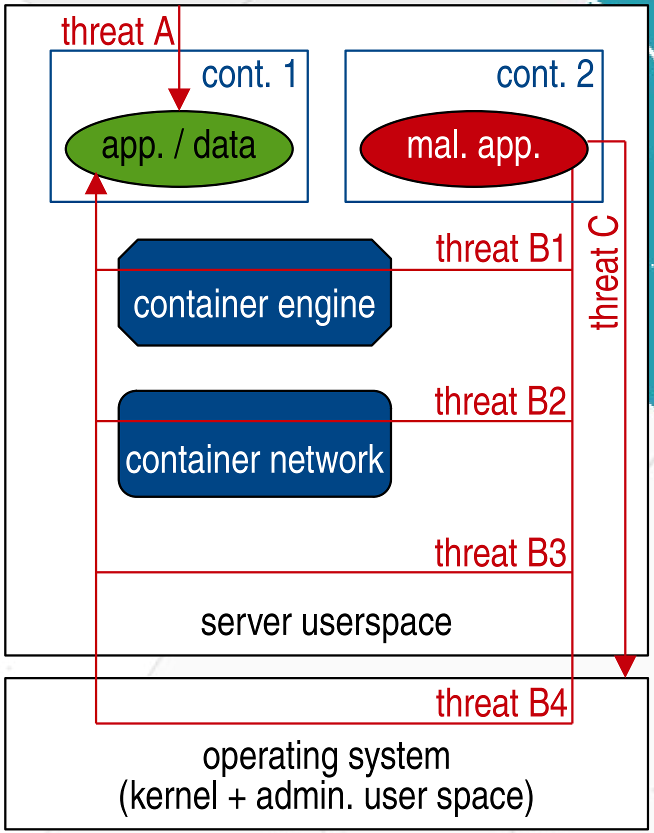
- Fundamental issue in the cloud: execute untrusted code
- With containers: on the same shared kernel
  - No mitigation when the kernel is compromised
  - Incompatibility of kernel-level security policies
    - No security namespacing
  - Vast attack surface and trusted code base
    - Virtual Machines (VMs) are better in this regard (but less flexible)
- Isolation of untrusted code
  - To protect containers from each other
  - To protect the system from containers

Threats

- Attack goals:
  - Disrupt service: bad neighbor, denial of service
  - Subvert service: usurp identity, steal resources
  - Steal data
- Cloud-oriented: applications are regular services
  - Containers are also good for system services
    - SSH, cron jobs, logs…
  - More privileged requirements = more care!

Threat models

- Threat models: attack…
  - A. from outside, on the containerized application
  - B. from a container, on another container
  - C. from a container, on the system



Threat models of containers

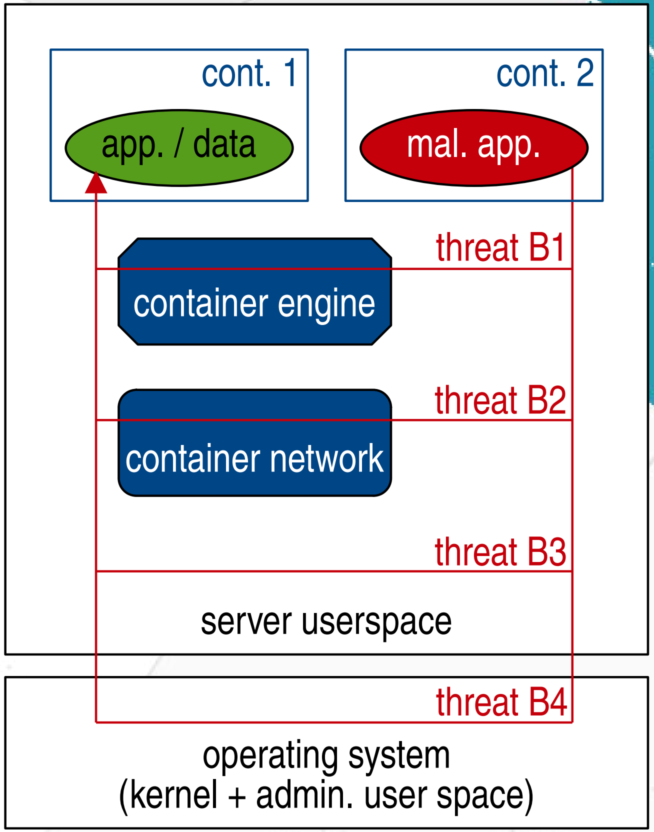
Threats from outside (A)

- For Internet-facing applications: containers are not magic
  - If your web server is vulnerable, it remains vulnerable
- However containers help:
  - Breach containment
    - Micro-service model
  - Easier to achieve secure configuration
    - More secure defaults, less knobs to tweak
  - Fast, easy distribution of security updates
    - Generic images from a centralized place
  - Simpler audit
    - Limited set of dependencies and software pieces

But this also works the other way around with vulnerable or compromised images (malicious updates or owner, typosquatting…) Use private repository of audited images!

Threats between containers (B)

- Containers run arbitrary code by definition
- B1: Leak to another container
  - Namespace bug
  - Filesystem leak
- B2: Abuse container network
  - Packet forging
  - Layer 2 attack
- B3: Escalate to root
  - Vulnerable SUID binaries
- B4: Execute arbitrary kernel code
  - Exploitable syscalls



Container-to-container threat models

Threats to the system (C)

- Containers run arbitrary code by definition
- Escape containment
  - Namespace bug
  - Filesystem leak
- Escalate to root
  - Vulnerable SUID binaries
- Execute arbitrary kernel code
  - Exploitable syscalls

Good practices (1/2)

- As a Docker user:
  - Audit public images
    - Fix versions but stay aware of security updates
  - Use micro-services model (pods) for intrusion detection and containment
    - Each micro-service can be “equipped” of its monitor
  - Mount read-only as much as possible
    - Images are already immutable with overlay FS
  - Drop capabilities
    - Docker drops many by default
- As a Docker image developer:
  - Use and build immutable container images
    - All deployments execute the same code eventually
  - Don’t run as root
    - Even with user namespaces
    - Eliminate SUID binaries for normal use

Good practices (2/2)

- As a Docker system administrator:
  - Harden the kernel
    - Enable MAC: SELinux / AppArmor; also seccomp
    - Use hardened kernel (GRSEC…)
    - Update kernel to vetted versions
      - It is shared by all containers: critical part
  - Configure container network tightly
    - Do not use host mode
    - Think about shared network namespaces, open ports…
      - In practice: abstracted by docker-compose, or Kubernetes, etc.
  - Go one step further: use virtual machines!
    - An application in a Docker container in a virtual machine in a container
    - Kata Containers, unikernels...

OS-level virtualization

- Virtualize the OS
  - Containers: lighter, faster, simpler
- Based on Linux kernel: namespaces, cgroups
  - Container engines bring usability and networking
- Enable new cloud-native application architecture: micro-services
  - Managed with orchestrators
- Security challenge:
  - Major cloud challenge: execution of arbitrary code
  - Specifically: vast attack surface, enables dangerous behaviors