Containerizing Linux Applications

ERIK HEMDAL

About me. . .

Adjunct Instructor / Software Engineering / Brandeis University

Graduate Professional Studies Division in the Rabb School

During the day. . .

Senior Support Specialist for InterSystems Corporation in Cambridge

- InterSystems IRIS Data Platform
- Cache‘ database system
- Ensemble integration engine
- HealthShare unified health-record software

Outline

Docker Basics

Docker “Mental Model”

DevOps concepts

Levels of security

- Infrastructure
- Build-time
- Runtime

VMs vs. Containers

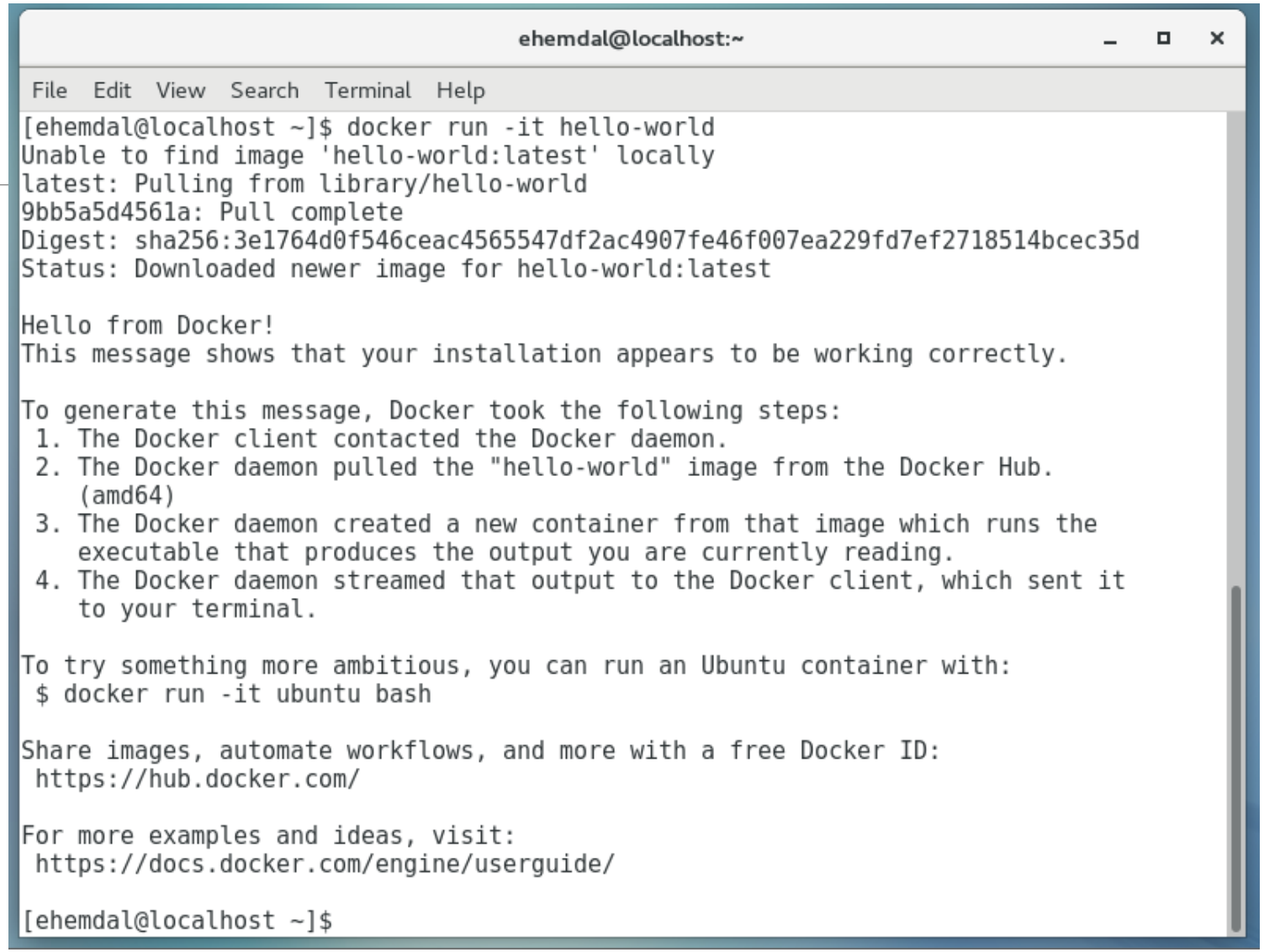
Virtual machines (VMs) emulate server hardware, on which you can install a “guest” operating system and your applications.

Containers virtualize the OS and allow you to run many images using a common OS.

Container IMAGES package individual applications and their dependencies into portable artifacts that can be run on any compatible operating system.

Hello World!

This shows that Docker is running and describes its data flow -→



“Mental Model”: Container vs. VM

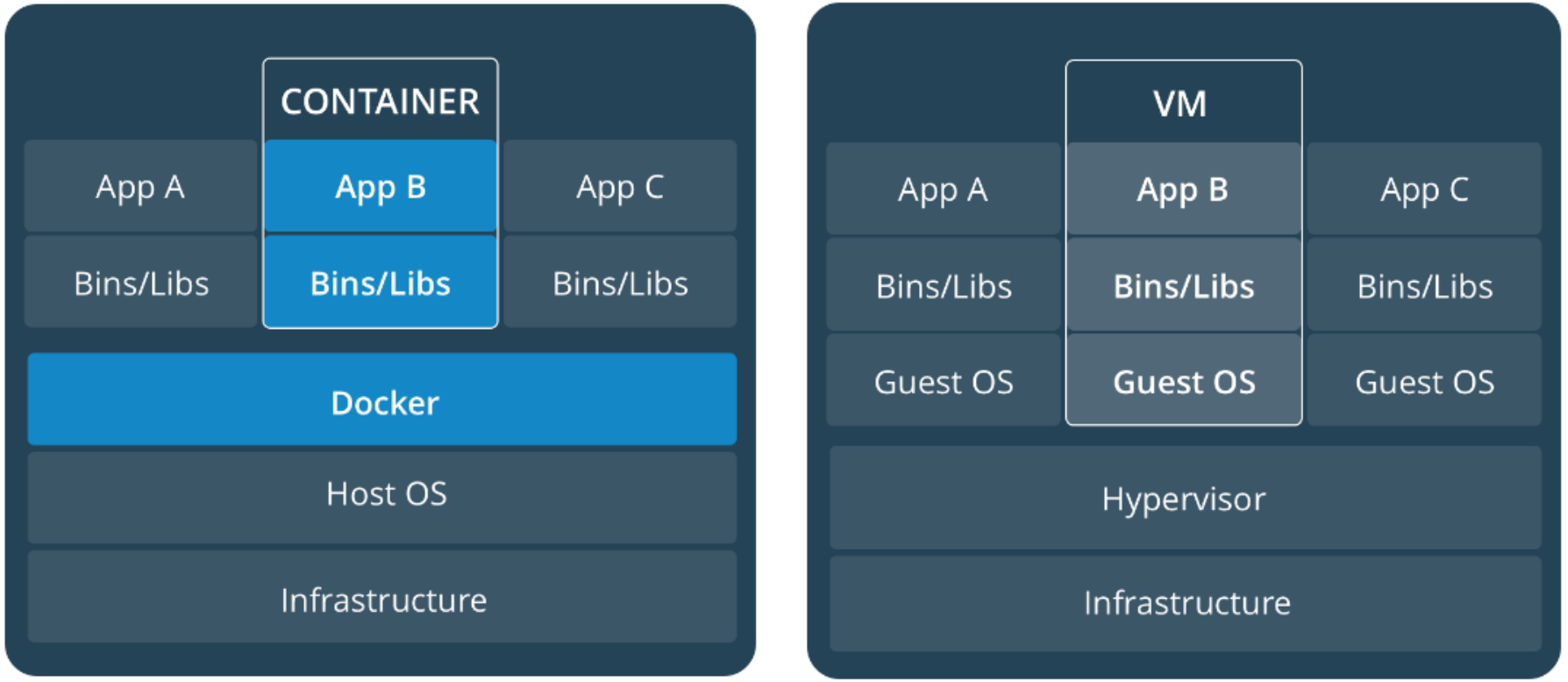
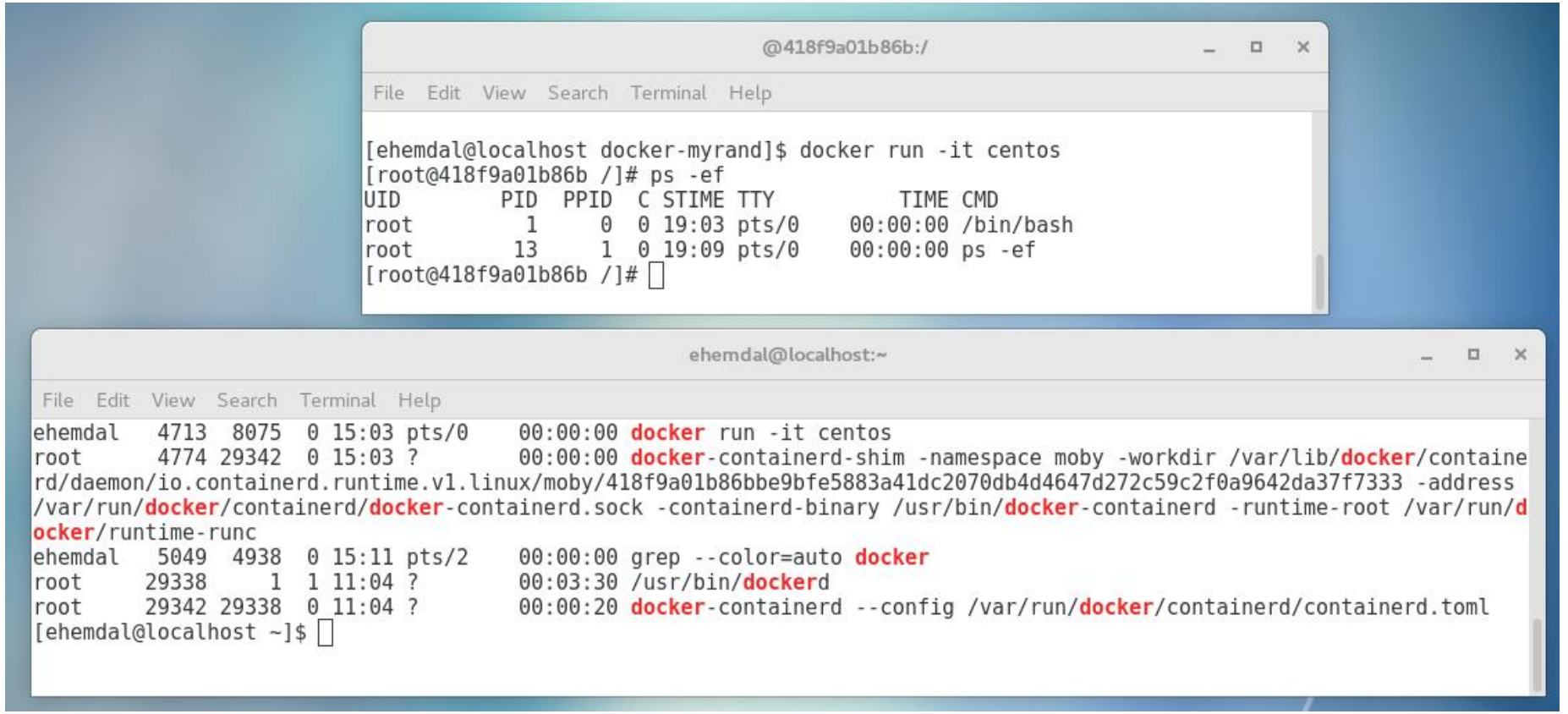


Image from https://docs.docker.com/get-started/#prepare-your-docker-environment

Docker container from OS level



OS:

PID 1: init (systemd) daemon

PID 29338: Docker daemon

PID 29342: Containerd daemon (runtime)

PID 4774: Container shim process

PID 4713: Docker client (my shell)

Container:

PID 1: ENTRYPOINT shell

PID 13: My command

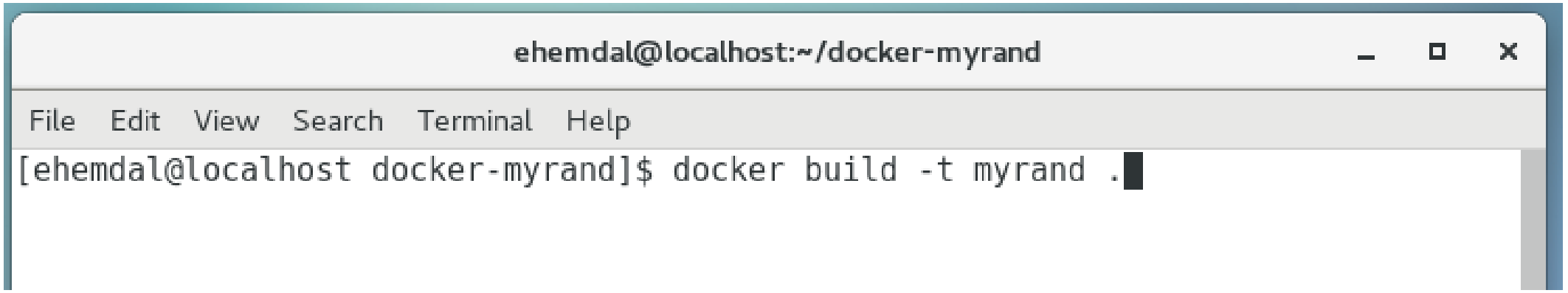
Basics (1 of 3): Simple Dockerfile

ehemdal@localhost:~/docker-myrand

File Edit View Search Terminal Help

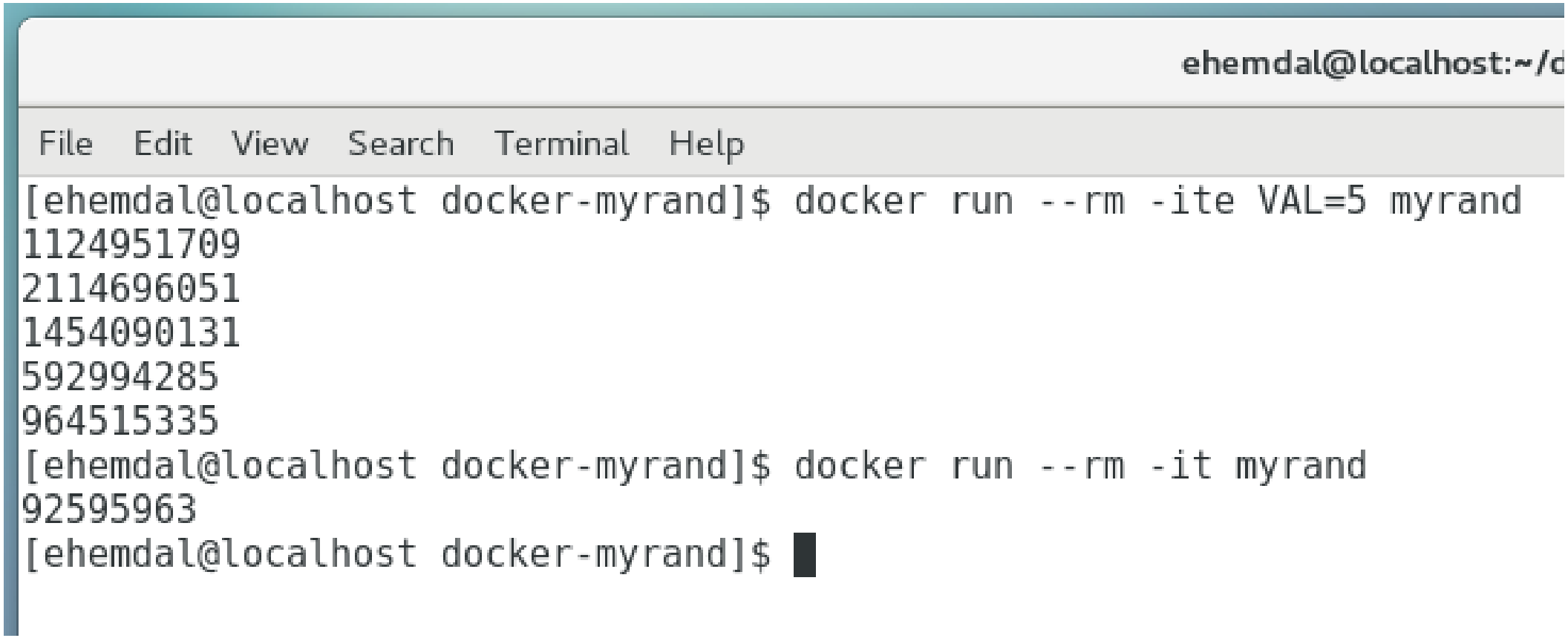
```
# Base image; coerce to CentOS 7.5
FROM centos:7.5.1804
MAINTAINER ehemdal@brandeis.edu
# Force to use GCC 4.8.5-28 and latest make
# Build the code from source
COPY ./myrand.c /app/myrand.c
RUN yum -y install gcc-4.8.5-28.el7_5.1.x86_64 make && \
    cd /app && make myrand && \
    yum -y remove gcc make cpp glibc-devel \
                  glibc-headers kernel-headers libgomp libmpc mpfr && \
    rm /app/myrand.c
ENV VAL=1
ENTRYPOINT /app/myrand $VAL
```

Basics (2 of 3): Build the image



This creates a Docker image containing my RNG. It takes as argument the number of random numbers to produce. The –t option tags the image with a convenient name.

Basics (3 of 3): Run the image in a container



DevOps Concepts

Immutable Infrastructure: Don't make changes in LIVE environment

- Update the Dockerfile and rebuild in DEV
- Test the new container image in QA
- Stop the old and start the new container in LIVE

Shift Left: DevOps concept to make changes early in the development pipeline

This pushes responsibility for security "leftward" and into the hands of developers

Infrastructure-as-code: The Dockerfile describes how to build your container images and can be version-controlled.

System Drift: "It works on my machine!": Containers allow you to prevent system drift.

Container isolation and security

Things to notice

- The container is run from a shim process at OS level
- The container terminates if I kill the Docker daemon or the shim job

“Containerize Everything!” Sure! Privileged access at OS level exposes the containers to this kind of misbehavior.

Remember that containerized applications still can invoke syscalls at OS level. The isolation isn’t as complete as what is provided by virtual machines.

Three fundamental levels of security**: Infrastructure security, Build-time security, Runtime security**

Infrastructure security

You are still running an operating system, so the basics still apply

- Firewalls
- OS patching
- SELinux for Linux hosts
- Logging/monitoring
- Etc.

Build-time security: Image size

In general, try to make images as small as possible, with as few packages/dependencies as possible.

For example, if you don’t need an editor, don’t include it

Small images -> smaller attack surface and smaller resource usage (running out of disk)

Remove packages that are only needed at build time

Chain RUN commands (as in the example)

Pick the right base image: a pre-built image for Ruby, Go, Python, etc. might be better for you than starting with a tiny image (like alpine) and adding dependencies

Balance size against flexibility: do you need diagnostic tools, shells, etc.?

Build-time security: Provenance of the image

Where did your base image come from?

- Motivation for Docker Trusted Registries, Notary (cryptographically-signed images)
- According to BanyanOps (container and virtualization startup), 30% of official images on Docker Hub contain known vulnerabilities (https://banyanops.com/blog/analyzing-docker-hub/).

Where do your dependencies come from? Are they vulnerable?

Scanners are available that can integrate with your CI pipeline and abort a build if vulnerable components are included. Examples:

Black Duck Software: https://www.blackducksoftware.com/

Clair Scanner: https://github.com/arminc/clair-scanner

Aqua: https://www.aquasec.com/use-cases/continuous-image-assurance/

Developers now drive security

- In LIVE environment, a patched OS is not enough if the container image brings in vulnerabilities
- Operations staff needs to collaborate with development

Runtime Security

Control the containers you will allow to run in your environment

Container affinity: Run “sensitive” containers on specific hosts or specific cores for further isolation

Keep your environment “clean”. Remove obsolete containers (docker image prune/docker system prune). Similar idea to removing old config files and back-rev scripts that might be lying around in working directories.

Watch for defaults: default passwords; open ports for applications and dashboards.

Options (CPU) for `docker run`

`--cap-add list`	Add Linux capabilities
`--cap-drop list`	Drop Linux capabilities
`--cgroup-parent string`	Optional parent cgroup for the container
`--cidfile string`	Write the container ID to the file
`--cpu-period int`	Limit CPU CFS (Completely Fair Scheduler) period
`--cpu-quota int`	Limit CPU CFS (Completely Fair Scheduler) quota
`--cpu-rt-period int`	Limit CPU real-time period in microseconds
`--cpu-rt-runtime int`	Limit CPU real-time runtime in microseconds
`-c, --cpu-shares int`	CPU shares (relative weight)
`--cpus decimal`	Number of CPUs
`--cpuset-cpus string`	CPUs in which to allow execution (0-3, 0,1)

Options (I/O) for `docker run`

--device-read-bps list

Limit read rate (bytes per second) from a device (default [])

--device-read-iops list

Limit read rate (IO per second) from a device (default [])

--device-write-bps list

Limit write rate (bytes per second) to a device (default [])

--device-write-iops list

Limit write rate (IO per second) to a device (default [])

--disable-content-trust

Skip image verification (default true)

Options (Health check) for `docker run`

`--health-cmd string`

Command to run to check health

`--health-interval duration`

Time between running the check (ms|s|m|h) (default 0s)

`--health-retries int`

Consecutive failures needed to report unhealthy

Options (Memory) for `docker run`

-m, --memory bytes Memory limit

--memory-reservation bytes Memory soft limit

--memory-swap bytes

Swap limit equal to memory plus swap

--memory-swappiness int

Tune container memory swappiness (0 to 100)

--oom-kill-disable=false

Whether to disable OOM Killer for the container or not.

Options (Memory) for `docker run`

`--storage-opt list`

Storage driver options for the container

`--sysctl map`

Sysctl options (default map[])

`--ulimit ulimit`

Ulimit options (default [])

`-u, --user string`

Username or UID (format: <name|uid>[:<group|gid>])