LINUX CONTAINER INTERNALS

How they really work

Jay Ryan (@jaywryan)
Senior Account Solutions Architect

Steve Ovens
Architect - Solutions and Technology Practice

Content adapted and modified from Scott McCarty (@fatherlinux) original Linux Container Internals workshops.

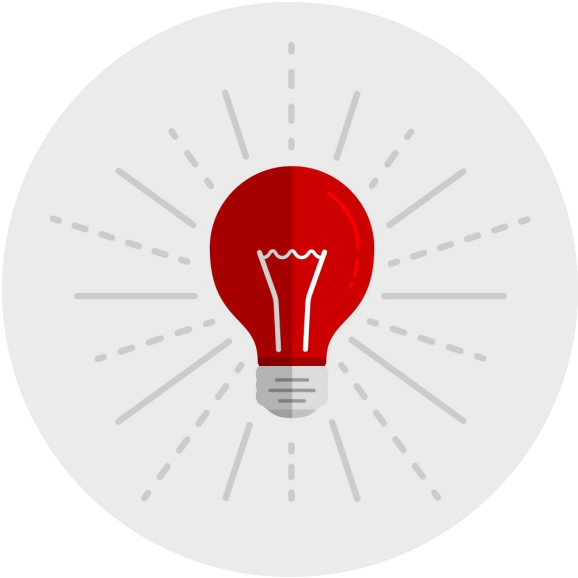
Red Hat

AGENDA

Morning Session 09:00→ 12:30 EST

Lunch Break 12:30→ 1:30 EST

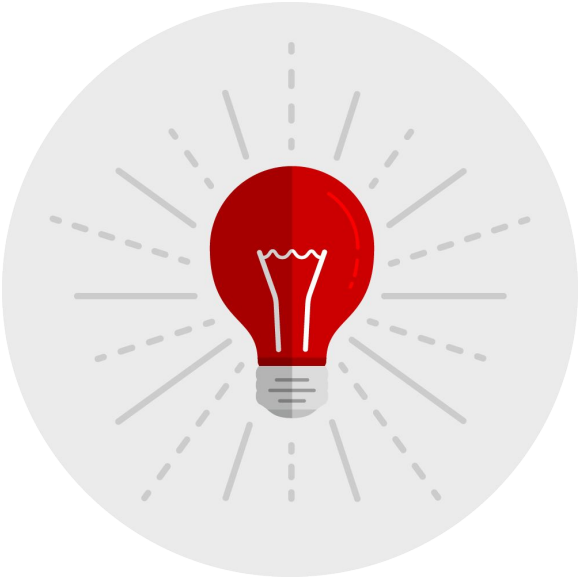
Afternoon Session 1:30→ 5:00 EST



AGENDA

Morning Session

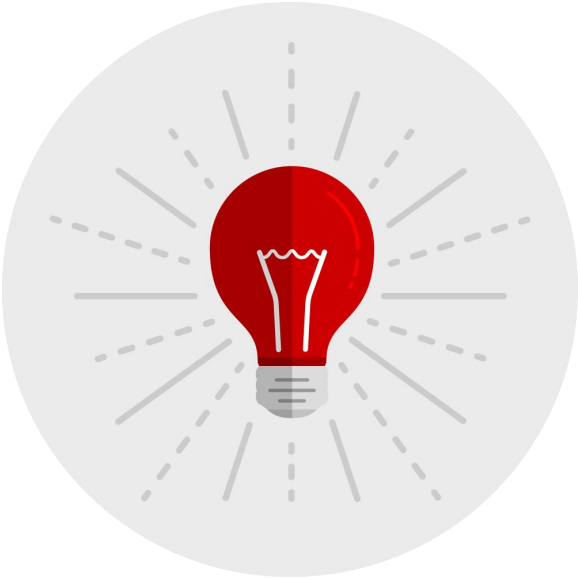
- Introduction to Container Technology
- Lab 1 - Intro to Containers
- Lab 8 - Container from Scratch
- Deep Dive - Images & Registries
- Lab 2 - Container Images
- Lab 3 - Container Registries



AGENDA

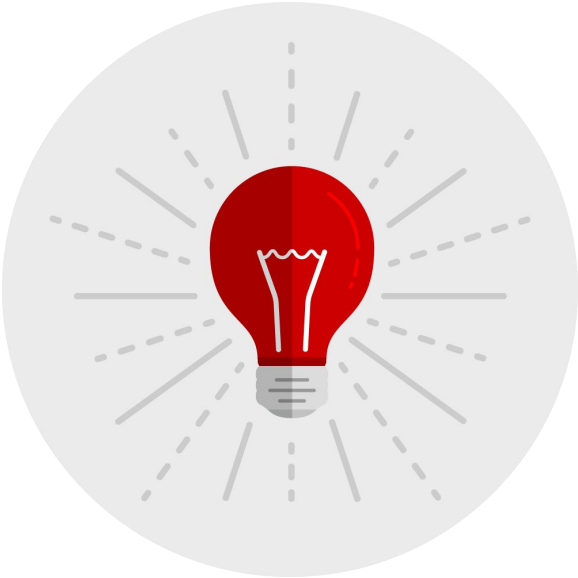
Afternoon Session

- Deep Dive - Container Hosts
  - Container Engine
  - Container Runtime
- LAB 4 - Container Hosts
- LAB 6 - OCI Specs
- BONUS LABS!!



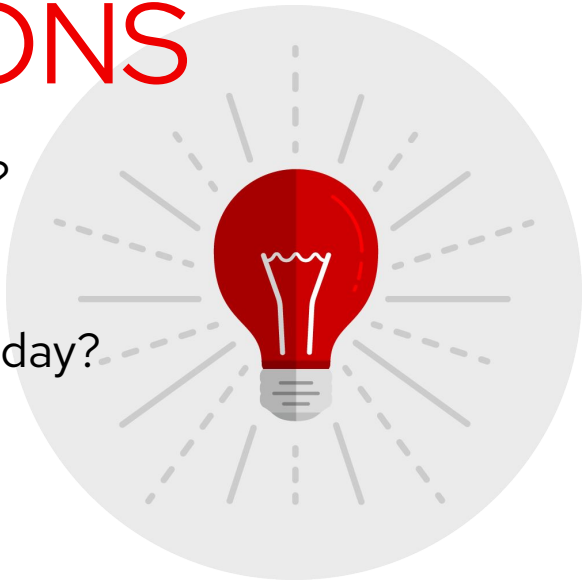
LOGISTICS

- Requirements – A laptop with internet access
- No food or drinks on the tables
- There are no stupid questions
- I don’t know / I’ll get back to you
- Small group / Let’s make it Interactive
- You will get out what you put in
- Building Logistics / Questions?



INTRODUCTIONS

- What Company or Organization are you from?
- What's your role (not your title!)
- What are you looking to get out of this Lab today?

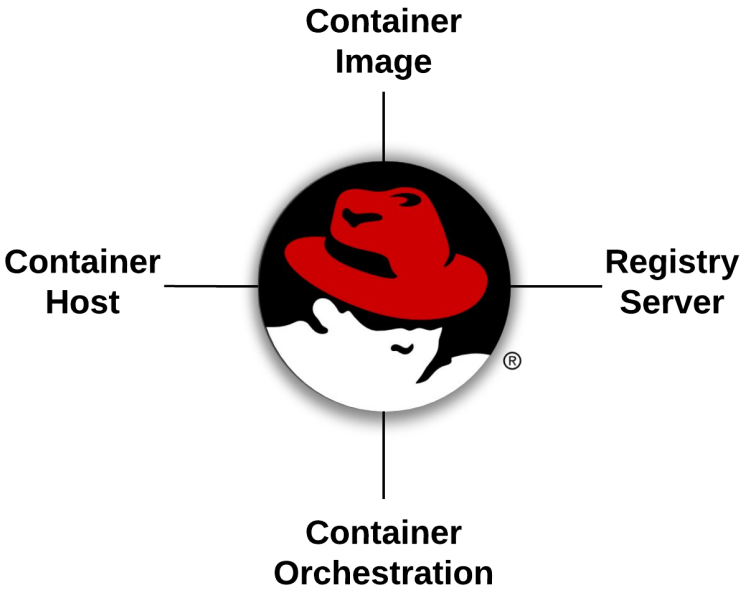


INTRO TO CONTAINER TECHNOLOGY

Red Hat

Production-Ready Containers

What are the building blocks you need to think about?



The History of Containers

1979:
UNIX
CHROOT
SYSCALL ADDED

2000:
JAILS ADDED
TO FREEBSD

2001:
LINUX -VSERVER
PROJECT

2003:
SELINUX
ADDED TO LINUX
MAINLINE

2005:
FULL RELEASE
OF SOLARIS
ZONES

2006:
PROCESS
CONFINEMENT

2007:
GPC RENAMED
CGROUPS

2008:
LINUX
CONTAINER
PROJECT (LXC)

2008:
KERNEL & USER
NAMESPACES

2013:
DOTCLOUD PYCON
LIGHTNING TALK

2013:
DOTCLOUD
BECOMES
DOCKER

2013:
RED HAT
ENTERPRISE
LINUX

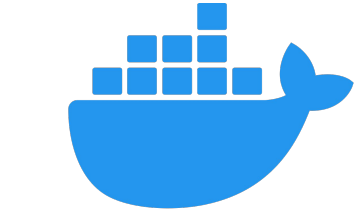
2014:
GOOGLE
KUBERNETES

1979 2000 2005 2010

https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016

Complexity begets docker

- 2013 - A Star is born!
- Makes all of this hard stuff easy for developers
- New constructs to build, deploy, store, run containers
    - Container Host - brings the kernel to the party
    - Container Runtime - Interface to the kernel
    - Container Engine - Interface to the operator (API/CLI)

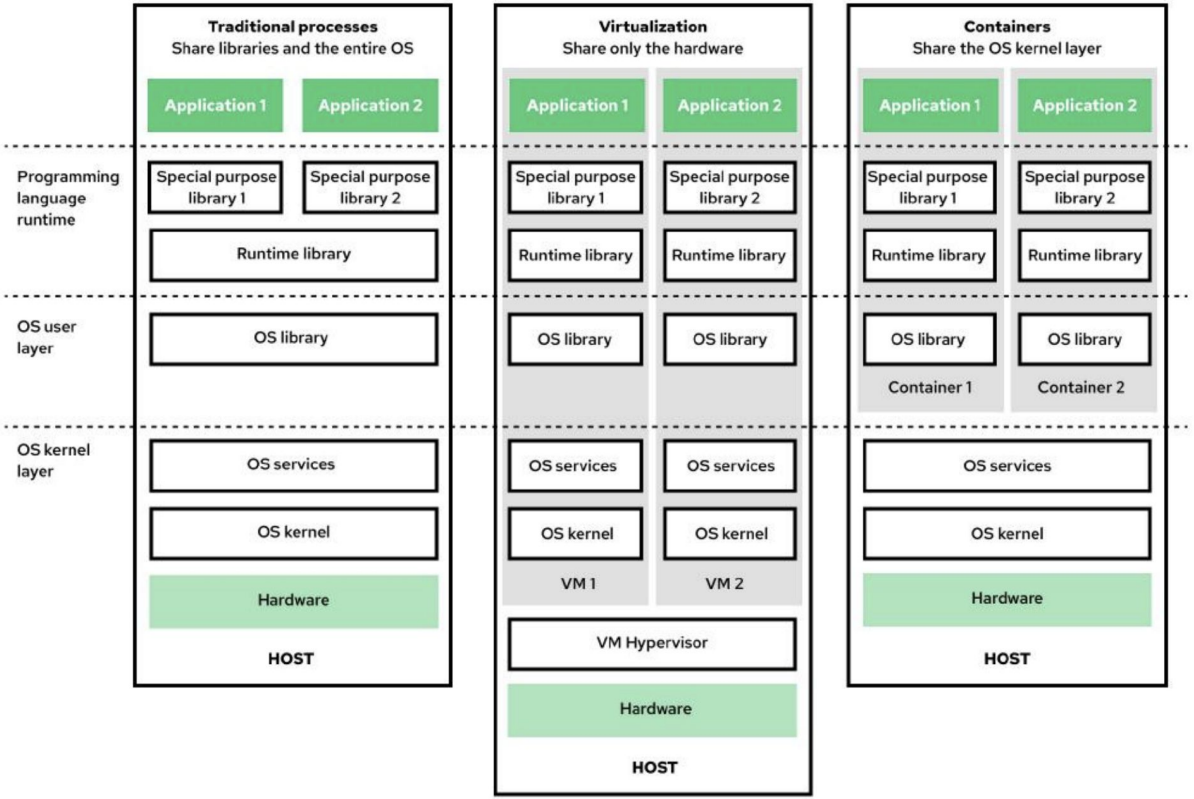


What is a container?

It depends who you ask!

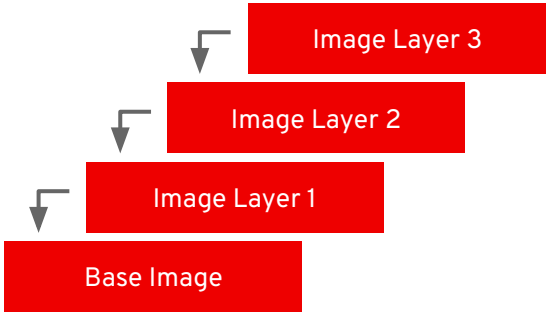
- **Google –** A unit of software that contains code and all its dependencies so that the application can be run in the same way in different computational environments.
- **System Administrator** - Portability for an app and its runtime dependencies
- **Developer** - Universal Application Packager
- **Operating system** - A process in a sandbox

What is a container?

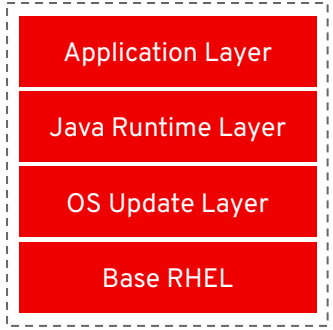


containers are created from container images

container images are structured in layers



Container Image Layers



Example Container Image

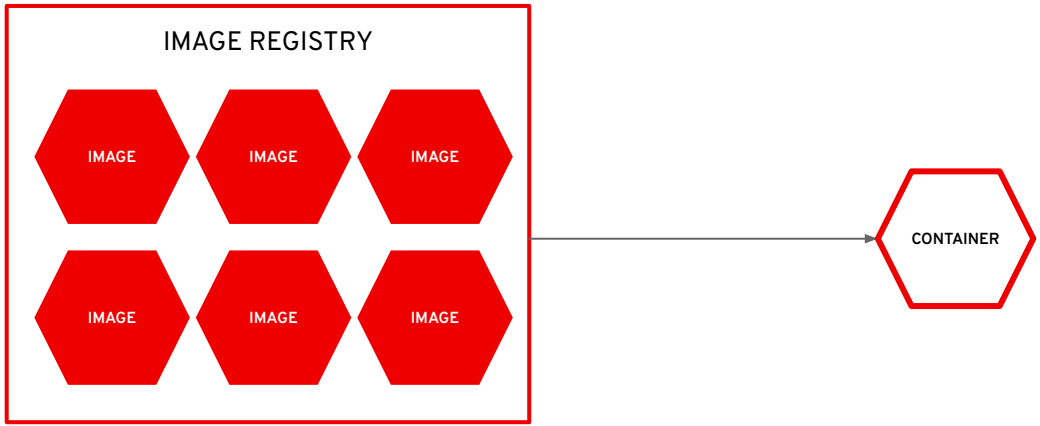
anatomy of a Dockerfile/Containerfile

```
FROM registry.access.redhat.com/ubi8/ubi
ENV foo=text
RUN dnf install -y java-11-openjdk
ADD my-app.jar /home/my-app.jar
EXPOSE 8080
CMD java -jar /home/my-app.jar
```

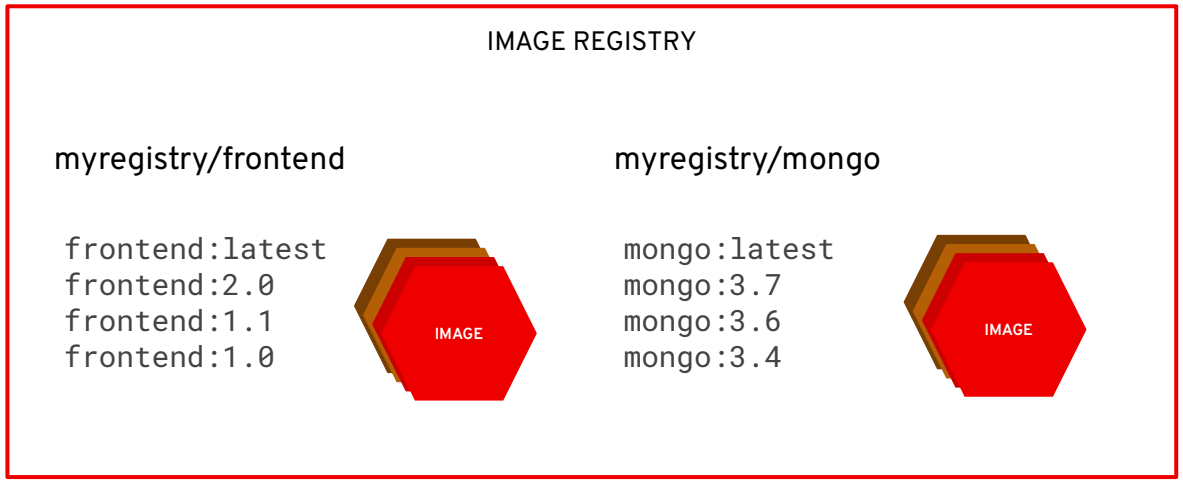
1. Inherit from a base image
2. Parameters as environment variables
3. Install dependencies (tooling from base image)
4. Add your app as a new Layer
5. Expose the port your app will use
6. Run the app

Example for Java app

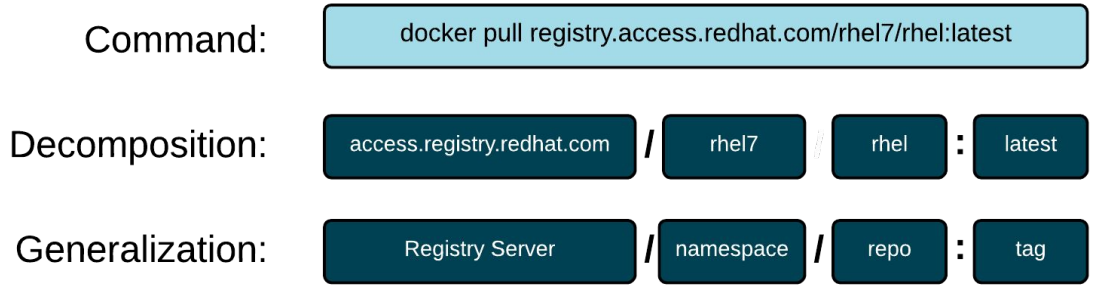
container images are stored in an image registry



an image repository contains all versions of an image in the image registry

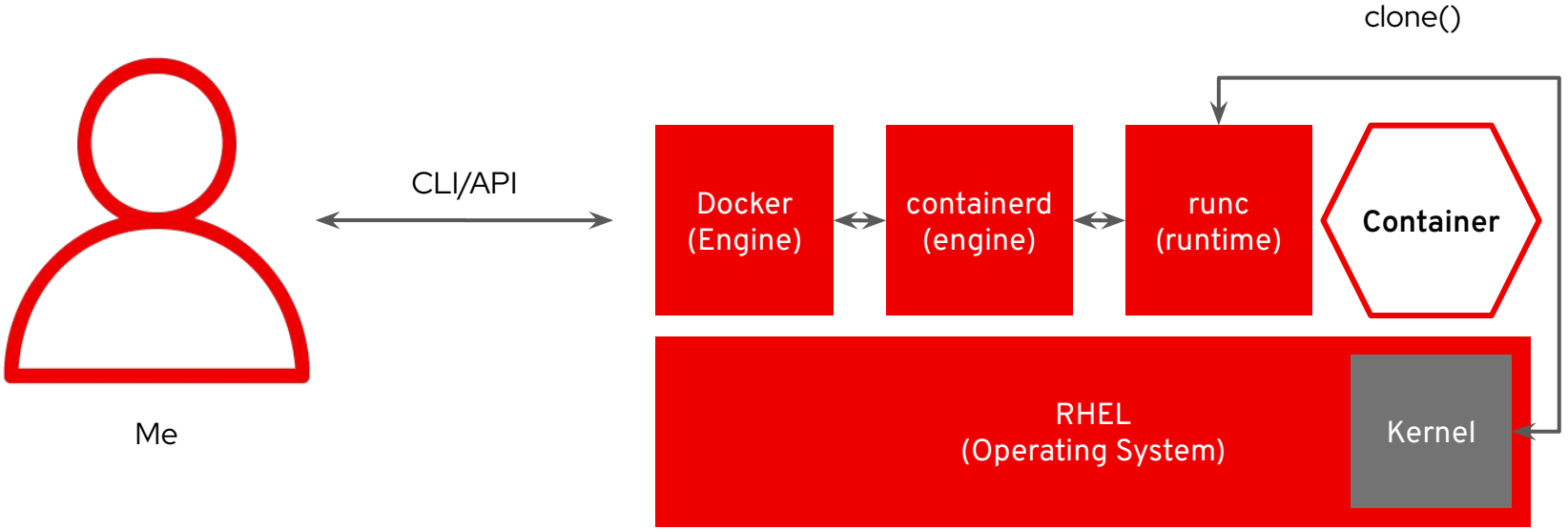


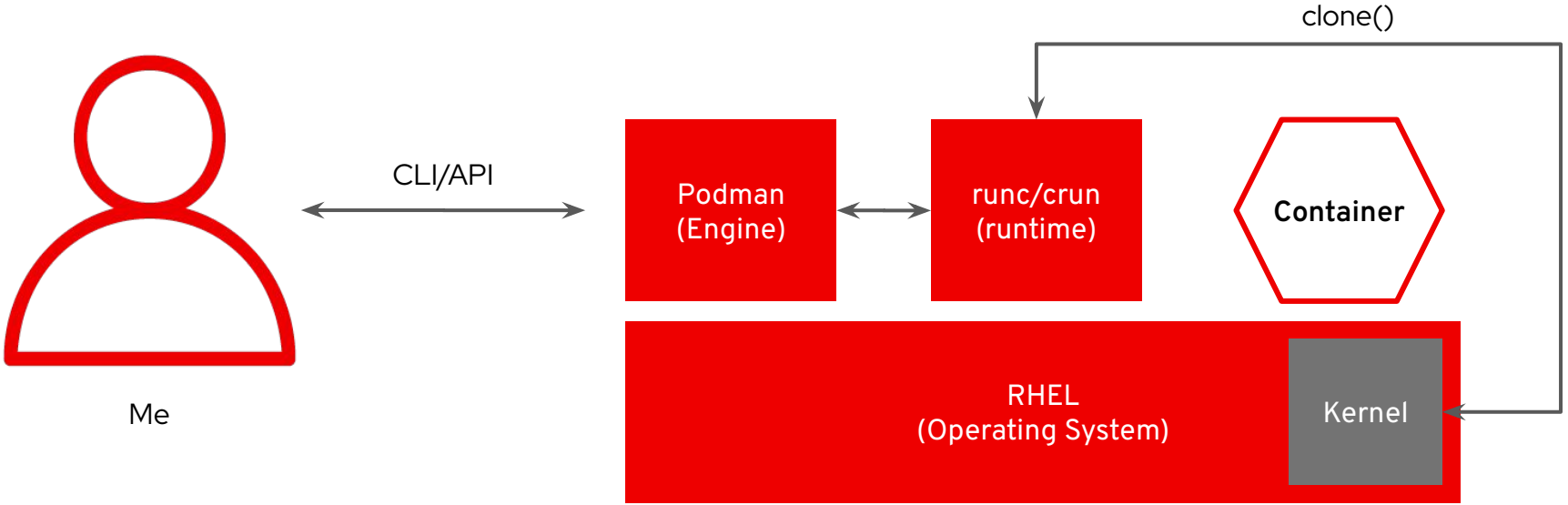
Registry Command Syntax

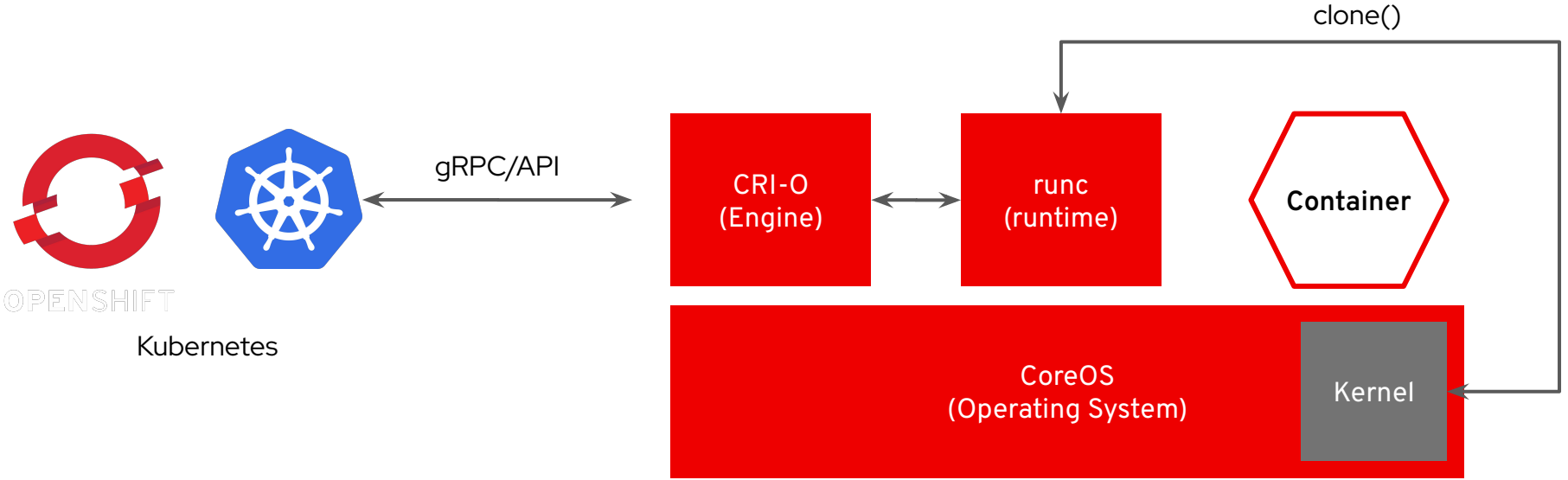


Courtesy of @fatherlinux - Container Internals


Google
docker architecture
Sign in
Docker Documentation
Docker overview | Docker Documentation
Whizlabs
Docker Archi
Geekflare
Docker Architecture and its Components ...
clone()
Docker Architecture
eduCBA
Docker Architecture | Learn the Objects ...
XenonStac
Docker Conta
Docker Resource ...
Container
Runtir
Kernel
GEEK FLARE
Docker Build
Docker Pull
Docker Run
Docker Daemon
Containers
Images
NGINX
openstack
Build
Pull
Run
educba.com
eduCBA
Docker Architecture | Learn the Objects ...
Simplilearn
Docker Architect
DZone
Docker Containers and Kubernetes: An ...
ainer? - Aqua Security
ResearchGate
Docker architecture ...
Docker Client
Docker Host/Server
Docker Registry/Hub
docker build
docker pull
docker run
docker push
Docker Daemon
Containers
Images
19
LINUX OS
CONTAINER
CONTAINER
NODE.JS
ALPINE
NGINX
BUSYBOX
DOCKER ENGINE
KERNEL
Docker Engine Architecture
Docker Engine - Enterprise
Docker Engine - Community
containerd
runc


clone()
CLI/API
Docker
(Engine)
containerd
(engine)
runc
(runtime)
Container
RHEL
(Operating System)
Kernel
Me


clone()
CLI/API
Podman
(Engine)
runc/crun
(runtime)
Container
Me
RHEL
(Operating System)
Kernel


clone()
OPENSHIFT
gRPC/API
CRI-O
(Engine)
runc
(runtime)
Container
Kubernetes
CoreOS
(Operating System)
Kernel

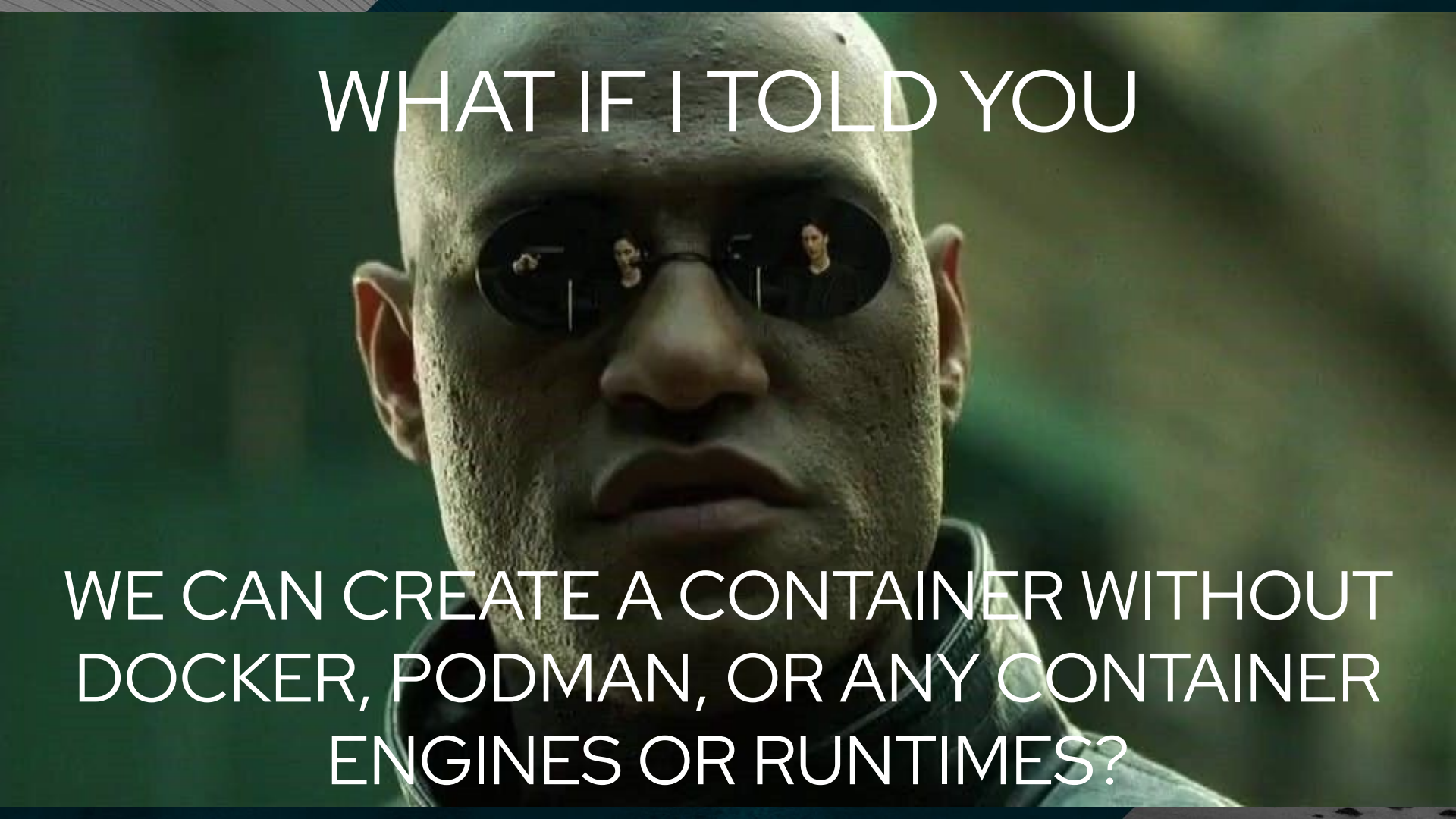
LAB

INTRODUCTION TO CONTAINERS (~30 min)

https://red.ht/olf-container

CONTAINERS FROM SCRATCH

Red Hat


WHAT IF I TOLD YOU
WE CAN CREATE A CONTAINER WITHOUT DOCKER, PODMAN, OR ANY CONTAINER ENGINES OR RUNTIMES?

How do we make one?

- It all starts with a process
- The we wrap some Linux/Unix boundaries around it that....
  - Partition kernel resources (filesystem, IPC, network, PID, users, more)
  - Limit resource usage
  - Prioritize or de-prioritize CPU share and Disk I/O
  - Control freezing, stopping, starting and checkpointing
  - Account for usage of resources managed above
  - Limit what/who they can talk to
  - Limit what they can say

Namespaces

- It all starts with a process
- The we wrap some boundaries around it that....
  - Partition kernel resources (filesystem, IPC, network, PID, users, more)
  - Limit resource usage
  - Prioritize or de-prioritize CPU share and Disk I/O
  - Control freezing, stopping, starting and checkpointing
  - Account for usage of resources managed above
  - Limit what/who they can talk to (MAC)
  - Limit what they can say

CGroups

- It all starts with a process
- The we wrap some boundaries around it that....
  - Partition kernel resources (filesystem, IPC, network, PID, users, more)
  - Limit resource usage
  - Prioritize or de-prioritize CPU share and Disk I/O
  - Control freezing, stopping, starting and checkpointing
  - Account for usage of resources managed above
  - Limit what/who they can talk to (MAC)
  - Limit what they can say

SELinux / AppArmor

- It all starts with a process
- The we wrap some boundaries around it that....
  - Partition kernel resources (filesystem, IPC, network, PID, users, more)
  - Limit resource usage
  - Prioritize or de-prioritize CPU share and Disk I/O
  - Control freezing, stopping, starting and checkpointing
  - Account for usage of resources managed above
  - Limit what/who they can talk to (MAC)
  - Limit what they can say

Stop Disabling SELINUX!

https://stopdisablingselinux.com/

#MakeSELINUXenforcingAgain
#SETENFORCE1

SECCOMP

- It all starts with a process
- The we wrap some boundaries around it that....
  - Partition kernel resources (filesystem, IPC, network, PID, users, more)
  - Limit resource usage
  - Prioritize or de-prioritize CPU share and Disk I/O
  - Control freezing, stopping, starting and checkpointing
  - Account for usage of resources managed above
  - Limit what/who they can talk to (MAC)
  - Limit what they can say

LAB

CONTAINERS FROM SCRATCH (~20 min)

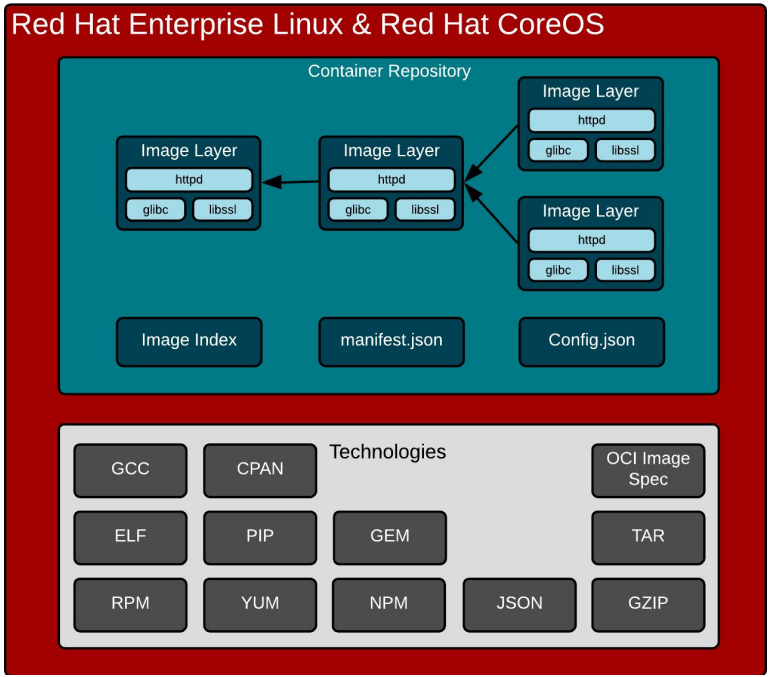
CONTAINER IMAGES

CONTAINER IMAGE

Open source code/libraries, in a Linux distribution, in a tarball

Even base images are made up of layers:

- Libraries (glibc, libssl)
- Binaries (httpd)
- Packages (rpms)
- Dependency Management (yum)
- Repositories (ubi8)
- Image Layer & Tags (ubi8:7-929)
- At scale, across teams of developers and CI/CD systems, consider all of the necessary technology

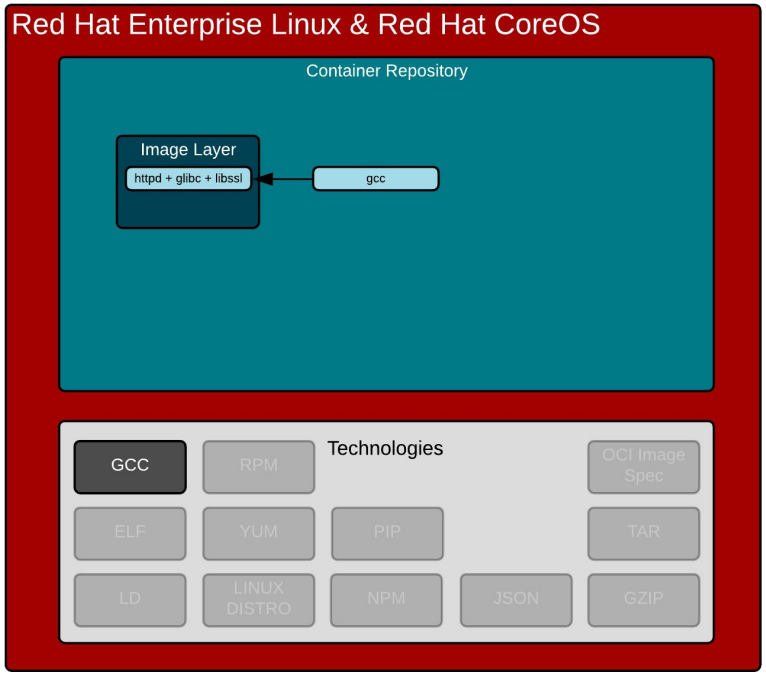


IT ALL STARTS WITH COMPILING

Statically linking everything into the binary

Starting with the basics:

- Programs rely on libraries
- Especially things like SSL - difficult to reimplement in for example PHP
- Math libraries are also common
- Libraries can be compiled into binaries - called static linking
- Example: C code + glibc + gcc = program

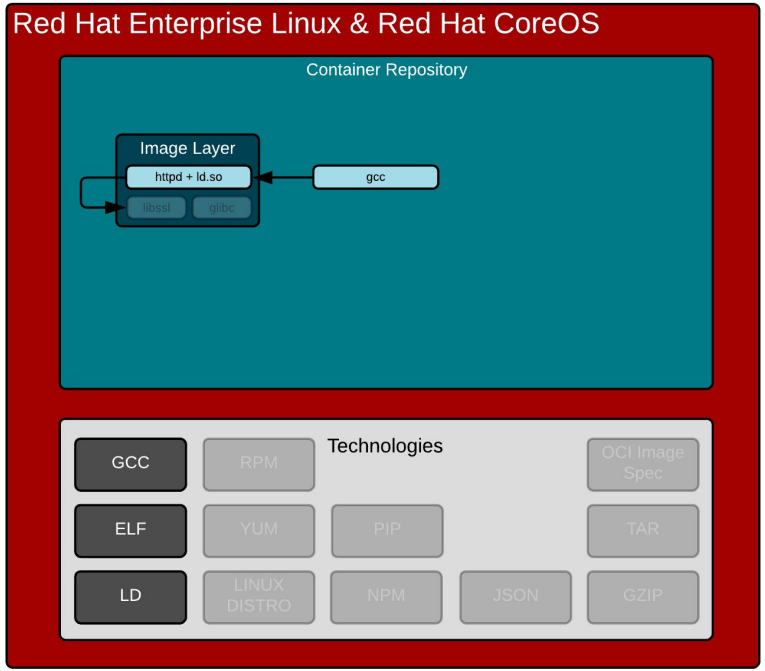


LEADS TO DEPENDENCIES

Dynamically linking libraries into the binary

Getting more advanced:

- This is convenient because programs can now share libraries
- Requires a dynamic linker
- Requires the kernel to understand where to find this linker at runtime
- Not terribly different than interpreters (hence the operating system is called an interpretive layer)

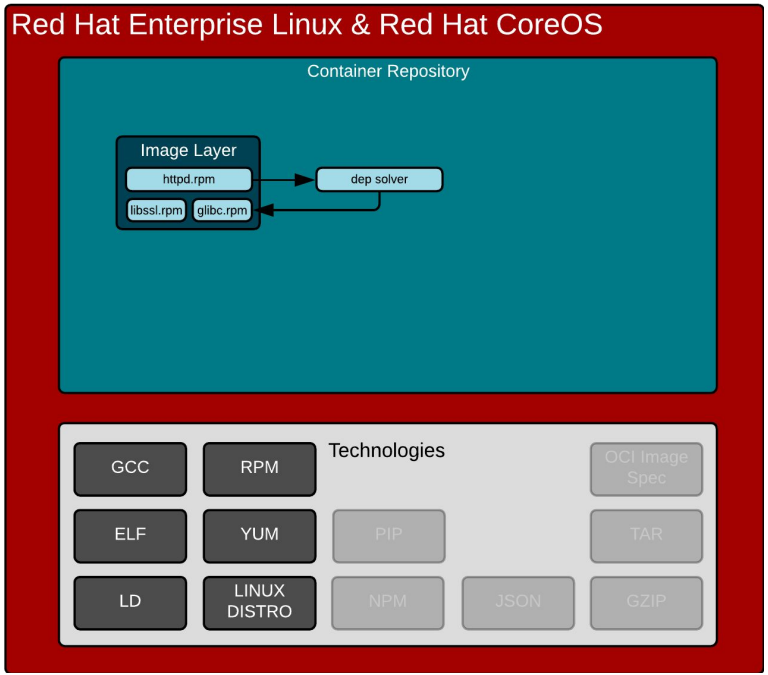


PACKAGING & DEPENDENCIES

RPM and Yum were invented a long time ago

Dependencies need resolvers:

- Humans have to create the dependency tree when packaging
- Computers have to resolve the dependency tree at install time (container image build)
- This is essentially what a Linux distribution does sans the installer (container image)

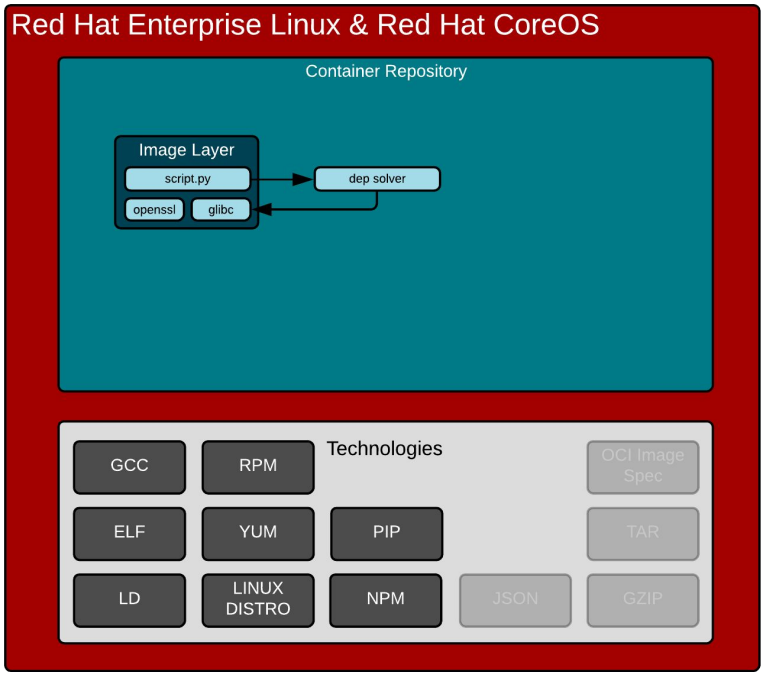


PACKAGING & DEPENDENCIES

Interpreters have to handle the same problems

Dependencies need resolvers:

- Humans have to create the dependency tree when packaging
- Computers have to resolve the dependency tree at install time (container image build)
- Python, Ruby, Node.js, and most other interpreted languages rely on C libraries for difficult tasks (ex. SSL)

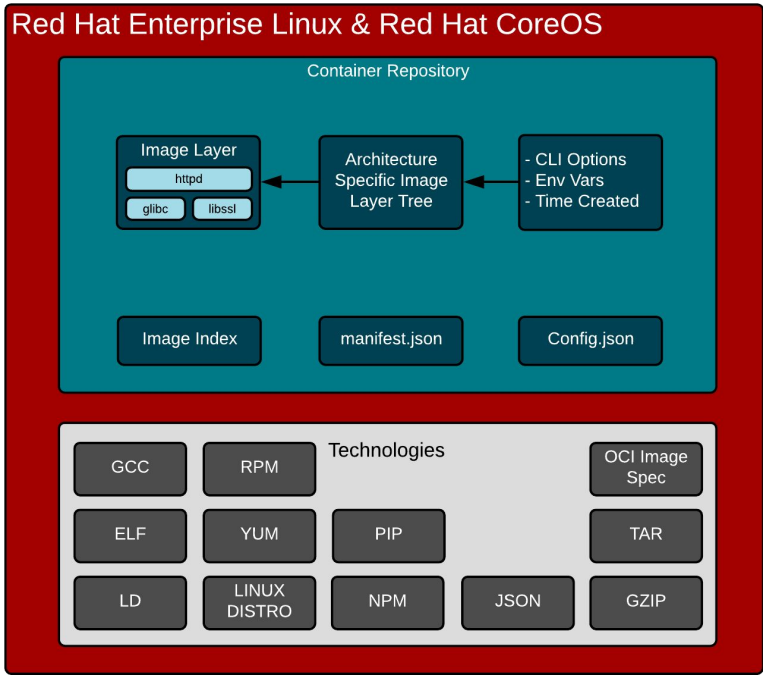


CONTAINER IMAGE PARTS

Governed by the OCI image specification standard

Lots of payload media types:

- Image layers provide change sets - adds/deletes of files
- Image Index/Manifest.json - provide index of image layers
- Config.json provides command line options, environment variables, time created, and much more
- Not actually single images, really repositories of image layers

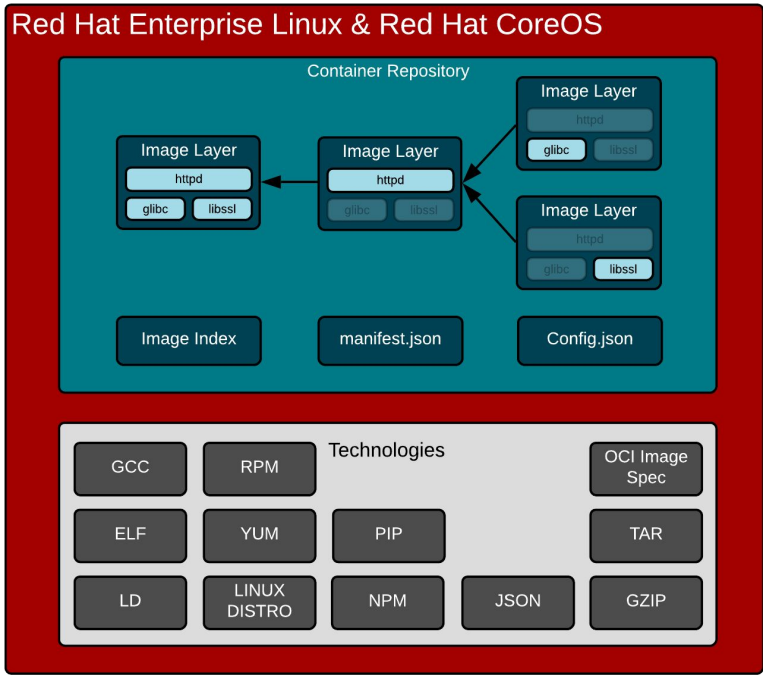


LAYERS ARE CHANGE SETS

Each layer has adds/deletes

Each image layer is a permutation in time:

- Different files can be added, updated or deleted with each change set
- Still relies on package management for dependency resolution
- Still relies on dynamic linking at runtime

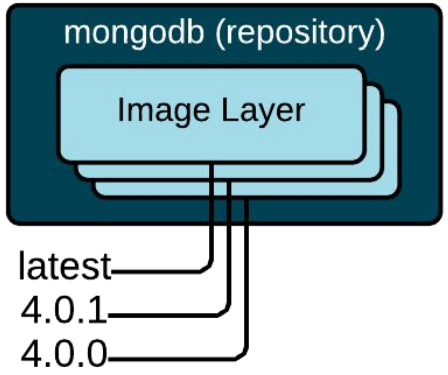


LAYERS ARE CHANGE SETS

Some layers are given a human readable name

Each image layer is a permutation in time:

- Different files can be added, updated or deleted with each change set
- Still relies on package management for dependency resolution
- Still relies on dynamic linking at runtime



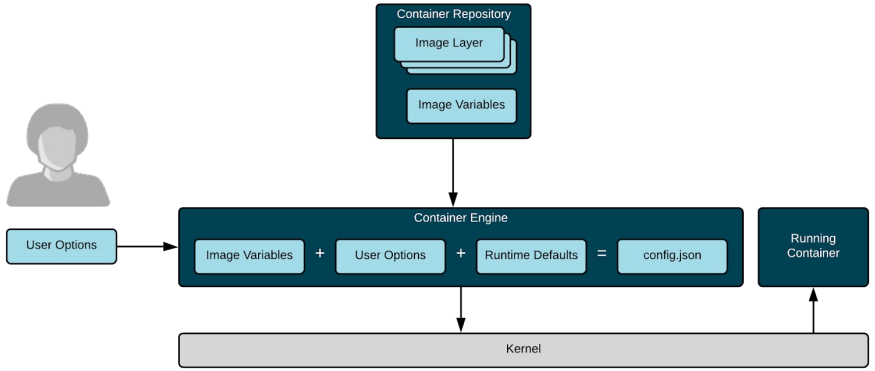
Layers and Tags

CONTAINER IMAGES & USER OPTIONS

Come with default binaries to start, environment variables, etc

Each image layer is a permutation in time:

- Different files can be added, updated or deleted with each change set
- Still relies on package management for dependency resolution
- Still relies on dynamic linking at runtime

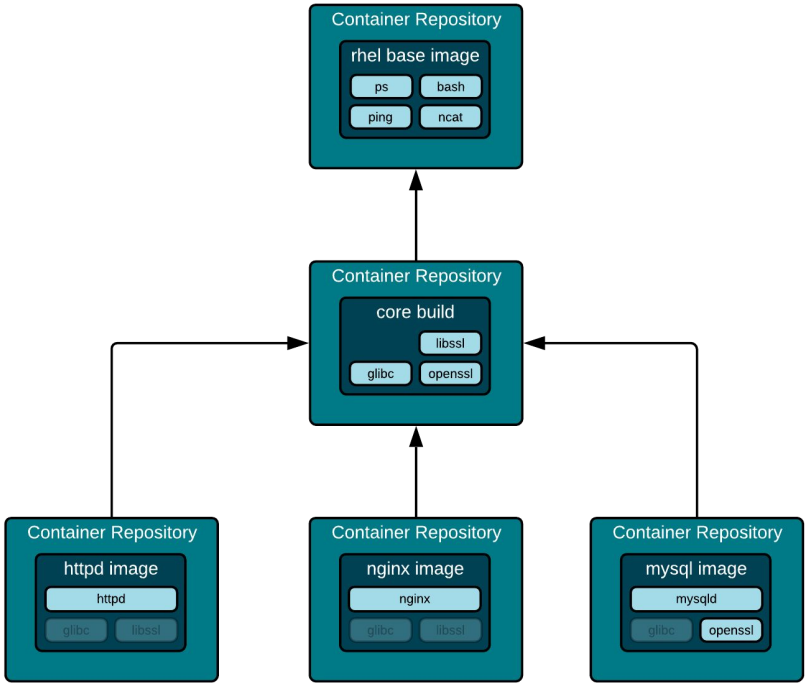


INTER REPOSITORY DEPENDENCIES

Think through this problem as well

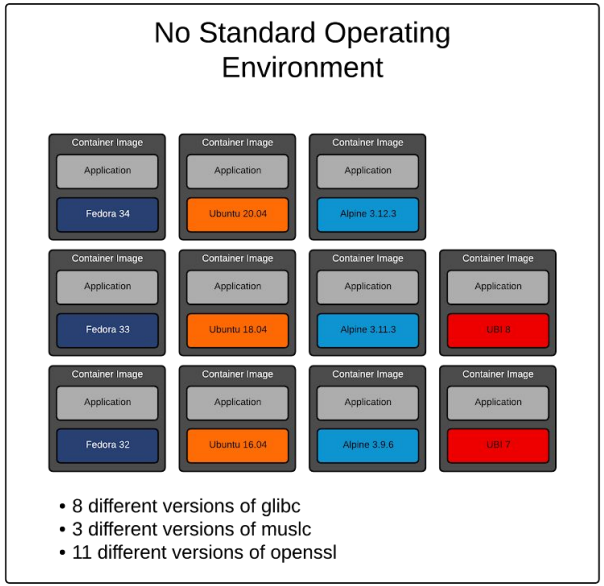
You have to build this dependency tree yourself:

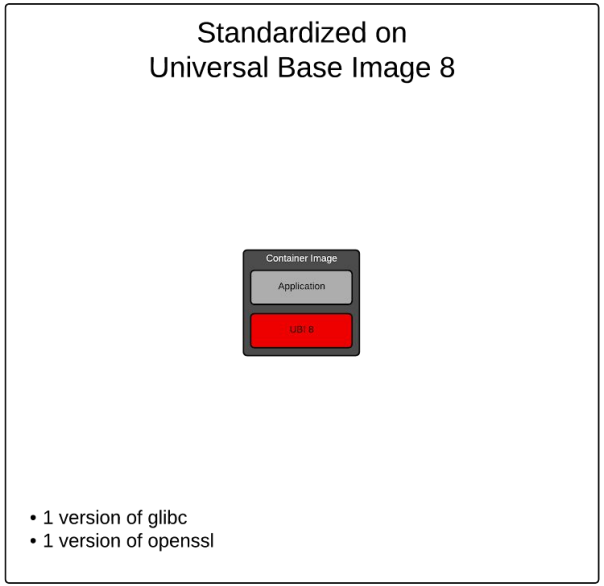
- DRY - Do not repeat yourself. Very similar to functions and coding
- OpenShift BuildConfigs and DeploymentConfigs can help
- Letting every development team embed their own libraries takes you back to the 90s



DEVELOPERS WILL MAKE A CHOICE

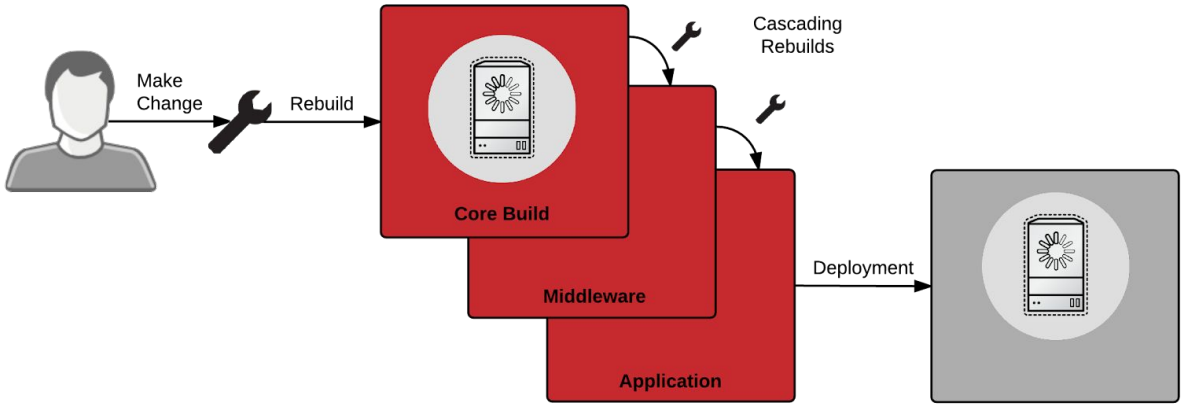
Which creates image sprawl





Fancy Files

The future of collaboration in the user space....



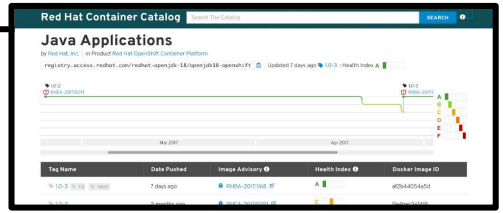
A Software Factory

Automated quality and security: because you can't inspect quality into a product

Automatically prohibit untrusted containers via Openshift policy

Trusted repos
registry.redhat.io

Private registry
- **RH Quay**
- JFrog Artifactory
- Sonatype Nexus



Secure Supply Chain of Quality Parts
- Quality Assurance
- Certifications
- Signing & Secure
- Distribution

CCB
RAPID DELIVERY

REQ → DEV → UNIT TEST → CODE QUAL → SEC SCAN → INT TEST → QA UAT → PROD

-**RH Dev Spaces**
Dependency
Analytics
- Snyk

-Cucumber
-Arquillian
-Junit

-Sonarqube
-Coverity

- **RH ACS**
- Aqua Sec
- Clair
- CyberArk
- Synopsys
- Sysdig
- Palo Alto (Twistlock)

AUTOMATED QUALITY

CM | CS

- **RH ACS**
- Aqua Sec
- CyberArk
- Hashicorp
- Sysdig
- Palo Alto (Twistlock)

OPENSHIFT SOFTWARE FACTORY

Source:
Red Hat Container Catalog: Java Applications
More about the Container Health Index

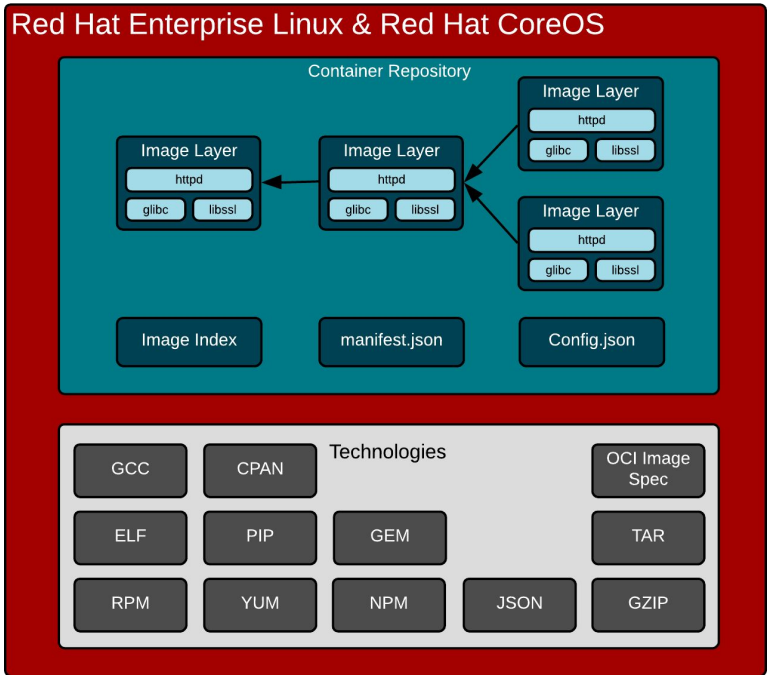
Red Hat

CONTAINER IMAGE

Open source code/libraries, in a Linux distribution, in a tarball

Even base images are made up of layers:

- Libraries (glibc, libssl)
- Binaries (httpd)
- Packages (rpms)
- Dependency Management (yum)
- Repositories (rhel7)
- Image Layer & Tags (rhel7:7.5-404)
- At scale, across teams of developers and CI/CD systems, consider all of the necessary technology



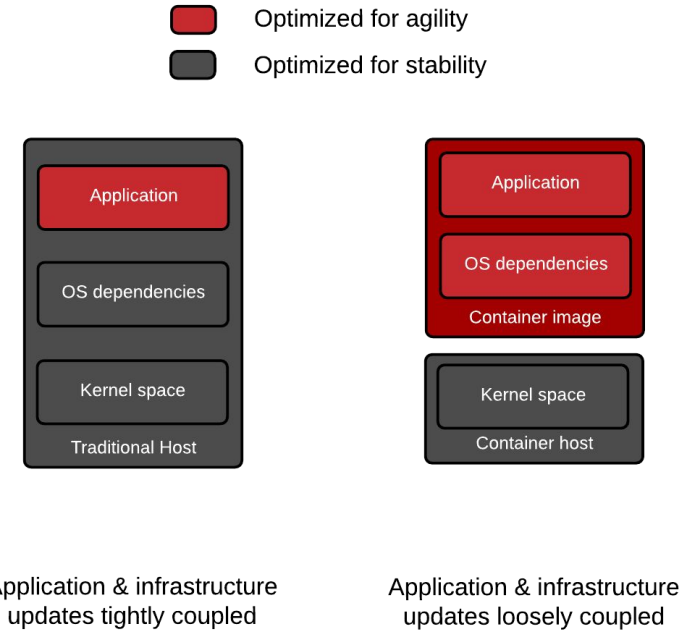
CONTAINER REGISTRIES

REGISTRY SERVERS

Better than virtual appliance market places :-)

Defines a standard way to:

- Find images
- Run images
- Build new images
- Share images
- Pull images
- Introspect images
- Shell into running container
- Etc, etc, etc

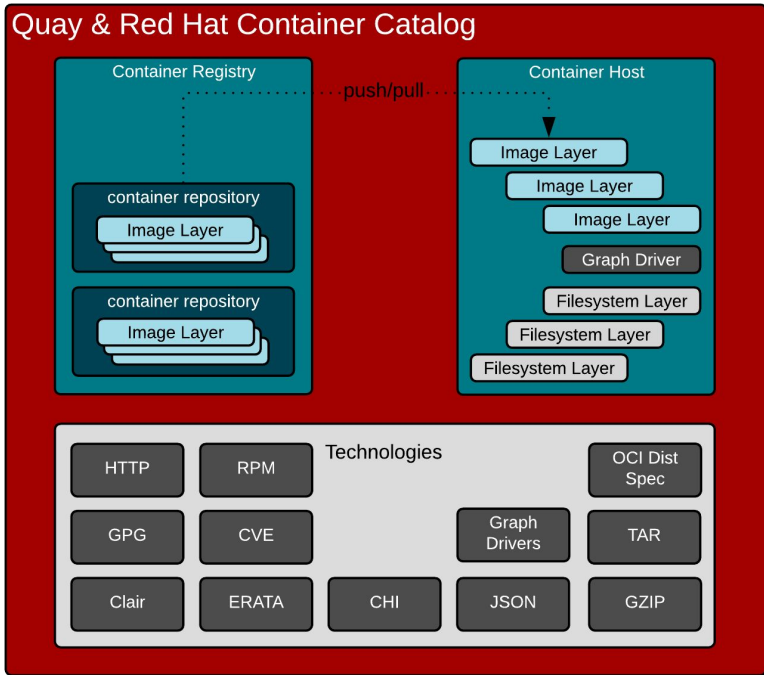


CONTAINER REGISTRY & STORAGE

Mapping image layers

Covering push, pull, and registry:

- Rest API (blobs, manifest, tags)
- Image Scanning (clair)
- CVE Tracking (errata)
- Scoring (Container Health Index)
- Graph Drivers (overlay2, dm)
- Responsible for maintaining chain of custody for secure images from registry to container host

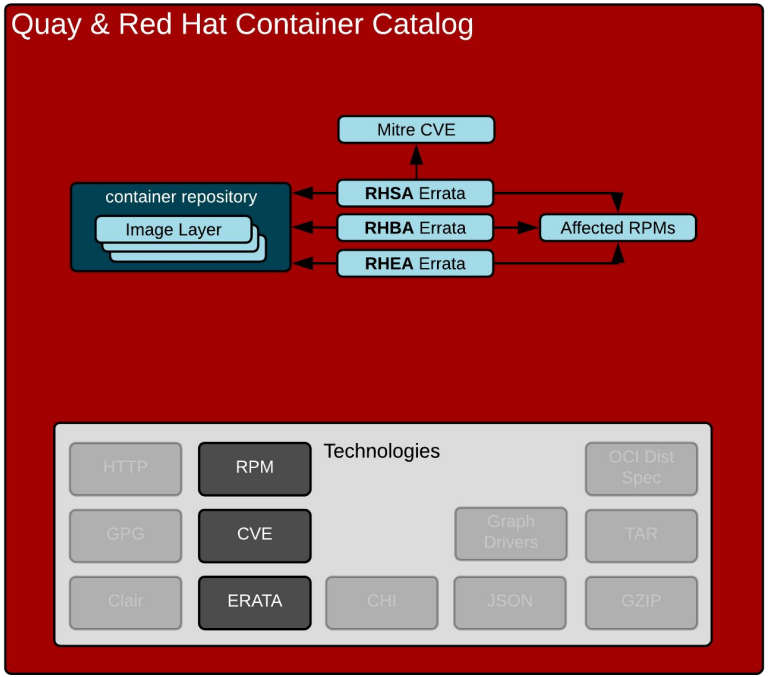


START WITH QUALITY REPOSITORIES

Repositories depend on good packages

Determining the quality of repository requires meta data:

- Errata is simple to explain, hard to build
  - Security Fixes
  - Bug Fixes
  - Enhancements
- Per container images layer (tag), often maps to multiple packages

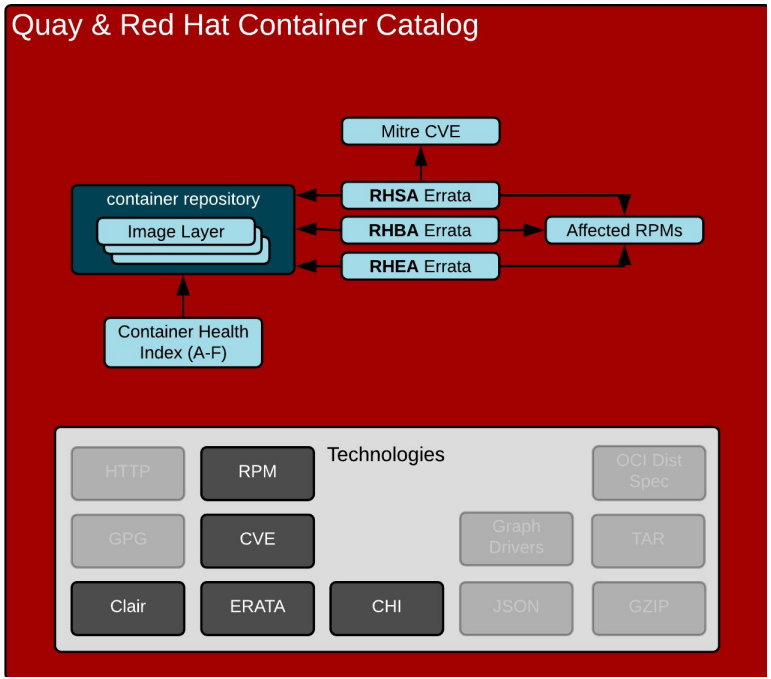


SCORING REPOSITORIES

Images age like cheese, not like wine

Based on severity and age of Security Errata:

- Trust is temporal
- Even good images go bad over time because the world changes around you

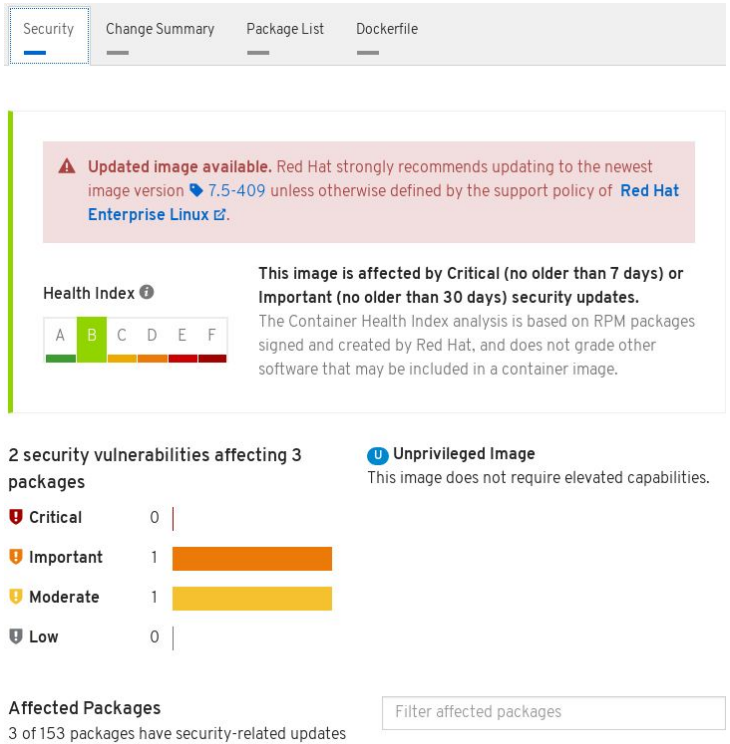


SCORING REPOSITORIES

Container Health Index

Based on severity and age of Security Errata:

- Trust is temporal
- Images must constantly be rebuilt to maintain score of “A”

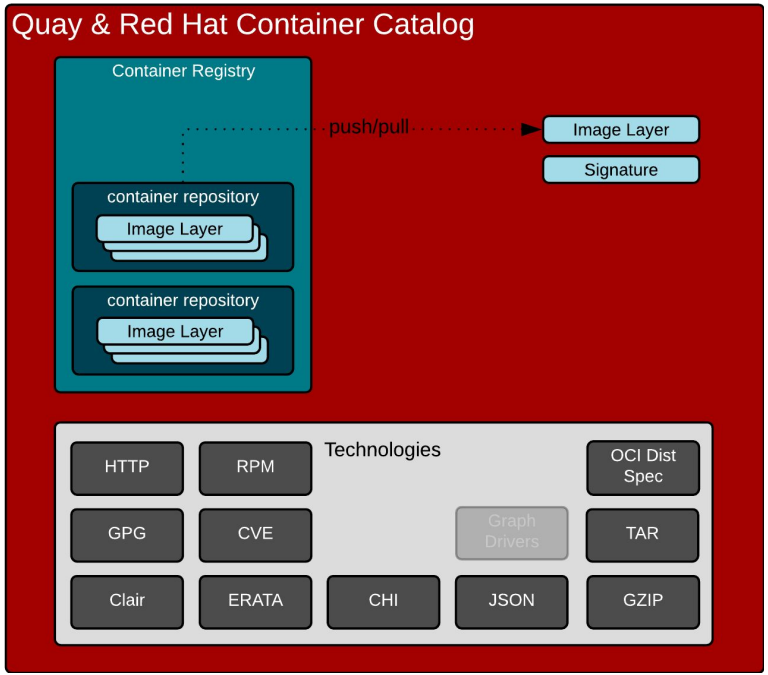


PUSH, PULL & SIGNING

Signing and verification before/after transit

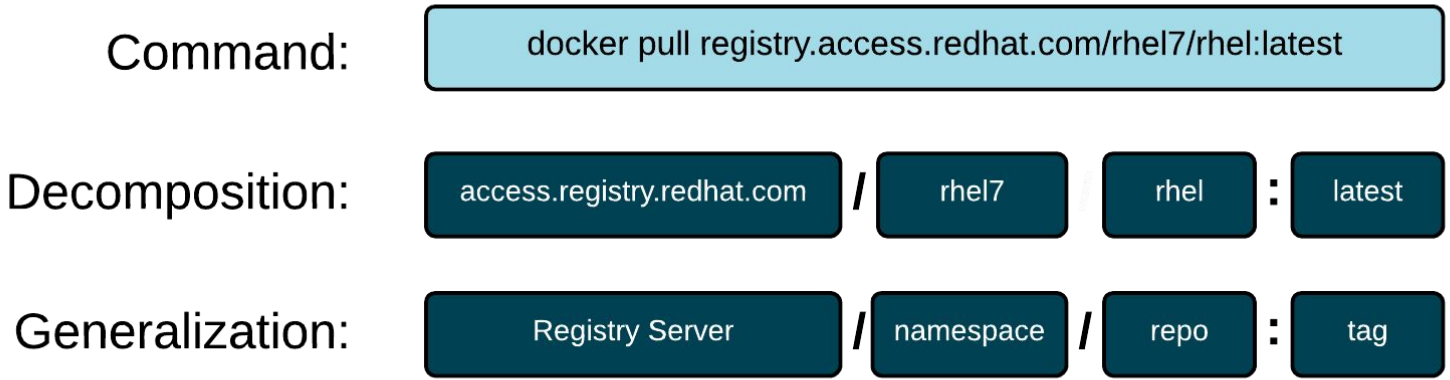
Registry has all of the image layers and can have the signatures as well:

- Download trusted thing
- Download from trusted source
- Neither is sufficient by itself



PUSH, PULL & SIGNING

Mapping image layers

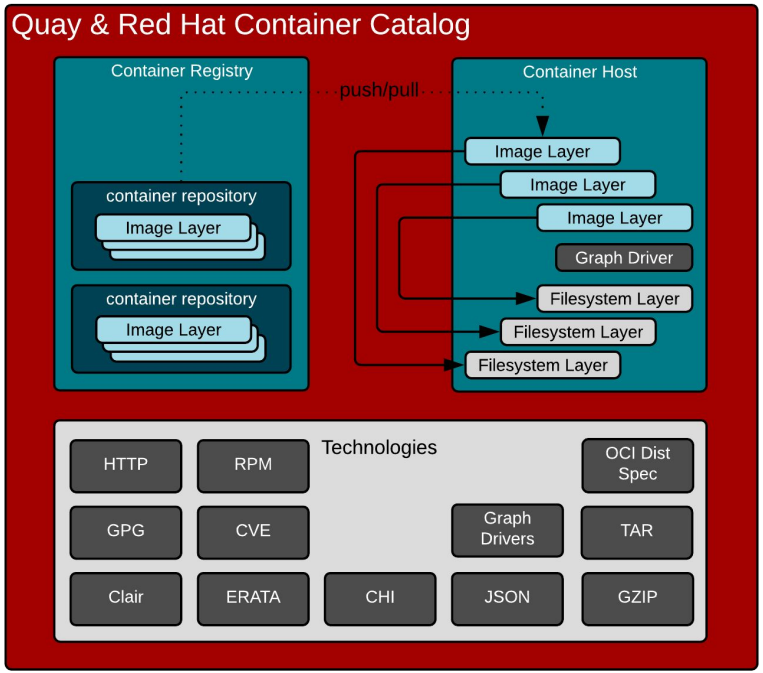


GRAPH DRIVERS

Mapping layers uses file system technology

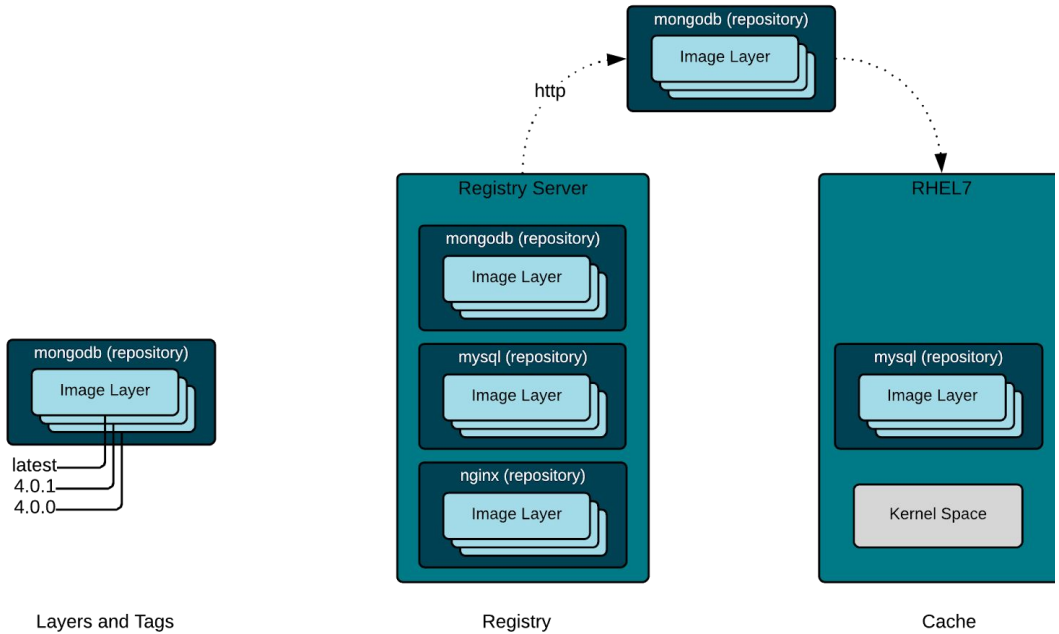
Local cache maps each layer to volume or filesystem layer:

- Overlay2 file system and container engine driver
- Device Mapper volumes and container engine driver



PUSH, PULL & SIGNING

Mapping image layers

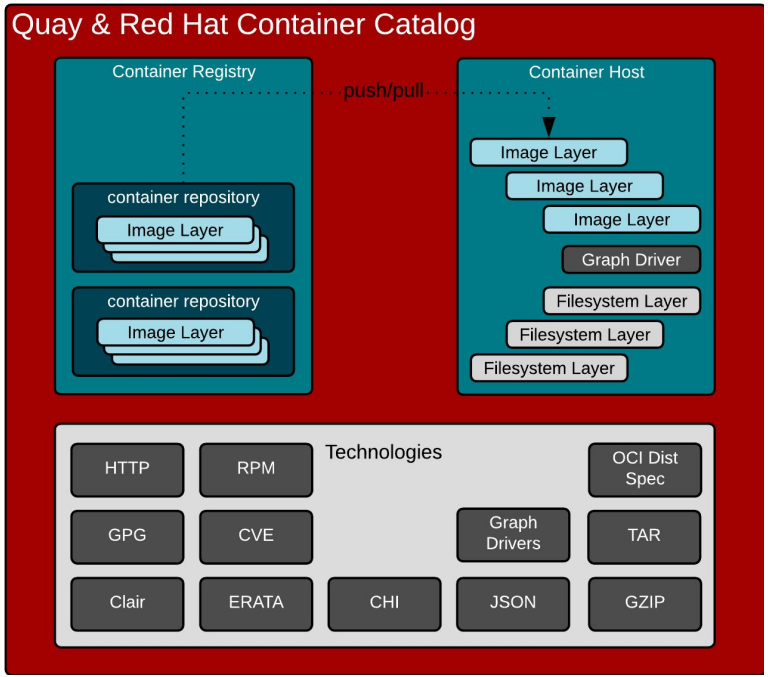


CONTAINER REGISTRY & STORAGE

Mapping image layers

Covering push, pull, and registry:

- Rest API (blobs, manifest, tags)
- Image Scanning (clair)
- CVE Tracking (errata)
- Scoring (Container Health Index)
- Graph Drivers (overlay2, dm)
- Responsible for maintaining chain of custody for secure images from registry to container host



LAB

CONTAINER IMAGES

CONTAINER REGISTRIES

CONTAINER HOSTS

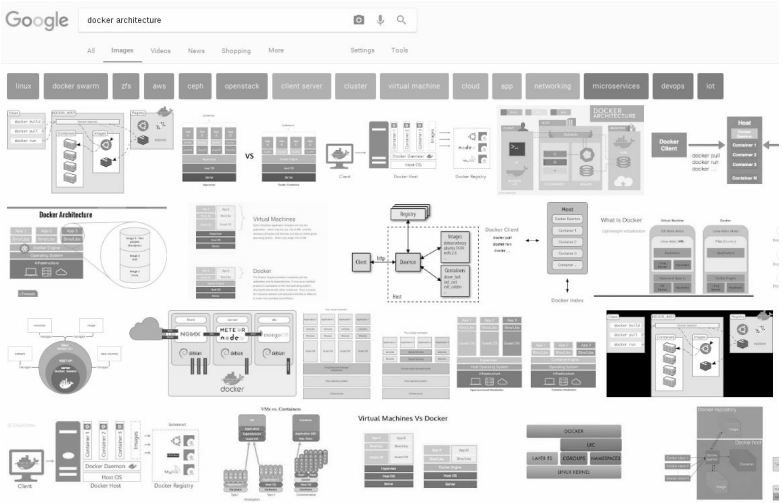
Container Engine, Runtime, and Kernel

CONTAINERS DON'T RUN ON DOCKER

The Internet is WRONG :-)

Important corrections

- Containers do not run ON docker. Containers are processes - they run on the Linux kernel. Containers are Linux processes (or Windows).
- The docker daemon is one of the many user space tools/libraries that talks to the kernel to set up containers

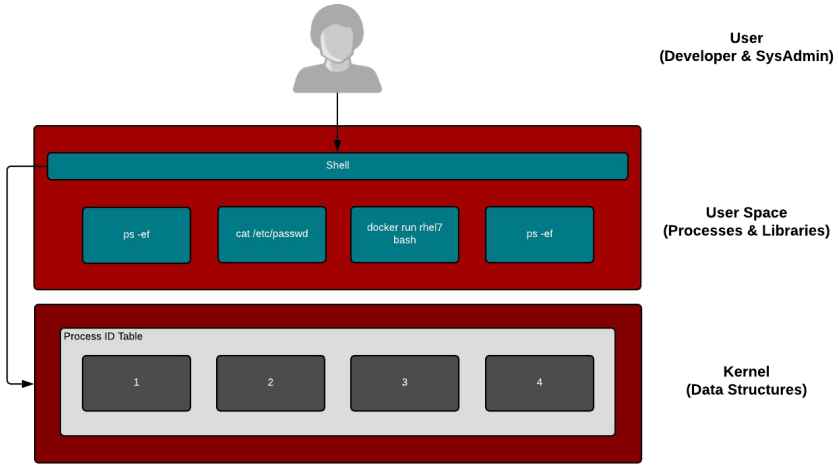


PROCESSES VS. CONTAINERS

Actually, there is no processes vs. containers in the kernel

User space and kernel work together

- There is only one process ID structure in the kernel
- There are multiple human and technical definitions for containers
- Container engines are one technical implementation which provides both a methodology and a definition for containers

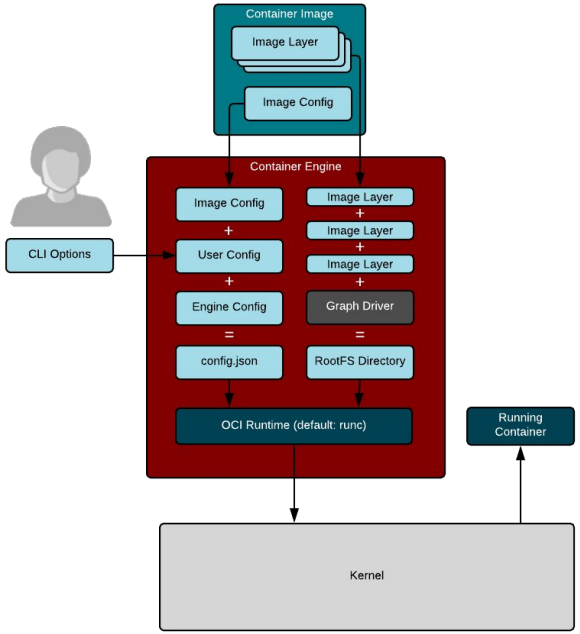


THE CONTAINER ENGINE IS BORN

This was a new concept introduced with Docker Engine and CLI

Think of the Docker Engine as a giant proof of concept - and it worked!

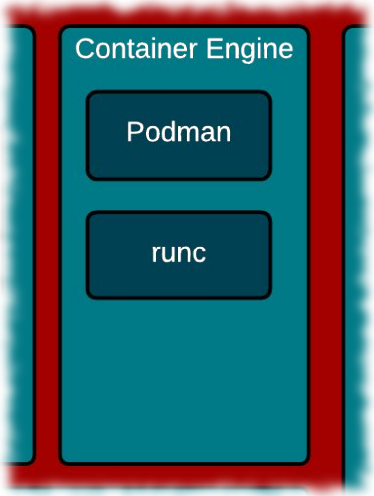
- Container images
- Registry Servers
- Ecosystem of pre-built images
- Container engine
- Container runtime (often confused)
- Container image builds
- API
- CLI



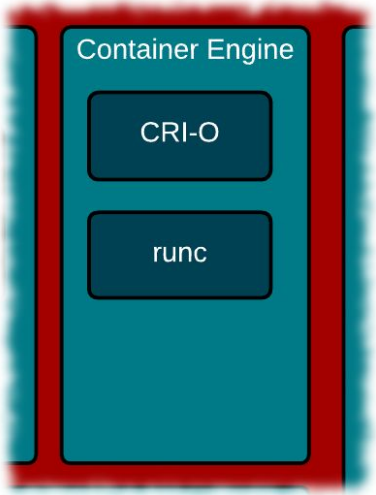
A LOT of moving pieces!

DIFFERENT ENGINES

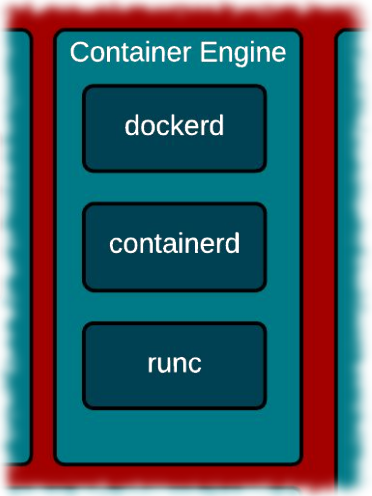
All of these container engines are OCI compliant



Podman



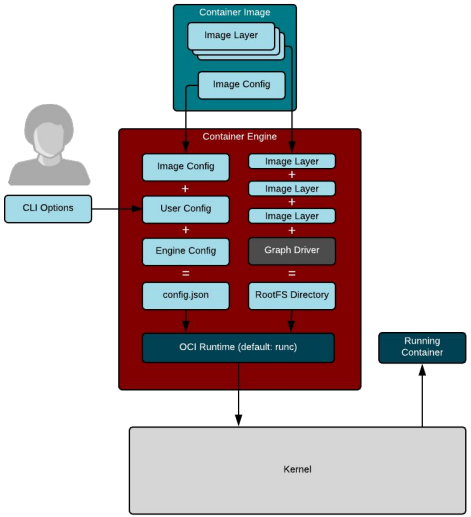
CRI-O



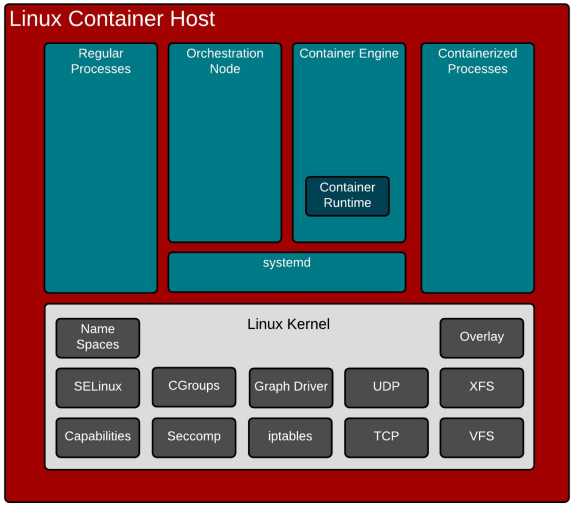
Docker

CONTAINER ENGINE VS. CONTAINER HOST

In reality the whole container host is the engine - like a Swiss watch



VS.

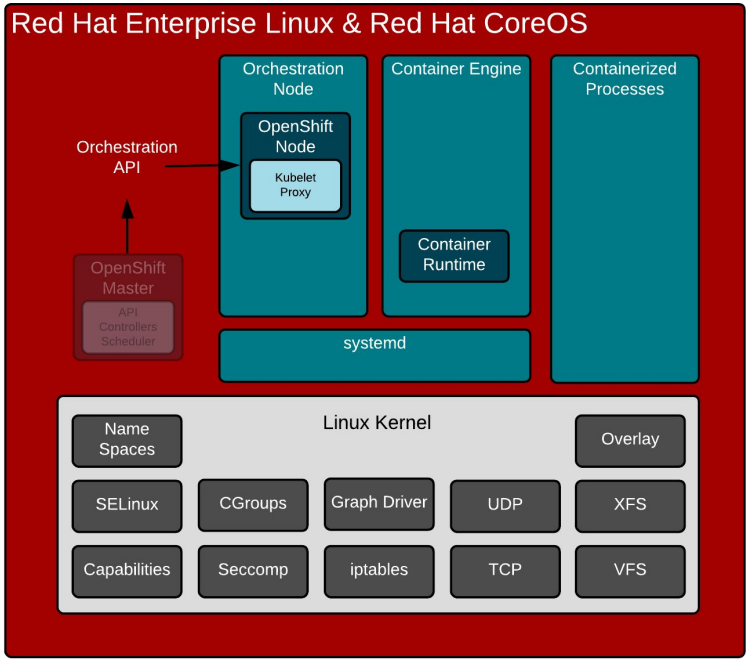


CONTAINER HOST

Released, patched, tested together

Tightly coupled communication through the kernel - all or nothing feature support:

- Operating System (kernel)
- Container Runtime (runc/crun)
- Container Engine (CRI-O)
- Orchestration Node (Kubelet)
- Whole stack is responsible for running containers



KERNEL

Defining a container

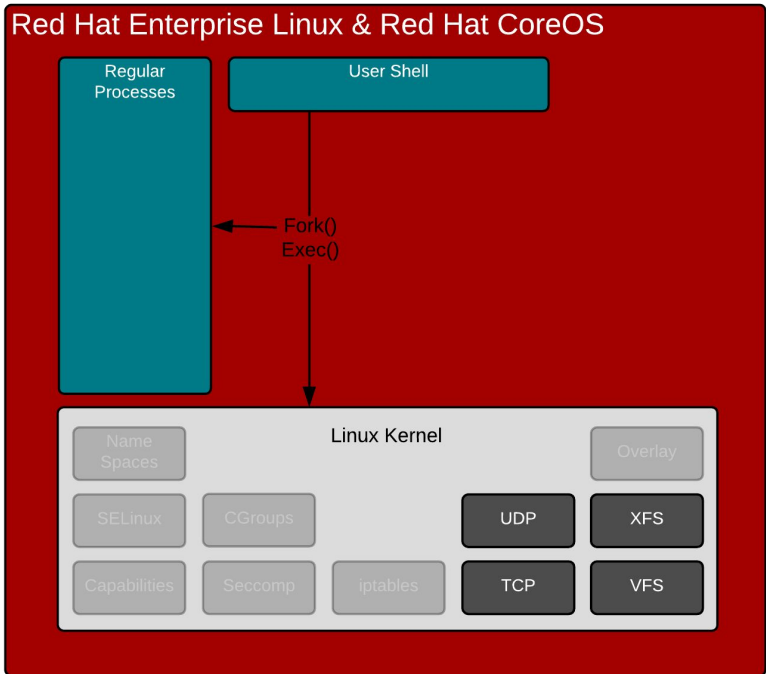
Red Hat

KERNEL

Creating regular Linux processes

Normal processes are created, destroyed, and managed with system calls:

- Fork() - Think Apache
- Exec() - Think ps
- Exit()
- Kill()
- Open()
- Close()
- System()

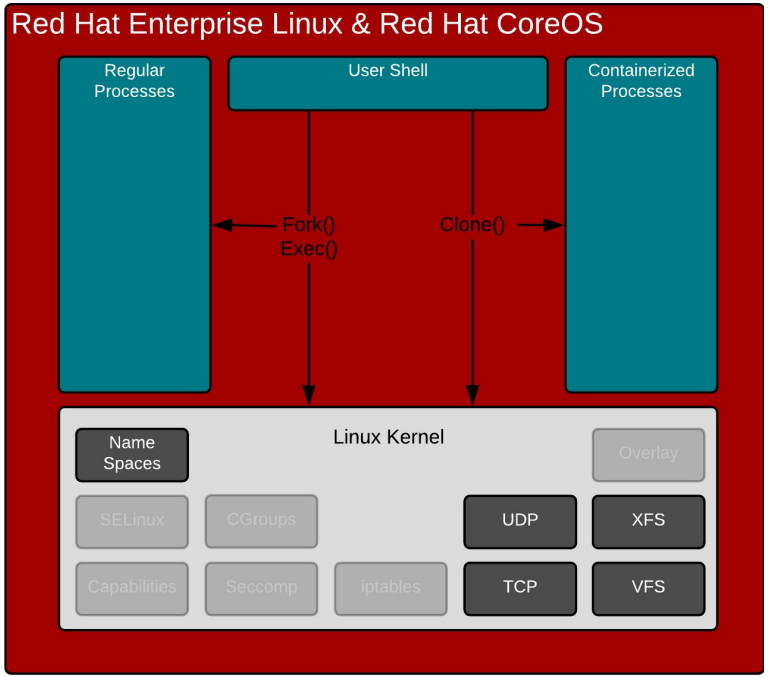


KERNEL

Creating “containerized” Linux processes

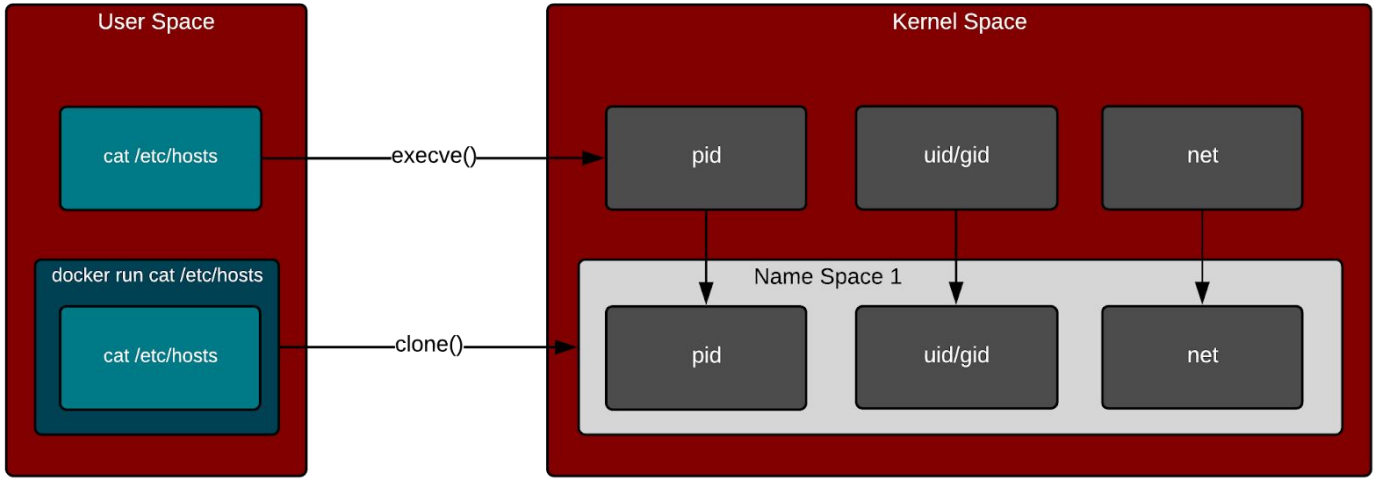
What is a container anyway?

- No kernel definition for what a container is - only processes
- Clone() - closest we have
- Creates namespaces for kernel resources
  - Mount, UTC, IPC, PID, Network, User
- Essentially, virtualized data structures



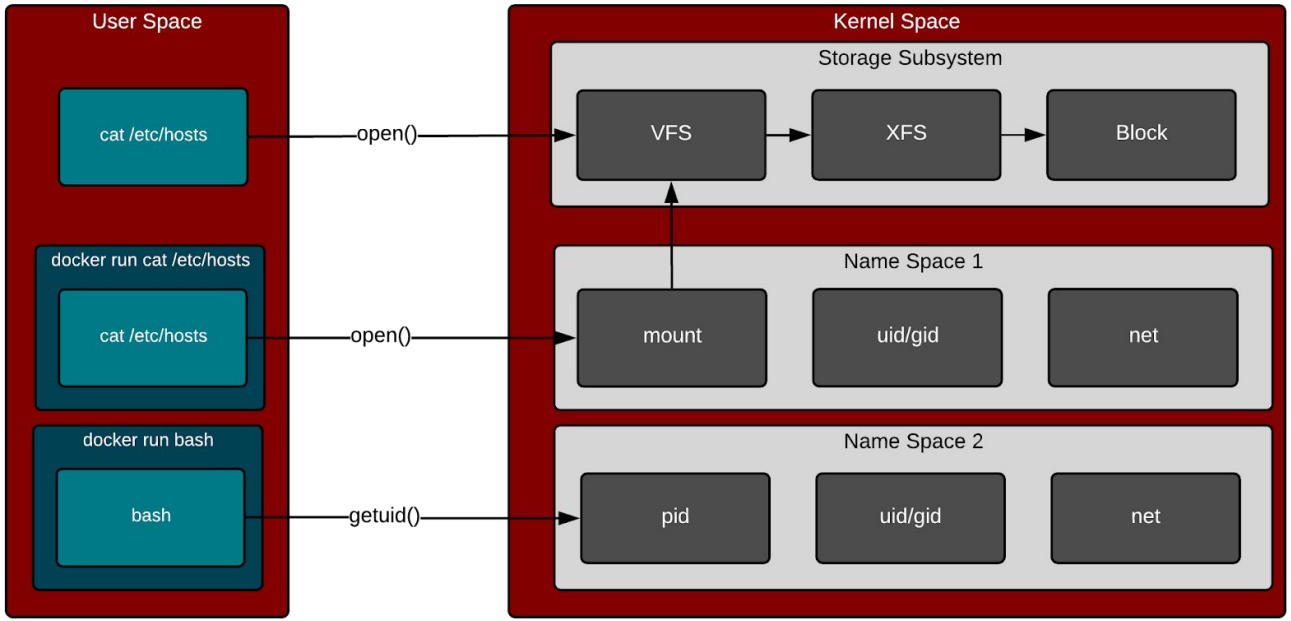
KERNEL

Namespaces are all you get with the clone() syscall



KERNEL

Even namespaced resources use the same subsystem code



CONTAINER RUNTIME

Defining a container

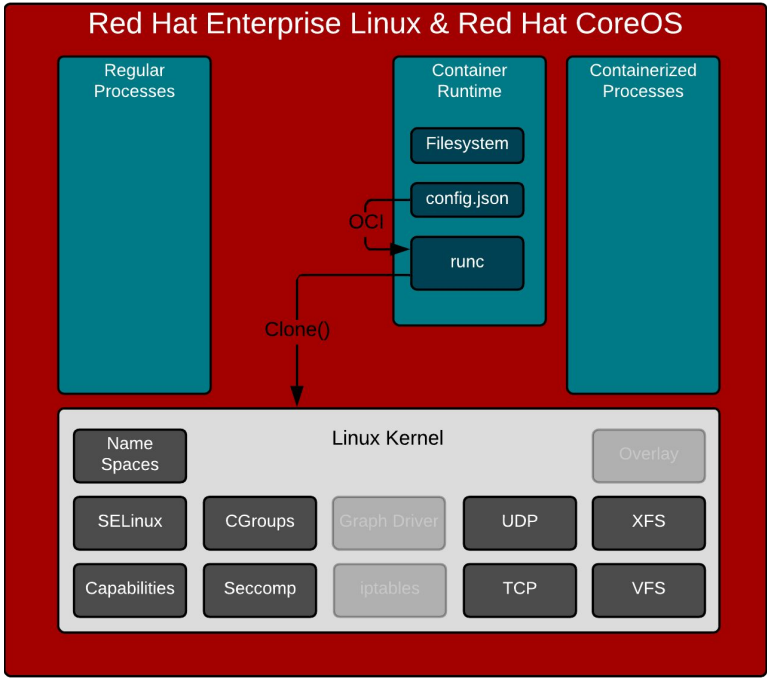
Red Hat

CONTAINER RUNTIME

Standarding the way user space communicates with the kernel

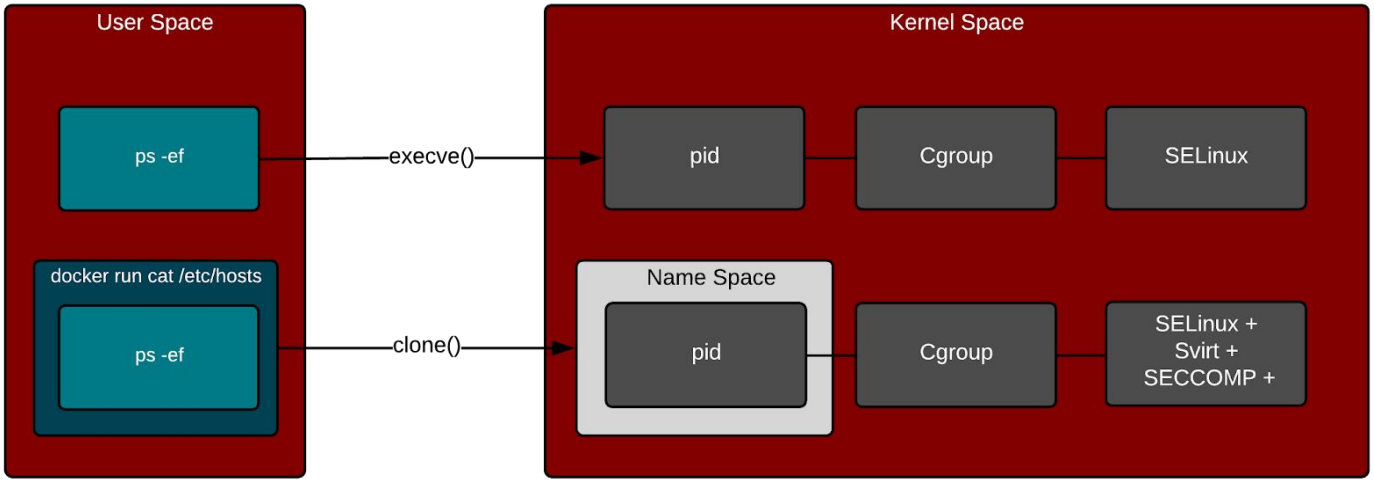
Expects some things from the user:

- OCI Manifest - json file which contains a familiar set of directives - read only, seccomp rules, privileged, volumes, etc
- Filesystem - just a plain old directory which has the extracted contents of a container image



CONTAINER RUNTIME

Adds in cgroups, SELinux, sVirt, and SECCOMP

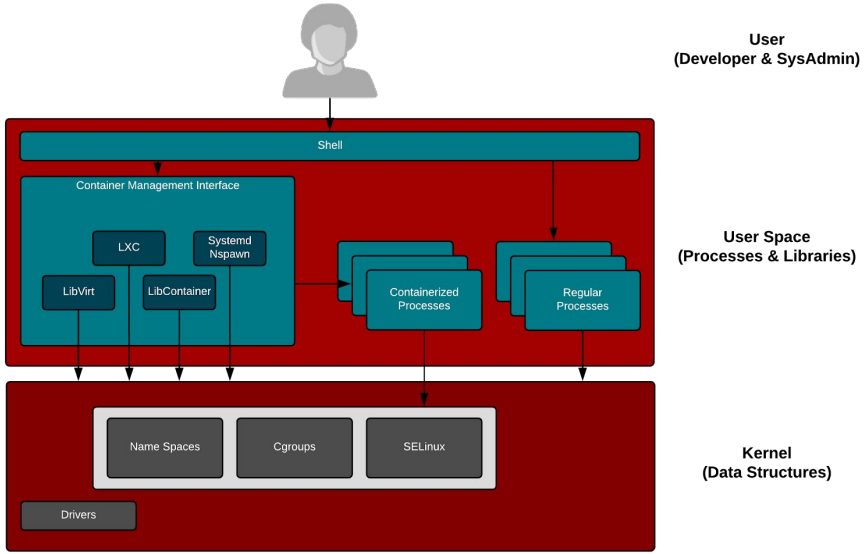


CONTAINER RUNTIME

But, there were others before runc, what's the deal?

There is a rich history of standardization attempts in Linux:

- LibVirt
- LXC
- Systemd Nspawn
- LibContainer (eventually became runc)



CONTAINER ENGINE

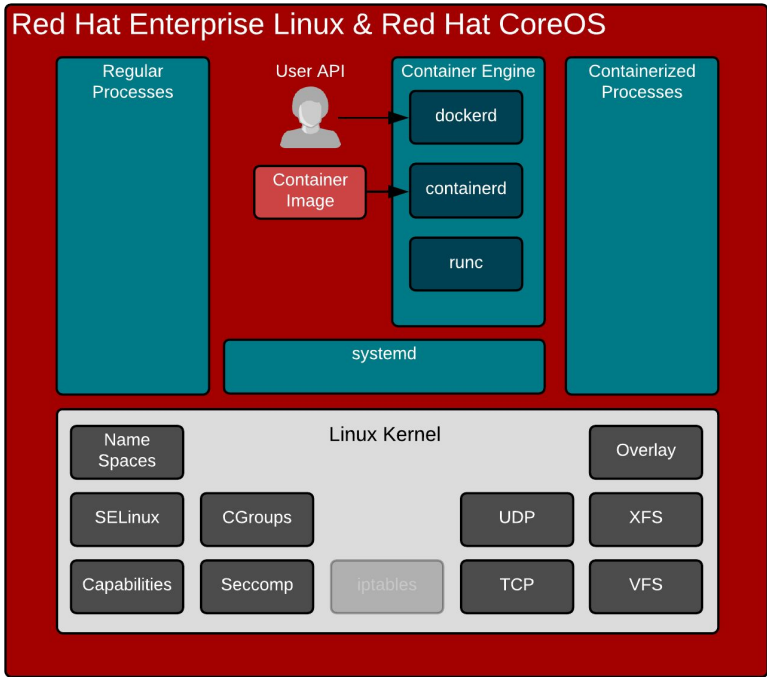
Defining a container

CONTAINER ENGINE

Provides an API prepares data & metadata for runc

Three major jobs:

- Provide an API for users and robots
- Pulls image, decomposes, and prepares storage
- Prepares configuration - passes to runc

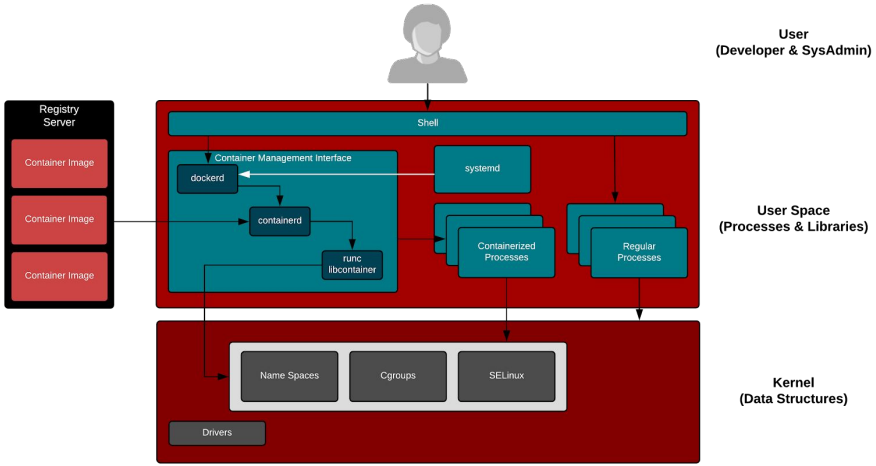


PROVIDE AN API

Regular processes, daemons, and containers all run side by side

In action:

- Number of daemons & programs working together
  - dockerd
  - containerd
  - runc

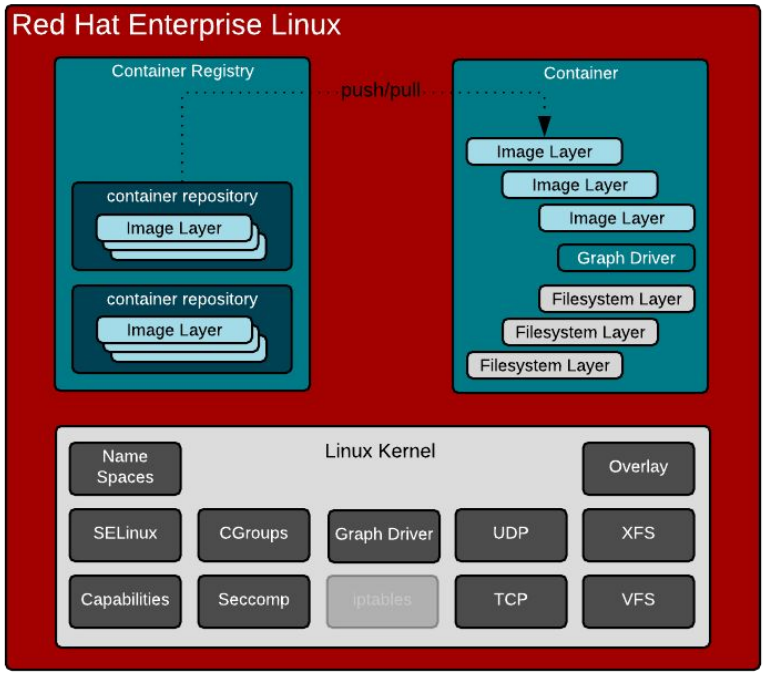


PULL IMAGES

Mapping image layers

Pulling, caching and running containers:

- Most container engines use graph drivers which rely on kernel drivers (overlay, device mapper, etc)
- There is work going on to do this in user space, but there are typically performance trade offs

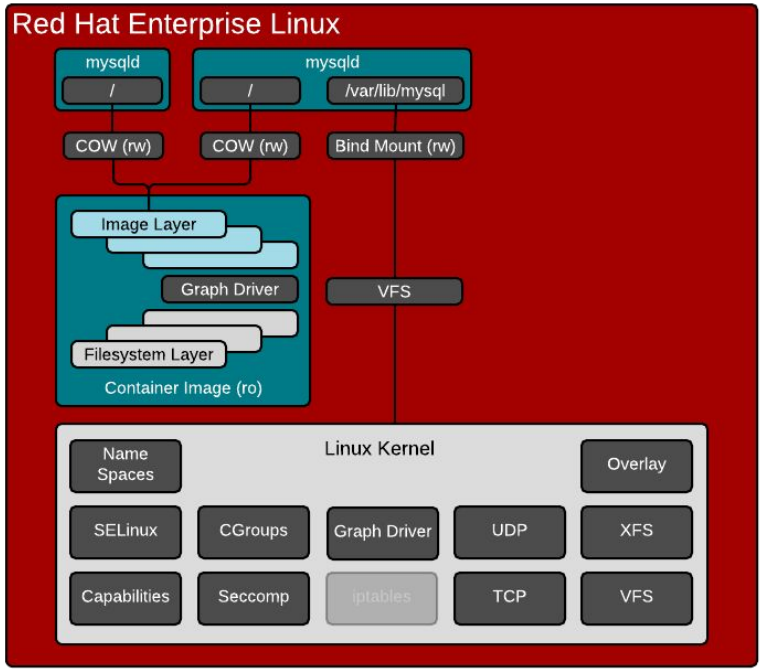


PREPARE STORAGE

Copy on write and bind mounts

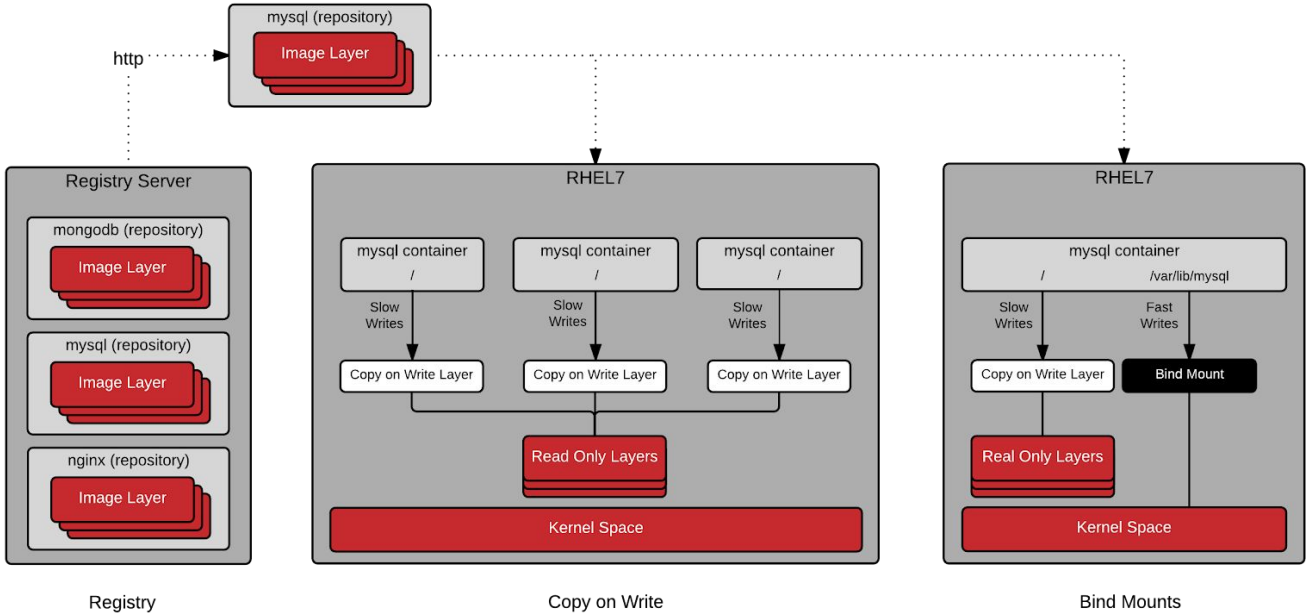
Understanding implications of bind mounts:

- Copy on write layers can be slow when writing lots of small files
- Bind mounted data can reside on any VFS mount (NFS, XFS, etc)



Mounts

Copy on write vs. bind mounts

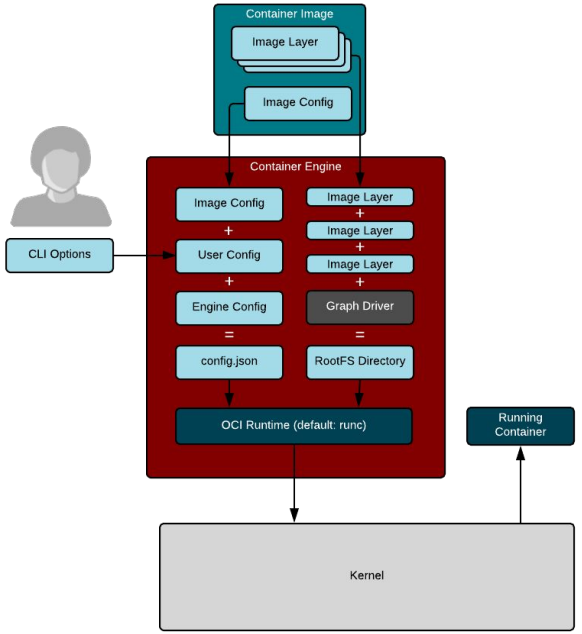


PREPARE CONFIGURATION

Combination of image, user, and engine defaults

Three major inputs:

- User inputs can override defaults in image and engine
- Image inputs can override engine defaults
- Engine provides sane defaults so that things work out of the box

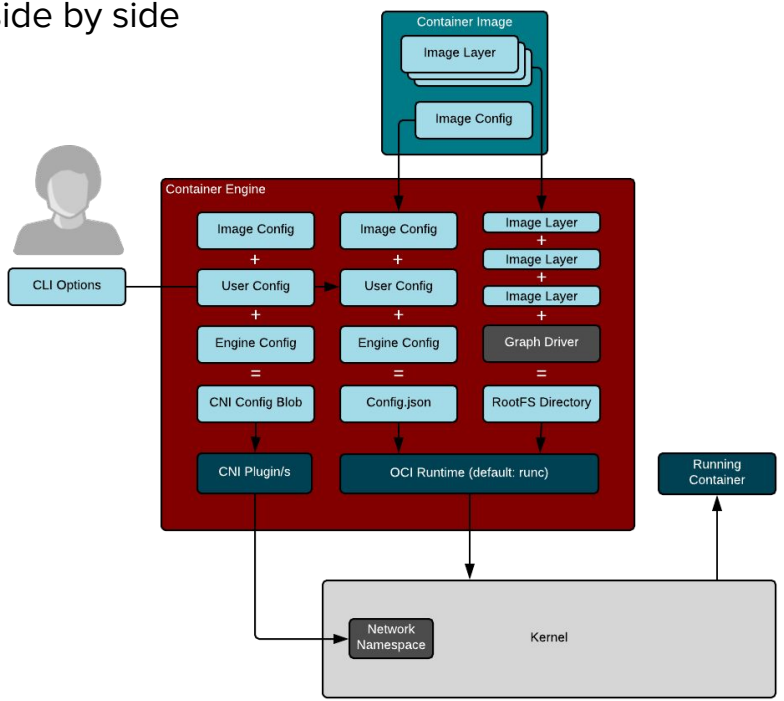


PREPARE CONFIGURATION + CNI

Regular processes, daemons, and containers all run side by side

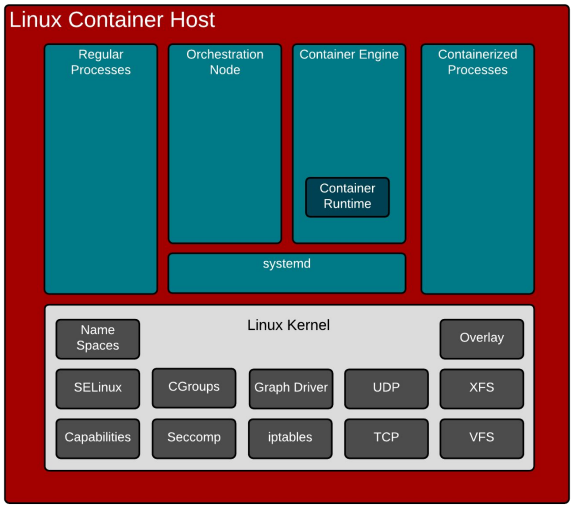
In action:

- Takes user specified options
- Pulls image, expands, and parses metadata
- Creates and prepares CNI json blob
- Hands CNI blob and environment variables to one or more plugins (bridge, portmapper, etc)

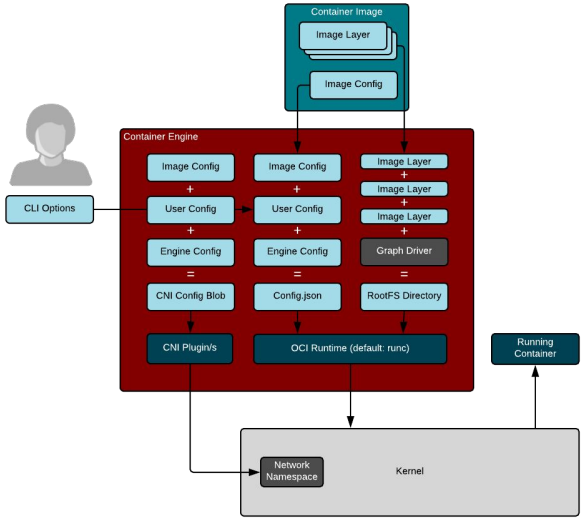


ENGINE, RUNTIME, KERNEL, AND MORE

All of these must revision together and prevent regressions together

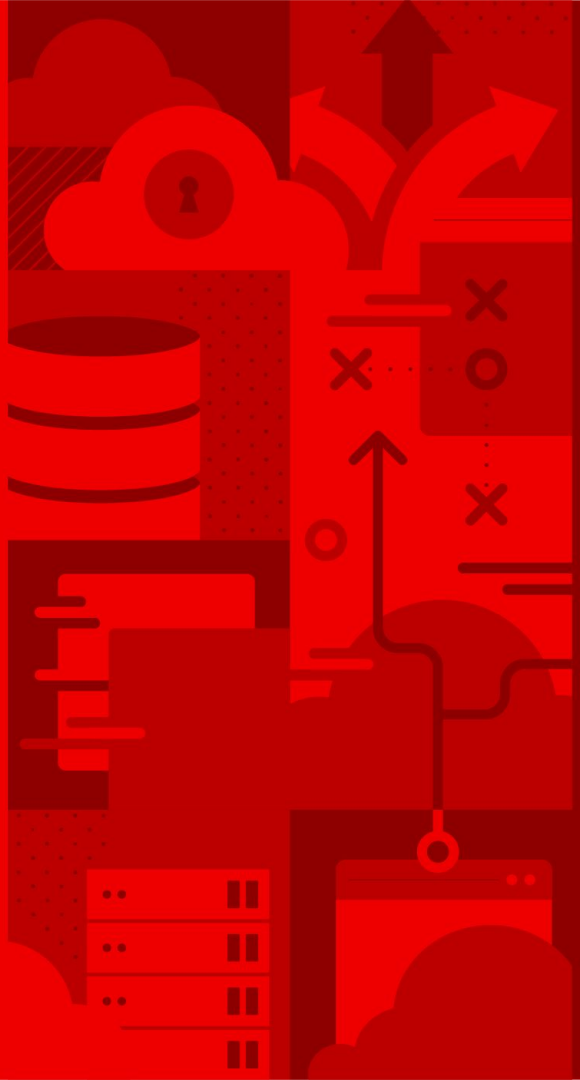


+



LAB

CONTAINER HOSTS



CGroups - Managing Access To Resources

What Are CGroups & Why You Should Care

- Linux Kernel is responsible for all hardware interaction
- i.e. the amount of RAM a system has to divide up amongst processes
- This is great when all applications play well together
- But what if they don't?
    - OOM Killer!

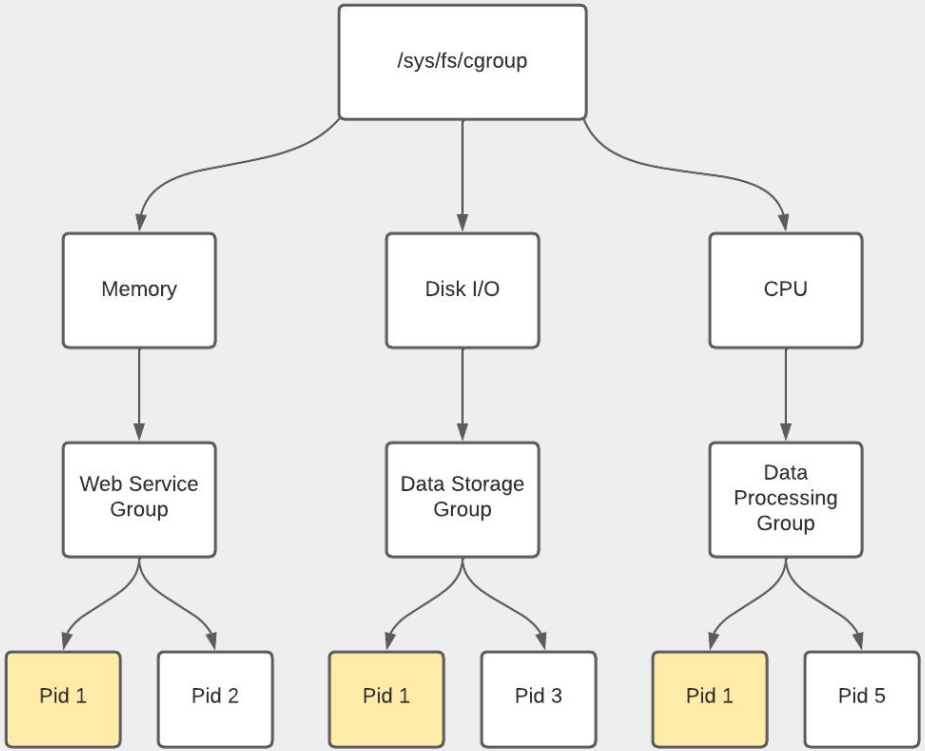
What Are CGroups & Why You Should Care

- Kernel facility which provides access to hardware
- You can limit the amount of a given resource an application can consume
- CGroups control:
  - The number of CPU shares per process.
  - The limits on memory per process.
  - Block Device I/O per process.
  - Which network packets are identified as the same type so that another application can enforce network traffic rules.

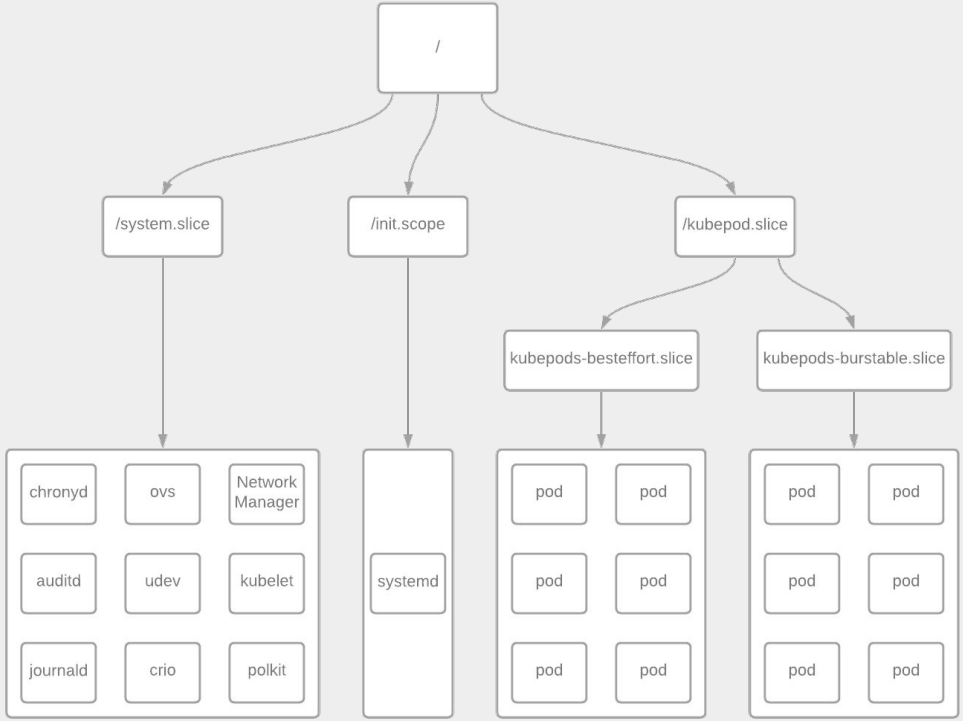
4 Main Tasks of CGroups

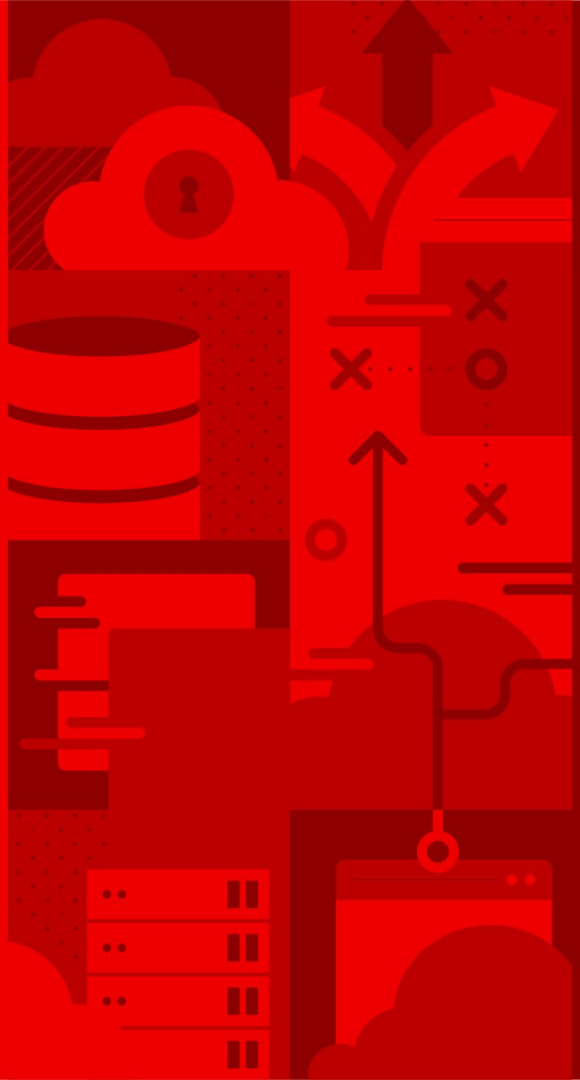
- Resource Limiting
- Prioritization
- Accounting
- Process Control

How Do CGroups Work?



How Do Shares Work?





Linux Capabilities

What Is This In Context of OCP

- We need to run “stuff”
- Don’t want to run as root
- Regular users might not be good enough
- Container cannot “sudo”
- Capabilities = Permission Model for containers

Traditional DAC on *nix

- Traditional DAC is through users/groups
- setuid and setgid bits can be set on binaries to provide more access

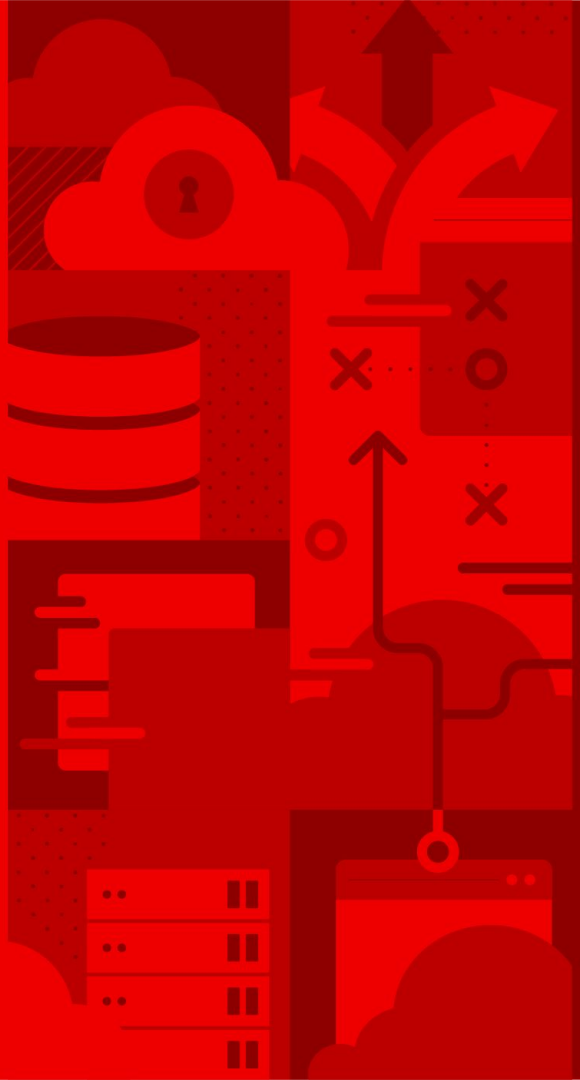
  (i.e. passwd util)
- Capabilities break permissions down into 40 different categories

Capabilities - Granular Control

- As of Linux Kernel 5.15 there are 40 capabilities
- Capabilities are tied to a user namespace
- Each namespace will have its ***own*** capabilities that only apply to themselves

Capabilities Sets - Applied Capabilities

- A *capability set* determines how a capability can be assigned to a thread
- There are 5 in total but we really only care about 2
- **Effective:** These are capabilities that are active. You are allowed to perform this action
- **Permitted:** These are not yet active but the process can decide to elevate its permissions into **effective** capabilities



Namespaces: User

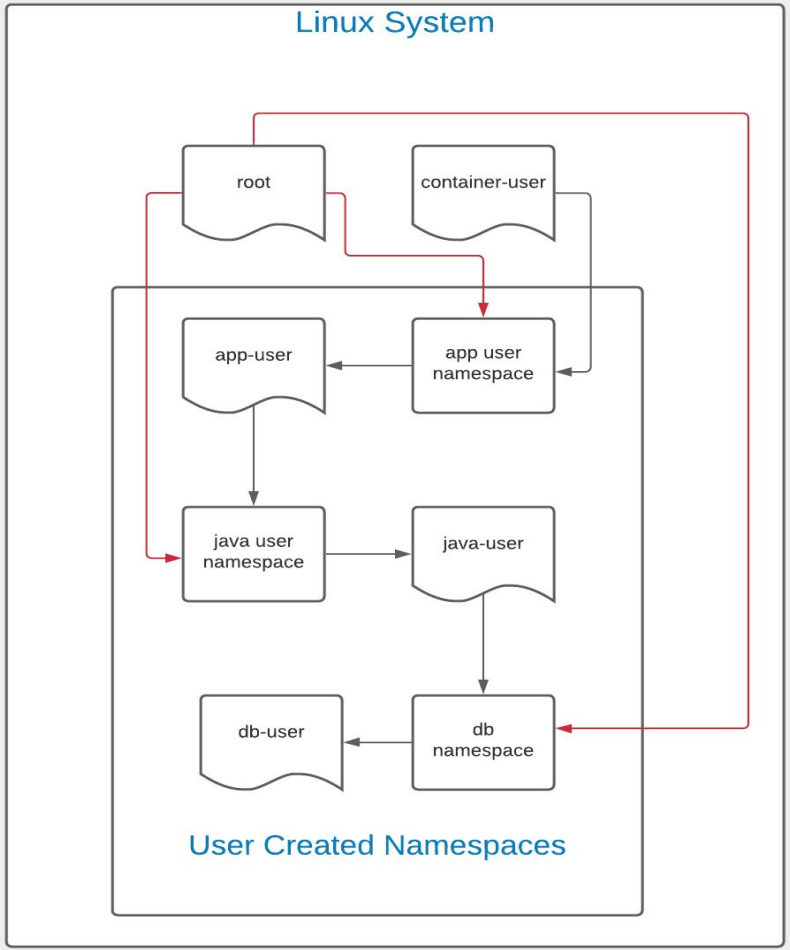
What Is This In Context of K8S

- Processes are associated with users
- What if I need the same user for lots of “stuff”
- What if I want a random UID
- How are permissions assigned per pod?
- User namespace = Container UID mapping

Theory

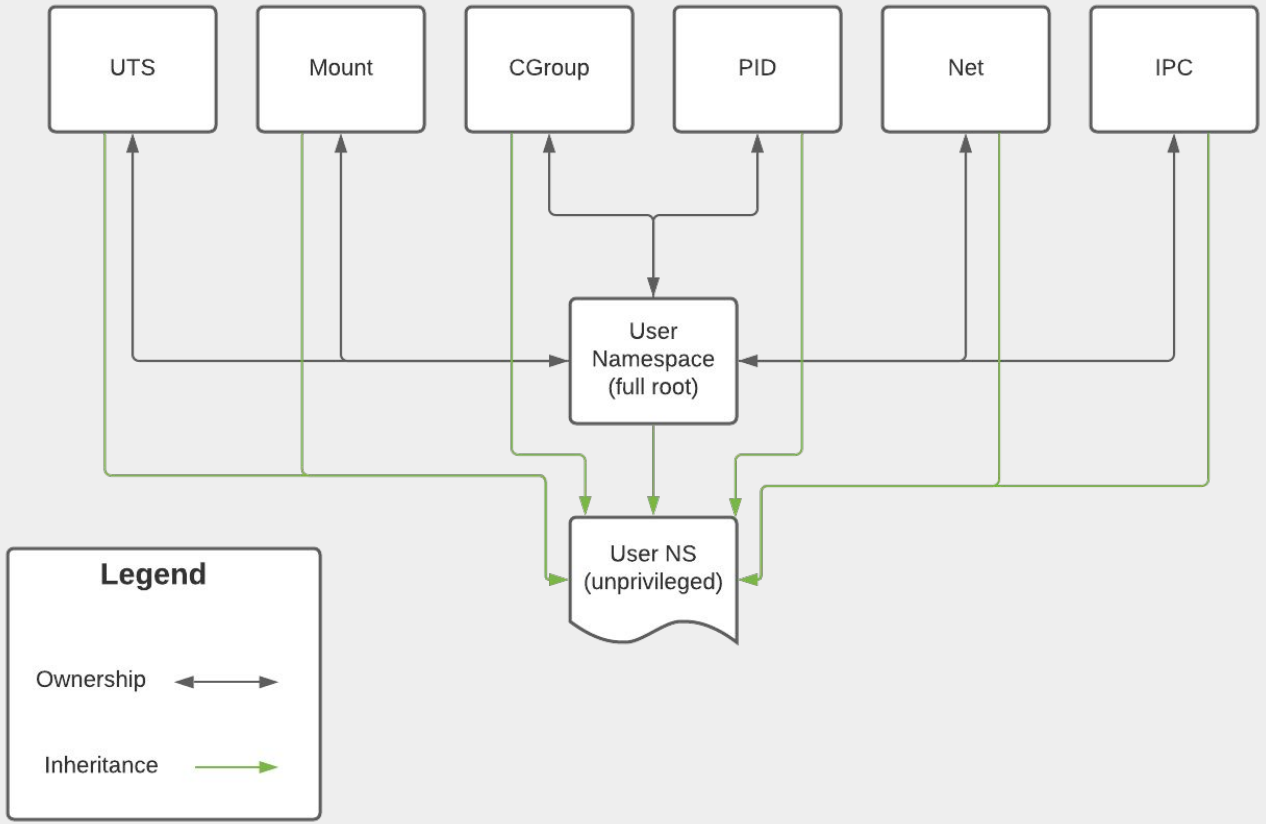
- As discussed UIDs are part of the Linux security strategy
- The user namespace is what allows the container to have a different set of permissions than the host itself
- Every container inherits permissions from the user who created the namespace

Overview



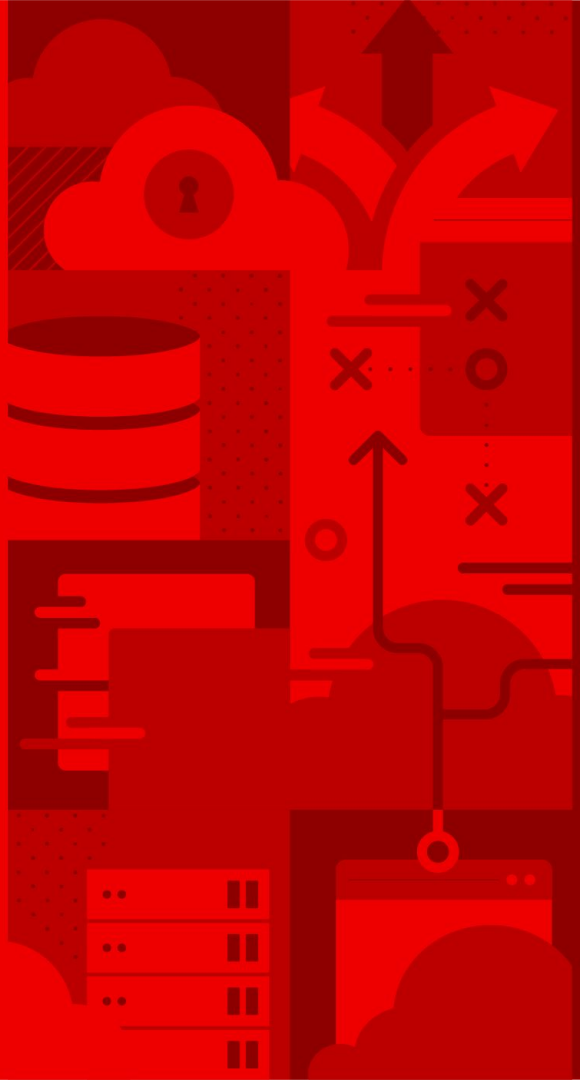
Important Notes

- By default there is no user mapping into a new namespace
  - This results in the user being given the id of **65534(nobody)**
  - It is possible to create a detailed UID mapping before hand
  - There is a flag that will do a root user mapping for you


UTS
Mount
CGroup
PID
Net
IPC
User Namespace (full root)
User NS (unprivileged)
Legend
Ownership
Inheritance

Summary

- Important… especially from permissions inheritance
- explains file permissions in containers
- they are most effective in combination



Namespaces: Mount

What Is This In Context of K8S

- Containers need discrete FS
- Should only access their own files
- “chroot” not good enough
- **Mount Namespace = Container File System separation**
- Container Overlay Directory = Mount NS root

Theory

- Creating a new mount space creates a copy of all mount points at the time the namespace is created
- Poorly configured **mnt** namespaces **WILL** impact the host
- Mount propagate by default because of a kernel feature called **shared subtree**
- There is metadata on each mount point that determines whether it will get propogated

Theory - Peer Groups

- **peer group** → a group of vfsmounts that propagate events to each other
  - Events → mounting a network share or unmounting an optical device
- Groups determine what can be seen

Theory - Mount State

- **Mount state** determines which groups receive events
- States are *per mount point*
- There are 5
- Container engines use **private** mount points

Theory - chroot?

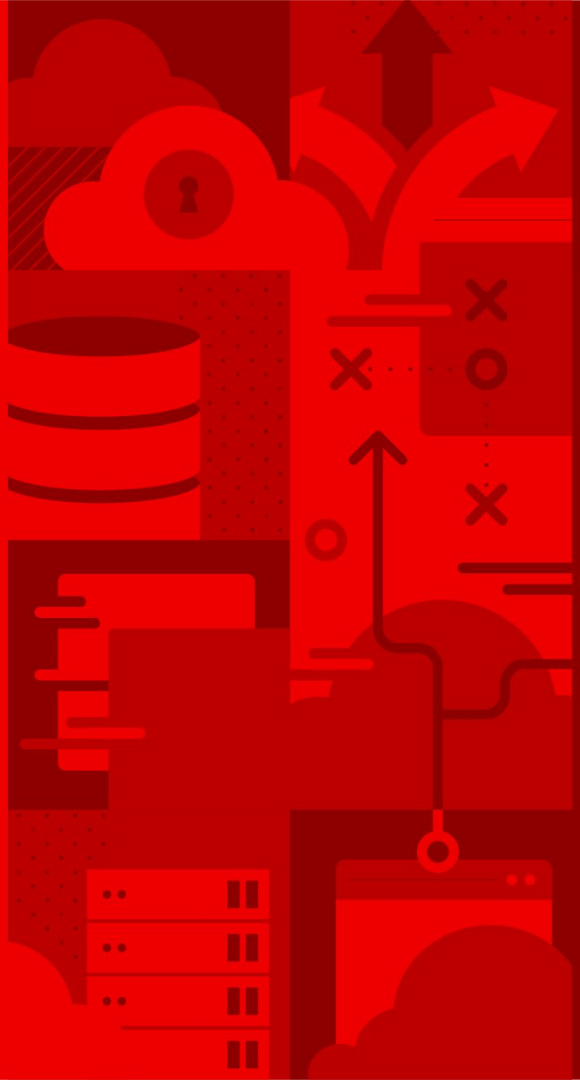
- chroot is often thought of as having extra security benefits.
- chroot can be very secure
  - chroot does restrict Linux capabilities
- chroot does not limit system calls to the kernel.
  - potent to escape a chroot

Hold On…

- I thought MNT was supposed to protect me?!?
- How do containers use MNT?
  - pivot_root → Unmount the root file system

Important Notes

- **/proc** is not included in the **mnt** namespace
  - it's a special file system that is tied into the PID namespace
- Because of Linux Capabilities in the user namespace you cannot mount **/proc**



Namespaces: Net

What Is This In Context of K8S

- Containers need their own IPs
- Containers should not impact other network functions
- Might want VPN or other similar tech
- **Net Namespace = Discrete networking for containers**

Theory

- Net - Controls Networking Stack
- Primary controls via the following commands:
  - *ip link*
  - *ip netns*
- You need a virtual switch of some sort for anything interesting

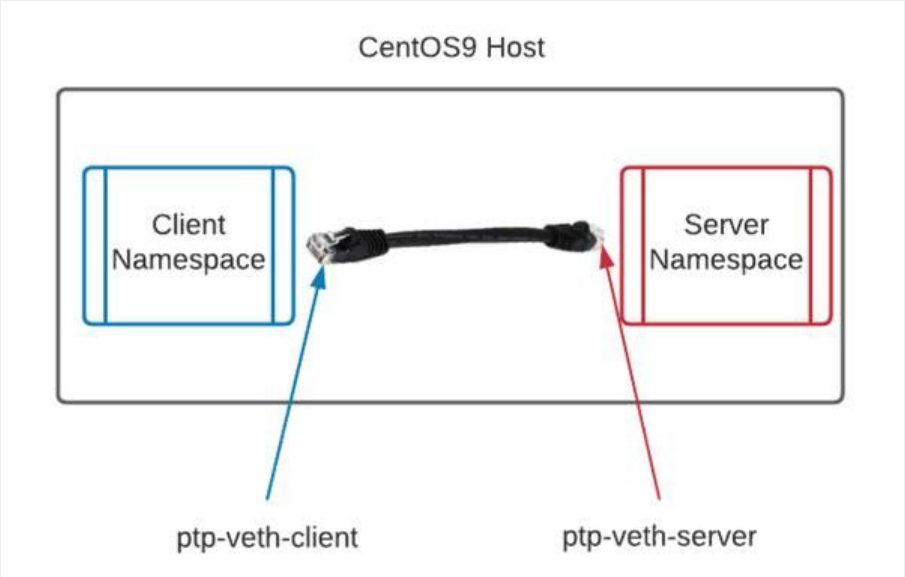
Theory

- Users Can Create But Not Modify Interfaces In Other Namespaces
- Can Segregate:
  - VPNs
  - DNS/DHCP
  - Host From Outside World

Theory

- We are going to create 2 namespaces
- Using a point to point link or a “crossover cable”
- It will look like this:

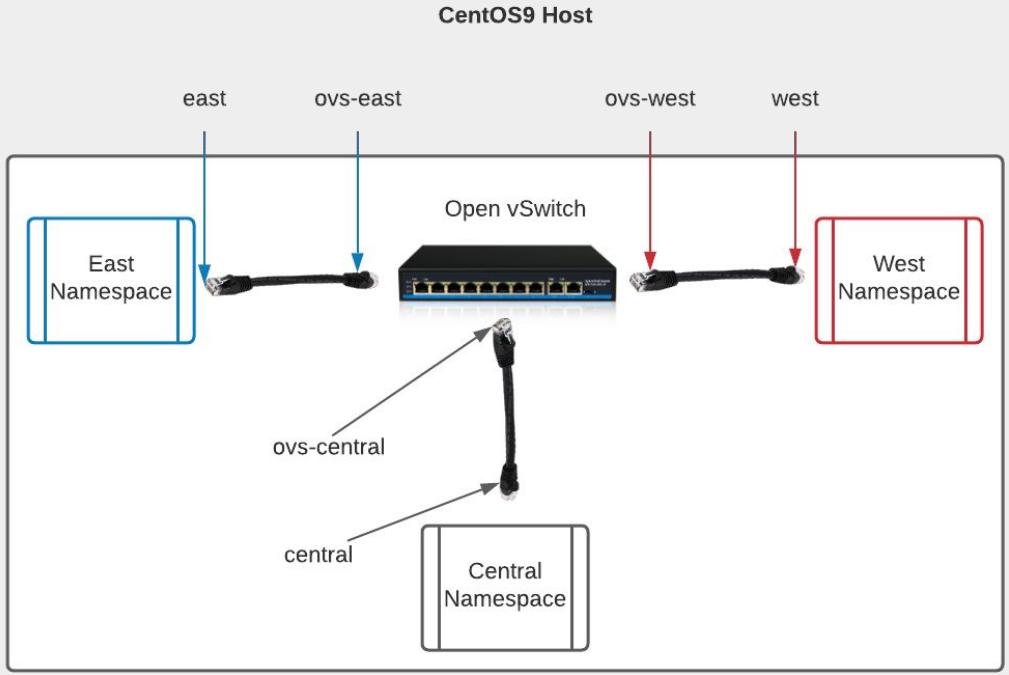
Theory



Theory

- Point to point is interesting…
- How does Kubernetes Work?
- Like actual networking, we need a switch

Theory



Remember…

- Containers need their own IPs
- **Net Namespace = Discrete networking for containers**
- Containers should not impact other network functions

WRAPPING UP

A walk down memory lane

Red Hat

THE HISTORY OF CONTAINERS

1979:
UNIX
CHROOT
SYSCALL ADDED

2000:
JAILS ADDED
TO FREEBSD

2001:
LINUX -VSERVER
PROJECT

2003:
SELINUX
ADDED TO LINUX
MAINLINE

2005:
FULL RELEASE
OF SOLARIS
ZONES

Cpusets
/
sys staff students
2006:
PROCESS
CONFINEMENT

2007:
GPC RENAMED
CGROUPS

2008:
LINUX
CONTAINER
PROJECT (LXC)

2008:
KERNEL & USER
NAMESPACES

2013:
DOTCLOUD PYCON
LIGHTNING TALK

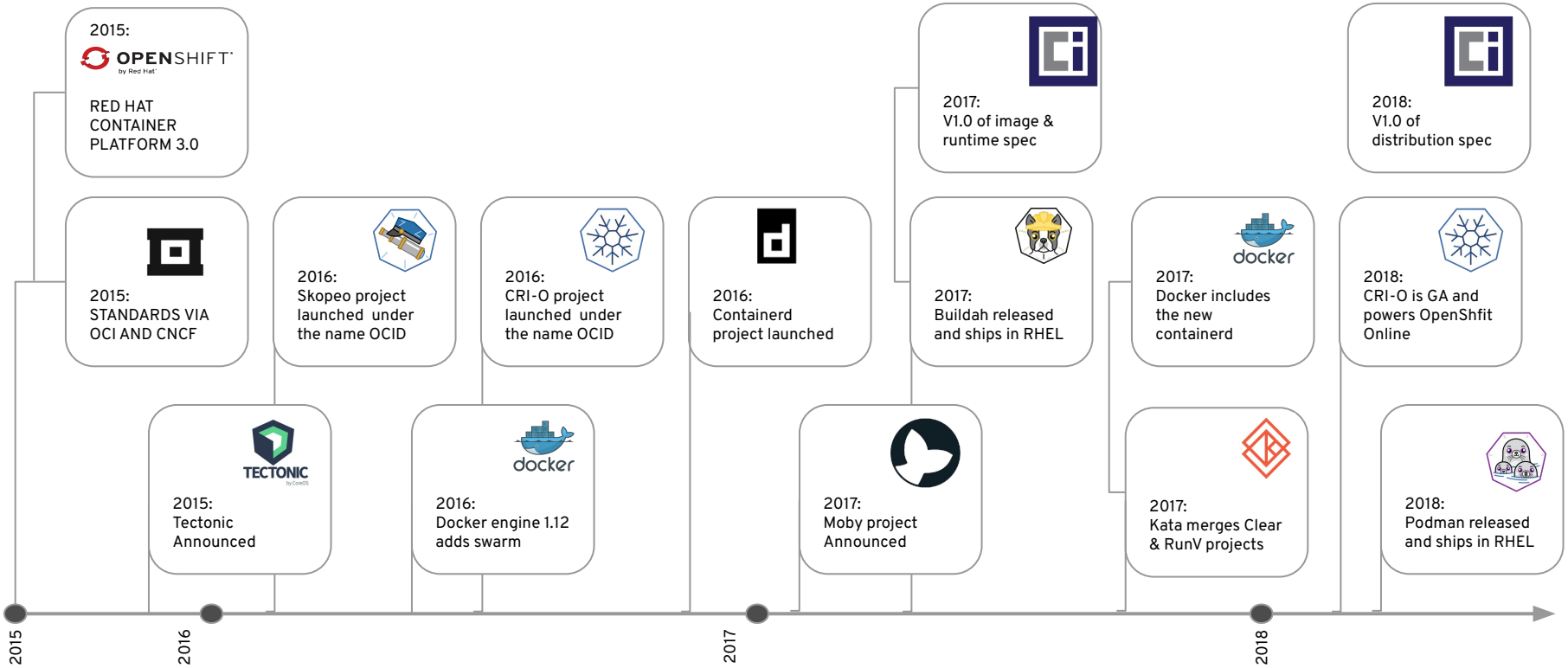
2013:
DOTCLOUD
BECOMES
DOCKER

2013:
RED HAT
ENTERPRISE
LINUX

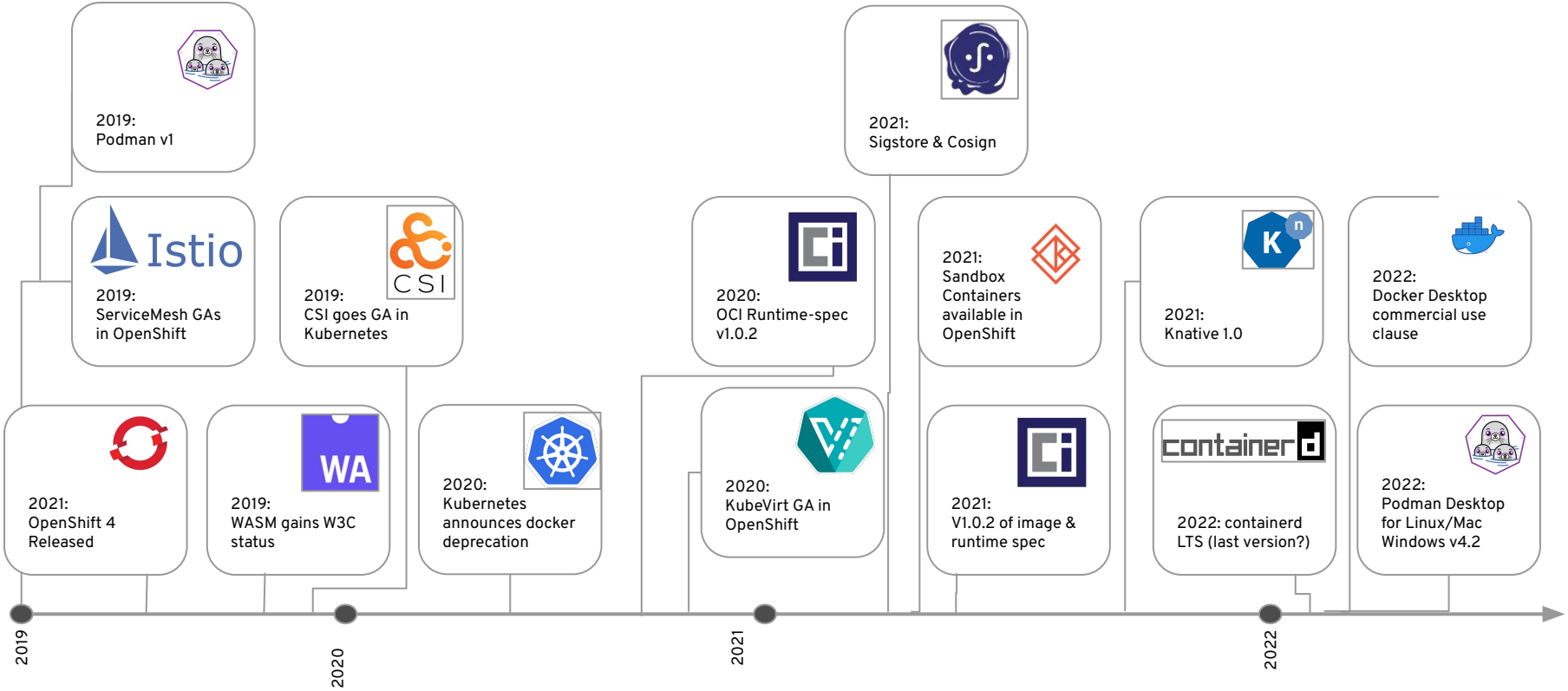
2014:
GOOGLE
KUBERNETES

1979 2000 2005 2010

CONTAINER INNOVATION ...



CONTAINER INNOVATION IS NOT FINISHED



Choose your own adventure!

If there is time, do them both!

LAB 9 - Down the Stack

- User Namespace
- Mount Namespace
- Network Namespace
- Cgroups

LAB 5 & 7 - Up the Stack

- Distributed Containers (OpenShift)
- Container Tooling
  - Podman
  - Buildah
  - Skopeo
  - Criu
  - Udica
  - oscap-podman

THANK YOU


linkedin.com/company/red-hat


youtube.com/user/RedHatVideos


facebook.com/redhatinc


twitter.com/RedHat

Red Hat

How did we do?

red.ht/olf-survey

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat

Sigstore

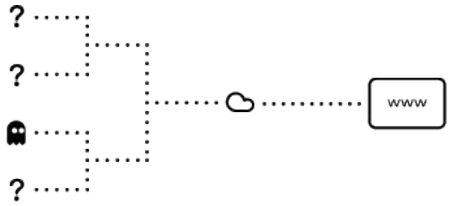
Artifact Signing for a Software Factory

Bill Bensing Managing Architect

The Problem

Open Source Security

Not knowing where all your software comes from means hard-to-spot risks to the integrity of your services. Without constant identity checks and safety protocols for keys and secrets, open source dependencies can open the door to breaches, exploits and supply chain attacks.

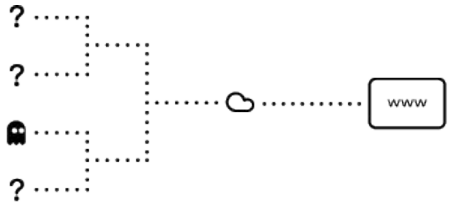


The Opportunity

Open Source Security

Improve supply chain technology for anyone using open source projects. It's for open source maintainers, by open source maintainers.

And it's a direct response to today's challenges, a work in progress for a future where the integrity of what we build and use is up to standard.



The Solution

SigStore

We've automated how you digitally sign and check components, for a safer chain of custody tracing software back to the source. We want to remove the effort, time and risk of error this usually comes with. And for anyone whose software depends on open source, future integrations can make it easier to check for authenticity, wherever it's come from.

Why SigStore

The Key Capabilities

Automatic Key Management

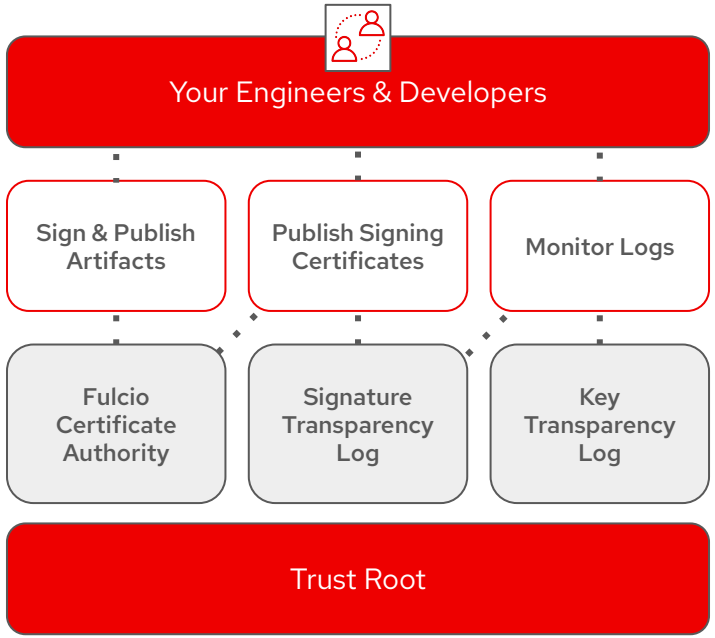
We use SigStore to generate the key pairs needed to sign and verify artifacts, automating as much as possible so there's no risk of losing or leaking them

Transparent Ledger Technology

A transparency log means anyone can find and verify signatures, and check whether someone's changed the source code, the build platform or the artifact repository.

SigStore For Your Organization

Bring These Capabilities To Your Organization



How Do You Use SigStore

The Key Capabilities

Sign Artifacts

Easy authentication and smart cryptography work in the background. Just push your code.

Verify Signatures

A transparency log stores data like who created something and how, so you know it hasn't been changed.

Monitor Activity

Logged data is readily auditable, for future monitors and integrations to build into your security workflow.

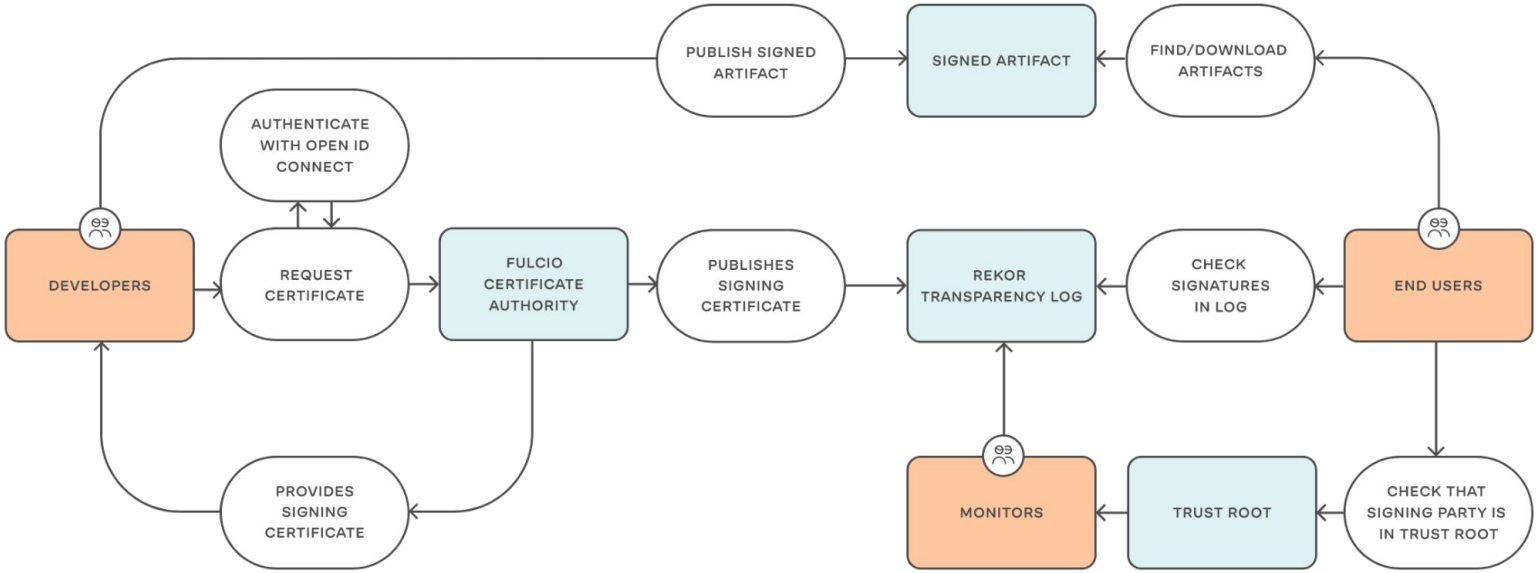
What's Behind The Scenes?

The Parts & Pieces Of SigStore

- **Cosign (Soon Within Podman)**
  - For container signing, verification and storage in an Open Container Initiative (OCI) registry, making signatures invisible infrastructure.
- **OpenID Connect**
  - An identity layer that checks if you're who you say you are. It lets clients request and receive information about authenticated sessions and users.
- **Certificate Authority**
  - A mechanism that generates certificates, binding cryptographic keys to an identity and an independent check over an artifact's information.
- **Rekor**
  - A built in transparency and timestamping service, Rekor records signed metadata to a ledger that can be searched, but can't be tampered with.
- **Fulcio**
  - A free root certification authority, issuing temporary certificates to an authorized identity and publishing them in the Rekor transparency log
- **Trust root**
  - The foundation for trust behind the whole of sigstore, our keyholders and ways of working to protect the root keys.

The Organizational Experience

Fitting This All Together

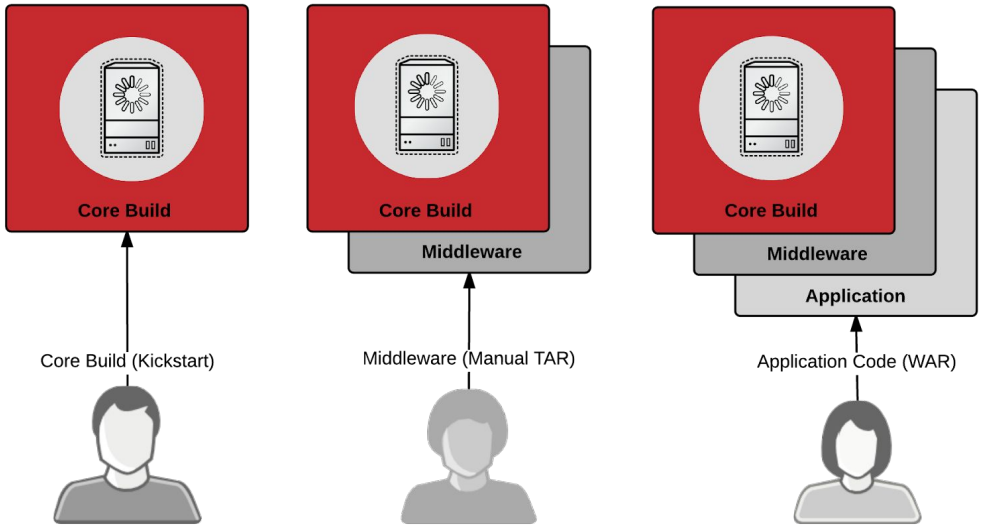



Cosign
cosign:demo dlorenc$

PRODUCTION IMAGE BUILDS

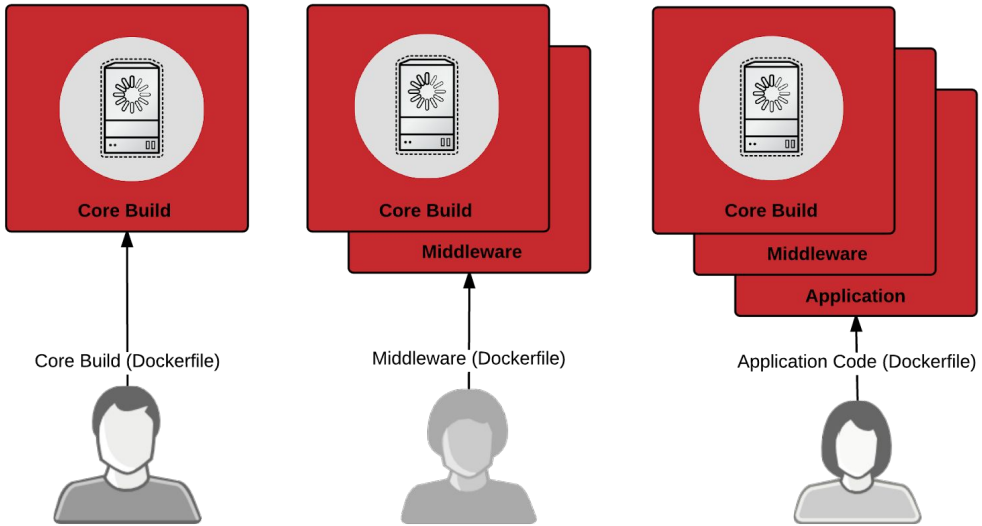
Fancy Files

How do we currently collaborate in the user space?



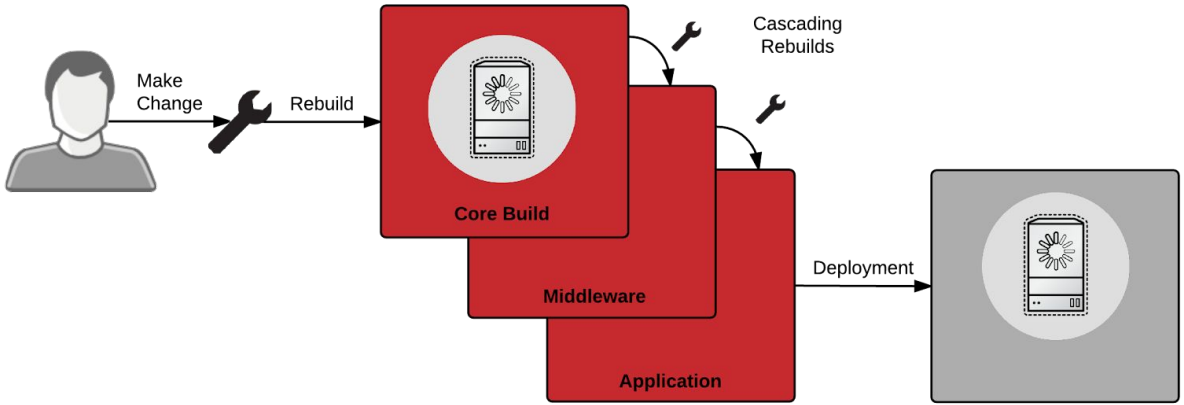
Fancy Files

The future of collaboration in the user space....



Fancy Files

The future of collaboration in the user space....



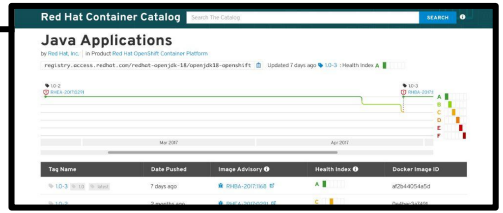
A Software Factory

Automated quality and security: because you can't inspect quality into a product

Automatically prohibit untrusted containers via Openshift policy

Trusted repos
registry.redhat.io

Private registry
- **RH Quay**
- JFrog Artifactory
- Sonatype Nexus



Secure Supply Chain of Quality Parts
- Quality Assurance
- Certifications
- Signing & Secure
- Distribution

CCB
RAPID DELIVERY

REQ → DEV → UNIT TEST → CODE QUAL → SEC SCAN → INT TEST → QA UAT → PROD

-**RH Dev Spaces**
Dependency
Analytics
- Snyk

-Cucumber
-Arquillian
-Junit

-Sonarqube
-Coverity

- **RH ACS**
- Aqua Sec
- Clair
- CyberArk
- Synopsys
- Sysdig
- Palo Alto (Twistlock)

AUTOMATED QUALITY

CM | CS

- **RH ACS**
- Aqua Sec
- CyberArk
- Hashicorp
- Sysdig
- Palo Alto (Twistlock)

OPENSHIFT SOFTWARE FACTORY

Source:
Red Hat Container Catalog: Java Applications
More about the Container Health Index

Red Hat

CONTAINER STANDARDS

THE PROBLEM

With no standard, there is no way to automate. Each box is a different size, has different specifications. No ecosystem of tools can form.

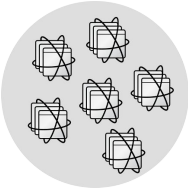
Image: Boxes manually loaded on trains and ships in 1921

WHY STANDARDS MATTER TO YOU

Click to add subtitle

Allow communities with competing interests to work together

There are many competing interests, but as a community we have common goals.



Enable ecosystems of products and tools to to form

Cloud providers, software providers, communities and individual contributors can all build tools.

Protect customer investment

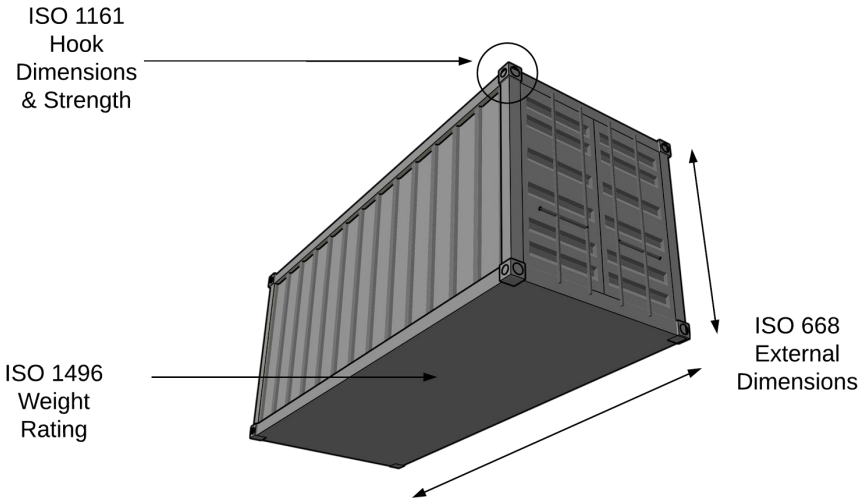
The world of containers is moving very quickly. Protect your investment in training, software, and building infrastructure.

SIMILAR TO REAL SHIPPING CONTAINERS

Standards in different places achieve different goals

The analogy is strikingly good. The importance of standards is critical:

- Failures are catastrophic in a fully automated environments, such as port in Shanghai (think CI/CD)
- Something so simple, requires precise specification for interoperability (Files & Metadata)
- Only way to protect investment in equipment & infrastructure (container orchestration & build processes)

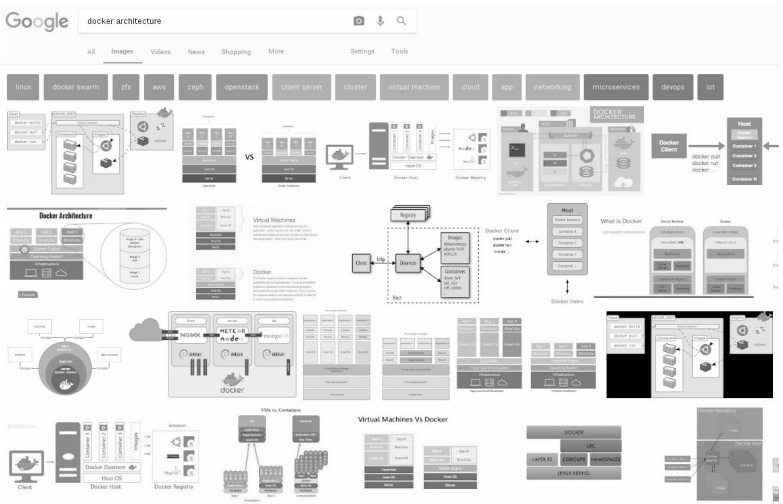


ARCHITECTURE

The Internet is WRONG :-)

Important corrections

- Containers do not run ON docker. Containers are processes - they run on the Linux kernel. Containers are Linux.
- The docker daemon is one of the many user space tools/libraries that talks to the kernel to set up containers

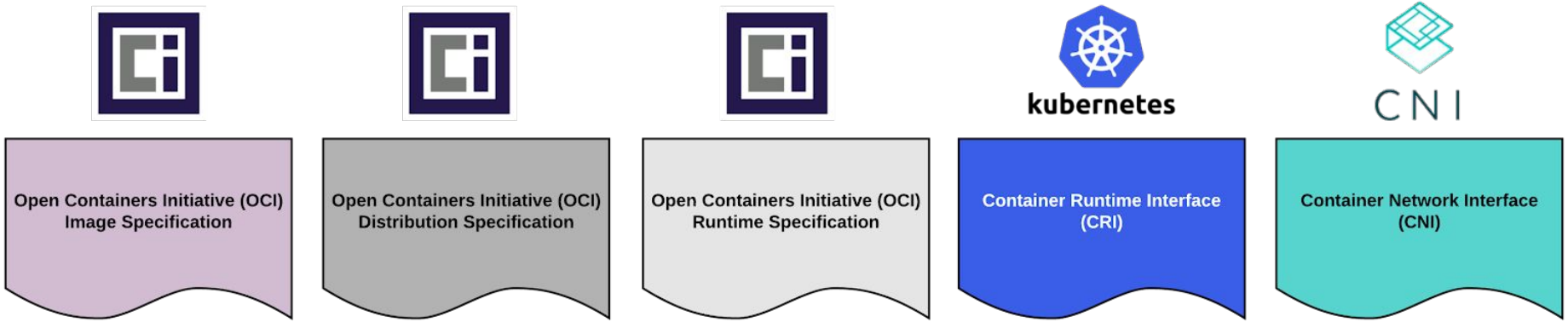


Containers Are Open

Established in June 2015 by Docker and other leaders in the container industry, the OCI currently contains three specifications which govern, building, running, and moving containers.

OVERVIEW OF THE DIFFERENT STANDARDS

Vendor, Community, and Standards Body driven



Many different standards

Advanced Isolation

MicroVMs and other runtimes

Red Hat

Containers With Advanced Isolation

Kata Containers, gVisor, and *KubeVirt (because deep down inside you want to know)*

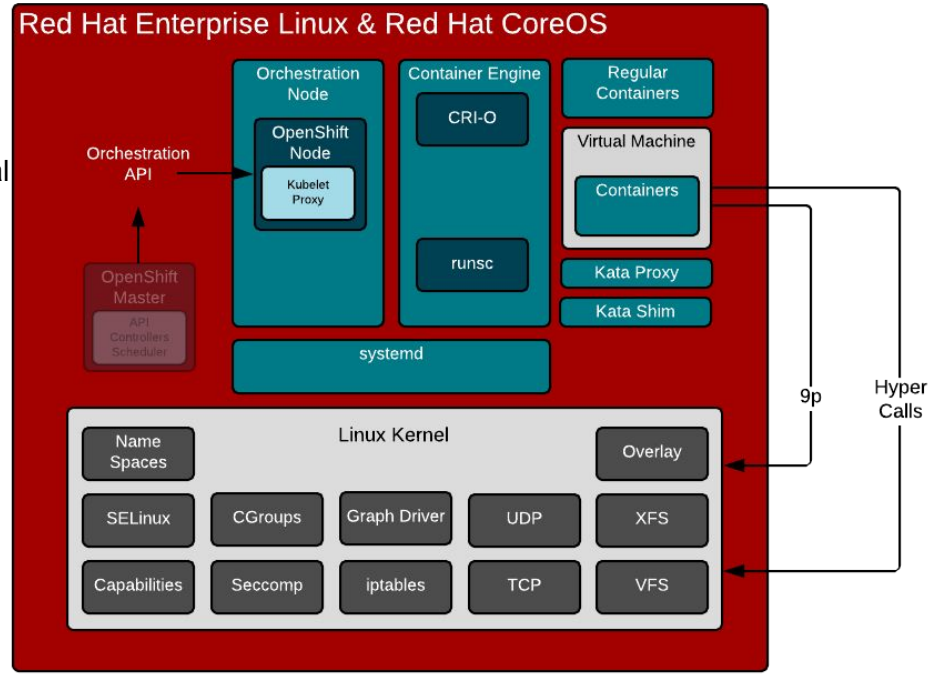
- **Kata Containers** integrate at the container runtime layer
- **gVisor** integrates at the container runtime layer
- **KubeVirt** not advanced container isolation. Add-on to Kubernetes which extends it to schedule VM workloads side by side with container workloads

Kata Containers

Containers in VMs

You still need connections to the outside world:

- Shim offers reaping of processes/VMs similar to normal
- Proxy allows serial access into container in VM
- P9fs is the communication channel for storage



gVisor

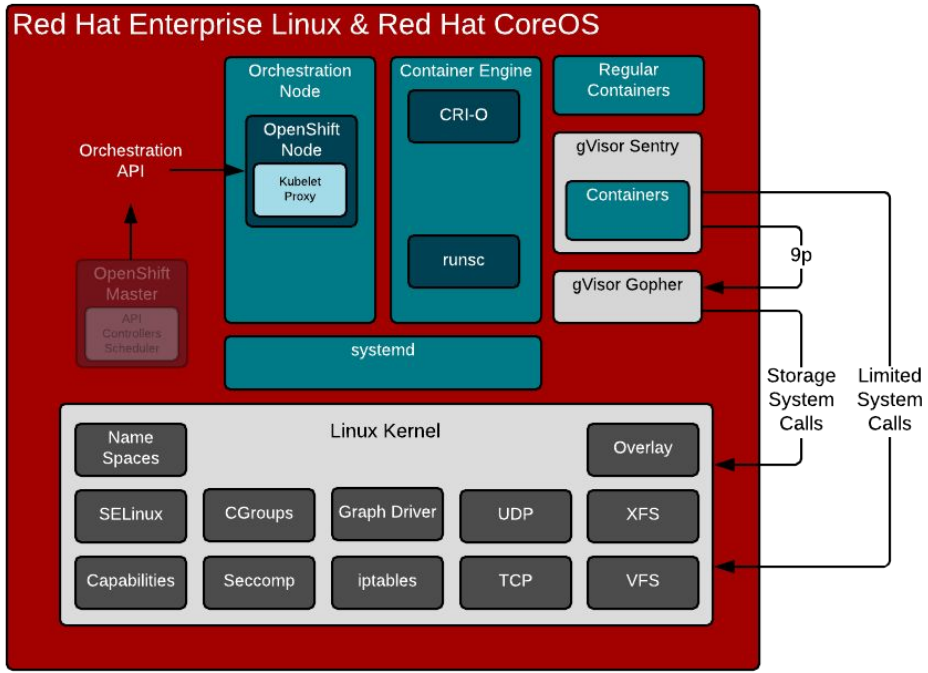
Anybody remember user mode Linux?

gVisor is:

- Written in golang
- Runs in userspace
- Reimplements syscalls
- Reimplements hardware
- Uses 9p for storage

Concerns

- Storage performance
- Limited syscall implementation



KubeVirt

Extension of Kubernetes for running VMs

KubeVirt is:

- Custom resource in Kubernetes
- Defined/actual state VMs
- Good for VM migrations
- Uses persistent volumes for VM disk

KubeVirt is not:

- Stronger isolation for containers
- Part of the Container Engine
- A replacement Container Runtime
- Based on container images