LINUX CONTAINER INTERNALS

How they really work

Scott McCarty
Principal Product Manager, Containers
07/17/2018
BASIC INFO
Introduction - Linux Container Internals

Wifi, Labs, Etc

- Labs
  - learn.openshift.com/training
Introduction

- At Red Hat we encourage networking and we'd like you to spend 2 to 3 minutes introducing yourselves to the person(s) next to you. Say what company or organization you're from, and what you're looking to learn from this tutorial.
- We will use a completely hosted solution called Katacoda for this lab:
  - All you need is a web browser and Internet access: SSID - OReilly18 Password - oscon2018
  - Instructions, code repositories, and terminal will be provided to a real, working virtual machine
  - All code is clickable, all you have to do is click on it and it will paste into the terminal
  - The environment can be reset at any time by refreshing (very nice)
- https://www.katacoda.com/fatherlinux/training/subsystems/
AGENDA

Introduction - Linux Container Internals

Introduction
Four new tools in your toolbelt

Container Images
The new standard in software packaging

Container Hosts
Container runtimes, engines, daemons

Container Registries
Sharing and collaboration

Container Orchestration
Distributed systems and containers
<table>
<thead>
<tr>
<th>AGENDA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advanced - Linux Container Internals</strong></td>
</tr>
<tr>
<td>Container Standards</td>
</tr>
<tr>
<td>Understanding OCI, CRI, CNI, and more</td>
</tr>
<tr>
<td>Container Tools Ecosystem</td>
</tr>
<tr>
<td>Podman, Buildah, and Skopeo</td>
</tr>
<tr>
<td>Production Image Builds</td>
</tr>
<tr>
<td>Sharing and collaboration between specialists</td>
</tr>
<tr>
<td>Intermediate Architecture</td>
</tr>
</tbody>
</table>
Production-Ready Containers

What are the building blocks you need to think about?
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 (rhel7)
- Image Layer & Tags (rhel7:7.5-404)
- At scale, across teams of developers and CI/CD systems, consider all of the necessary technology
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 Index/Manifest.json - provide index of image layers
- Image layers provide change sets - adds/deletes of files
- 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
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
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
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

Command: docker pull registry.access.redhat.com/rhel7/rhel:latest

Decomposition: access.registry.redhat.com / rhel7 / rhel : latest

Generalization: Registry Server / namespace / repo : tag
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
CONTAINER HOSTS
CONTAINER HOST BASICS

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
CONTAINER HOST
Released, patched, tested together

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

- Operating System (kernel)
- Container Runtime (runc)
- Container Engine (Docker)
- Orchestration Node (Kubelet)
- Whole stack is responsible for running containers
CONTAINER ENGINE

Defining a container
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()
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
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
There is a rich history of standardization attempts in Linux:

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

Other related technology
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 containers
- 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
CONTAINER ORCHESTRATION
KUBERNETES & OPENSIFT

It’s a 10 ton dump truck that handles pretty well at 200 MPH

Two major jobs:

- Scheduling - distributed systems computing. Resolving where to put containers in the cluster and allowing users to connect to them
- Provide an API - can be consumed by users or robots. Defines a model state for the application. Completely new way of thinking.
SCHEDULING CONTAINERS

Defining the desired state

- Requires thinking in a completely new way - distributed systems
- Fault tolerance must be designed into the system
MODELING THE APPLICATION

Defining the desired state

Modeling the application using defined state, actual state. Resolving discrepancies:

- The end user defines the desired state
- The system continuously resolves discrepancies by taking action
- Automation can also modify the desired state - Inception
ADVANCED MODELING

Many other resource can be defined
ADVANCED MODELING

Humans interact with these resource through defined state
ADVANCED MODELING

These resources are virtual, but map to real world infrastructure
ADVANCED MODULES
AGENDA

Advanced - Linux Container Internals

Container Standards
Understanding OCI, CRI, CNI, and more

Advanced Architecture
Building in resilience

Container Tools Ecosystem
Podman, Buildah, and Skopeo

Container History
Context for where we are today

Production Image Builds
Sharing and collaboration between specialists

Intermediate Architecture
Production environments
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

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

Enable ecosystems of products and tools to form
Cloud providers, software providers, communities and individual contributors can all build tools.

Allow communities with competing interests to work together
There are many competing interests, but as a community we have common goals.
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)
Where are we going?

2015:
- Tectonic Announced
- Red Hat Container Platform 3.0
- Standards via OCI and CNCF

2016:
- Skopeo project launched under the name OCID
- CRI-O project launched under the name OCID
- Docker engine 1.12 adds swarm

2017:
- Buildah released and ships in RHEL
- Moby project Announced
- Kata merges Clear & RunV projects

2018:
- Podman released and ships in RHEL
- CRI-O is GA and powers OpenShift Online
- V1.0 of distribution spec

2017:
- Containerd project launched
- Docker includes the new containerd
- V1.0 of image & runtime spec
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
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.
Standards Are Well Governed

- Governed by The Linux Foundation
- Ecosystem includes:
  - Vendors
  - Cloud Providers
  - Open Source Projects
OVERVIEW OF THE DIFFERENT STANDARDS

Vendor, Community, and Standards Body driven

- Open Containers Initiative (OCI) Image Specification
- Open Containers Initiative (OCI) Distribution Specification
- Open Containers Initiative (OCI) Runtime Specification
- Container Runtime Interface (CRI)
- Container Network Interface (CNI)

Many different standards
Different standards are focused on different parts of the stack.

- Container Images & Registries
- Container Runtimes
- Container Networking
WHAT ARE CONTAINERS ANYWAY?
Data and metadata

Container images need to express user’s intent when built and run.

- How to run
- What to run
- Where to run
IMAGE AND RUNTIME SPECIFICATIONS

Powerful standards which enable communities and companies to build best of breed tools

Fancy files and fancy processes
WORKFLOW OF CONTAINERS

The building blocks of how a container goes from image to running process

- **允** users to build container images with any tool they choose. Different tools are good for different use cases.

- The container engine is responsible for creating the config.json file and unpacking images into a root file system.

- OCI compliant runtimes can consume the config.json and root filesystem, and tell the kernel to create a container.

- OCI compliant runtimes can be built for multiple operating systems including Linux, Windows, and Solaris.
TYING IT ALL TOGETHER

These standards are extremely powerful
Different standards are focused on different parts of the stack.

- Tools like crictl use the CRI standard
- Tools like Podman use standard libraries
- Tools like runc are widely used
THE COMMUNITY LANDSCAPE

Open Source, Leadership & Standards

The landscape is made up of committees, standards bodies, and open source projects:

- Docker/Moby
- Kubernetes/OpenShift
- OCI Specifications
- Cloud Native Technical Leadership
CONTAINER ECOSYSTEM
AN OPEN SOURCE SUPPLY CHAIN

One big tool, or best of breed Unix like tools based on standards
BASIC CONTAINERS ARE SIMILAR TO PDF?

Find, Run, Build, and Share. Collaboration with any reader/writer
MINIMUM TO BUILD OR RUN A CONTAINER?
Standards and open source code

- A standard definition for a container at rest
  - OCI Image Specification - includes image and metadata in a bundle
- A standard mechanism to pull the bundle from a container registry to the host
  - OCI Distribution Specification - specifies protocol for registry servers
  - github.com/containers/image
- Ability to uncompress and map the OCI image bundle to local storage
  - github.com/containers/storage
- A standard mechanism for running a container
  - OCI Runtime Specification - expects only a root file system and config.json
  - The default runc implementation of the Runtime Spec (same tool Docker uses)
WHAT ELSE DOES KUBERNETES NEED?

Standards and open source code

- The minimum to build or run a container

AND

- A standard way for the Kubelet to communicate with the Container Engine
  - Container Runtime Interface (CRI) - the protocol between the Kubelet and Engine
- A daemon which communicates with CRI
  - gRPC Server - a daemon or shim which implements this server specification
- A standard way for humans to interface with the gRPC server to troubleshoot and debug
  - cri-ctl - a node based CLI tool that can list images, view running containers, etc
THERE ARE NOW ALTERNATIVES

Moving to Podman in RHEL 8 and CRI-O in OpenShift 4
THE UNDERLYING ECOSYSTEM

Many tools and libraries

- containers
- podman
- buildah
- cri-o
- skopeo
- OpenSCAP
- CRIU
CREATING DOWNSTREAM PRODUCTS

Release timing is critical to solving problems
THE JOURNEY
Can start anywhere

Traditional Development

Cloud Native

FIND     RUN     BUILD     SHARE     INTEGRATE     DEPLOY
RHEL (Podman/Buildah/Skopeo)  Quay  OpenShift (Kubernetes)
# CUSTOMER NEEDS

Mapping customer needs to solutions

<table>
<thead>
<tr>
<th>Capability</th>
<th>Platform</th>
<th>Product</th>
<th>Container Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Node</td>
<td>Linux &amp; Container Tools</td>
<td>Red Hat Enterprise Linux</td>
<td>Podman</td>
</tr>
<tr>
<td>Multi Node</td>
<td>Linux &amp; Kubernetes</td>
<td>OpenShift</td>
<td>CRI-O</td>
</tr>
</tbody>
</table>
Red Hat Enterprise Linux 8

The container tools module
PODMAN ARCHITECTURE
Find, Run, Build, and Share. Collaboration with any reader/writer

How containers run with a container engine
APPLICATION STREAMS USE MODULES

Modules are the mechanism of delivering multiple streams (versions) of software within a major release. This also works the other way round, a single stream across multiple major releases.

Modules are collections of packages representing a logical unit e.g. an application, a language stack, a database, or a set of tools. These packages are built, tested, and released together.

Each module defines its own lifecycle which is closer to the natural life of the app rather than the RHEL lifecycle.
THE CONTAINER TOOLS RELEASES

One Module delivered with multiple Application Streams based on different use cases:

- The rhel8 stream delivers new versions for developers
- The versioned, stable streams provide stability for operations
  - Created once a year, supported for two years
  - Only backports of critical fixes

![Diagram of Container Tools Releases]

- rhel8
  - Fast Stream
- Rolling Stream
- V1
  - Stable Stream
  - 2 years of updates
- V2
  - Stable Stream
  - 2 years of updates
OpenShift 4
CRI-O and Buildah as a library
CRI-O ARCHITECTURE

Run containers

How containers run in a Kubernetes cluster
BUILD DAH ARCHITECTURE

Build and share containers

Building when all you can do is run containers
IN LOCKSTEP WITH KUBERNETES

All components for running containers released, tested, and supported together for reliability:

- CRI-O moves in lock-step with the underlying Kubernetes
- The runc container runtime is delivered side by side
- Buildah delivered as a library specifically for OpenShift. No commands for users.
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....
INTERMEDIATE ARCHITECTURE
The orchestration toolchain adds the following:

- More daemons (it’s a party) :-(
- Scheduling across multiple hosts
- Application Orchestration
- Distributed builds (OpenShift)
- Registry (OpenShift)
THE LOGIC
Bringing it All Together
ADVANCED ARCHITECTURE
TYPICAL ARCHITECTURE

Bringing it All Together

In distributed systems, the user must interact through APIs.
THE HISTORY OF CONTAINERS

1979: UNIX
- CHROOT SYSCALL ADDED

2000: JAILS ADDED TO FREEBSD

2003: SELINUX ADDED TO LINUX MAINLINE

2005: FULL RELEASE OF SOLARIS ZONES

2006: PROCESS CONFINEMENT

2008: LINUX CONTAINER PROJECT (LXC)

2008: KERNEL & USER NAMESPACES

2013: DOTCLOUD BECOMES DOCKER

2014: GOOGLE KUBERNETES

2008: RED HAT ENTERPRISE LINUX

1979: GPC RENAMED CGROUPS

2000: LINUX VSERVER PROJECT

2001: LINUX VSERVER PROJECT

2003: SELINUX ADDED TO LINUX MAINLINE

2006: PROCESS CONFINEMENT

2008: KERNEL & USER NAMESPACES

2013: DOTCLOUD PYCON LIGHTNING TALK

2014: GOOGLE KUBERNETES
CONTAINER INNOVATION IS NOT FINISHED

2015:
- Tectonic Announced
- Skopeo project launched under the name OCID
- RED HAT CONTAINER PLATFORM 3.0
- STANDARDS VIA OCI AND CNCF

2016:
- Docker engine 1.12 adds swarm
- CRI-O project launched under the name OCID
- Containerd project launched

2017:
- Moby project Announced
- Kata merges Clear & RunV projects
- V1.0 of image & runtime spec

2018:
- Podman released and ships in RHEL
- CRI-O is GA and powers OpenShift Online
- V1.0 of distribution spec
Mounts

Copy on write vs. bind mounts
AGENDA

L103118 - Linux container internals

10:15AM—10:25AM  
INTRODUCTION

10:25AM—10:40AM  
ARCHITECTURE

10:40AM—11:05AM  
CONTAINER IMAGES

11:05AM—11:35PM  
CONTAINER HOSTS

11:35AM—12:05PM  
CONTAINER ORCHESTRATION

12:05PM—12:15PM  
CONCLUSION
Materials
The lab is made up of multiple documents and a GitHub repository

CONTACT INFORMATION

We All Love Questions

- Jamie Duncan: @jamieeduncan jduncan@redhat.com
- Billy Holmes: @gonoph111 biholmes@redhat.com
- John Osborne: @openshiftfed josborne@redhat.com
- Scott McCarty: @fatherlinux smccarty@redhat.com