Virtual Machines VS Containers

From an isolation perspective
21 February 2020

Francesco Romani
Senior Software Engineer, Red Hat
fromani {gmail,redhat}

whoami

- sweng @ Red Hat: opinions and mistakes are my own!
- daily toolset: golang, kubernetes, podman
- worked with VMs: ~2013 - ~2018
- worked with containers: ~2018 - ...
- interested in: more, golang, more containers, lisp (someday!)
- happy linux user (red hat linux, debian, ubuntu, fedora)
- geek

Outline

- Dramatis personae
- How a container is made
- How a (K)VM is made
- The Fallout

STANDARD DISCLAIMER

Software Engineer, not security expert!

Mistakes may happen - please point them out!

All opinions are mine only

Versus?

Mid-Late 2000s (~2004 - ~2010) was all about VMs
Mid-Late 2010s (~2014 - ~2020) is all about containers

Containers (initially?) advertised as "better" (easier, simpler less resources) virtualization

Different tools, some overlap in the use cases, large overlap in the technology stack.

From a security perspective?

The cloud use case

It's (mostly) about isolating workloads

What about software distribution?

As usual, a lots of tradeoff are involved

Dramatis personae

Virtual Machines

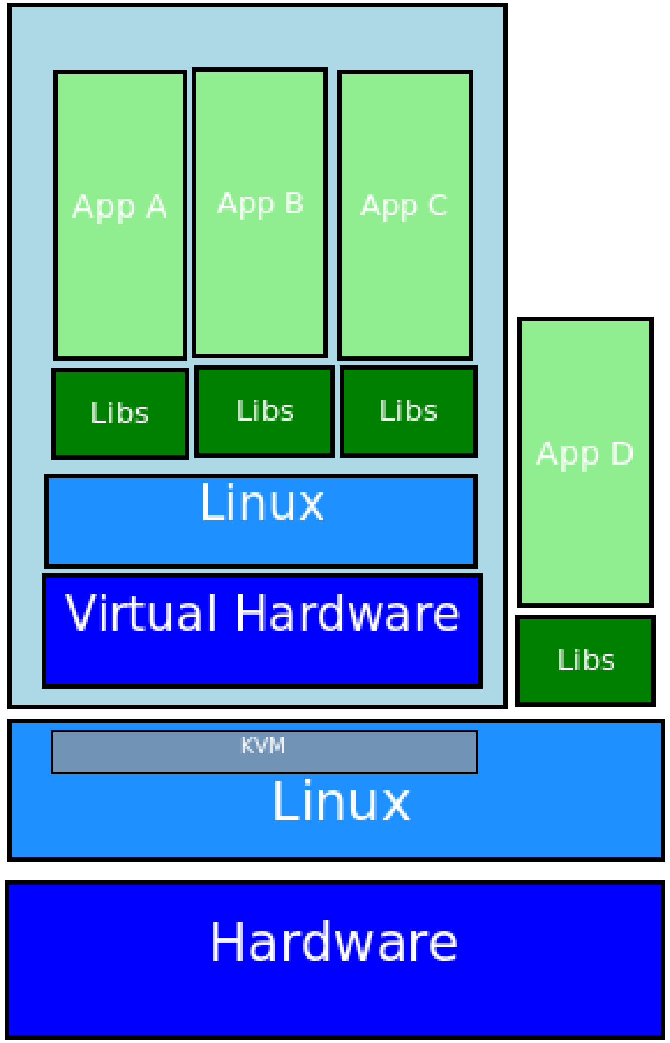
"A virtual machine (VM) is an emulation of a computer system. Virtual machines are based on computer architectures and provide functionality of a physical computer."

"[...] virtual machines [...] provide a substitute for a real machine. They provide functionality needed to execute entire operating systems."

"Modern hypervisors use hardware-assisted virtualization, virtualization-specific hardware, primarily from the host CPUs."

source: wikipedia (https://en.wikipedia.org/wiki/Virtual_machine)

Virtual Machines (2)



"(Linux) Virtual machine block diagram" - (C) Francesco Romani 2019 - CC by-sa 4.0

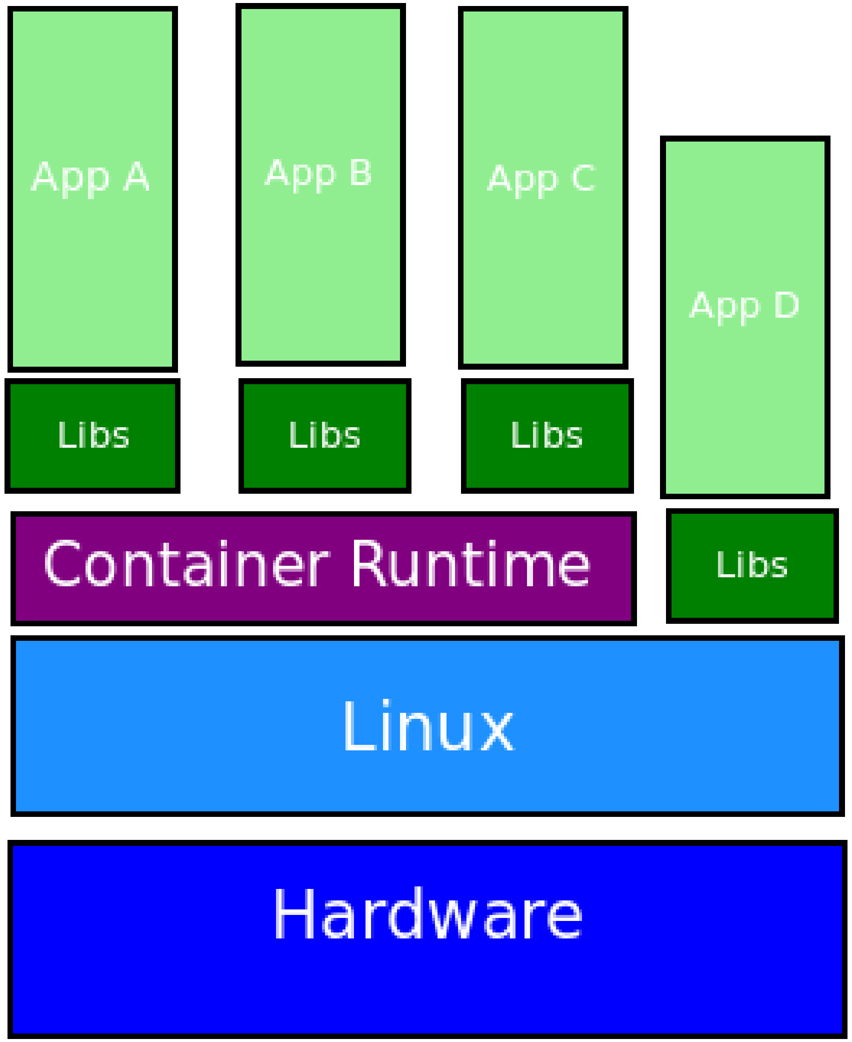
Containers

A (Linux) container is a set of one or more processes isolated from the rest of the system, using facilities of the Linux kernel

> source: not actual quote, amalgamation. (https://en.wikipedia.org/wiki/List_of_Linux_containers)

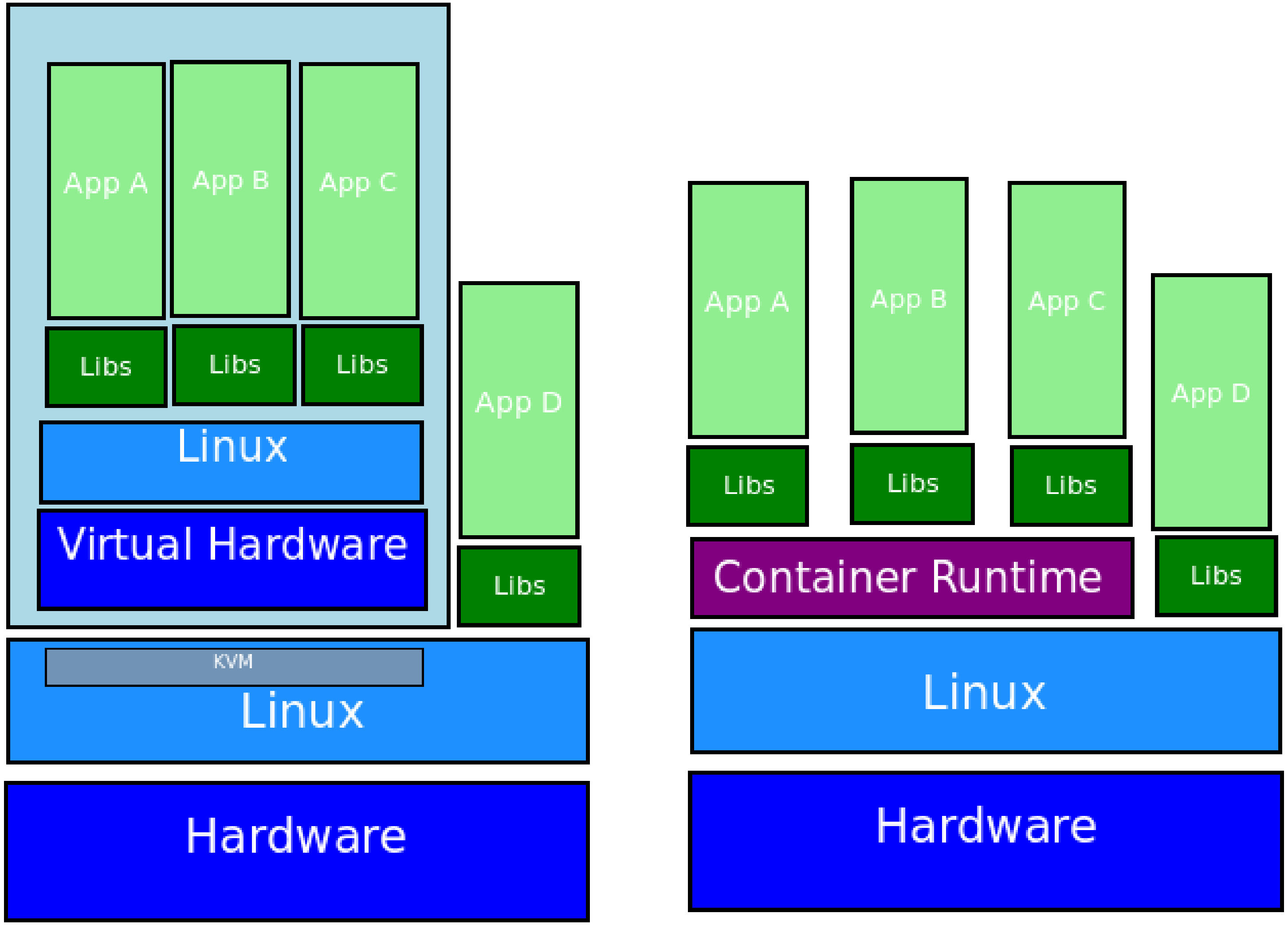
That's cgroups+seccomp+selinux+namespaces... All conveniently packed in a container runtime (cri-o, docker, rkt...)

Containers (2)



"(Linux) Containers block diagram" - (C) Francesco Romani 2019 - CC by-sa 4.0

Virtual Machines vs Containers



"(Linux) VMs vs Containers block diagram" - (C) Francesco Romani 2019 - CC by-sa 4.0

How a container is made

Meet the containers

"Containers are being loaded on the container ship MSC Sola at the container terminal of Bremerhaven in Germany" by Tvabutzku1234, public domain, from Wikimedia Commons

A recipe for containers

The basic building blocks:
- namespaces: process isolation
- cgroups: resource limits

Security enforcement tools:
- seccomp: limit syscall usage
- SELinux: mandatory access control
- linux capabilities: finer-grained privileges

Namespaces: Intro

Inception: ~2002; major developments ~2006 and onwards.

A namespace...

```
wraps a global system resource in an abstraction that makes it appear to the processes
within the namespace that they have their own isolated instance of the global resource.
[...]
One use of namespaces is to implement containers.
```

Namespaces are *ephemeral* by default: they are tied to the lifetime of a process.
Once that process is gone, so is the namespace.

more documentation (http://man7.org/linux/man-pages/man7/namespaces.7.html)

Namespaces: API

A **Kernel** API, syscalls:

- unshare(2): move calling process in new namespace(s) - and more.
- setns(2): make the calling process join existing namespace(s)
- clone(2): create a new process, optionally joining a new namespace - and **much** more.

Namespaces: what we can unshare?

- cgroup: cgroup root directory (more on that later)
- ipc: System V IPC, POSIX message queues
- network: network devices, stacks, ports, etc.
- mount: mount points
- pid: process id hierarchy
- user: user and group IDs
- uts: hostname and NIS domain name
- time: the very last addition (linux 5.6)

more documentation (http://man7.org/linux/man-pages/man7/namespaces.7.html)

Namespaces DIY: unshare

PID of the current shell:

```
///samurai7/~># echo $$
5184
```

We start a new process (bash) with different network and PID namespaces

```
///samurai7/~># unshare --net --fork --pid --mount-proc bash
///samurai7/~># echo $$
1
///samurai7/~># ifconfig
///samurai7/~>#
```

Let's doublecheck:

```
///samurai7/~># ls -lh /proc/{1,5184,5282}/ns/pid
lrwxrwxrwx. 1 root root 0 Feb 21 19:54 /proc/1/ns/pid -> pid:[4026531836]
lrwxrwxrwx. 1 root root 0 Feb 21 19:53 /proc/5184/ns/pid -> pid:[4026531836]
lrwxrwxrwx. 1 root root 0 Feb 21 19:54 /proc/5282/ns/pid -> pid:[4026532544]
```

Namespaces DIY: nsenter

Let's enter the namespaces context we created in the slide before:

```
///samurai7/~># nsenter -a -t 5282 /bin/sh
sh-4.4# ps -fauxw
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root        32  0.0  0.0 122680  3864 pts/4    S    20:00   0:00 /bin/sh
root        33  0.0  0.0 149756  3700 pts/4    R+   20:00   0:00  \_ ps -fauxw
root         1  0.0  0.0 123884  5108 pts/2    S+   19:53   0:00 bash
sh-4.4# echo $$
32
```

Namespaces: wrap up

Namespaces allow us to have separate instances of system resources.

Operating System resources are still **shared**

With the linux namespaces, we have the bare bones of a simpl{e,istic} container engine!

But much more is needed.

cgroups: intro

Inception: ~2007. Major update: ~2013

Linux **C** ontrol **Groups**: allow process to be organized in hierarical groups to do limiting and accounting of certain system resources.

Most notably, memory and CPU time (and more: block I/O, pids...)

Powerful and easy-as-possible resource control mechanism

But still quite complex to manage

cgroups: what can we control?

- blkio: limits on input/output access to and from devices
- cpu: uses the scheduler to provide cgroup tasks access to the CPU
- cpuacct: automatic reports on CPU resources used by tasks
- cpuset: assigns individual CPUs and memory nodes to tasks
- memory: sets limits on memory and reports on memory resources
- perf_event: performance analysis.

Specific Linux Distribution (e.g. RHEL) may offer more cgroups.

Add your own!

cgroups: API

Just use sysfs:

```
echo browser_pid > /sys/fs/cgroup/<restype>/<userclass>/tasks
```

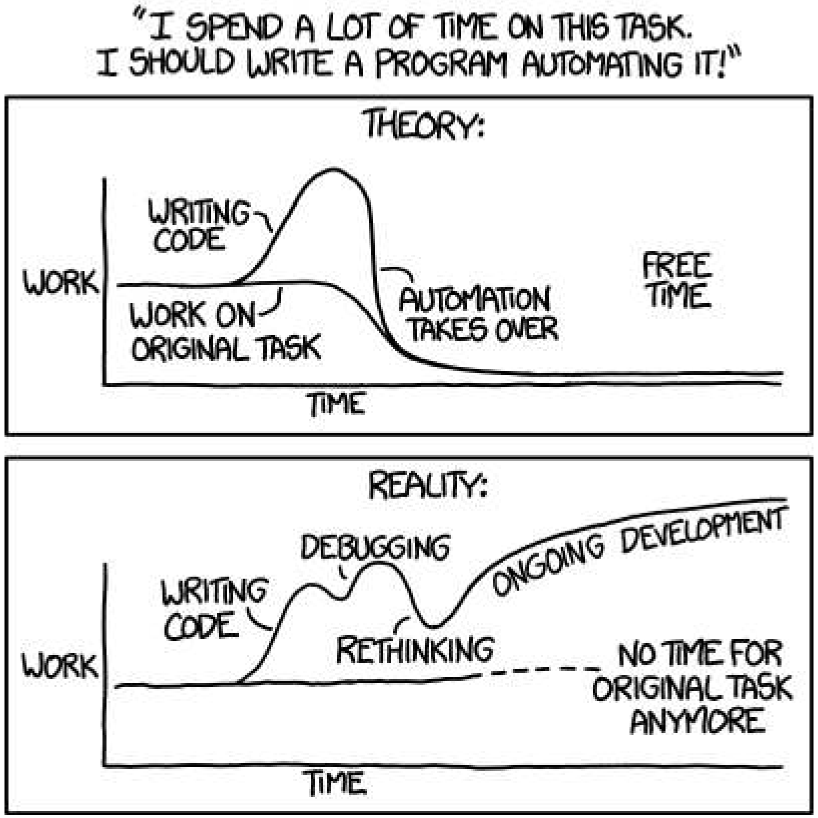
command line tools: cgcreate, cgexec, and cgclassify (from libcgroup).

Or just let your management engine do that for you:

- systemd
- docker
- libvirt (spoiler!!)

cgroups: DIY

Mostly, you don't want to do it :)



Seriously, the management tool (whatever it is) almost always Just Works (tm) and it is simpler to tune.

cgroups: wrap-up

CGroups provide resource `limit` and `accounting`

Organized in hierarchies

A **LOT** of subtleties with respect to accounting and sensible limits

Here's why you should not DIY - don't reinvent a square wheel

Deserves a (long) talk on its own

seccomp

Inception: ~2005; Major update ~2012

Operational modes:

- 0 disabled
- 1 for strict: only *four* system calls: read, write, exit, sigreturn
- 2 for filter: allow developers to write filters to determine if a given syscall can run

seccomp: API & DIY

Kernel API (syscall), so just prctl(2) and seccomp(2)

And obviously `procfs` interface.

You can add your own syscall filters using **BPF** language (!!!)

Again, better don't reinvent the wheel, just use profiles from your management engine

If you really want to DIY, maybe start here (https://lwn.net/Articles/656307/)

SELinux

Inception: ~1998

Adds Mandatory Access Control (MAC) and Role Based Access Control (RBAC) to the linux kernel

Linux, being UNIX-Like, previously supported only Discretionary Access Control

SELinux: DAC vs MAC vs RBAC

WARNING: brutal semplification ahead

DAC: access control is based on the discretion of the owner: root can do anything.

MAC: the system (and not the users) specifies which can access what: no, even root *cannot* do that.

RBAC: in a nutshell, generalization of MAC: create and manage *Roles* to specify which entity can access which data.

beware: Again: the world is much more complex than that... (https://en.wikipedia.org/wiki/Role-based_access_control)

SELINUX: Daily usage

Mostly used on CentOS, Fedora, RHEL, RHEL-derived distributions

SELinux used to be perceived as overly complex, and overly annoying too.

"Just disable SELinux" was a recurrent advice up until not so long ago

It got **EXTREMELY** better: most of time, you don't even notice it is running. Just Works (tm)

Except when it prevents exploits :)

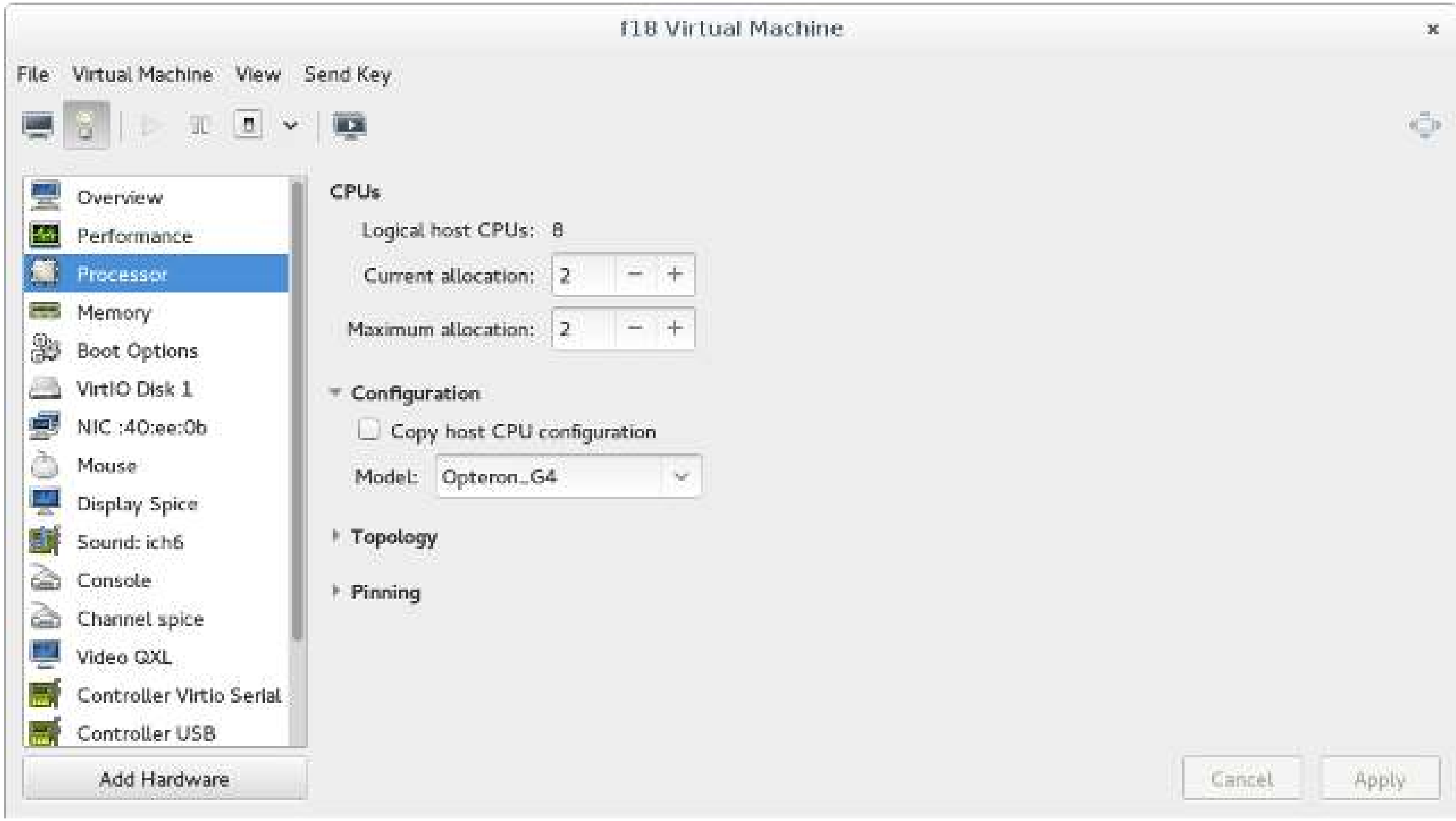
If you need to troubleshoot something, `audit2why` is usally a great start

Again, most often just use the profiles your distribution/management engine provides

- Lots of documentation available (http://selinuxproject.org/page/Main_Page)

How a (K)VM is made

Virtual Machines



virt-manager screenshot, from https://www.virt-manager.org/wp-content/uploads/2014/01/details.png

Preamble

VMs went a long way - even on x86

We will focus only on the "winner stack": [VT-x/SVM +] KVM + QEMU (e.g. not Xen)

The history is more complex

The modern Linux virtualization stack

- HW-assisted virtualization: Intel VT-x, AMD SVM
- KVM: Linux (lightweight but complete) Hypervisor, makes use of HW-assisted virtualization
- QEMU: System emulator, provides I/O, management layer, uses KVM for acceleration
- Libvirt: better management layer, adds isolation/containment to QEMU instances

Concepts: Hypervisor

A hypervisor a supervisor-of-supervisor

The kernel is a supervisor

A hypervisor allows to run Virtual Machines (OSes inside OSes)

KVM makes the Linux kernel a hypervisor

Of course you can still run regular processes alongside VMs!
Linx + KVM is both a hypervisor and a supervisor (not always the case).

Concepts: x86 hw-assisted virtualization

We'll just cover the basics - otherwise there's material worth few slide decks...

- New x86 instructions (like MMX, SSE*, AVX...)
- Introduced by Intel (2005) and AMD (2006)
- From user perspective, nowadays (2020) pretty much equivalent
- Both supported by KVM
- Both allowes nested VMs (VMs inside VMs)

Concepts: virtualized vs paravirtualized

(Full-)virtualization: the guest OS is not aware it runs in a VM.

Paravirtualization: the guest OS is aware it is running in a VM.
- Special device/device drivers (virtio)
- The guest OS may adjust itself (e.g. scheduler, host-provided hints)

HW-assisted x86 virtualization, in a nutshell

- New CPU operational mode `root` and `non-root`.
- New modes orthogonal to both cpu mode (real, protected, long) and privilege (0-3).
- Hypervisor run in root mode
- VMs run in non-root mode.
- Provileged instructions **which also change the context of the CPU** (clock, interrupt regs, control regs) cannot be executed in non-root mode.

Some VT-x instructions

- **VMXON**: enables virtualization support. Must be called first. Leaves CPU in root mode.
- **VMLAUNCH**: creates a VM and enters non-root mode.
- **VMRESUME**: (re-)enters non-root mode for an existing VM.
- **VMREAD / VMWRITE**: access VMCS.

VMEXITs

vmexit: When a VM tries to execute a CPU-state-changing operation, disallowed in non-root mode, the CPU switches back to root mode (like a trap).

After a vmexit, the hypervisor must take actions to let the VM resume its operations, and then call **VMRESUME**

How does the hypervisor know **WHY** a vmexit happened?

VMCSes

Each VM instance has a **Virtual Machine Control Structure** (VMCS),
a 4 KiB segment which contains the VM context.

The VMCS holds the virtual CPU state (as seen by the guest)
The VMCS holds the reason why a vmexit happen.

The x86 virtualization in a nutshell

The key component of the X86 virtualization is the interaction between root and non-root code:

hypervisor -> VMLAUNCH -> vmexit -> [hypervisor actions from VMCS data] -> VMRESUME

KVM

In a nutshell
- Turns Linux into a hypervisor
- built on top of hardware virtualization (VT-x, SVM)
- API as device `/dev/kvm`, `ioctl()s`

Do not use directly! (use qemu! or kvmtool or pretty much any other linux tool)

QEMU uses it as accelerator

QEMU

- Can emulate hardware
- Used in the virtualization stack to handle I/O (device emulation)
- Uses KVM to achieve near-native execution speed
- I/O speed close to native with paravirtualization
- Large, complex software
- command line only tool - not easy to manage

Libvirt

- toolkit to manage virtualization platform
- QEMU+KVM is the most popular (and developed) target
- stable interface
- applies additional **isolation layers** around QEMU

libvirt + systemd = {cgroups, selinux} around QEMU

Wrapping up - and some musings about security

VMs

- OS-inside-OS
- **perceived** as heavyeight, slow to spin up, hard to manage
- guest apps interact with guest Kernel
- actually two layers of operating system around your code
- more layers -> more code -> more bugs
- VM escape techniques do exist
- still the greatest possible isolation

Containers

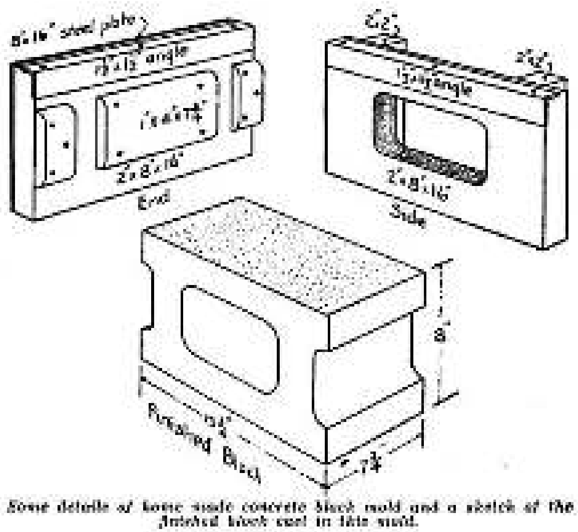
- shared kernel with host OS
- easy and lightweight to get started - aka nice scaling down
- guest apps interact with host kernel - but they believe they are alone :)
- made popular by docker
- friendlier tooling overall?
- weak isolation

The fallout - and more musings about security

Containers as amalgamation of technologies

Containers don't exist -YET- as proper linux objects

Containers are made of a set of linux technologies which create isolation layer(s) around regular processes



"19th century knowledge mechanisms homemade concrete block mold parts" by Henry Colin Campbell, Public Domain, from Wikimedia Commons

Containers are turbo-charged processes

Wait, QEMU is a process too!

So what does prevent us to use the same isolation technologies around Virtual Machines?

Columbus Breaking the Egg, CC0, From Wikimedia Commons

Extra-isolated VMs

The modern linux (virtualization) stack **IS** using a good chunk of isolation technologies around VMs

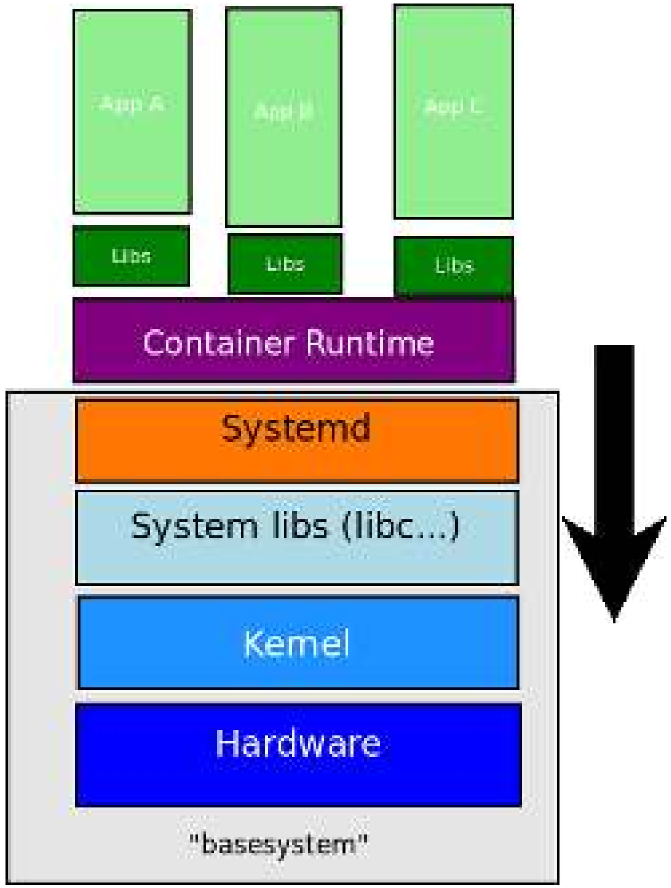
Defense in depth

Libvirt uses SELinux (if available) to restrict the VM behaviour

Systemd provides cgroup isolation out of the box

Container building blocks integration

The technologies powering containers are being pushed down the linux kernel stack



"(Linux) container stack evolution block diagram" - (C) Francesco Romani 2020 - CC by-sa 4.0

Container building blocks integration /2

The modern linux systems are gaining more and more container-like capabilities out of the box

Will container just disappear in the future?

Meaning, will they just become yet another type of service units?

What's a container, really?

If a container is a way to run isolated workloads, the basic linux system are gaining capabilities to run them
- systemd (and more to come)
- podman?

If a container is a way to *ship* software, that's a completely different story.

Let's not open the pandora's box of (linux) software packaging.

So are container going to disappear?

It's hard to make predictions, especially about the future :)

Are VM going to disappear?

It's hard to make predictions, especially about the future :)

But VMs survived the container revolution.

VMs provide a *different* toolset.

Do we really have to choose?

VM resurgence!
VMs and containers are going to be integrated:

See:
- kubevirt
- kata containers
- ...

Q? A!

Thank you

Francesco Romani
Senior Software Engineer, Red Hat
fromani {gmail,redhat}
http://github.com/{mojaves,fromanirh} (http://github.com/%7Bmojaves,fromanirh%7D)