Title:	Container Mythbusters
Author(s):	Jennings, Michael E.
Intended for:	Sandia SSESS Seminar, 2021-12-08 (None (MS Teams), New Mexico, United States)
Issued:	2021-12-10 (rev.1)

Los Alamos NATIONAL LABORATORY

Container Mythbusters

Debunking the Nonsense, Dissecting the Misconceptions, and Distilling the Facts of High-Performance Containering

Michael Jennings (mej@lanl.gov)

High-Performance Computing Systems

Los Alamos National Laboratory

Scientific Software Engineering Seminar Series (SSESS)
Sandia National Laboratories
8 December 2021

LA-UR-21-32001

MYTH: Containers are …*insert definition here*…

FACT: "Container" is a term used somewhat indiscriminately to mean different things to different people & projects!

"Container" sometimes refers to the entire stack/collection of individual layers and metadata that compose a final, tagged filesystem tree.

- Docker calls each layer an "**image**" and the tagged grouping a "**repository**."
- The latter is *also* referred to as an "image," especially in day-to-day speech and in writing.
- Each tag points only to a single layer, but since layers are limited to a single parent, the terms wind up being somewhat interchangeable even if a bit vague/confusing.
- Related to this, "container" is frequently used to refer to the merged/unified filesystem, often composed by the container runtime, which acts as the root filesystem for the containerized application.

"Container" is also used to refer to the **process** at runtime which is invoked by the container runtime engine (e.g., Docker) and is the entrypoint (usually PID 1) of the containerized application.

- This is generally considered the "correct" definition.
- I sometimes mess this up myself, and if I do (or you're not sure), feel free to **stop me and ask**!

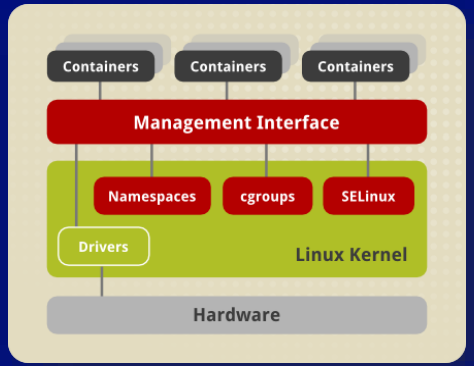


Image credit: Red Hat

MYTH: Containers are the new `chroot()`.

FACT: Linux employs several kernel features, system calls, and services to “containerize” processes.

Modern kernel features allow us to instruct the kernel to “lie” to our applications about various attributes of the system, including filesystem mounts, process IDs, hostnames, network stacks, and more.

- 7 Privileged Namespaces (require `CAP_SYS_ADMIN` to create)
  - `mount` – Private filesystem mount points, recursion/propagation controls
  - `pid` – Private view of process IDs and processes, `init` semantics
  - `uts` – Private hostname and domainname values
  - `net` – Private network resources (devices, IPs, routes, ports, etc.)
  - `ipc` – Private IPC resources (SysV IPC objects, POSIX msg queues)
  - `cgroup` – Private control group hierarchy (Linux 4.6+ only)
  - `time` – Private offsets for `MONOTONIC` and `BOOTTIME` clocks (5.6+ only)
- 1 Unprivileged Namespace (requires no special capabilities to create)
  - `user` – Private UID and GID mappings; can be combined with other namespaces, *even if unprivileged*
- System Call API: `unshare(2)`, `clone(2)`, `setns(2)`

FACT: Linux employs several kernel features, system calls, and services to “containerize” processes.

The Linux kernel has several additional subsystems that containers sometimes use:

- **`cgroups` – Control hierarchical resource management and usage constraints**
  - Latest kernels (4.6+) even have namespaces for this!
  - Schedulers/RMs use to track/control job resource utilization
- **`seccomp-bpf` – Berkeley Packet Filter-based syscall filtering**
  - Frequently used to prevent containers from exceeding their scope
- **`prctl(PR_SET_NO_NEW_PRIVS)` – Prevent privilege escalation**
  - Kernel-level flag that prevents `execve()` granting privileges.
  - Persists across all calls to `fork()`, `clone()`, and `execve()`
  - Privileged containerization is unsafe without this.
- **SELinux – MLS/MAC Labeling system for files/processes**
  - Allows admins precise control over actions, roles of applications
- **AppArmor – Profile-based MAC system for limiting apps’ abilities**
  - Similar to SELinux but without filesystem labeling features

MYTH: Containers are lightweight/more efficient VMs.

MYTH: Containers should be used to replace/virtualize entire servers.

FACT: Containers couple applications to their OS environment. Their flexibility allows them many uses, though.

In the Docker/OCI ecosystem, when you build an **application container**, you specify a “command” or an “entrypoint:” the command to run when the container starts up.

- All other processes in the container are children of this single parent command.
- The analogue of an application container is an application, not a machine.
- The term “operating system virtualization” is often misunderstood; it simply means that containerized applications have a unique/altered view of the underlying OS *but not of the kernel*!
- From the perspective of the kernel, containers are *always* a group of processes and their children.
- Some container runtimes allow for the creation of virtual networks, volume mounts, etc. At minimum, though, containers have distinct views of the filesystem mount table, including the OS.

Depending on the runtime, certain details may differ. Some runtimes actually *are* intended to virtualize/abstract entire hosts! So there are exceptions:

- The `systemd-nspawn` container system expects to “boot” the container.
- LXD offers VM-/cloud-like functionality like replication and live migration.
- Even with Docker, it’s possible to convert hosts into containers. But if that’s the goal, Docker may not be the best tool for that job. At least not by itself.
- HPC jobs & microservices use app containers; containerized hosts are a different beast altogether and should use a different engine/runtime.

MYTH: Containers contain.

MYTH: Containers don’t contain.

FACT: Containers contain passively, not actively. Think buckets, not prisons.

Typical (privileged) containers are an abstraction & encapsulation tool, not a security measure.

- The Linux kernel does not go out of its way to prevent containerized processes from escaping namespaces or crossing between them. In fact, it explicitly allows this (via the `setns()` syscall)!
- Additionally, numerous endpoints in the `/proc` filesystem offer opportunities to "escape" or cross over the namespace boundary and move "outside" the container.
- That's where the additional kernel features come in. Privileged containers need additional security measures to be "safe" (e.g., SELinux/AppArmor, `seccomp-bpf`).

Unprivileged containers get safety measures imposed by the kernel.

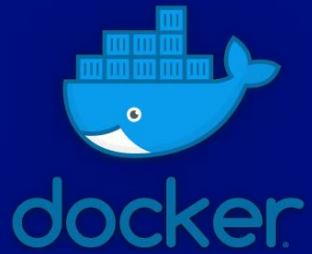
- Capabilities-based, kernel-enforced policies govern interaction/ movement between namespaces.
- Extensive testing and R&D has gone into user namespaces to make them usable & secure.
- *Something* must manage the privilege boundary between contained process(es) and the system.

MYTH: “Container” is shorthand for “Docker Container.”

FACT: There are many container runtimes and related technologies; most are built around/leverage the OCI standards.

Docker did popularize Linux containers by making them portable, reproducible, and composable.

- Other players in the space took exception to certain design choices Docker, Inc., made and revolted.
- A global standards body was set up under The Linux Foundation as a Collaborative Project.
- The Open Container Initiative publishes Runtime and Image specifications, bootstrapped by Docker but developed and governed openly by representatives from key member organizations.



FACT: There are many container runtimes and related technologies; most are built around/leverage the OCI standards.

High-Performance Computing has a unique set of challenges not seen in the web-app world. Docker's client/server architecture and root-only access model is not well suited to address them.

- NERSC's Shifter came first; it uses a privileged runtime and parallel filesystem storage to scale.
- LANL's Charliecloud went the other direction, using user namespaces to facilitate unprivileged runtime; backend image distribution at scale is left up to the user (only safe due to lack of privilege).
- Singularity began as a non-container `chroot()`-based amalgamation of old technologies with poorly understood behavior, was rewritten, and has since incompatibly reproduced much of the greater container community's standards.
- While not focused on the use cases of HPC, Red Hat's `podman` offers `runc`-based OCI compliance *and* addresses many of Docker's "issues." Unprivileged containers/builds are now fully supported.

MYTH: Containers are hard & require complex tools like Docker or Rkt.

FACT: Setting up, running, and using containers is easy; you can even write your own container-based solutions in BASH!

Recall the system call API is only 3 functions:

- `unshare(2)`: Creates one or more new namespaces and moves the current process into them;
- `clone(2)`: Creates a new process/thread, optionally putting it in one or more new namespaces; and
- `setns(2)`: Places the calling process/thread into the specified new namespace.

Recent versions of util-linux include 2 shell commands that wrap 2 of the 3 calls:

- `unshare(1)`: Runs a new program with one or more namespaces unshared from the parent; and
- `nsenter(1)`: Enters the namespace(s) of other process(es), then executes shell/specified program.

Namespace directives are also supported in `systemd` unit files, making it easy to containerize services.

As long as you're using an existing (unprivileged whenever possible) runtime, containers are pretty straightforward. Even using those existing building blocks to create something larger isn't terribly difficult. The gory details of ***writing*** a runtime, however, is ***very*** complex and nuanced! Use an existing runtime, and understand the technical rationale for your choice.

Image Credit: Toca do Tux

MYTH: Docker is insecure.

FACT: Docker's security record is pretty scary.

Since early 2017, there have been ~30 vulnerabilities that could lead to kernel panics, host information leaks, and privilege escalation inside or outside the container!

- One particular release fixed a total of SIX vulnerabilities, including 2 buffer overflows. No CVE IDs.
- More than once, a single CVE covered multiple vulnerabilities, including the ability to join and affect the root namespace, test for arbitrary file existence as `root`, and add content to `/usr/bin`.
- 7 of the 9 releases in 2018 contained security fixes, almost all high severity. No CVEs until 7/2018.
- 2017-2021 (5 years), 38.3% of releases fixed vulnerabilities (~43% in 2017/2020, ~78% in 2018).

Security experts and container experts have expressed serious concerns about its design/code:

- "*I found the code of the setuid binaries quite difficult to read. It feels like upstream somewhen lost the focus on the "minimal and clean" design that set*id programs require.*"
- "*Mixing user controlled data with "trusted" data generated by the setuid binary itself in the same registry makes the code hard to read or to trust, respectively.*"
- "*After fixing the major security issues and doing some additional hardening we can keep [it]…since the binaries are only accessible to members of [its UNIX] group. I wouldn't like to see world access for those setuid binaries.*"

FACT: Singularity's security record is pretty scary.

Since early 2017, there have been ~30 vulnerabilities that could lead to kernel panics, host information leaks, and privilege escalation inside or outside the container!

- One particular release fixed a total of SIX vulnerabilities, including 2 buffer overflows. No CVE IDs.
- More than once, a single CVE covered multiple vulnerabilities, including the ability to join and affect the root namespace, test for arbitrary file existence as `root`, and add content to `/usr/bin`.
- 7 of the 9 releases in 2018 contained security fixes, almost all high severity. No CVEs until 7/2018.
- 2017-2021 (5 years), 38.3% of releases fixed vulnerabilities (~43% in 2017/2020, ~78% in 2018).

Security experts and container experts have expressed serious concerns about its design/code:

- "*I found the code of the setuid binaries quite difficult to read. It feels like upstream somewhen lost the focus on the "minimal and clean" design that set*id programs require.*"
- "*Mixing user controlled data with "trusted" data generated by the setuid binary itself in the same registry makes the code hard to read or to trust, respectively.*"
- "*After fixing the major security issues and doing some additional hardening we can keep [it]…since the binaries are only accessible to members of [its UNIX] group. I wouldn't like to see world access for those setuid binaries.*"

FACT: Most reports of Docker being "insecure" are "pilot error." The `docker` CLI requires privilege for a reason!

Docker is, by design, only accessible to the `root` user.

- Docker Enterprise Edition allows an authorization plugin to control access to the API.
- Most sites/users don't bother exploring all the security features/options available in Docker, such as customized `seccomp-bpf` filters, fine-grained capability control, privilege flag control, and more.
- As a result of the access model, true vulnerabilities in Docker are (arguably) limited to repo creators.

Looking at CVEs 2017-2021 (all 4 production quality), Docker compares favorably, though not great:

	Charliecloud	Docker	Shifter	Singularity
Vulnerability Count	0	<=23	0	29+

Even so, it's 2021! We have much better options today, especially **unprivileged** runtimes!

- Multiple schedulers & RMs support Docker, always by restricting direct user access to Docker API.
- Most security professionals agree using `root`-owned daemons or setuid binaries is unnecessarily risky.
- Current versions of all major Linux distributions, including RHEL & SLES, support user namespaces.
- Thanks to security expert Dan Walsh, Red Hat offers compatible/competing tools (`podman`, et al.).
- All major runtimes (almost?) now support unprivileged operations/workflows!

MYTH: Containers (or specific container runtimes) solve the problem of reproducibility in computational and data science.

FACT: Reproducible Builds is an area of study unto itself; *no* single existing solution fully solves the reproducibility problem.

Docker and Singularity both offer solutions to prescriptive container image generation.

- The `Dockerfile` format is supported by almost all container build engines. Build instructions are preserved in the output via JSON-encoded layer metadata along with labels, lineage, etc.
- Singularity supports an RPM-specfile-like "recipe" syntax (not to be confused with Chef's) with similar, but incompatible, format/purpose. User Guide seems to confuse "reproducible" with "immutable."
- Docker's format facilitates "reproducible" layered images; each build directive creates a new, unique layer which directly depends on the previous layer and records the directive used to create it.
- Docker/OCI image format uses Content-Addressable Storage for content assurance/persistence.

Many challenges still exist around reproducibility that are not solved, or even addressed, by containers.

- There are no guarantees that build instruction artifacts/effects are consistent across time. Nothing says that "`yum install foo`" or "`FROM centos:7`" will have the same result in 5 years…or 5 months…or even a week.
- As Aleksa Sarai points out, the `tar` archive format is fraught with reproducibility roadblocks.
- Using CAS hashes to identify layers/images consistently requires infinite, eternal artifact archive.
- Reproducibility via containers ignores the key differentiator of containers vs. VMs – the kernel!

MYTH: Containers are secure as long as the user's UID inside the container matches the user's UID outside the container.

FACT: Container security is multifaceted and highly nuanced. That claim reflects incomplete/insufficient understanding.

The kernel/userspace interface for containers is simple; the security model, however, is not.

- A number of issues were found early on that revealed overlooked corner cases.
- Numerous strange/subtle quirks are needed to deal with combinations of namespaces and common HPC use cases (e.g., in-memory `rootfs`). (Charliecloud examples document many of them.)
- The complex interplay of identity, privileges, permissions, capabilities, kernel settings, and so forth is challenging enough to get correct without hiding crucial details from the ultimate arbiter of access!

Example: If I told you to do `chmod 4755 /bin/bash` and that it's safe because you'd have the same uid "inside" the shell as you had "outside" it, would you do it? or would you think I'd taken leave of my senses?

- There's a lot that happens between typing `bash` and the shell prompt being displayed.
- There could be exploits that are useless on their own but effective with `root` privileges.
- *Privileged operations are privileged for good reason*; override at your own peril!

```
-bash-4.2$ ls -Fla /bin/bash
-rwsr-xr-x 1 root root 964608 Oct 30 17:07 /bin/bash*
-bash-4.2$ /bin/bash
bash-4.2$ id
uid=1000(mej) gid=1000(mej) groups=1000(mej)
bash-4.2$
```

MYTH: User namespaces are too new to be considered secure.

FACT: User namespaces were introduced in Linux 3.8 (2013) and have remained substantially unchanged since 3.19 (2015).

Vulnerabilities in user namespaces have been minimal recently:

- Last CVE attributable to the unprivileged user namespace implementation was CVE-2014-8989.
- Vulnerabilities enabled by user namespaces have happened, roughly 2-4 each year. (SELinux typically, though not always, prevents exploitation of these.)
- Container solutions which leverage unprivileged user namespaces (Charliecloud, PodMan, Rootless RunC) were unaffected by nested user namespace issue (CVE-2018-18955); they also protect against the RunC binary replacement issue (CVE-2019-5736) when correctly configured.

Most experts working on end-user containers are focused on user namespaces.

- For all the reasons we already talked about: in particular, the kernel-based trust and security model.
- The safest path is the one where the bulk of the brain trust has its focus.
- It's fine to invent your own solution, but that's a lot to own. Make sure technical rationale is sound!

Standards are good for everyone!

• Charliecloud

LANL's Container Runtime

- Available on GitHub: https://github.com/hpc/charliecloud

2018 R&D 100 Winner!

- Supercomputing 2017 Paper by Reid Priedhorsky and Tim Randles
  - "Charliecloud: Unprivileged Containers for UDSS in HPC"
  - Los Alamos Tech Report LA-UR-17-30438
  - http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-17-30438
- ;login: Article "Linux Containers for Fun & Profit in HPC" by Reid Priedhorsky
  - https://www.usenix.org/publications/login/fall2017/priedhorsky
- "Minimizing privilege for building HPC containers"
  - https://dx.doi.org/10.1145/3458817.3476187
- Documentation: https://hpc.github.io/charliecloud (includes tutorials!)
- Source Code: https://github.com/hpc/charliecloud
- Mailing List: charliecloud@groups.io || https://groups.io/charliecloud
- Contact Reid (reidpr@lanl.gov), Tim (trandles@lanl.gov), or Michael (mej@lanl.gov, @mej0 on Twitter)

Any Questions?

Michael Jennings

Los Alamos National Laboratory

mej@lanl.gov **||** ***mej@eterm.org***

f/kainx || t/@mej0 || i/kainx