Los Alamos
NATIONAL LABORATORY
EST.1943

LA-UR-19-21161
Approved for public release; distribution is unlimited.

Title: Container Mythbusters

Author(s): Jennings, Michael E.

Intended for: HPC/AI Advisory Council 2019 Stanford Conference,
2019-02-14/2019-02-15 (Palo Alto, California, United States)

Issued: 2019-02-20 (rev.1)

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

National Nuclear Security Administration
Operated by Triad National Security, LLC for the U.S. Department of Energy's NNSA

Container Mythbusters

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

Michael Jennings (@mej0) – *mej@lanl.gov*
Platforms Team Lead, HPC Systems Group
Los Alamos National Laboratory

2019 Stanford Conference
HPC/AI Advisory Council
Stanford University, Palo Alto, CA
15 February 2019

Operated by Triad National Security, LLC for the U.S. Department of Energy's NNSA

UNCLASSIFIED

LA-UR-19-21161

Los Alamos National Laboratory

Introduction

- Established in 1943 as “Site Y” of the Manhattan Project
- Mission: To solve National Security challenges through Scientific Excellence
- One of the largest science and technology institutes in the world, conducting multidisciplinary research in fields such as national security, space exploration, renewable energy, medicine, nanotechnology, and supercomputing.

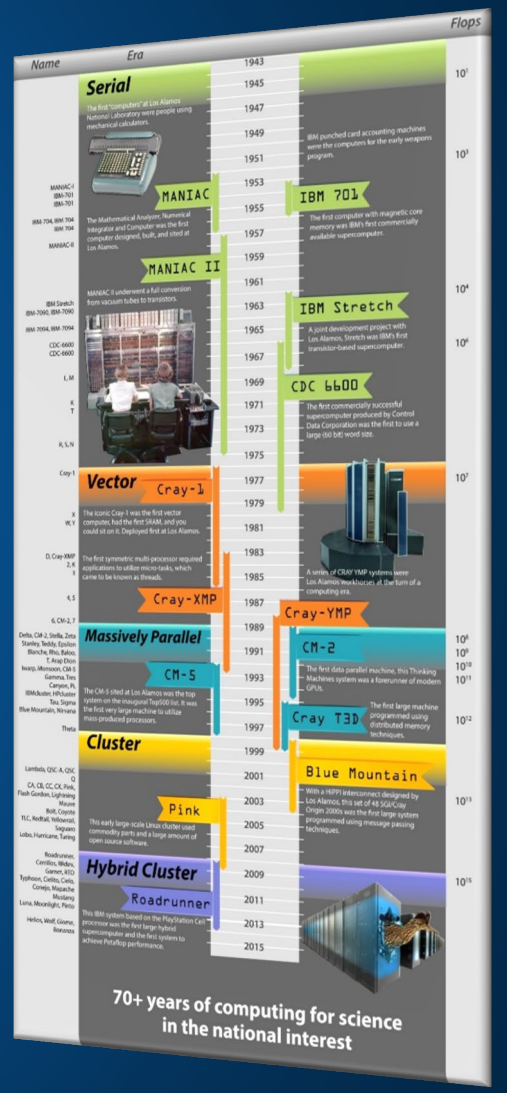
- Funded primarily by the Department of Energy, we also do extensive work for/with the Departments of Defense and Homeland Security, the Intelligence Community, et al.
- Our strategy reflects US government priorities including nuclear security, intelligence, defense, emergency response, nonproliferation, counterterrorism, and more.
- We help to ensure the safety, security, and effectiveness of the US nuclear stockpile.
- Since 1992, the United States no longer performs full-scale testing of nuclear weapons. This has necessitated continuous, ongoing leadership in large-scale simulation capabilities realized through investment in high-performance computing.

LANL High-Performance Computing Division

Introduction

- **LANL's history in HPC dates back to the early '50s.**
- **Accomplishments include:**
  - Helped IBM develop Stretch, the 1st transistor-based supercomputer
  - The 1st vector computer, Cray-1, deployed here
  - Our CM-5 was #1 on the inaugural Top500 List
  - 1st hybrid supercomputer (using IBM POWER and PlayStation Cell processors), Roadrunner, was also 1st to break the PetaFLOP/s barrier
- **Led by Gary Grider, creator of Burst Buffer technology**

- We support over 2000 unique users across more than 100 different classified/open science projects on 20+ clusters



LANL has been a leader in HPC since before HPC was HPC!

MYTH: Containers are …*insert definition here*…

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

What are containers?

"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."
- Frequently this concept 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 and *is the one we'll use*.
- I'm not perfectly consistent about this either, so if the meaning isn't clear from context, feel free to **ask**!

Containers are, fundamentally, processes! More on that to come…

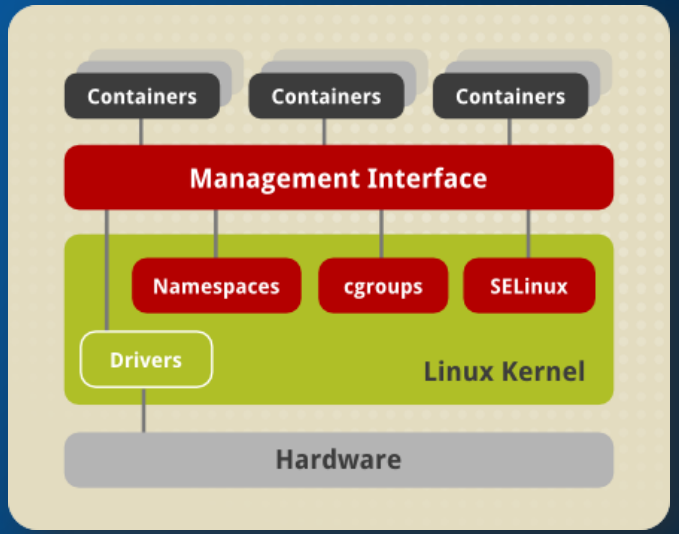


Image credit: Red Hat

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.

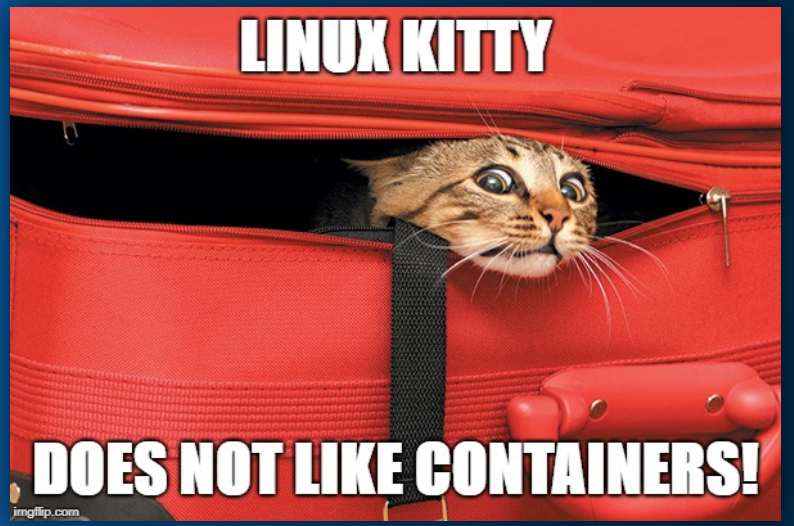
- **6 Privileged Namespaces** (require `CAP_SYS_ADMIN` to create)
  - `mount` – Private filesystem mount points, recursion/propagation controls
  - `pid` – Private view of process IDs and processes, `init` semantics
  - `uts` – Private hostname and domainname values
  - `net` – Private network resources (devices, IPs, routes, ports, etc.)
  - `ipc` – Private IPC resources (SysV IPC objects, POSIX msg queues)
  - `cgroup` – Private control group hierarchy (Linux 4.6+ only)
- **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)`

Containers are lies we tell ourselves. Or, rather, our applications.

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(*_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



Containers are lies we tell ourselves. Or, rather, our applications.

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.

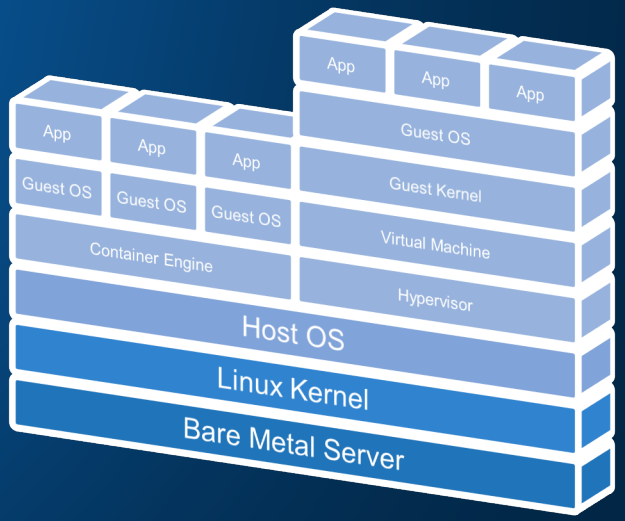
Application Containers

In the Docker/OCI ecosystem, when you build a 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* 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. So there are exceptions!

- The `system-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 job containers are app containers. Microservices containers aren't!



Container runtimes differ. Ask your doctor which one is right for you!

MYTH: Containers contain.

MYTH: Containers don’t contain.

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

Container Containment

Containers are primarily an abstraction & encapsulation technique, 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.

There's "Secure," and there's "Not Exactly." Make sure you choose the right one!

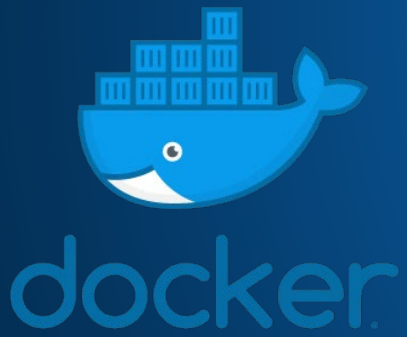
MYTH: “Container” is shorthand for “Docker Container.”

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

The Vast Container Landscape

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.



Like any tool, Docker isn't always the right choice. Expand your toolbox!

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

The Vast Container Landscape

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 model 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 privileged runtime).
- 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 ecosystem.
- While not focused on the use cases of HPC, Red Hat's `podman` offers `runc`-based OCI compliance *and* addresses many of the issues with Docker. Unprivileged containers are now fully supported.

Like any tool, Docker isn't always the right choice. Expand your toolbox!

MYTH: Containers are hard and require complicated tools like Docker or Rkt.

FACT: Containers are easy, at least for the basics. These days, you can even write your own container-based solutions in BASH!

Simply Contained

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.

The gory details, however, ***are*** complex…so use an existing solution, and understand why!

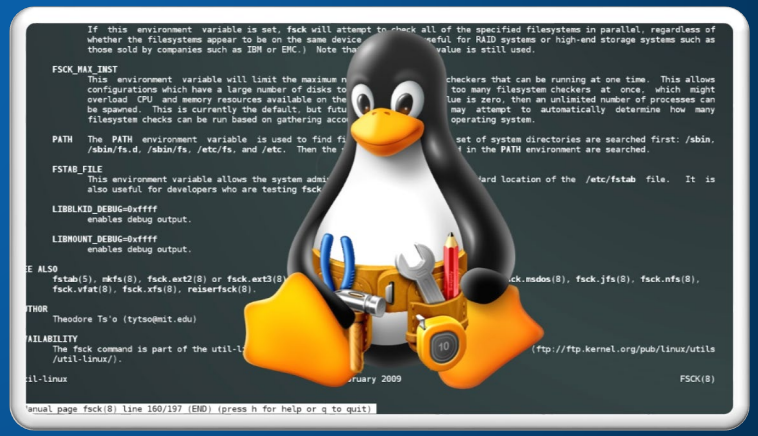


Image Credit: Toca do Tux

Unless you can clearly articulate the technical rationale, don't write your own!

MYTH: Docker is insecure.

FACT: Docker's security record is sub-optimal.

Security through Insecurity?

Since early 2017, there have been **17** vulnerabilities that could lead to kernel panics, host information leaks, and privilege escalation inside or outside the container!

- Only 2 CVEs were obtained, both of which were within the past 6 months.
- One particular release fixed a total of **SIX** vulnerabilities, including 2 buffer overflows. No CVE IDs.
- At least 1 CVE covers multiple vulnerabilities, including the ability to join and affect the root namespace, test for arbitrary file existence as `root`, and escalate to `root` by adding content to `/usr/bin`.
- 7 of the 9 releases in 2018 were for fixes to vulnerabilities, almost all of which were high severity.

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.*"

"There is no supported means for privilege escalation…so no additional controls [are needed]."

FACT: 's security record is sub-optimal.

Security through Insecurity?

Since early 2017, there have been **17** vulnerabilities that could lead to kernel panics, host information leaks, and privilege escalation inside or outside the container!

- Only 2 CVEs were obtained, both of which were within the past 6 months.
- One particular release fixed a total of **SIX** vulnerabilities, including 2 buffer overflows. No CVE IDs.
- At least 1 CVE covers multiple vulnerabilities, including the ability to join and affect the root namespace, test for arbitrary file existence as `root`, and escalate to `root` by adding content to `/usr/bin`.
- 7 of the 9 releases in 2018 were for fixes to vulnerabilities, almost all of which were high severity.

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.*"

"There is no supported means for privilege escalation…so no additional controls [are needed]."

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

Docker access IS root access!

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 since 2016 (the first year all 4 were available publicly), Docker compares favorably:

	Charliecloud	Docker	Shifter	Singularity
Vulnerability Count	0	0 (or 5)	0	~~2~~ 17+

Even so, it's 2019! We have much better options today.

- 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.).

If you open up access to it, then Docker isn't what's vulnerable…YOU ARE!

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, and no single existing solution fully solves the reproducibility problem.

Relatively Reproducible?

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 and 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 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!

Many folks mistakenly say "reproducible" when they really mean "prescriptive."

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 a multifaceted and highly nuanced issue. That claim reflects incomplete/insufficient understanding.

Security Oversimplified.

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 required to deal with combinations of namespaces and scenarios common to HPC (e.g., in-memory root filesystems). (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$
```

Exposing privileged operations to unprivileged users requires deep expertise!

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).

The Road Not Not Taken

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 namespace root access have happened, 2-4 each year 2015-2017.
- Container solutions which leverage unprivileged user namespaces (Charliecloud, PodMan, Rootless RunC) were unaffected by recent nested user namespace issue (CVE-2018-18955); they also protect against the new 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!

If your container vendor is blazing their own trail, ask yourself…how fireproof are you?

Charliecloud

Speaking of which…

- **LANL's Container Runtime**
  - Available on GitHub: https://github.com/hpc/charliecloud
- **2018 R&D 100 Winner!**

- **Recent developments in version 0.9.x (currently 0.9.7):**
  - New `Vagrantfile` for generating Charliecloud-enabled (and Docker-enabled) VM images based on CentOS 7 Virtualbox image.
  - New example containers and tutorials based on MPICH, Spack, `spokeo`, `umoci`, OpenMPI 3.1.3, and more.
  - New `ch-fromhost` utility to seamlessly integrate host-based resources into Charliecloud containers (HSN, GPU, libraries, etc.)
  - Improved spec file for potential future inclusion in upstream distros.
  - Significantly improved documentation (and how it gets generated on RHEL-based platforms)

Any Questions?