Rootless containers with Podman and fuse-overlayfs

Giuseppe Scrivano
@gscrivano
Introduction
Rootless Containers

• “Rootless containers refers to the ability for an unprivileged user (i.e. non-root user) to create, run and otherwise manage containers.” ([https://rootlesscontaine.rs/](https://rootlesscontaine.rs/))

• Not just about running the container payload as an unprivileged user

• Container runtime runs also as an unprivileged user
Don’t confuse with...

- `sudo podman run --user foo`
  - Executes the process in the container as non-root
  - Podman and the OCI runtime still running as root

- `USER` instruction in Dockerfile
  - same as above
  - Notably you can’t `RUN dnf install ...`
Don't confuse with...

- `podman run --uidmap`
  - Execute containers as a non-root user, using user namespaces
  - Most similar to rootless containers, but still requires podman and runc to run as root
Motivation of Rootless Containers

• To mitigate potential vulnerability of container runtimes

• To allow users of shared machines (e.g. HPC) to run containers without the risk of breaking other users environments

• To isolate nested containers
Caveat: Not a panacea

• Although rootless containers could mitigate these vulnerabilities, it is not a panacea, especially it is powerless against kernel (and hardware) vulnerabilities – CVE 2013-1858, CVE-2015-1328, CVE-2018-18955 😞

• Castle approach🏰: it should be used in conjunction with other security layers such as seccomp and SELinux
Podman
Podman is a daemon-less alternative to Docker

• $ alias docker=podman

• Better integration with systemd
## Rootless Podman

<table>
<thead>
<tr>
<th></th>
<th>Root</th>
<th>Rootless</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage</strong></td>
<td>/var/lib/containers</td>
<td>$HOME/.local/share/containers</td>
</tr>
<tr>
<td><strong>Runtime data</strong></td>
<td>/run/libpod</td>
<td>$XDG_RUNTIME_DIR/libpod (/run/user/1000/libpod)</td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>/etc/containers</td>
<td>$HOME/.config/containers</td>
</tr>
</tbody>
</table>
$ podman run --name nginx -d -p 8080:80 -v ~/public_html:/usr/share/nginx/html:ro nginx

Podman process (euid=$UID)
  - Join existing user+mount namespace, or create one

Podman process (euid=0)
  - conmon
  - OCI runtime
  - container process

Setup the network namespace with slirp4netns

Setup the storage and the fuse-overlayfs mount
Implementation details
User Namespaces

• The key component of rootless containers.
  – Map UIDs/GIDs in the guest to different UIDs/GIDs on the host.
  – Unprivileged users can have (limited) root inside a user namespace!

• Root in a user namespace has UID 0 and full capabilities, but obvious restrictions apply.
  – Inaccessible files, inserting kernel modules, rebooting, ...
User Namespaces

Host namespace

```
... 1000 ...
... 110000 110001 110002 110003 ...
```

1st level user namespace

```
0 1 2 3 4 ...
```

2nd level user namespace

```
0 1
...```

Red Hat
User Namespaces

- To allow multi-user mappings, shadow-utils provides `newuidmap` and `newgidmap` (packaged by most distributions).
  - SETUID binaries writing mappings configured in `/etc/sub[ug]id`

/`etc/subuid`:
```
1000:420000:65536
```

/`proc/42/uid_map`:
```
0  1000   1
1 420000  65536
```

- Provided by the admin (real root)
- User can configure map UIDs after unsharing a user namespace
User Namespaces

Problems:

• SETUID binary can be dangerous
  • newuidmap & newgidmap had two CVEs so far:
    • CVE-2016-6252 (CVSS v3: 7.8): integer overflow issue
    • CVE-2018-7169 (CVSS v3: 5.3): supplementary GID issue
• Hard to maintain subuid & subgid
  • Having 65536 sub-IDs should be ok for most cases, but to allow nesting user namespaces, an enormous number of sub-IDs would be needed
    • Potential sub-ID starvation
User Namespaces

Alternative way: Single-mapping mode

• Single-mapping mode does not require `newuidmap/newgidmap`
• There is only one UID/GID available in the container

Limit the privileges of `newuidmap/newgidmap`

• Install them using file capabilities rather than SETUID bit
  – Only `CAP_SETUID` and `CAP_SETGID` are needed
Network Namespaces

• An unprivileged user can create network namespaces along with user namespaces

• With network namespaces, the user can
  – create iptables rules
  – isolate abstract (pathless) UNIX sockets
  – set up overlay networking with VXLAN
  – run tcpdump
  – ...

Red Hat
Network Namespaces

• But an unprivileged user cannot set up `veth` pairs across the host and namespaces, i.e. No internet connection.
Network Namespaces

Prior work: LXC uses SETUID binary (*lxc-user-nic*) for setting up the *veth* pair across the host and containers

Problem: SETUID binary can be dangerous! ⚠️
- CVE-2017-5985 (CVSS v3: 3.3): netns privilege escalation
- CVE-2018-6556 (CVSS v3: 3.3): arbitrary file `open(2)`
we use a completely unprivileged usermode network ("slirp") with a TAP device

send fd as SCM_RIGHTS cmsg via an UNIX socket
Network Namespaces

Benchmark of several “Slirp” implementations:

<table>
<thead>
<tr>
<th></th>
<th>MTU=1500</th>
<th>MTU=4000</th>
<th>MTU=16384</th>
<th>MTU=65520</th>
</tr>
</thead>
<tbody>
<tr>
<td>vde_plug</td>
<td>763 Mbps</td>
<td>Unsupported</td>
<td>Unsupported</td>
<td>Unsupported</td>
</tr>
<tr>
<td>VPNKit</td>
<td>514 Mbps</td>
<td>526 Mbps</td>
<td>540 Mbps</td>
<td>Unsupported</td>
</tr>
<tr>
<td>slirp4netns</td>
<td>1.07 Gbps</td>
<td>2.78 Gbps</td>
<td>4.55 Gbps</td>
<td>9.21 Gbps</td>
</tr>
<tr>
<td>cf. rootful veth</td>
<td>52.1 Gbps</td>
<td>45.4 Gbps</td>
<td>43.6 Gbps</td>
<td>51.5 Gbps</td>
</tr>
</tbody>
</table>

- slirp4netns (based on QEMU Slirp) is the fastest because it avoids copying packets across the namespaces
Multi-node networking

- Flannel VXLAN is known to work
  - Encapsulates Ethernet packets in UDP packets
  - Provides L2 connectivity across rootless containers on different nodes

- Other protocols should work as well, except ones that require access to raw Ethernet
/sys/fs/cgroup is a roadblock to many features we want in rootless containers (accounting, pause and resume, even getting a list of PIDs!).

• By default completely owned by root (and managed by systemd).

Some workarounds:

• LXC’s pam_cgfs requires installation of a PAM module (and only works for logged-in users). It needs to be used carefully as it gives cgroupv1 write access to unprivileged users.
• cgroup namespaces (with nsdelegate) only work in cgroupv2.
cgroups v2

- Safe to use for unprivileged user
- An entire subtree is delegated to the user
- The file path is not the only difference

/sys/fs/cgroup/memory/foo/bar/memory.limit_in_bytes
/sys/fs/cgroup/cpu/foo/bar/cpu.shares

...
cgroups v2

- OCI runtime specs are designed around cgroup v1
- supporting cgroup v2 will require changes in the OCI specs
Storage
The container root filesystem has to live somewhere. Many filesystem features used by “rootful” container runtimes aren’t available.

- Ubuntu allows overlayfs in a user namespace, but this isn't supported upstream (due to security concerns).

- BTRFS allows unprivileged subvolume management, but requires privileges to set it up beforehand.

- Devicemapper is completely locked away from us.
A “simple” work-around is to just extract images to a directory!

- It works … but people want storage deduplication.

Alternatives:

- Reflinks to a "known good" extracted image (inode exhaustion).
  - (Can use on XFS, btrfs, ... but not ext4.)
- Unprivileged userspace overlayfs using FUSE (Kernel 4.18+).
fuse-overlayfs

- Overlayfs implementation using FUSE
- Layers deduplication as for root containers
- Fast setup for a new container
- Built-in support for shifting UIDs/GIDs

- Adds complexity
fuse-overlayfs UIDs/GIDs shifting

- When creating a user namespace, we must ensure proper ownership of the files in the RO layers.

- The file system “lies” about the owner, so that it has the correct UID/GID in the user namespace and the same layer on disk can be used by different user namespaces.

- Less expensive alternative to `cp -r` and `chown`'ing the entire image and layers.
fuse-overlayfs UIDs/GIDs shifting

Namespace configuration

1000  ->  0
110000:4096  ->  1..4096

1000  ->  0
118000:4096  ->  1..4096

From the host

/usr/bin/ls  1000:1000
/usr/bin/write 1000:110004

From the container

/usr/bin/ls  0:0
/usr/bin/write 0:5

/usr/bin/ls  0:0
/usr/bin/write 0:5
Questions?

gscrivan@redhat.com

@gscrivano