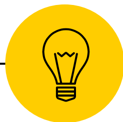
Rooting out Root: User namespaces in Docker



Phil Estes

Senior Technical Staff Member, Open Technologies, IBM Cloud

@estesp

estesp@gmail.com

I work for IBM's Cloud division

We have a large organization focused on open cloud technologies, including **CloudFoundry**, **OpenStack**, and **Docker**.

I have been working upstream in the Docker community since **July 2014**, and am currently a Docker core **maintainer**.

I have interests in **runC** (IBM is a founding member of OCI), **libnetwork**, and the **docker/distribution** project (Registry v2)

Trivia: I worked in IBM's **Linux** Technology Center for over 10 years!

Why user namespaces?

Security

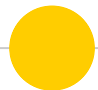
Currently, by default, the user inside the container is **root**; more specifically `uid = 0, gid = 0`. If a breakout were to occur, the container user is **root** on the host system.

Multitenancy

Sharing Docker compute resources among more than one user requires isolation between tenants. Providing `uid`/`gid` ranges per tenant will allow for this separation.

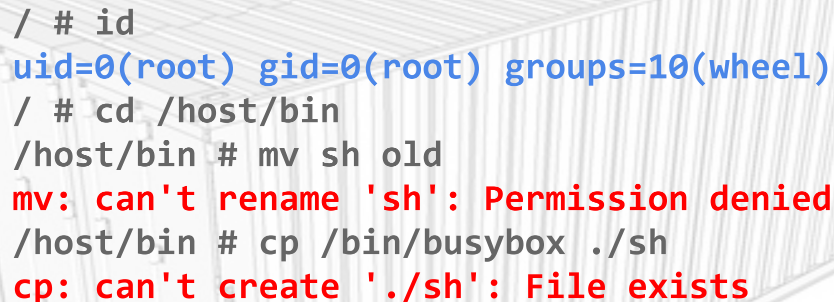
User Accounting

Any per-user accounting capabilities are useless if everyone is **root**. Specifying unique `uid`s enables resource limitations specific to a user/`uid`.



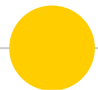
Added Security

```
$ docker run -v /bin:/host/bin -ti busybox /bin/sh
```

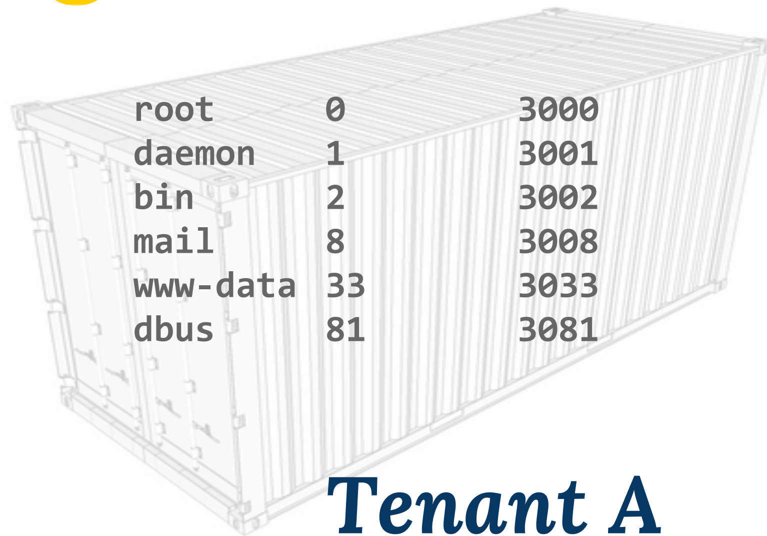


```
/ # id
uid=0(root) gid=0(root) groups=10(wheel)
/ # cd /host/bin
/host/bin # mv sh old
mv: can't rename 'sh': Permission denied
/host/bin # cp /bin/busybox ./sh
cp: can't create './sh': File exists
```

Host **root** ≠ Container **root**

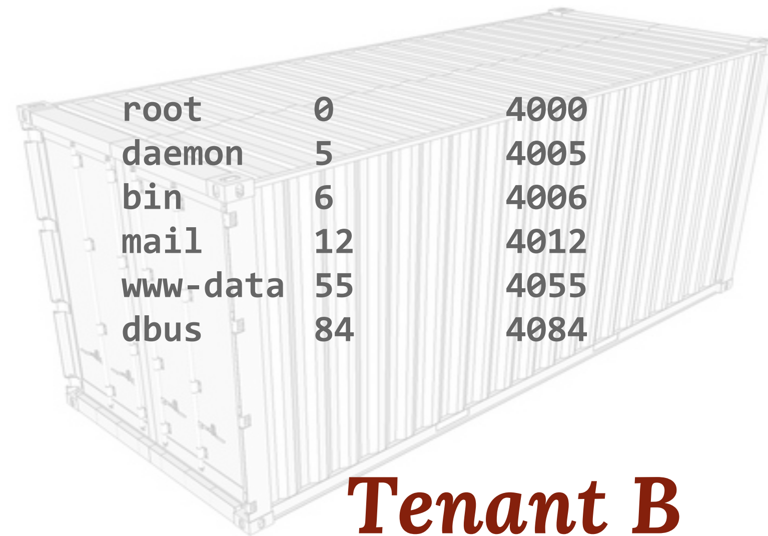


Multitenant Services



root	0	3000
daemon	1	3001
bin	2	3002
mail	8	3008
www-data	33	3033
dbus	81	3081

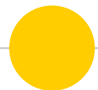
Tenant A



root	0	4000
daemon	5	4005
bin	6	4006
mail	12	4012
www-data	55	4055
dbus	84	4084

Tenant B

Full UID and GID namespace separation between tenants in the same hosted cloud



User Accounting/Limits

```
$ ulimit -n
2048
$ docker run lotsofiles
```

Container root uid = 2000

```
# of Open Files Limit:
1024
-----------------------
Can't open temp file #1021, error: open
/tmp/zzz378562286: too many open files
```

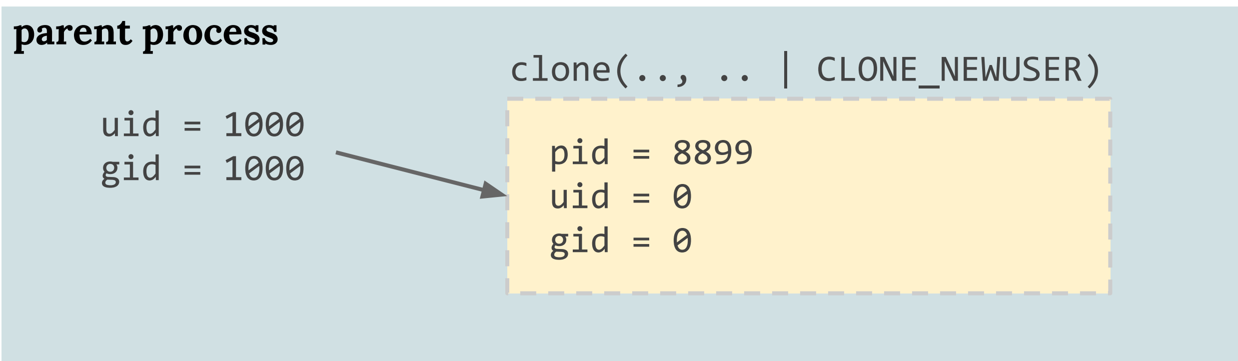
Docker Security

User namespaces are only **one** piece of the puzzle.

AppArmor/SELinux, Notary, image security, and proper environment/network security all play a part in the overall Docker security picture.

Linux user namespaces

- Available as a `clone()` flag [`CLONE_NEWUSER`] in Linux kernel 3.8 (some work completed in 3.9)
- Per-process namespace to map user and group IDs to a specified set of numeric ranges



“Most notably, a process can have a nonzero user ID outside a namespace while at the same time having a user ID of zero inside the namespace; in other words, the process is unprivileged for operations outside the user namespace but has root privileges inside the namespace.”

https://lwn.net/Articles/532593/
Michael Kerrisk, February 27, 2013

User namespaces and Go

- Available since Go version 1.4.0 (October 2014) as fields within the `syscall.SysProcAttr` structure: arrays `UidMappings` and `GidMappings`
- Thanks to good work from **Mrunal Patel** and **Michael Crosby** laying the Go-lang groundwork for user namespace capability within Docker/libcontainer
(https://github.com/golang/go/issues/8447)

Go user namespaces example

```
var sys *syscall.SysProcAttr

sys.UidMappings = []syscall.SysProcIDMap{{
     ContainerID: 0,
     HostID:      1000,
     Size:        1,
}}
sys.GidMappings = []syscall.SysProcIDMap{{
     ContainerID: 0,
     HostID:      1000,
     Size:        1,
}}

sys.Cloneflags = syscall.CLONE_NEWUSER

cmd := exec.Cmd{
      Path:         "/bin/bash",
      SysProcAttr: sys,
}
```

When we run this code we'll have a command that, when executed, will appear to be running as ***root*** *(uid/gid = 0), but will actually be the non-privileged user with uid/gid = 1000* ***mapped*** *inside the user namespace to* ***root****.*

Well, that was easy!

What's the holdup?

At this point you might be asking yourself: "So why doesn't Docker have user namespace support yet!?"

Let's take a deeper look at some of the challenges…

1

File Ownership

Who can read the metadata, image layers, and associated files across Docker's runtime store?

Docker Filesystem Hierarchy

```
drwx------ root:root  /var/lib/docker
```

Docker’s metadata tree is rooted (by default) at `/var/lib/docker` and only accessible to user **root**

```
drwx------ root:root  /var/lib/docker/containers
```

The container metadata also contains files which are bind-mounted into the container as **`root:root`** today

```
drwx------ root:root  /var/lib/docker/aufs
                                      btrfs
                                      devicemapper
                                      overlay
                                      vfs
                                      zfs
```

Depending on your chosen storage driver, the actual layer content of Docker images will be placed here in a root-owned directory path

```
drwx------ root:root  /var/lib/docker/{tmp, volumes, ..}
```

Other various locations exist which may need to be accessed by container processes

File Ownership Solution

- At Docker daemon startup, if remapped root is enabled, create a new subtree owned by the remapped root's `uid` and `gid`

```
# docker daemon --root=2000:2000 ...
drwxr-xr-x root:root   /var/lib/docker
drwx------ 2000:2000   /var/lib/docker/2000.2000
```

- Whenever the Docker daemon components create a directory structure, take remapped root into account
- This solves an additional issue around file ownership that we'll discuss next

2

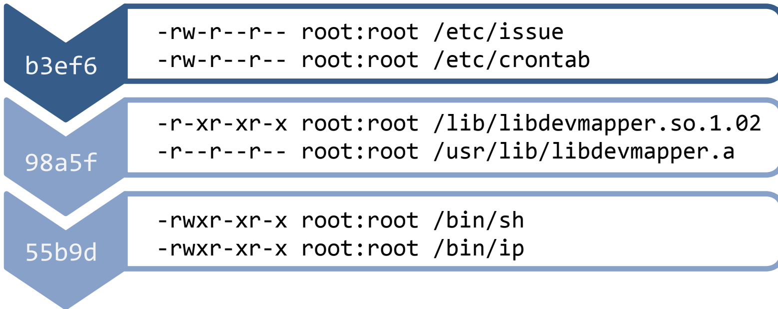
Layer Sharing

Docker images are downloaded to the local daemon's cache from a registry and expanded into the storage driver's subtree by ID

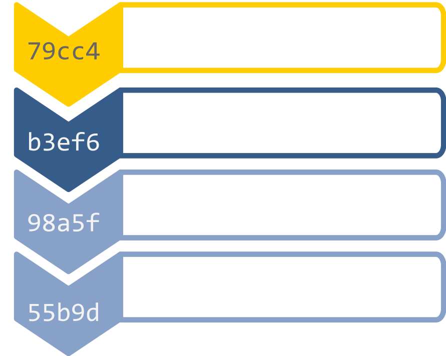
Docker Image Layer Details

useless:1.0

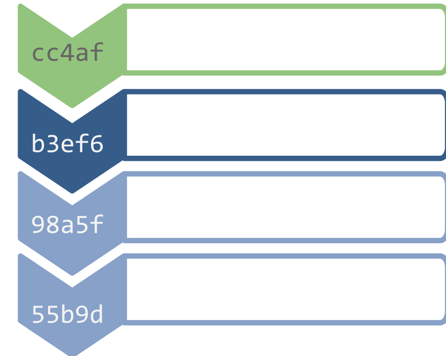
```
FROM ubuntu:15.04
…
RUN apt-get install libdevmapper \
    libdevmapper-dev
…
COPY issue crontab /etc/
```



`docker run useless:1.0 /bin/sh`



`docker run useless:1.0 /bin/sh`



Layer Sharing ~~Solution~~ *restriction?*

Given:

- Already mentioned: one metadata subtree per **remapped root**
- Remapped root setting is daemon-wide (for all containers running in this instance)

Therefore we:

- Untar all layers per the user namespace `uid/gid` mapping provided at daemon start
- All layers are usable (correct ownership) by any container in this daemon instance

Layer Solution Pros/Cons

Pros

No ugly `chown -R uid:gid <huge file tree>` work to do at container start time.

For a daemon-wide user namespace setting, this solution works perfectly for the general "don't be **root**" case.

Cons

Restarting the daemon with/without remapped roots resets the metadata cache (must re-pull images, no prior container history)

Some increased disk cost if daemon is started with unique remappings or turned on/off

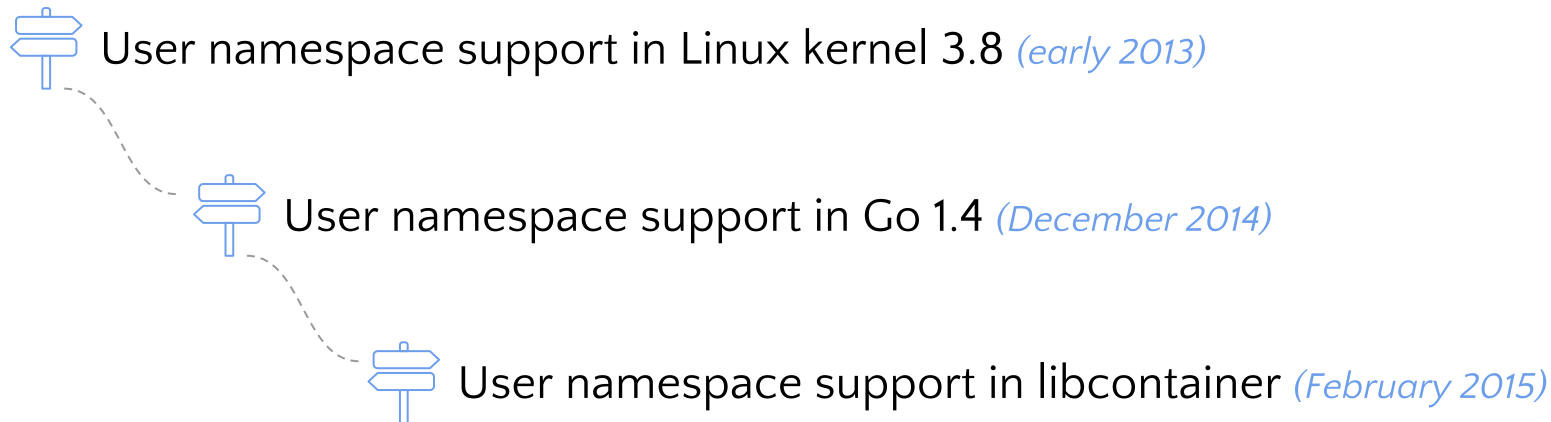
3

Namespace Order

When not using `clone()` to create all namespaces, joining other namespace types (PID, UTC, Network) may not work properly depending on the order of operations

Namespace Sharing/Ordering

- Joining a namespace (e.g. network) which was not created in the context of a user namespace will not work as expected.
- Prior to Docker 1.7, this meant `--net=<container>` was impacted by adding the user namespace function.
- In Docker 1.7, **libnetwork** took over the role of Linux network namespace creation, introducing the same ordering problem as `--net=<container>`
- These issues are being resolved now; more info in https://github.com/docker/docker/issues/15187



So where are we ***now***?

User Namespace Status

- Namespace sharing/ordering details & design are resolved; implementation/changes underway in **runC** and **libnetwork**
  - runC hooks PR: https://github.com/opencontainers/runc/pull/160
  - libnetwork tracker: https://github.com/docker/libnetwork/issues/429
- “**Phase 1**” user namespace implementation (remapped root per daemon instance) targeted for **Docker 1.9**
  - tracking issue: https://github.com/docker/docker/issues/15187
  - code PR: https://github.com/docker/docker/pull/12648
- “**Phase 2**”--providing full maps and allowing per-container maps--is still under discussion

"Phase 1" Usage Overview

```
# docker daemon --root=2000:2000 ...
drwxr-xr-x root:root  /var/lib/docker
drwx------ 2000:2000  /var/lib/docker/2000.2000
```

Start the daemon with a remapped root setting (in this case uid/gid = 2000/2000)

```
$ docker run -ti --name fred --rm busybox /bin/sh
/ # id
uid=0(root) gid=0(root) groups=10(wheel)
```

Start a container and verify that inside the container the uid/gid map to root (0/0)

```
$ docker inspect -f '{{ .State.Pid }}' fred
8851
$ ps -u 2000
  PID TTY          TIME CMD
 8851 pts/7    00:00:00 sh
```

You can verify that the container process (PID) is actually running as user 2000

Thanks!

Any questions?

@estesp

github.com/estesp

estesp@gmail.com

http://integratedcode.us

Credits

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival