Linuxcon 2015

# Using Docker for existing installed OS and applications, running half inside half outside the container

**http://marc.merlins.org/linux/talks/DockerLocalDisk-LC2015-JP/**
**http://goo.gl/PWjume** (short link)

Marc MERLIN
marc_soft@merlins.org

# Another docker talk? Isn't everyone using docker already?

## With a cool logo like this, you have no excuse :)

## Docker: the shipping container model

- Package your application once, make it work on any OS
- …
- Profit!

## Why Use Containers for security?

- Virtual machines are of course better separated and more secure.
- But VMs require more resources, run their own kernel, duplicate an entire system, and run slower than native.
- Containers give separation of resources. Root in the container is reasonably protected from accessing resources outside.
- Containers also give control over CPU/memory/I/O resources used by containers

## Why was I not using containers and LXC?

- I had a TODO from 2011 to migrate to LXC, but never got around to it.
- I don't want to maintain 2 or more operating systems (the base system and each docker image's OS) for security updates or local tweaks/patches
- Containers weren't that secure back then, you could escape them pretty easily.
- Setting them up wasn't trivial.

## What does Docker offer over LXC?

- Standard images of linux distributions ready for use (if you are ok with trusting them).
- A standard interface compatible across linux distributions (shipping container model)
- Images that can be modified at run time and rolled back to their clean install state when restarted
- Network integration, virtual filesystems, data sharing between containers, and better security out of the box

# Docker: Security bits that come for free

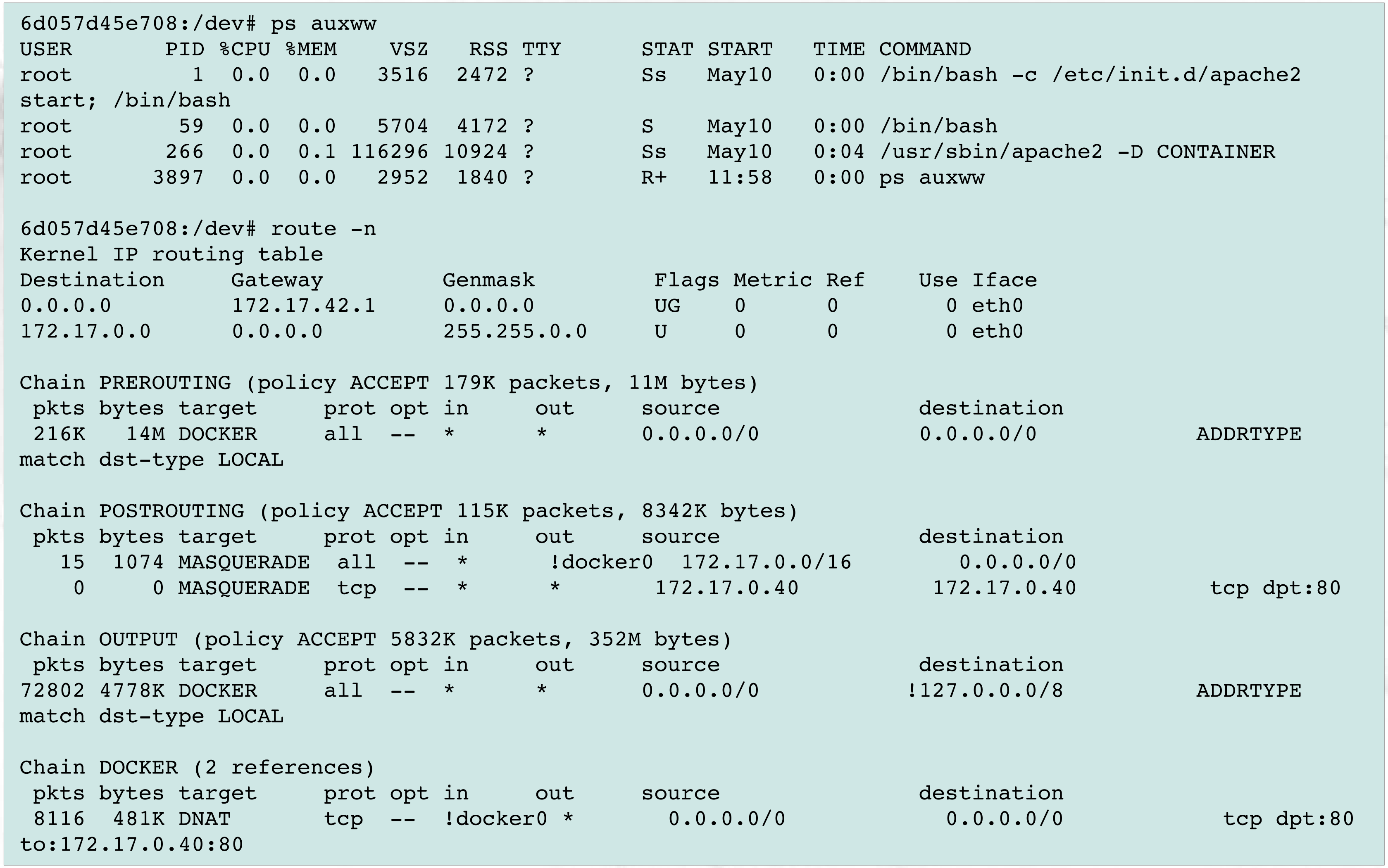
- Docker does a good job of securing known bits to prevent a container from accessing the host
- For /proc it uses bind mounts to disable dangerous entry points:

```
proc on /proc/asound type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/bus type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/fs type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/irq type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sys type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sysrq-trigger type proc (ro,nosuid,nodev,noexec,relatime)
tmpfs on /proc/kcore type tmpfs (rw,nosuid,mode=755)
tmpfs on /proc/latency_stats type tmpfs (rw,nosuid,mode=755)
tmpfs on /proc/timer_stats type tmpfs (rw,nosuid,mode=755)
```

# Docker protects device nodes

```
6d057d45e708:/dev# ls -l
total 0
crw------- 1 root root 136,  13 May 12 11:51 console
lrwxrwxrwx 1 root root       13 May 10 21:59 fd -> /proc/self/fd
crw-rw-rw- 1 root root   1,   7 May 10 21:59 full
c--------- 1 root root  10, 229 May 10 21:59 fuse
lrwxrwxrwx 1 root root       11 May 10 21:59 kcore -> /proc/kcore
drwxrwxrwt 2 root root       40 May 10 21:59 mqueue
crw-rw-rw- 1 root root   1,   3 May 10 21:59 null
lrwxrwxrwx 1 root root        8 May 10 21:59 ptmx -> pts/ptmx
drwxr-xr-x 2 root root        0 May 10 21:59 pts
crw-rw-rw- 1 root root   1,   8 May 10 21:59 random
drwxrwxrwt 2 root root       40 May 10 21:59 shm
lrwxrwxrwx 1 root root       15 May 10 21:59 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root       15 May 10 21:59 stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root       15 May 10 21:59 stdout -> /proc/self/fd/1
crw-rw-rw- 1 root root   5,   0 May 10 21:59 tty
crw-rw-rw- 1 root root   1,   9 May 10 21:59 urandom
crw-rw-rw- 1 root root   1,   5 May 10 21:59 zero
6d057d45e708:/dev# mknod sda b 8 0
6d057d45e708:/dev# fdisk /dev/sda
fdisk: unable to open /dev/sda: Operation not permitted
6d057d45e708:/dev# mknod sda1 b 8 1
6d057d45e708:/dev# mount ./sda1 /mnt
mount: permission denied
```

# Docker protects processes, users, networking



```
6d057d45e708:/dev# ps auxww
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   3516  2472 ?        Ss   May10   0:00 /bin/bash -c /etc/init.d/apache2
start; /bin/bash
root        59  0.0  0.0   5704  4172 ?        S    May10   0:00 /bin/bash
root       266  0.0  0.1 116296 10924 ?        Ss   May10   0:04 /usr/sbin/apache2 -D CONTAINER
root      3897  0.0  0.0   2952  1840 ?        R+   11:58   0:00 ps auxww

6d057d45e708:/dev# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.17.42.1     0.0.0.0         UG    0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0

Chain PREROUTING (policy ACCEPT 179K packets, 11M bytes)
 pkts bytes target     prot opt in     out     source               destination
 216K   14M DOCKER     all  --  *      *       0.0.0.0/0            0.0.0.0/0            ADDRTYPE
match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 115K packets, 8342K bytes)
 pkts bytes target     prot opt in     out     source               destination
   15  1074 MASQUERADE  all  --  *      !docker0  172.17.0.0/16        0.0.0.0/0
    0     0 MASQUERADE  tcp  --  *      *       172.17.0.40          172.17.0.40          tcp dpt:80

Chain OUTPUT (policy ACCEPT 5832K packets, 352M bytes)
 pkts bytes target     prot opt in     out     source               destination
72802 4778K DOCKER     all  --  *      *       0.0.0.0/0           !127.0.0.0/8          ADDRTYPE
match dst-type LOCAL

Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
 8116  481K DNAT       tcp  --  !docker0 *       0.0.0.0/0            0.0.0.0/0            tcp dpt:80
to:172.17.0.40:80
```

## Why didn't I try docker earlier?

- Docker came to make containers easier to install and deploy everywhere
- Didn't address my use-case of containerizing an existing already installed system.
- I didn't want to trust an outside linux image (beware: docker image pull checksums aren't secure,this is being worked on)
- If docker gets compromised and its images changed, the fallout would be pretty horrible.
- I have a very customized Debian image, I didn't want to redo all these changes in a new image I'd build myself from scratch.

# Why didn't I try docker earlier? (2)

- I don't want to waste disk and more importantly RAM by having multiple slightly different copies of the same code and multiple copies of similar shared libraries
- It took me a long time to get custom apps working on my base system and don't really want to do the work a 2nd time in a fresh container

If it ain't broke
don't fix it

## Sharing the same base image in the enterprise

Google uses the same boot image for the server and its containers

- Easier maintenance and ensuring that everyone is using a secure image
- Everyone is using the same version of libraries. This saves memory and prevents version and API drift
- When you upgrade the server image, all containers automatically pick up the new image. This can be a huge plus for maintenance efforts.

## Which approach is best?

- Neither has to be the only correct solution
- Docker's default of image separation works for some and is desirable in some cases like needing to run an old application in a different version of the OS
- My approach of server image re-use has advantages we just listed
- Docker lets you mix and match, so feel free to use the best one for each application

## Running applications half in and half out of a container

- This is a great plus you get from my approach
- You install your application as usual on the host system
- Part of that application accesses raw devices and internal networks you don't want to expose to the outside
- So you run the backend on the server as usual
- And run the untrusted php frontend inside a container
- All based on the same package and binaries, no need to install and configure the package twice and keep both sides in sync.

## Docker install: before you start

Docker uses copy on write to allow the container to write in the image without modifying it.

You'll need one of:

- The old AUFS union filesystem was never merged due to code quality
- Device mapper or Device mapper loopback
- OverlayFS in recent kernels (3.18+)
- Btrfs (not very stable in older kernels)

More details on this RH blog: http://goo.gl/kUxTs5

# Install and setup time (on debian)

```
apt-get install -t testing docker.io
The following NEW packages will be installed:
  aufs-tools cgroupfs-mount docker.io mountall plymouth
The following packages will be upgraded:
  dmsetup libdevmapper1.02.1

=> bug in the package, you can remove mountall and plymouth

/usr/share/docker.io/contrib/check-config.sh
info: reading kernel config from /proc/config.gz ...

Generally Necessary:
- cgroup hierarchy: properly mounted [/sys/fs/cgroup]
- CONFIG_NAMESPACES: enabled
- CONFIG_NET_NS: enabled
- CONFIG_PID_NS: enabled
- CONFIG_IPC_NS: enabled
- CONFIG_UTS_NS: enabled
- CONFIG_DEVPTS_MULTIPLE_INSTANCES: enabled
- CONFIG_CGROUPS: enabled
- CONFIG_CGROUP_CPUACCT: enabled
- CONFIG_CGROUP_DEVICE: enabled
- CONFIG_CGROUP_FREEZER: enabled
- CONFIG_CGROUP_SCHED: enabled
(...)
```

# Install and setup time (2)

Make sure you do not have cgroup mounted as a single directory:

```
Filesystem      Size  Used Avail Use% Mounted on
cgroup             0     0     0    - /sys/fs/cgroup
```

This will cause docker to fail:

```
FATA[0000] Error response from daemon: Cannot start container
37b628f60d8d88aa47cb88199372d40f845e8acb73ed56343028fd9cf5bce238: [8]
System error: write
/sys/fs/cgroup/docker/37b628f60d8d88aa47cb88199372d40f845e8acb73ed563430
28fd9cf5bce238/cgroup.procs: no space left on device
```

Remove the /sys/fs/cgroup mount from /etc/fstab, and reboot (due to a kernel bug, unmount is not enough). Then make sure you have this from either the docker initscript or cgroupfs-mount:

```
mount |grep cgroup
cgroup on /sys/fs/cgroup type tmpfs (rw,relatime,size=12k)
cgroup on /sys/fs/cgroup/cpuset type cgroup
cgroup on /sys/fs/cgroup/cpu type cgroup
cgroup on /sys/fs/cgroup/cpuacct type cgroup
cgroup on /sys/fs/cgroup/memory type cgroup
cgroup on /sys/fs/cgroup/devices type cgroup
cgroup on /sys/fs/cgroup/freezer type cgroup
cgroup on /sys/fs/cgroup/net_cls type cgroup
cgroup on /sys/fs/cgroup/blkio type cgroup
cgroup on /sys/fs/cgroup/perf_event type cgroup
cgroup on /sys/fs/cgroup/net_prio type cgroup
```

# Install and setup time (3)

```
legolas:~# docker run -t --entrypoint bash debian -c "ls -al /; cat
/etc/debian_version"
Unable to find image 'debian:latest' locally
latest: Pulling from debian
3cb35ae859e7: Pull complete
41b730702607: Already exists
debian:latest: The image you are pulling has been verified. Important:
image verification is a tech preview feature and should not be relied on
to provide security.
Digest:
sha256:5c8ab02dd3a1faa611adb36005087deb52452b1a17582cc63f7b57f9e91f5e12
Status: Downloaded newer image for debian:latest
total 0
drwxr-xr-x   1 root root  174 May 12 22:51 .
drwxr-xr-x   1 root root  174 May 12 22:51 ..
-rwxr-xr-x   1 root root    0 May 12 22:51 .dockerenv
-rwxr-xr-x   1 root root    0 May 12 22:51 .dockerinit
(...)
drwxr-xr-x   1 root root    0 Apr 28 23:52 srv
dr-xr-xr-x  13 root root    0 May 12 22:51 sys
drwxrwxrwt   1 root root    0 Apr 28 23:57 tmp
drwxr-xr-x   1 root root   70 Apr 28 23:52 usr
drwxr-xr-x   1 root root   90 Apr 28 23:52 var
8.0
```

## Making your own super small base image

- Docker is not meant to be used to run a container entirely from your host filesystem
- But it can be made to do so by giving it any image, and mounting your host filesystem as directories on top of that image
- So let's start with the smallest docker image that doesn't depend on outside code
- We'll use a single binary: sash (single user shell), but we could have used busybox-static instead

# Making my own base image

```
gargamel:~/docker_base/_build# find
./bin
./bin/sash
./Dockerfile
gargamel:~/docker_base/_build# cat Dockerfile
FROM scratch
COPY /bin/sash /bin/sash
#COPY /bin/busybox /bin/busybox
RUN ["/bin/sash", "-c", "-mkdir /root"]
RUN ["/bin/sash", "-c", "-mkdir /run"]
RUN ["/bin/sash", "-c", "-mkdir /run/lock"]
EXPOSE 80
CMD ["/bin/sash"]
gargamel:~/docker_base/_build# docker build -t empty .
Sending build context to Docker daemon 455.7 kB
Sending build context to Docker daemon
Step 0 : FROM scratch
 ---> 511136ea3c5a
Step 1 : COPY /bin/sash /bin/sash
 ---> d32464b3bc02
Removing intermediate container acb530e0c375
Step 2 : CMD /bin/sash
 ---> Running in 41dff2fb9acb
 ---> 3ee781c323da
Removing intermediate container 41dff2fb9acb
Successfully built 3ee781c323da
```

# Looking at an empty image

```
gargamel:~# docker run -t -i empty
Stand-alone shell (version 3.8)
> -ls -l /
drwxr-xr-x  1 0        0               88  May 12 23:47 .
drwxr-xr-x  1 0        0               88  May 12 23:47 ..
-rwxr-xr-x  1 0        0                0  May 12 23:47 .dockerenv
-rwxr-xr-x  1 0        0                0  May 12 23:47 .dockerinit
drwxr-xr-x  1 0        0                8  May 10 17:16 bin
drwxr-xr-x  5 0        0              380  May 12 23:47 dev
drwxr-xr-x  1 0        0               56  May 12 23:47 etc
dr-xr-xr-x583 0        0                0  May 12 23:47 proc
drwxr-xr-x  1 0        0                0  May 10 17:16 root
drwxr-xr-x  1 0        0                8  May 10 21:59 run
dr-xr-xr-x 13 0        0                0  May 10 17:33 sys
> -ls -l /etc
drwxr-xr-x  1 0        0               56  May 12 23:47 .
drwxr-xr-x  1 0        0               88  May 12 23:47 ..
-rw-r--r--  1 0        0               13  May 12 23:47 hostname
-rw-r--r--  1 0        0              175  May 12 23:47 hosts
lrwxrwxrwx  1 0        0               12  May 12 23:47 mtab
-rw-r--r--  1 0        0               87  May 12 23:47 resolv.conf
> -ls -l /run
drwxr-xr-x  1 0        0                8  May 10 21:59 .
drwxr-xr-x  1 0        0               88  May 12 23:47 ..
drwxr-xr-x  1 0        0                0  May 10 21:59 lock
```

# Looking at an empty image (2)

```
> -more /proc/mounts
/dev/mapper/cryptroot / btrfs rw,noatime,compress=lzo 0 0
proc /proc proc rw,nosuid,nodev,noexec,relatime 0 0
tmpfs /dev tmpfs rw,nosuid,mode=755 0 0
devpts /dev/pts devpts rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=666
shm /dev/shm tmpfs rw,nosuid,nodev,noexec,relatime,size=65536k 0 0
mqueue /dev/mqueue mqueue rw,nosuid,nodev,noexec,relatime 0 0
sysfs /sys sysfs ro,nosuid,nodev,noexec,relatime 0 0
/dev/mapper/cryptroot /etc/resolv.conf btrfs rw,noatime,compress=lzo 0 0
/dev/mapper/cryptroot /etc/hostname btrfs rw,noatime,compress=lzo 0 0
/dev/mapper/cryptroot /etc/hosts btrfs rw,noatime,compress=lzo 0 0
devpts /dev/console devpts rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000
proc /proc/asound proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/bus proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/fs proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/irq proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/sys proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/sysrq-trigger proc ro,nosuid,nodev,noexec,relatime 0 0
tmpfs /proc/kcore tmpfs rw,nosuid,mode=755 0 0
tmpfs /proc/latency_stats tmpfs rw,nosuid,mode=755 0 0
tmpfs /proc/timer_stats tmpfs rw,nosuid,mode=755 0 0
```

# Files automatically created by docker

Docker uses file bind mounts to add autogenerated networking files in /etc

```
 > -more /etc/resolv.conf
search svh.merlins.org merlins.org
nameserver 192.168.205.3
nameserver 192.168.205.254
> -more /etc/hostname
2f1cfdcbc70c
> -more /etc/hosts
172.17.0.48     2f1cfdcbc70c
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

## Mounting your base filesystem in docker

- Docker lets you mount your host filesystem inside of your container.
- You can't mount /, but you can mount subdirectories
- Default mount is read-write, therefore unsafe
- I can now run bash (and my entire system) using binaries from the host
- This works but only protects my local network, process list, and host disk not mounted here

```
gargamel:~/docker_base# docker run  -v /:/ -i -t empty /bin/bash
docker: Invalid bind mount: destination can't be '/'. See 'docker run
--help'.
gargamel:~/docker_base# docker run  -v /tmp:/tmp -v /lib:/lib -v
/lib64:/lib64 -v /usr:/usr -v /var:/var -v /etc:/etc  -v /bin:/bin -i -t
empty /bin/bash
dee1143f0c1d:/#
```

## Mounting portions of your filesystem, read only

- The goal is to mount enough of your system for your your programs to work in the containers
- As much should be mounted read only as possible

```
docker run  -v /tmp:/tmp -v /lib:/lib:ro -v /lib64:/lib64:ro -v
/usr:/usr:ro -v /var:/var -v /bin:/bin:ro -v /etc:/etc:ro -i -t empty
/bin/bash
358076e71312:/usr# touch file
touch: cannot touch 'file': Read-only file system
358076e71312:/usr# mount
(...)
/dev/mapper/cryptroot on /etc/resolv.conf type btrfs (rw)
/dev/mapper/cryptroot on /etc/hostname type btrfs (rw)
/dev/mapper/cryptroot on /etc/hosts type btrfs (rw)
/dev/mapper/cryptroot on /bin type btrfs (ro)
/dev/mapper/cryptroot on /etc type btrfs (ro)
/dev/mapper/cryptroot on /lib type btrfs (ro)
/dev/mapper/cryptroot on /lib64 type btrfs (ro)
/dev/mapper/cryptroot on /tmp type btrfs (rw)
/dev/mapper/cryptroot on /usr type btrfs (ro)
/dev/mapper/cryptroot on /var type btrfs (rw)
```

# Docker can't create mountpoints on read only mounts

Be careful when you make a custom read only mountpoint with other mounts on top, to have the mountpoints already created, since docker will not be able to create them on a read only filesystem

```
D=/root/docker_base; docker run  -v $D/tmp:/tmp -v $D/lib:/lib:ro -v
$D/lib64:/lib64:ro -v $D/usr:/usr:ro -v $D/var:/var -v $D/bin:/bin:ro -v
$D/etc:/etc:ro -v /etc/apache2:/etc/apache2:ro -i -t debian /bin/bash

setup mount namespace creating new bind mount target mkdir
/var/lib/docker/devicemapper/mnt/e18389e125a58c4ee9004aa173116a4a44789b5
81f8d8a154ab3065427e38a5a/rootfs/etc/apache2: read-only file system
2015/04/26 18:53:43 Error response from daemon: Cannot start container
e18389e125a58c4ee9004aa173116a4a44789b581f8d8a154ab3065427e38a5a: setup
mount namespace creating new bind mount target mkdir
/var/lib/docker/devicemapper/mnt/e18389e125a58c4ee9004aa173116a4a44789b5
81f8d8a154ab3065427e38a5a/rootfs/etc/apache2: read-only file system
```

## Not showing sub mounts

- If you mount /usr in docker, it will also make /usr/local (a separate filesystem) visible
- This is done by creating a separate FS tree with bind mounts
- Bind mounts cannot be read-only if their source is read-write, so we use -v /usr:/usr:ro in docker instead

```
legolas:~/docker_base# for i in lib lib64 usr var bin; do
mkdir -p $i; mount -o bind /$i $i; done

D=/root/docker_base; docker run -v $D/tmp:/tmp -v $D/lib:/lib:ro -v
$D/lib64:/lib64:ro -v $D/usr:/usr:ro -v $D/var:/var -v $D/bin:/bin:ro -v
$D/etc:/etc:ro -i -t empty /bin/bash
```

## Not showing all subdirectories

- If you mount /var, you may want to shadow/hide /var/log on the same filesystem
- This is done on the host, not inside the container with lots of -v /tmp:/var/log -v /tmp:/var/lib/dpkg since it would make the docker command line very long
- With D=/root/docker_base
- Instead of running docker with “-v $D/var:/var -v /tmp:/var/log”
- we run mount -o bind /tmp $D/var/log
- And run docker with just “-v $D/var:/var”
- Docker sees that tmp is mounted in $D/var/log and automatically mounts it in your image
- Unfortunately we cannot use mount -o bind,ro /tmp $D/var/log

## Custom /etc

- /etc is full of files you don't want to show, like private keys, password shadow file, and more
- You want to share a portion of /etc/ as read only files
- We create $D/etc as an empty directory
- We then hardlink all the files we want in /etc
- Bind mount /etc/vim /etc/profile.d etc... in $D/etc/
- mkdir $D/etc/apache2
- Mount container specific directories with docker -v $D/etc/apache2:/etc/apache2:ro
- This avoids leaking your apache config in a mysql container.

# Custom /etc creation

- We make a skeleton $D/etc directory
- Hardlink the files we want to expose from the base system
- Because of docker -v $D/etc:/etc:ro, those files will not be modifiable in the image.

```
cd /root/docker_base
mkdir -p etc/init.d
for f in etc/
{bash.bashrc,bash_completion,bashrc,environment,hosts,inputrc,ld.so.cache
,ld.so.conf,localtime,magic,magic.mime,mailcap,mailcap.order,mime.types,n
sswitch.conf,profile,protocols,resolv.conf,rpc,shells,timezone,init.d/apa
che2}
do
    ln -f /$f $f
Done
# Add files local to just this container:
cp -a etc_local/* etc/
```

# Putting it all together: before

```
/dev/mapper/cryptroot / btrfs rw
proc /proc proc rw
tmpfs /dev tmpfs rw
devpts /dev/pts devpts rw
shm /dev/shm tmpfs rw
mqueue /dev/mqueue mqueue rw
sysfs /sys sysfs ro
/dev/mapper/cryptroot /bin btrfs ro
/dev/mapper/cryptroot /etc btrfs ro
/dev/mapper/cryptroot /etc/vim btrfs rw
/dev/mapper/cryptroot /lib btrfs ro
/dev/mapper/cryptroot /lib64 btrfs ro
/dev/mapper/pool1 /tmp btrfs rw
/dev/mapper/cryptroot /usr btrfs ro
/dev/mapper/cryptroot /var btrfs ro
/dev/mapper/pool1 /var/change btrfs rw
/dev/mapper/pool1 /var/local/nobck btrfs rw << bad, info leak
/dev/mapper/cryptroot /var/local/nobckd2 btrfs rw
/dev/mapper/pool2 /var/local/space btrfs rw
/dev/mapper/cryptroot /var/lib/docker/btrfs btrfs rw
^^^ very bad, read-write access to all docker instances
/dev/mapper/cryptroot /var/lib/docker/btrfs/subvolumes/XXX btrfs rw
proc /var/lib/docker/btrfs/subvolumes/XXX/proc proc rw
tmpfs /var/lib/docker/btrfs/subvolumes/XXX/dev tmpfs rw
devpts /var/lib/docker/btrfs/subvolumes/XXX/dev/pts devpts rw
shm /var/lib/docker/btrfs/subvolumes/XXX/dev/shm tmpfs rw
```

# Putting it all together: before (2)

```
(...)
mqueue /var/lib/docker/btrfs/subvolumes/XXX/dev/mqueue mqueue rw
sysfs /var/lib/docker/btrfs/subvolumes/XXX/sys sysfs ro
/dev/mapper/cryptroot /var/lib/docker/btrfs/subvolumes/XXX/bin btrfs ro
/dev/mapper/cryptroot /var/lib/docker/btrfs/subvolumes/XXX/etc btrfs ro
/dev/mapper/cryptroot /var/lib/docker/btrfs/subvolumes/XXX/etc/vim btrfs rw
/dev/mapper/cryptroot /var/lib/docker/btrfs/subvolumes/XXX/lib btrfs ro
/dev/mapper/cryptroot /var/lib/docker/btrfs/subvolumes/XXX/lib64 btrfs ro
/dev/mapper/pool1 /var/lib/docker/btrfs/subvolumes/XXX/tmp btrfs rw
/dev/mapper/cryptroot /var/lib/docker/btrfs/subvolumes/XXX/usr btrfs ro
/dev/mapper/cryptroot /etc/resolv.conf btrfs rw
/dev/mapper/cryptroot /etc/hostname btrfs rw
/dev/mapper/cryptroot /etc/hosts btrfs rw
devpts /dev/console devpts rw
proc /proc/asound proc ro
proc /proc/bus proc ro
proc /proc/fs proc ro
proc /proc/irq proc ro
proc /proc/sys proc ro
proc /proc/sysrq-trigger proc ro
tmpfs /proc/kcore tmpfs rw
tmpfs /proc/latency_stats tmpfs rw
tmpfs /proc/timer_stats tmpfs rw
```

# Putting it all together: after

```
/dev/mapper/cryptroot / btrfs rw
proc /proc proc rw
tmpfs /dev tmpfs rw
devpts /dev/pts devpts rw
shm /dev/shm tmpfs rw
mqueue /dev/mqueue mqueue rw
sysfs /sys sysfs ro
/dev/mapper/cryptroot /bin btrfs ro
/dev/mapper/cryptroot /etc btrfs ro
/dev/mapper/cryptroot /etc/vim btrfs rw << docker mounted with wrong perms (rw)
/dev/mapper/cryptroot /lib btrfs ro
/dev/mapper/cryptroot /lib64 btrfs ro
/dev/mapper/cryptroot /tmp btrfs rw
/dev/mapper/cryptroot /usr btrfs ro
/dev/mapper/cryptroot /var btrfs ro
/dev/mapper/cryptroot /etc/resolv.conf btrfs rw
/dev/mapper/cryptroot /etc/hostname btrfs rw
/dev/mapper/cryptroot /etc/hosts btrfs rw
devpts /dev/console devpts rw
proc /proc/asound proc ro
proc /proc/bus proc ro
proc /proc/fs proc ro
proc /proc/irq proc ro
proc /proc/sys proc ro
proc /proc/sysrq-trigger proc ro
tmpfs /proc/kcore tmpfs rw
tmpfs /proc/latency_stats tmpfs rw
tmpfs /proc/timer_stats tmpfs rw
```

## Fixing rw bind mounts automatically mounted in image

- It's great to be able to have docker automatically mount subdirectories of mount points
- But you cannot control read-write bind mounts also being mounted read-write in your image
- Solution #1: mount a read only snapshot of those directories in your $D tree (if you have read only snapshots)
- Solution #2: specify each mount on the docker command line with -v $D/etc/vim:/etc/vim:ro
- Solution #3: don't bind mount /etc/vim $D/etc/vim and symlink hardlink the files instead

## Every host mount mounted ro, including /var

- Make every host mount read only (including /var)
- The host's /var/log should be shadowed by mounting another directory in that location
- You can hide other dirs like /var/backups /var/account, /var/spool
- /var/run and /var/lock should be symlinked to /run if they aren't already
- /var/tmp is likely not actually written to. You might not even want a writeable /tmp in your container (it's used by some rootkits).

## Putting it all together: getting php5 of an installed app working in your container, along with apache2

- I had to create /root which was missing from my empty image and was causing errors about writing to ~/.bash_history
- I was missing /run and /run/lock that /var/run and /var/locks are linked to on my host system

```
b1c1d226cdda:~# /etc/init.d/apache2 start
[FAIL] Starting web server: apache2 failed!
[warn] The apache2 configtest failed. ... (warning).
Output of config test was:
mkdir: cannot create directory '/var/run': File exists
mktemp: failed to create directory via template
'/var/lock/apache2.XXXXXXXXXX': No such file or directory
chmod: missing operand after '755'
```

## Moving other session and locks to /run

- /run is part of your COW filesystem in the image, so it gets automatically reset for you every time you restart the image
- My php5 needed sessions moved to /run

```
gargamel:/var/lib/php5# ls -ld sessions
drwx-wx-wt 1 root root 0 May 10 15:39 sessions/
gargamel:/var/lib/php5# mv sessions /run; ln -s /run/sessions .
```

## Now apache works, but not connections to mysql

- Thankfully no need to open ports and play routing tricks
- Mysql can be accessed through a socket that can be mounted in the container.

```
FATAL: Cannot connect to MySQL server on 'localhost'. Please make sure you
have specified a valid MySQL database name in 'include/config.php'
Could not connect to database: Can't connect to local MySQL server through
socket '/var/run/mysqld/mysqld.sock' (2)

gargamel:~# l /run/mysqld/
total 4
drwxr-xr-x  2 mysql root    80 May  5 08:15 ./
drwxr-xr-x 34 root  root  1580 May 24 19:24 ../
-rw-rw----  1 mysql mysql    6 May  5 08:15 mysqld.pid
srwxrwxrwx  1 mysql mysql    0 May  5 08:15 mysqld.sock=

Simply run docker with -v /run/mysqld:/run/mysqld
```

**Success!**

- Apache, mysql, and php webapps are installed and working on the host system
- Apache can be started in the container, using the same files and same config
- The backend of the php apps can be run on the host system and access data not visible in the container
- The unsafe php5 frontends are now run inside the container and have no access or no write access to the host system's resources
- They access the data they serve through mysql or a shared filesystem read-only to them.

# Docker instance start script

```
grep $(pwd) /proc/mounts | awk '{ print $2 }' |xargs --no-run-if-empty umount

# FIXME: unsafe, docker makes the etc/xxx directories writeable in the host.
for d in lib lib64 usr var bin sbin etc/
{alternatives,bash_completion.d,mc,profile.d,vim}
do
    mkdir -p $d
    #mount -o bind,ro /$d $d will not work if the original dirs were rw
    # So instead we rely on the docker mount to be read only
    grep $(pwd)/$d /proc/mounts || mount -o bind /$d $d
done

for f in etc/
{bash.bashrc,bash_completion,bashrc,environment,inputrc,ld.so.cache,ld.so.conf,localtime
,magic,magic.mime,mailcap,mailcap.order,mime.types,nsswitch.conf,profile,protocols,resol
v.conf,rpc,shells,timezone,init.d/apache2}
do
    rm "$f"
    ln /"$f" "$f"
done

# Extra directories that aren't mounted here, but mapped by docker if needed by that
instance
# They need to exist because etc is mounted ro and docker cannot create mount point
directories
for mp in etc/{ssl,apache2,php5,zm,cacti,phpmyadmin}
do
    mkdir -p "$mp"
done
```

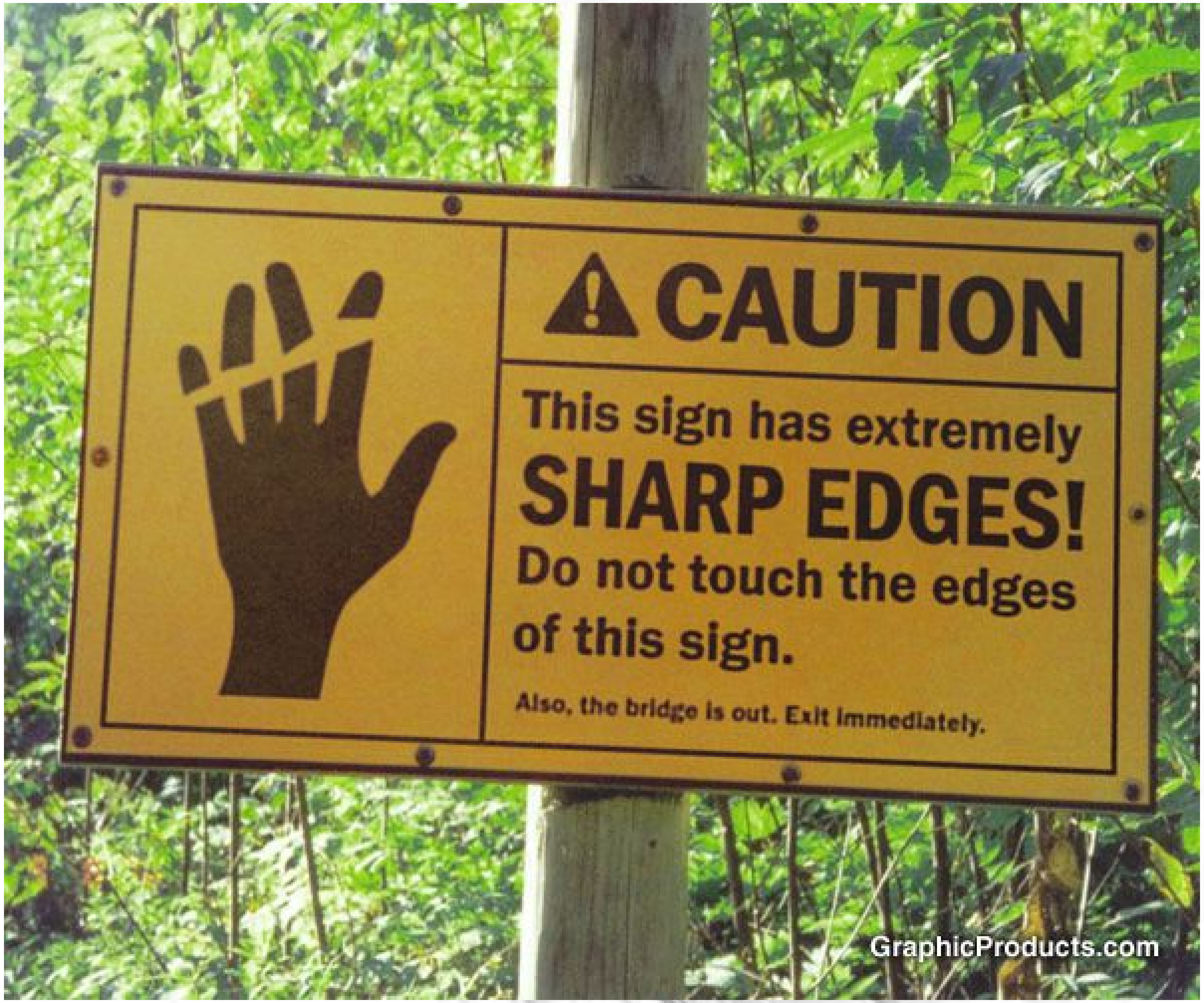
# Docker instance start script (cont)

```
# Directories to hide by mounting tmp on top of them
for hidedir in var/src var/backups var/account var/spool usr/local
do
    grep -q $(pwd)/$hidedir /proc/mounts || mount -o bind tmp $hidedir
done

# Local files separate from the host system's copies
# Suggested list: fstab, group, hostname, hosts, passwd, shadow
cp -a etc_local/* etc/
# Local directory which will get mounted under /var/log
mkdir -p local/log

D=/root/docker_base; docker run  -v $D/lib:/lib:ro -v $D/lib64:/lib64:ro -v
$D/usr:/usr:ro -v $D/var:/var:ro -v $D/bin:/bin:ro -v $D/sbin:/sbin:ro -v $D/etc:/etc:ro
-v $D/local/log:/var/log  -v /etc/apache2:/etc/apache2:ro -v /etc/php5:/etc/php5:ro
-v /etc/phpmyadmin:/etc/phpmyadmin:ro -v /etc/ssl:/etc/ssl:ro -v
/etc/cacti:/etc/cacti:ro -v /var/lib/cacti:/var/lib/cacti:ro  -v /etc/zm:/etc/zm:ro
-v /run/mysqld:/run/mysqld -p 80:80 -i -t --entrypoint /bin/bash empty -c
"/etc/init.d/apache2 start; /bin/bash"
```

# Warnings


⚠ CAUTION
This sign has extremely
SHARP EDGES!
Do not touch the edges
of this sign.
Also, the bridge is out. Exit immediately.
GraphicProducts.com

## Warnings

- Direct mounting is very powerful
- But also dangerous
- Make sure you only expose what you need
- And make everything read only outside of specific directories like a custom /var/log and /tmp
- You are losing the copy on write feature of docker for every directory mounted from the host.
- Anything written on a host mounted directory is persistent, affects others containers, and the host.

## Warnings about docker, btrfs, and backups

- If you use COW snapshots, they likely won't get properly backed up or restored.
- With btrfs, snapshots used by docker are difficult to backup, btrfs doesn't allow backing up a filesystem and all its subvolumes.
- But if you have an empty image you can recreate from scratch, it's quicker than doing backups/restores of your containers.
- Nothing gets stored in your container since you're mounting the local disk, so it gets backed up along with your host system.

## Beware...

## Summary: Doing it the docker way

- Not using the native docker way of doing things because it's hard to reproduce your app in a container may mean that your configuration management is lacking
- Doing it the docker way lets you know the minimal bits required for your app and control exactly what dependencies are available to your app
- The docker way lets your app be run on any host that can run docker by only moving your docker file

## Summary: Doing it the shared system way

- If your system is hard to reproduce/dockerize, it's still better to dockerize potentially unsafe front ends with the shared disk way
- It's great to have the same app hard to fully run in container run inside and outside of the container without running 2 separate copies maybe not in sync
- Some systems can benefit from sharing the same system image between the host and container(s) for disk and memory use

## Thanks to Docker Maintainers and Contributors

- Docker is probably the best documented open source project I've seen so far. It has nice tutorials and information cut in pieces as you need it.
- While I'm not using Docker the way it was meant to be used, it's flexible enough to be used in ways it wasn't meant for
- And more importantly, it makes LXC more accessible to all by doing the work to get containers up and configured properly.

# 質問がありますか？
# Questions?
# Want a job at Google?

**Talk slides for download:**
**http://marc.merlins.org/linux/talks/DockerLocalDisk-LC2015-JP/**
**http://goo.gl/PWjume**

Marc MERLIN
marc_soft@merlins.org