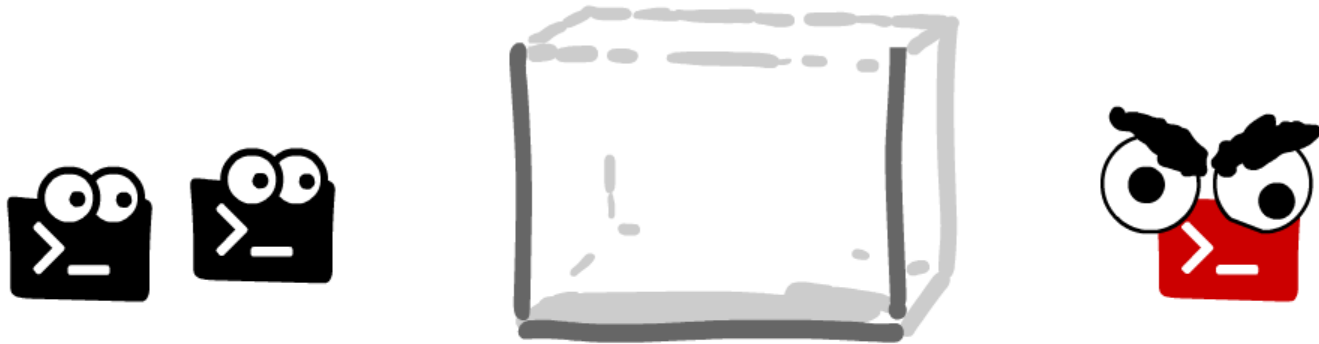




Let's Build And Break A Container By Hand



Reinhard Kugler, SBA Research



Kubernetes Blog

2020

A Custom Kubernetes
Scheduler to Orchestrate
Highly Available
Applications

Kubernetes 1.20: Pod
Impersonation and
Short-lived Volumes in
CSI Drivers

Third Party Device
Metrics Reaches GA

Kubernetes 1.20:
Granular Control of
Volume Permission

SBA Research, 2020

Don't Panic: Kubernetes and Docker

Wednesday, December 02, 2020

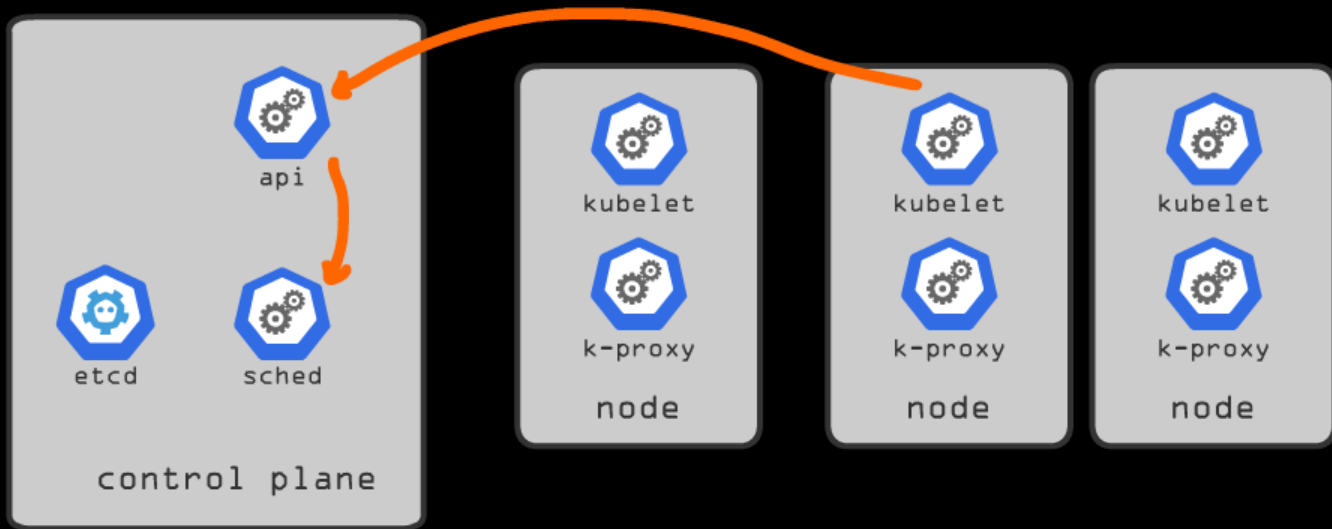
Authors: Jorge Castro, Duffie Cooley, Kat Cosgrove, Justin Garrison, Noah Kantrowitz, Bob Killen, Rey Lejano, Dan "POP" Papandrea, Jeffrey Sica, Davanum "Dims" Srinivas

Kubernetes is [deprecating Docker](#) as a container runtime after v1.20.

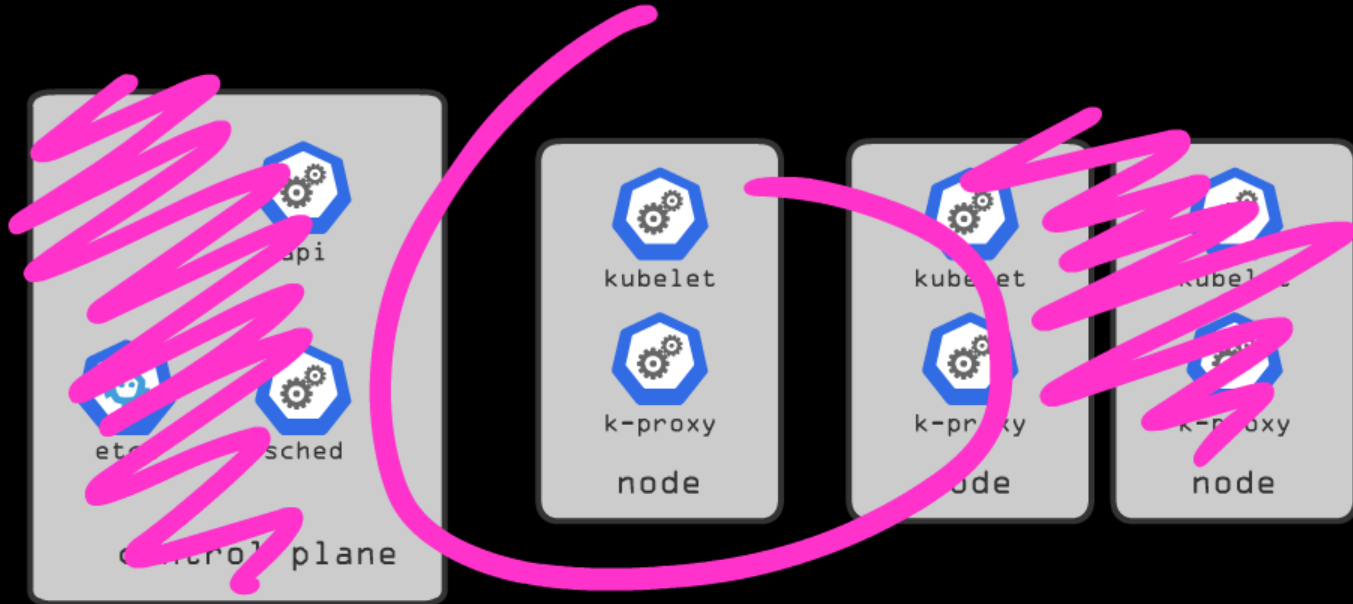
You do not need to panic. It's not as dramatic as it sounds.

TL;DR Docker as an underlying runtime is being deprecated in favor of runtimes that use the [Container Runtime Interface \(CRI\)](#) created for Kubernetes. Docker-

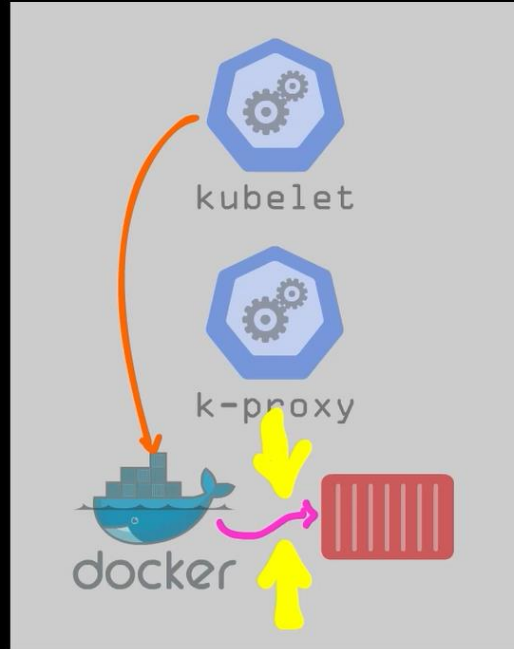
Kubernetes is a complex system. It consists of many components.



This talk focuses on the container creation, which takes place on the node.

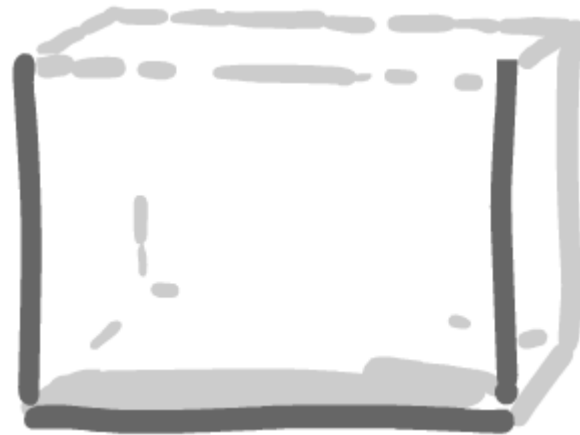


Kubernetes created Containers by talking to Docker. They dropped Docker – so this changes. We want to replace that ...

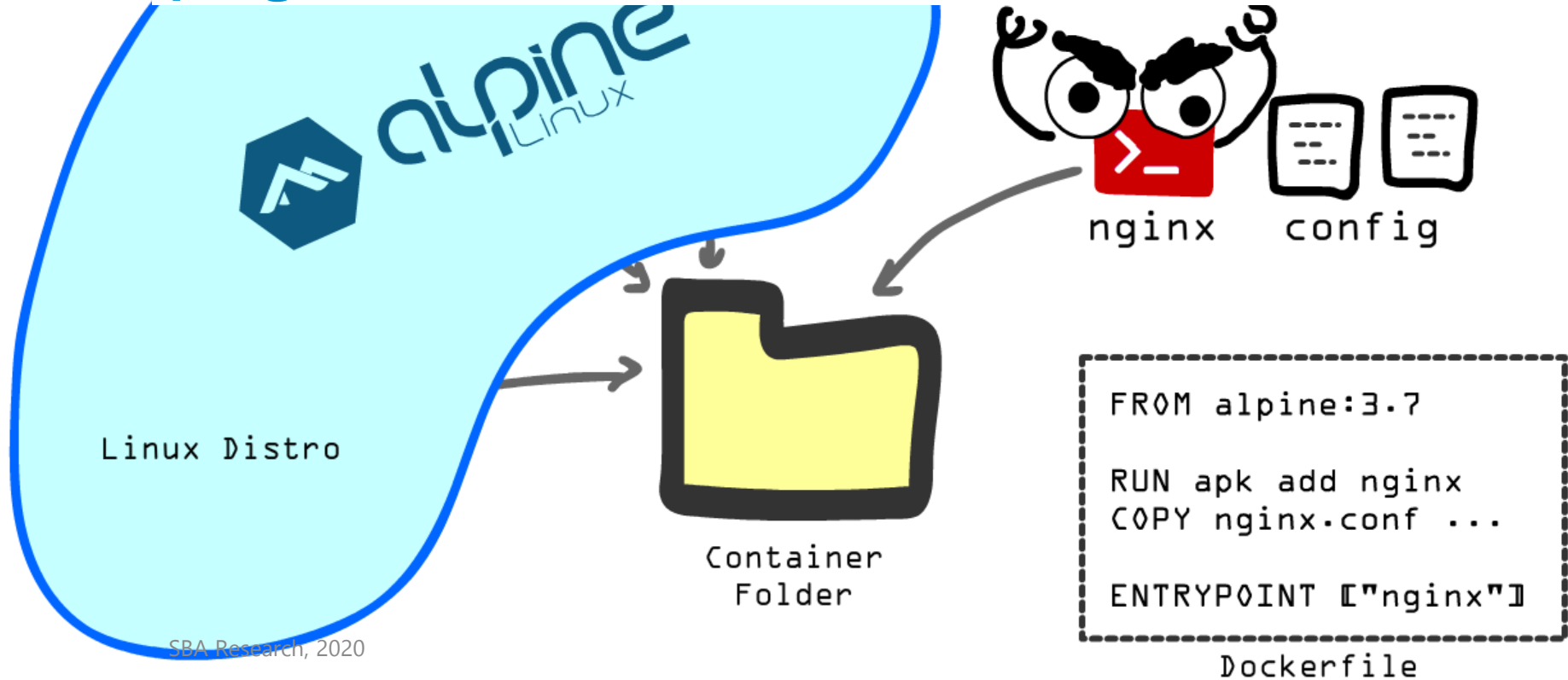


We want to build a container by hand. The checklist for creating a Container is:

- 0) Prepare an Image
- 1) Isolate the filesystem
- 2) Restrict access to devices
- 3) Drop user capabilities
- 4) Isolate network
- 5) Fix breakouts



The basis of a container is a separate image. This contains all the libraries, config files and programs.



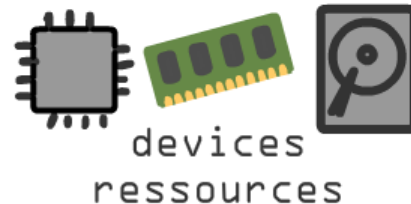
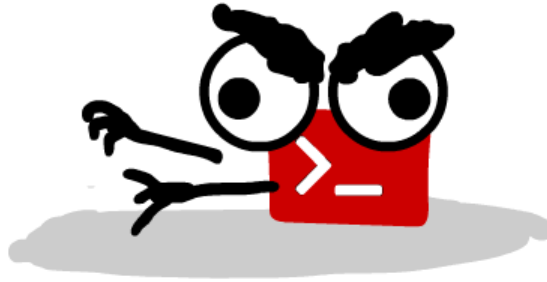
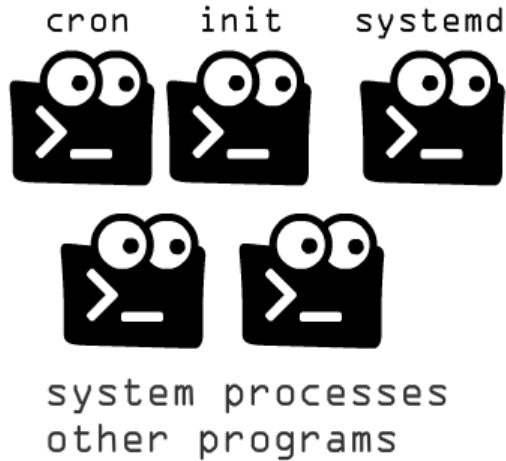
We create an image by using a alpine base

```
# curl -o alpine-minirootfs-3.9.0-x86_64.tar.gz http://dl-  
cdn.alpinelinux.org/alpine/v3.9/releases/x86_64/alpine-  
minirootfs-3.9.0-x86_64.tar.gz  
# mkdir container  
# tar -xzipf alpine-minirootfs-3.9.0-x86_64.tar.gz -C container
```

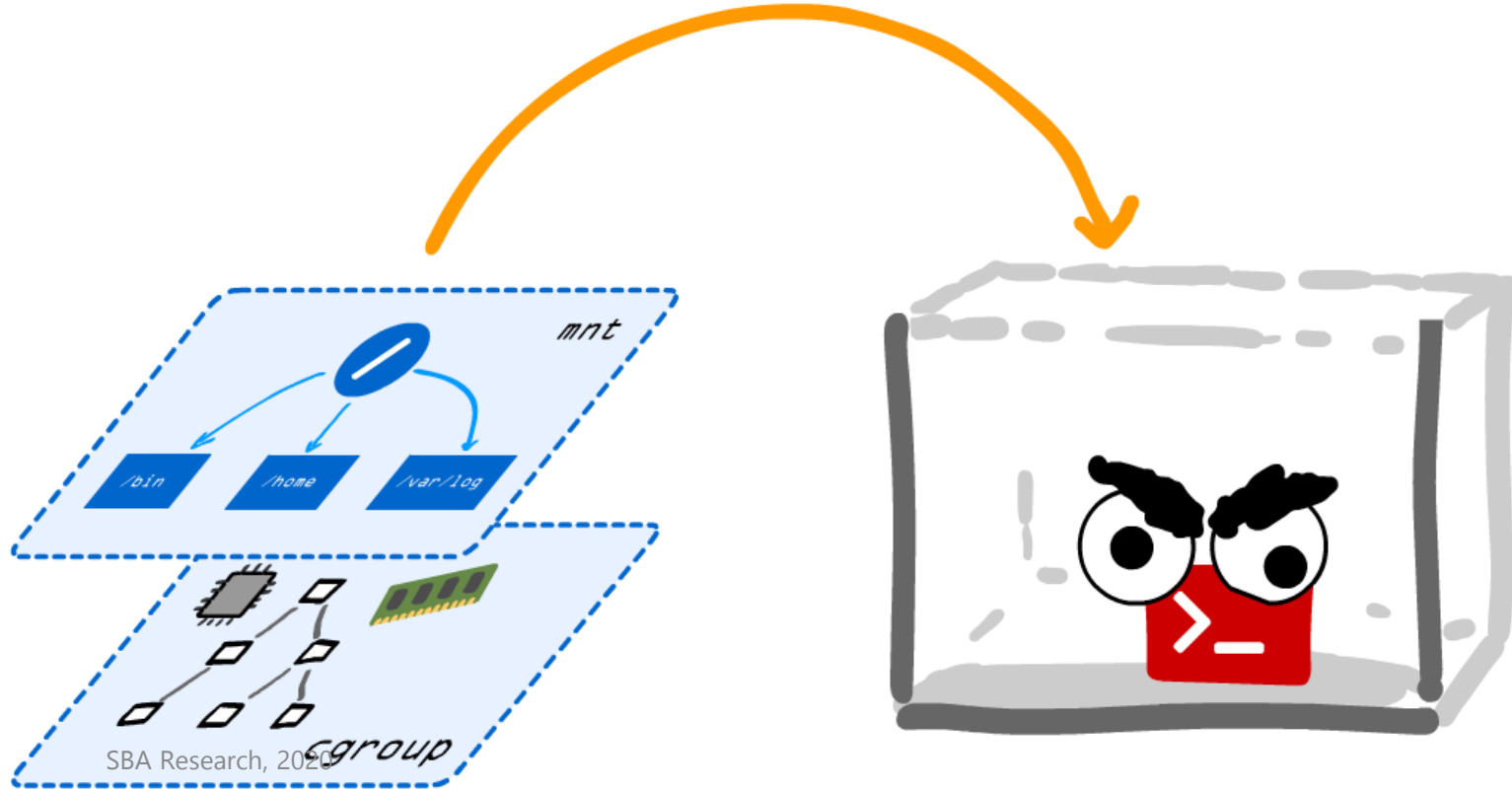
...and install nginx inside the Container

```
# echo "nameserver 8.8.8.8,, > container/etc/resolv.conf  
# sudo chroot container/ apk add nginx
```

The process (e.g. nginx) has to be treated as evil. We have to lock it down, otherwise an attacker could steal data and corrupts other processes



Namespaces and cgroups are the walls of containers around our image



We can examine the used namespaces in /proc – we see that many processes use the same mount namespace

```
sba@linux:~/sec4dev$ ls -la /proc/self/ns/
total 0
dr-x--x--x 2 sba sba 0 Jan 31 13:57 .
dr-xr-xr-x 9 sba sba 0 Jan 31 13:57 ..
lrwxrwxrwx 1 sba sba 0 Jan 31 13:57 cgroup → 'cgroup:[4026531835]'
lrwxrwxrwx 1 sba sba 0 Jan 31 13:57 ipc → 'ipc:[4026531839]'
lrwxrwxrwx 1 sba sba 0 Jan 31 13:57 mnt → 'mnt:[4026531840]'
lrwxrwxrwx 1 sba sba 0 Jan 31 13:57 net → 'net:[4026531992]'
lrwxrwxrwx 1 sba sba 0 Jan 31 13:57 pid → 'pid:[4026531836]'
lrwxrwxrwx 1 sba sba 0 Jan 31 13:57 pid_for_children → 'pid:[4026531836]'
lrwxrwxrwx 1 sba sba 0 Jan 31 13:57 time → 'time:[4026531834]'
lrwxrwxrwx 1 sba sba 0 Jan 31 13:57 time_for_children → 'time:[4026531834]'
lrwxrwxrwx 1 sba sba 0 Jan 31 13:57 user → 'user:[4026531837]'
lrwxrwxrwx 1 sba sba 0 Jan 31 13:57 uts → 'uts:[4026531838]'
sba@linux:~/sec4dev$ sudo ls -la /proc/1/ns/mnt
lrwxrwxrwx 1 root root 0 Jan 31 13:57 /proc/1/ns/mnt → 'mnt:[4026531840]'
```

Create a new mnt namespace! It's a copy of the mount list. Then umount old mounts.



```
>_
```

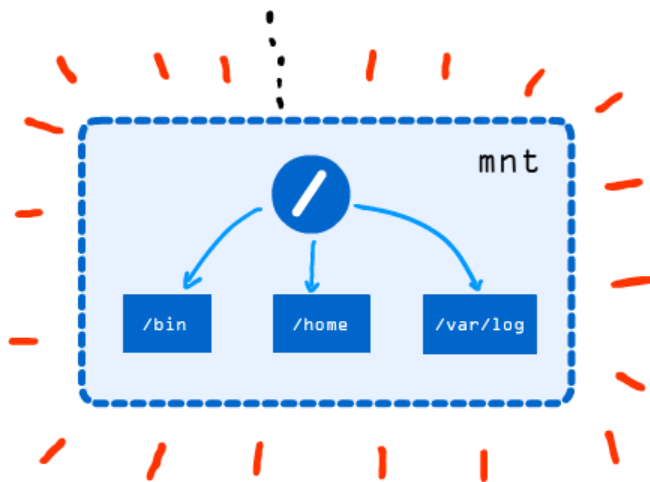
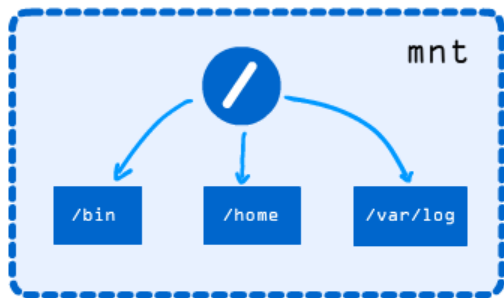
```
/sbin/init
```

```
>_
```

```
/bin/bash
```



```
container
```



We prepare the pivot (root) into the Container and exec the shell (or nginx)

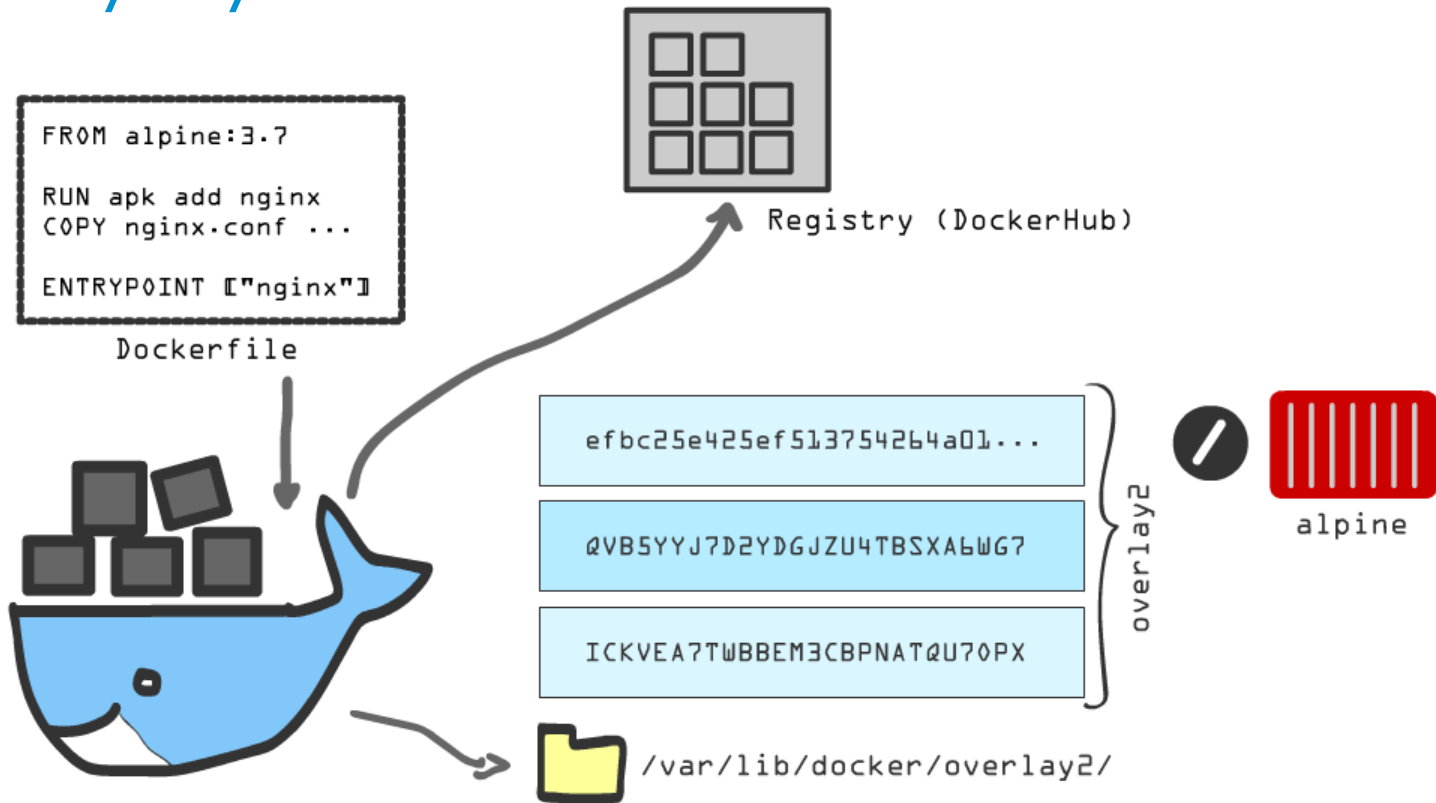
```
# mount --bind $PWD/container $PWD/container/  
# chmod 755 container  
# cd container  
# mkdir oldroot  
# pivot_root . oldroot/  
# umount -l oldroot
```

```
# cd /  
# exec /bin/sh  
# mount -t proc proc /proc
```



We're now inside the container!

Docker stores the image layers in folders in /var/lib/docker



Docker assembles the layers via overlay(fs). This is the root of a container (inside a mount namespace)

```
/ # mount
overlay on / type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay2/l/ICKVEA7TWBBEM3CBPNATQU70PX:/var/lib
/docker/overlay2/l/QVB5YYJ7D2YDGJZU4TBSXA6WG7,upperdir=/var/lib/docker/overlay2/efbc25e425ef513754264a0197ddcd
bc5c63c9ac3ee738f8519739e80cb166ba/diff,workdir=/var/lib/docker/overlay2/efbc25e425ef513754264a0197ddcd/bc5c63c
9ac3ee738f8519739e80cb166ba/work)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev type tmpfs (rw,nosuid,size=65536k,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=666)
sysfs on /sys type sysfs (ro,nosuid,nodev,noexec,relatime)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,relatime,mode=755)
cgroup on /sys/fs/cgroup/systemd type cgroup (ro,nosuid,nodev,noexec,relatime,xattr,name=systemd)
cgroup on /sys/fs/cgroup/cpuset type cgroup (ro,nosuid,nodev,noexec,relatime,cpuset)
cgroup on /sys/fs/cgroup/devices type cgroup (ro,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (ro,nosuid,nodev,noexec,relatime,net_cls,net_prio)
```

We have a separate filesystem, but the attacker still has access to devices (harddisk)

0) Prepare an Image

1) Isolate the filesystem

2) Restrict access to devices

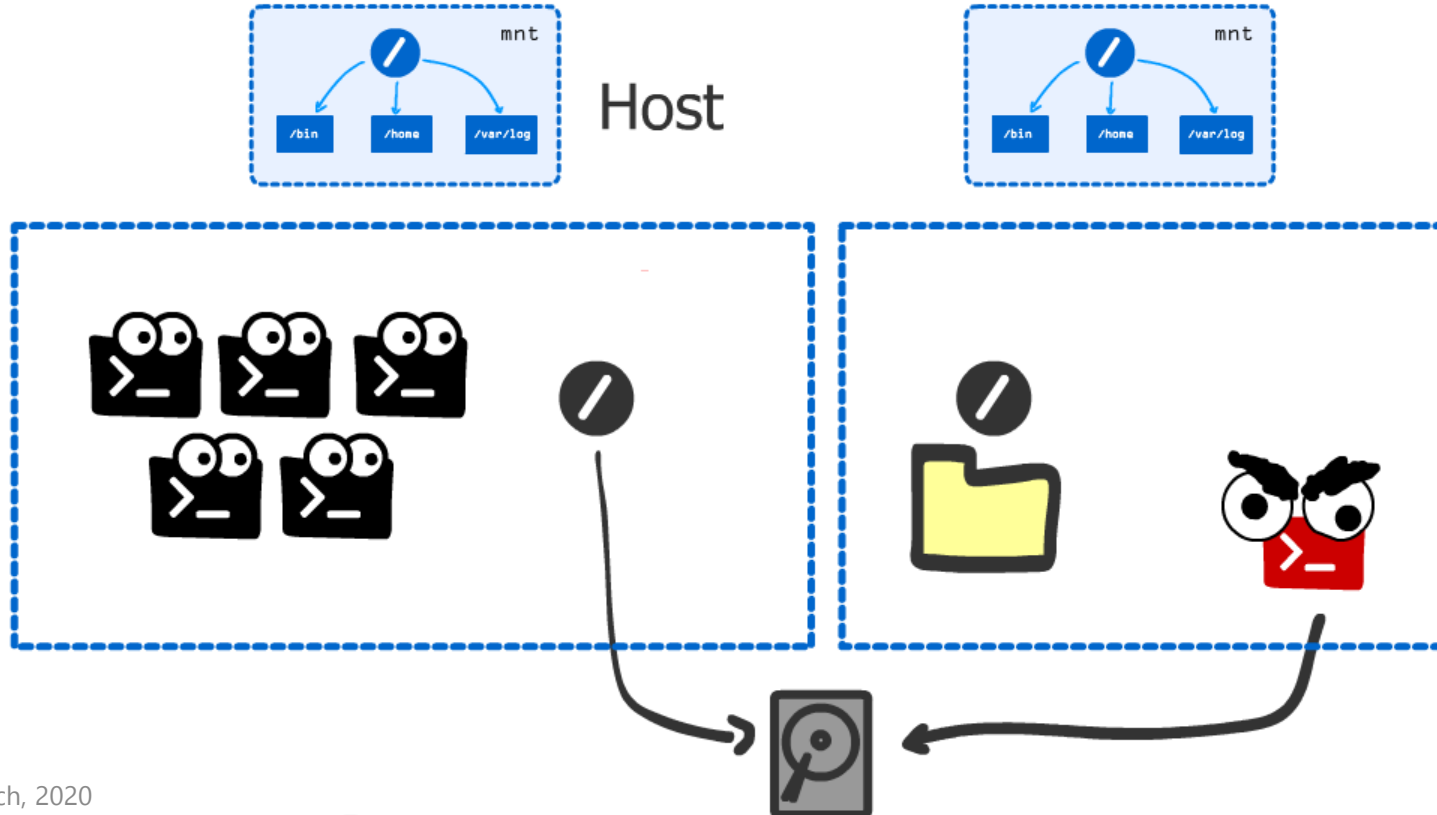
3) Drop user capabilities

4) Isolate network

5) Fix breakouts




The devices are addressed by major and minor numbers, as well as device types.



The attacker can break out via device nodes (harddisk). After that the host is corrupted!

```
# mknod /dev/sda1 b 8 1
# dd if=/dev/sda1 of=diskdump.bin bs=2048 count=1

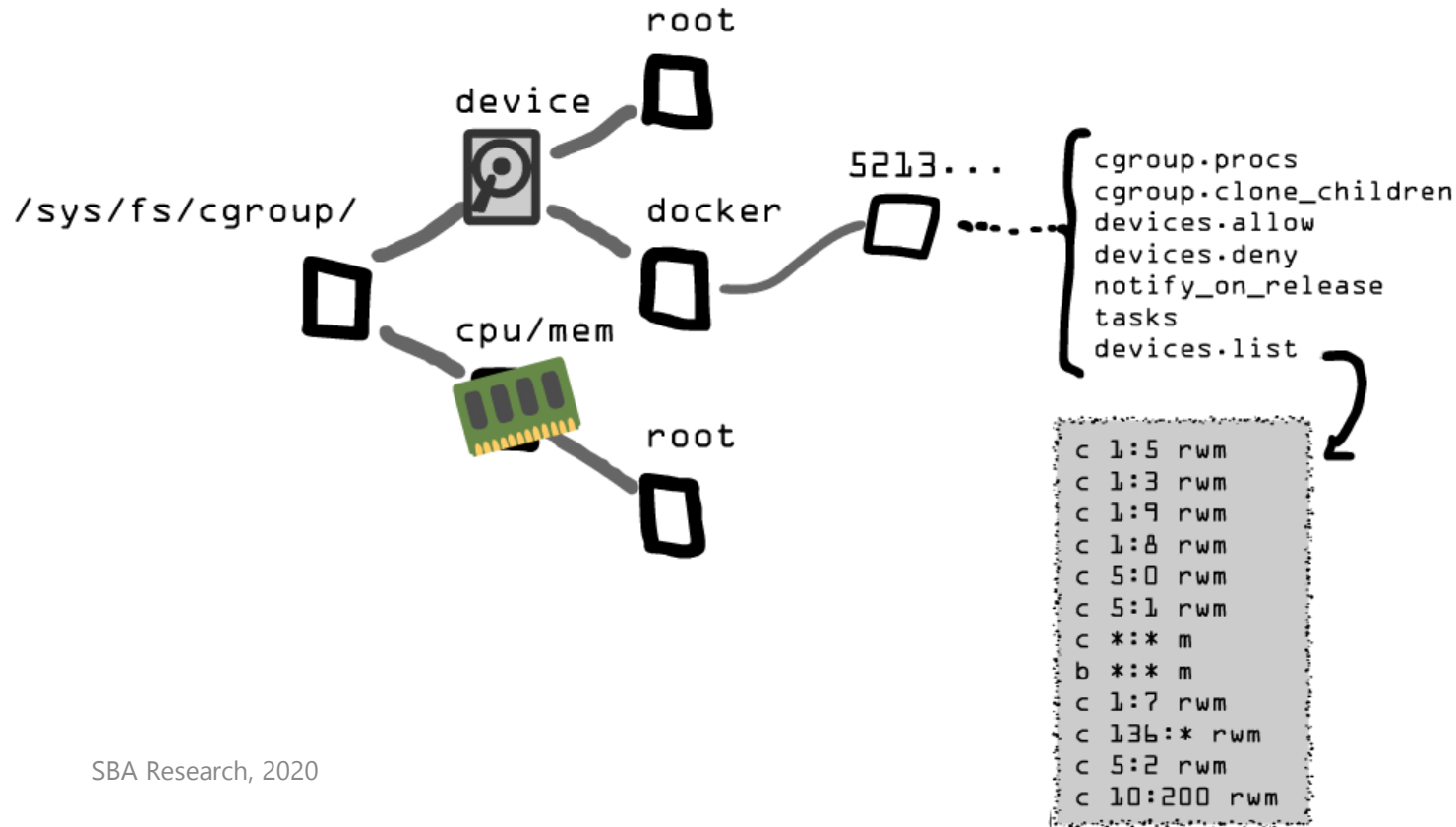
# mount /dev/sda1 /mnt
# chroot /mnt
```



We changed the root back to the original filesystem. We broke out!

```
drwxrwxrwt 16 root root 12288 Jan 31 14:09 tmp
drwxr-xr-x 14 root root 4096 May 16 2020 usr
drwxr-xr-x 12 root root 4096 May 16 2020 var
lrwxrwxrwx 1 root root 30 May 16 2020 vmlinuz → boot/vmlinuz-5.6.0-kali1-amd64
lrwxrwxrwx 1 root root 30 May 16 2020 vmlinuz.old → boot/vmlinuz-5.4.0-kali3-amd64
root@linux:/#
```

The Devices cgroup restrict access to device types (c, b, ...) for specific operations (r/w)



Create a subgroup „mycontainer1“ and move the Container process to this restricted one.

(on the host)

```
# mkdir /sys/fs/cgroup/devices/mycontainer1
```

```
# echo ,b *:* rwm' | sudo tee  
/sys/fs/cgroup/devices/mycontainer1/devices.deny
```

```
# cat /sys/fs/cgroup/devices/mycontainer1/devices.list
```

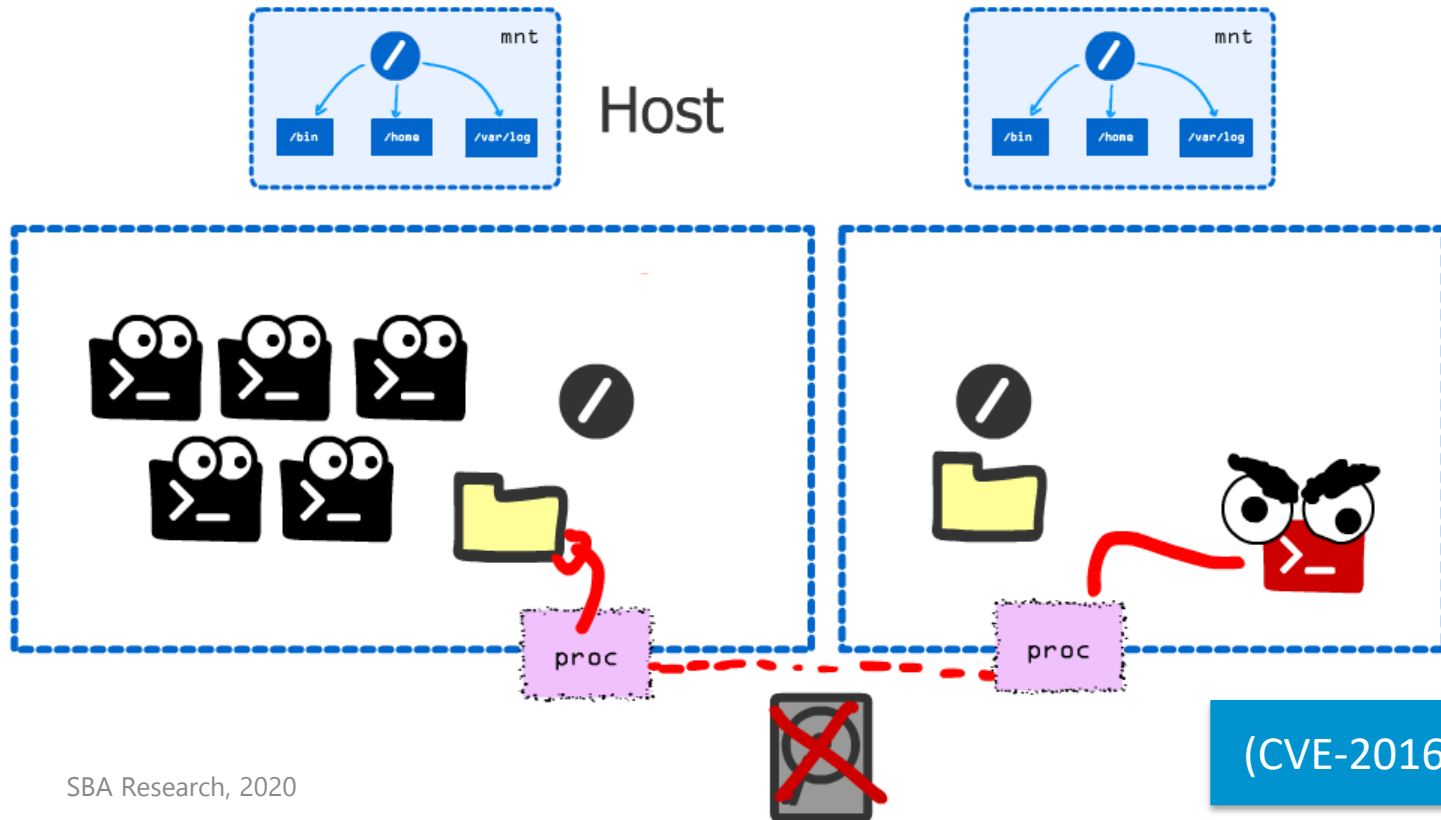
```
# echo 5966 | sudo tee  
/sys/fs/cgroup/devices/mycontainer1/cgroup.procs
```

This works. Docker does this too: block devices are restricted from read/write.

```
/ # echo $$
5966
/ # dd if=/dev/sda1 of=diskdump.bin bs=2048 count=1
dd: can't open '/dev/sda1': Operation not permitted
/ # mount /dev/sda1 /mnt/
mount: permission denied (are you root?)
/ # id
uid=0(root) gid=0(root) groups=0(root)
/ # █

sba@linux:~$ cat /sys/fs/cgroup/devices/docker/6c3ea20
8da61a4ecb64d8b7bfc3072b269/devices.list
c 1:5 rwm
c 1:3 rwm
c 1:9 rwm
c 1:8 rwm
c 5:0 rwm
c 5:1 rwm
c 1:10 rwm
```

Escape via /proc/<pid>/fd and chroot (overprivileged+no pid-ns)



We still see all other processes (same pid namespace). We can use the proc as tunnel

```
l-wx----- 1 root root 64 Jan 30 18:18 15 → pipe:[19468]
lr-x----- 1 root root 64 Jan 30 18:18 16 → pipe:[19469]
lrwx----- 1 root root 64 Jan 30 18:18 2 → socket:[17015]
lrwx----- 1 root root 64 Jan 30 18:18 3 → anon_inode:[eventfd]
lr-x----- 1 root root 64 Jan 30 18:18 4 → pipe:[17663]
l-wx----- 1 root root 64 Jan 30 18:18 5 → pipe:[17663]
l-wx----- 1 root root 64 Jan 30 18:18 6 → /var/log/lightdm/lightdm.log
lrwx----- 1 root root 64 Jan 30 18:18 7 → anon_inode:[eventfd]
lrwx----- 1 root root 64 Jan 30 18:18 8 → socket:[17119]
lr-x----- 1 root root 64 Jan 30 18:18 9 → /var/lib/lightdm/data
```

```
/proc/6359/fd:
```

```
total 0
```

```
lrwx----- 1 1000 1000 64 Jan 31 15:32 0 → /dev/pts/2
```

```
/ # chroot /proc/622/fd/9/ .. / .. / .. /
```

```
root@linux:/# ls
```

```
bin  etc  initrd.img  lib32  lost+found  opt  run  swapfile  usr  vmlinuz.old
boot home  initrd.img.old  lib64  media  proc  sbin  sys  var
dev  HOST  lib  libx32  mnt  root  srv  tmp  vmlinuz
```

```
root@linux:/#
```

We have too many privileges (as root, e.g. We can run mount or chroot) – let's change that!

0) Prepare an Image

1) Isolate the filesystem

2) Restrict access to devices

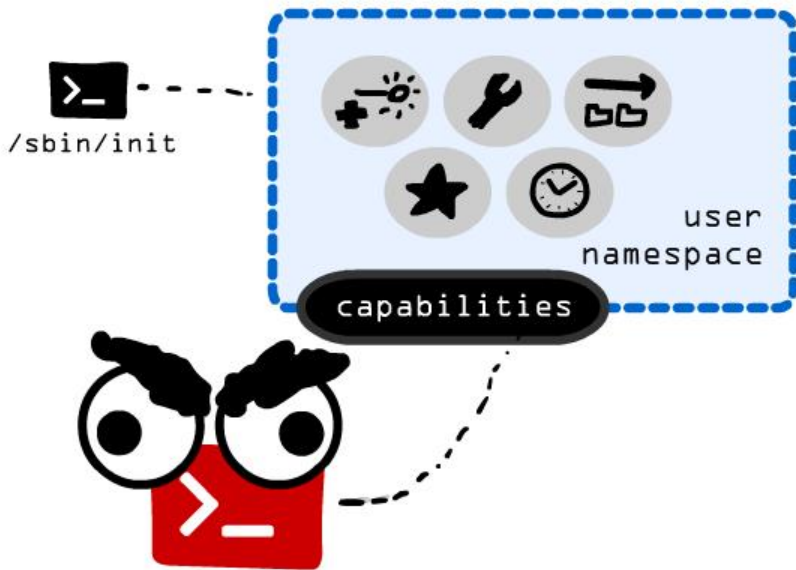
3) Drop user capabilities

4) Isolate network

5) Fix breakouts



Linux split root into multiple Capabilities. We only need CAP_NET_BIND_SERVICE



```
# id
uid=0(root) gid=0(root) groups=0(root)
```

```
# cat /proc/self/status | grep Cap
CapInh: 0000000000000000
CapPrm: 0000003ffffff
CapEff: 0000003ffffff
```

CAP_MAC_OVERRIDE (since Linux 2.6.25)

Override Mandatory Access Control (MAC). Implemented for the Smack LSM.

CAP_MKNOD (since Linux 2.4)

Create special files using `mknod(2)`.

CAP_NET_ADMIN

Perform various network-related operations:

- * interface configuration;
- * administration of IP firewall, masquerading, and accounting;
- * modify routing tables;
- * bind to any address for transparent proxying;
- * set type-of-service (TOS);
- * clear driver statistics;
- * set promiscuous mode;
- * enabling multicasting;
- * use `setsockopt(2)` to set the following socket options: **SO_DEBUG**, **SO_MARK**, **SO_PRIORITY** (for a priority outside the range 0 to 6), **SO_RCVBUFFORCE**, and **SO_SNDBUFFORCE**.

CAP_NET_BIND_SERVICE

Using su to switch to a user drops all Capabilities. Is there a way in between?

```
linux:~$ capsh --print
```

```
Current: =
```

```
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill  
,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,  
cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,  
cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,  
cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,  
cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read
```

```
Ambient set =
```

```
Securebits: 00/0x0/1'b0
```

```
secure-noroot: no (unlocked)
```

```
secure-no-suid-fixup: no (unlocked)
```

```
secure-keep-caps: no (unlocked)
```

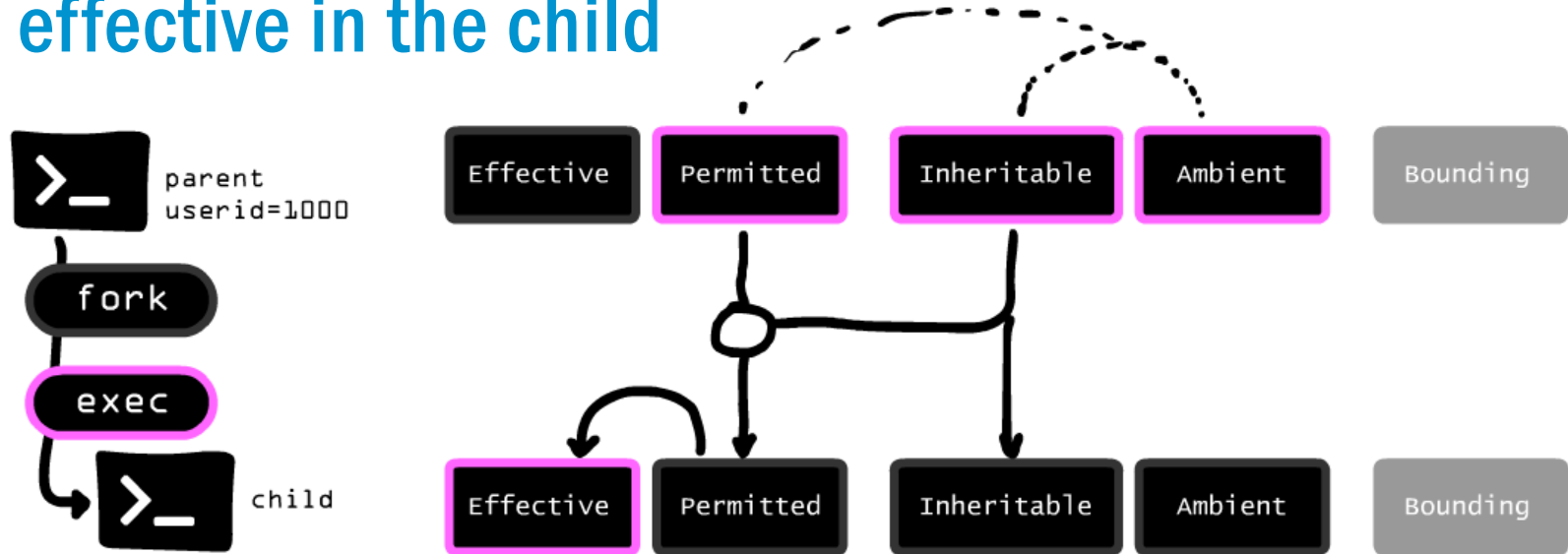
```
secure-no-ambient-raise: no (unlocked)
```

```
uid=1000(www)
```

```
gid=1000(www)
```

```
groups=1000(www)
```

We only need NET_BIND_SERVICE in the effective set of the child process. Execve needs permitted+inheritable+ambient to set effective in the child



Drop capabilities to only net_bind_service in the child process

```
# capsh --keep=1 --caps="cap_net_bind_service+epi  
cap_setpcap,cap_setgid,cap_setuid+ep" --user=www-data --  
addamb=cap_net_bind_service --
```

```
/ $ id  
uid=1000(www) gid=1000(www) groups=1000(www)  
/ $ capsh --print  
Current: = cap_net_bind_service+eip  
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search  
,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_s  
t,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap
```


Docker drops automatically dangerous Capabilities (for you)

```
sba@linux:~$ sudo capsh --decode=00000000a80425fb
0x00000000a80425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
sba@linux:~$ sudo capsh --decode=00000003ffffffff
0x00000003ffffffff=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read
sba@linux:~$
```

Next up: networking

0) Prepare an Image

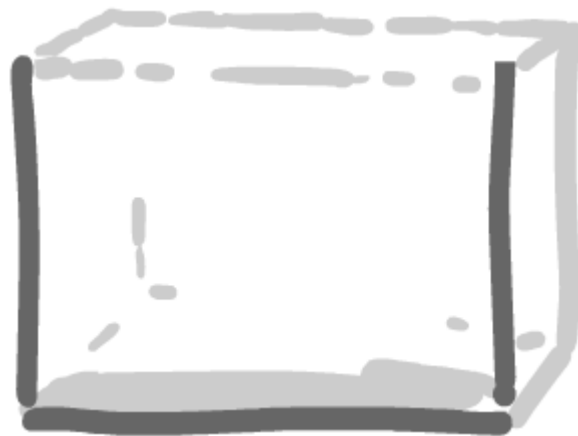
1) Isolate the filesystem

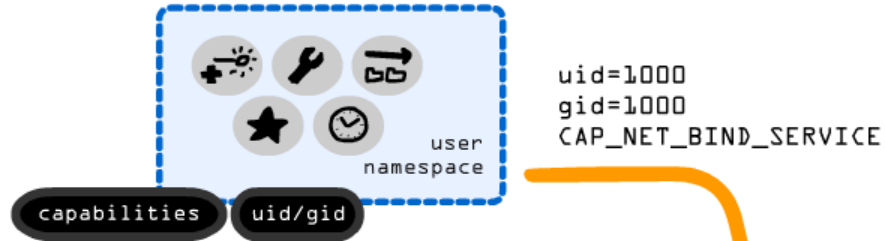
2) Restrict access to devices

3) Drop user capabilities

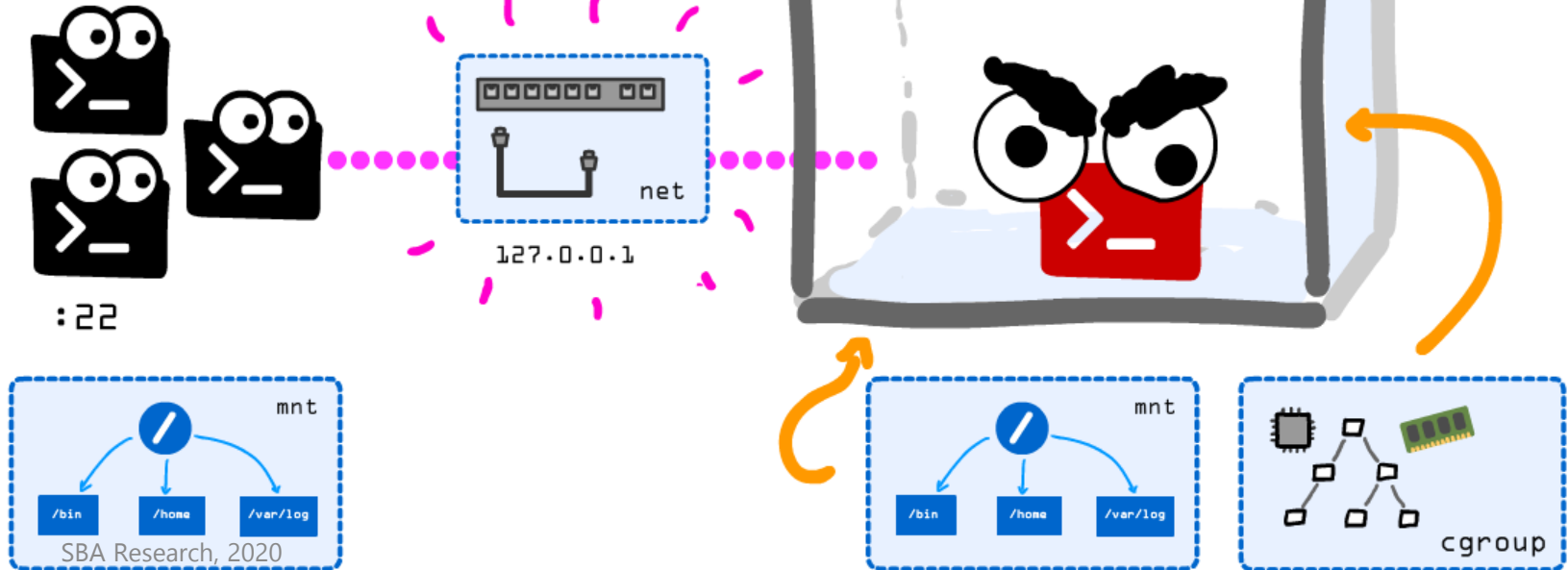
4) Isolate network

5) Fix breakouts

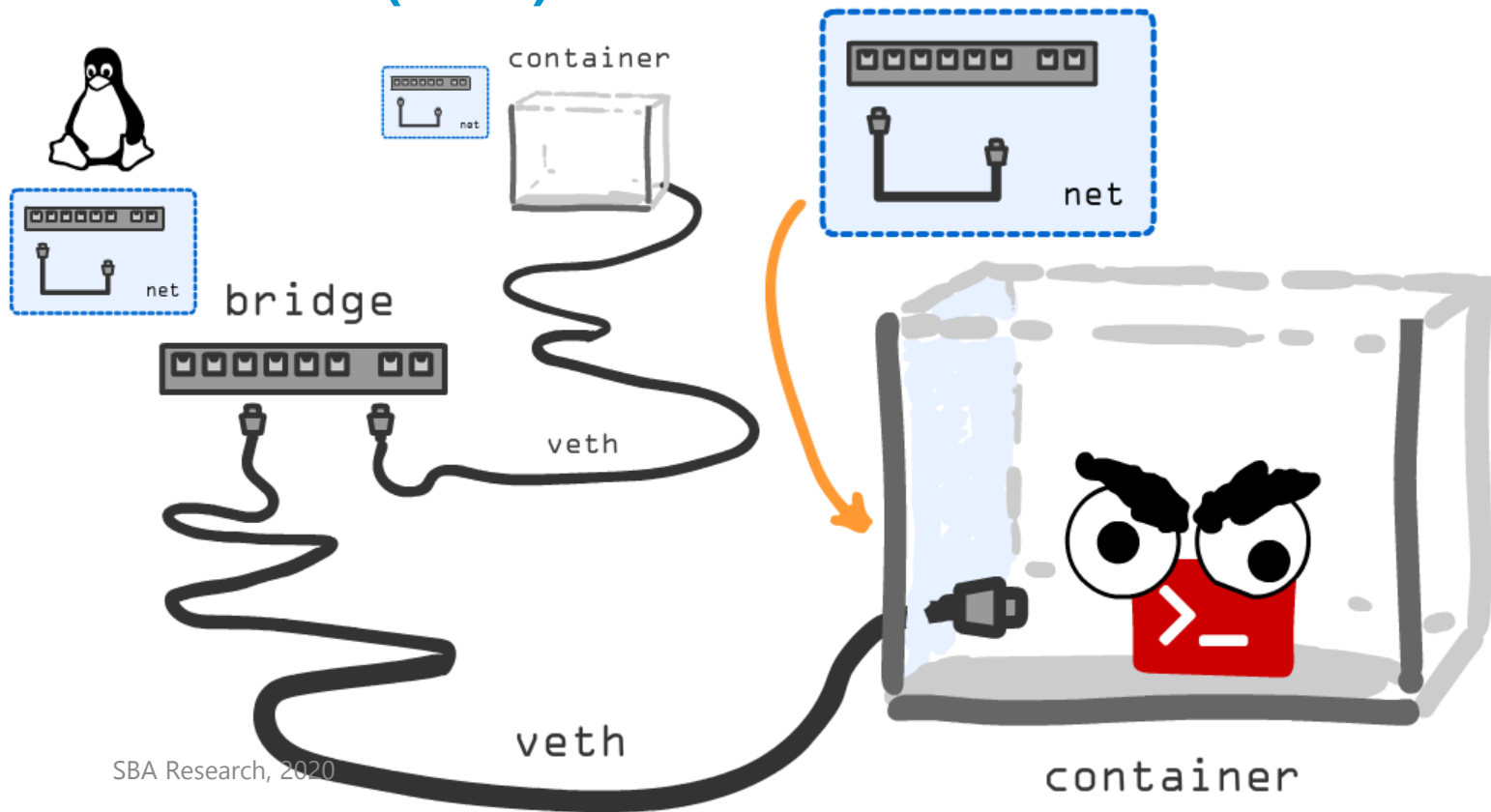




We still have a network connection from the Container to the Host



We create a new network namespace and cables (veth) to isolate the host



Create a bridge and connect containers and or other network interfaces

```
# export ID=42
# ip netns attach connetns$ID 1443
# ip link add vethHOST$ID type veth peer name vethCON$ID
# ip link set vethCON$ID netns connetns$ID
# ip netns exec connetns$ID ip addr add 192.168.42.2/24 dev vethCON$ID
# ip netns exec connetns$ID ip link set dev vethCON$ID up
# ip link set dev vethHOST$ID up
# ip route add 192.168.42.1/32 dev vethHOST$ID
# ip netns exec connetns$ID ip route add default via 192.168.42.1 dev vethCON$ID
# ip link add name br0 type bridge
# ip link set dev vethHOST$ID master br0
# ip addr add 192.168.42.1/24 dev br0
# ip link set dev br0 up
```

Almost done – is there still a breakout?

- 0) Prepare an Image
- 1) Isolate the filesystem
- 2) Restrict access to devices
- 3) Drop user capabilities
- 4) Isolate network
- 5) Fix breakouts**



What about malicious images? Are we safe from hacks?



Maybe an attacker provided a present for us?

We have to react to compromised or malicious images! This one uses a suid binary to escalate to root

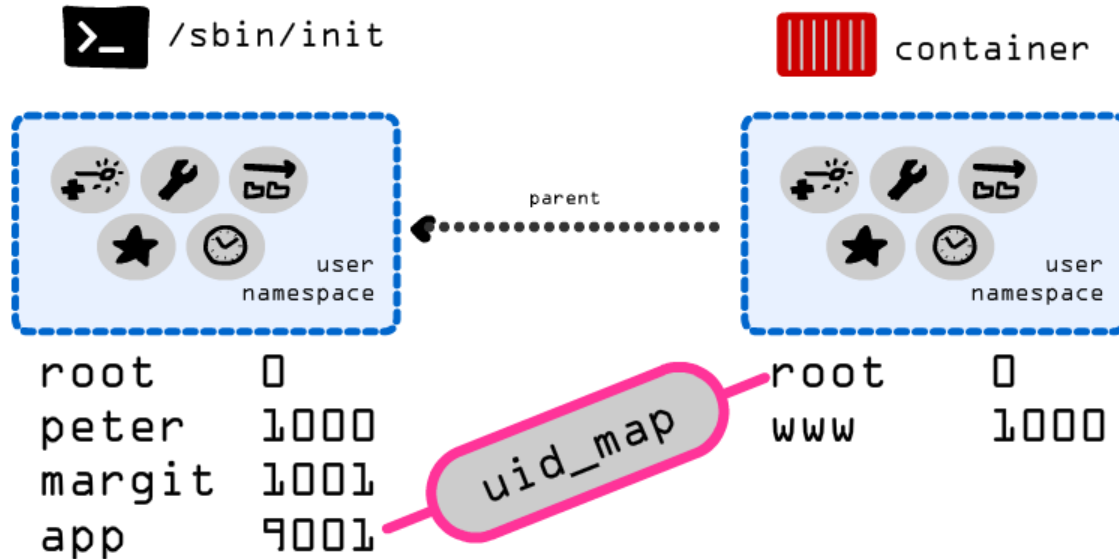
```
drwxr-xr-x 16 www www 4096 Jan 31 16:21 etc
-rwsr-xr-x 1 root www 778656 Jan 31 17:44 hacksudo
drwxr-xr-x 5 www www 4096 Jan 31 16:19 home
drwxr-xr-x 5 www www 4096 Jan 29 2019 lib
drwxr-xr-x 5 www www 4096 Jan 29 2019 media
drwxr-xr-x 2 www www 4096 Jan 29 2019 mnt
```

```
/ $ ./hacksudo /bin/sh
/ # id
uid=0(root) gid=1000(www) groups=1000(www)
/ # █
```



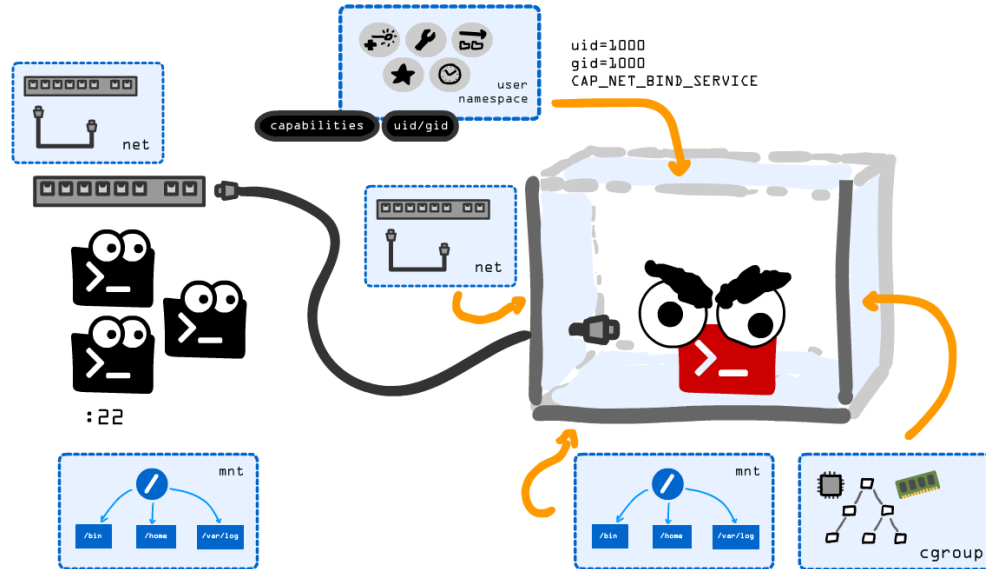
Root on the
host

User namespaces got introduced to map a user (root) inside the container to a user (>0) on the host



container ID	host id start	how many	
0	9001	1	root is app on host
1	9002	999..	everyone else is

Result: The Container has a new root/file system (mnt ns), a restricted cgroup for devices, own network (net ns), restricted capabilities and restricted user id



Done! A Container made without Docker.

0) Prepare an Image

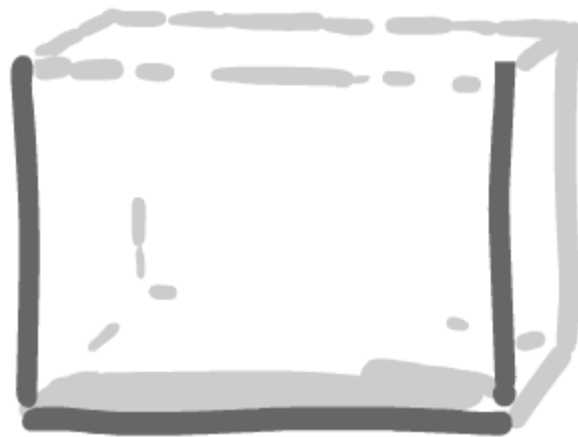
1) Isolate the filesystem

2) Restrict access to devices

3) Drop user capabilities

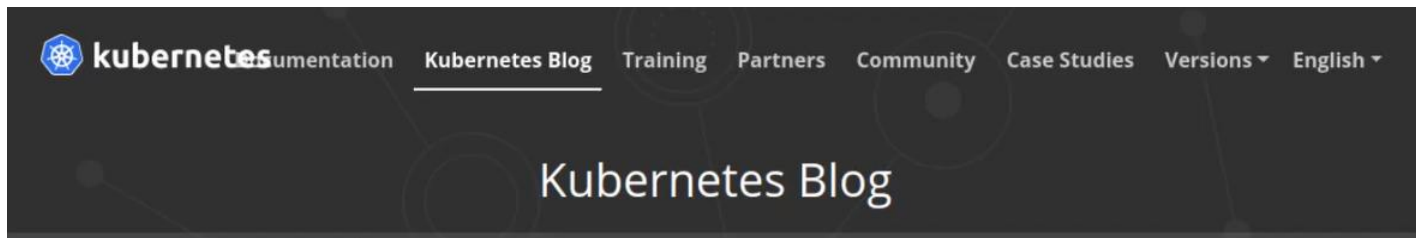
4) Isolate network

5) Fix breakouts?



Do try this at home. But not in production!

Kubernetes moved to the Container Native Interface (CNI) and use runc



2020

A Custom Kubernetes Scheduler to Orchestrate Highly Available Applications

Kubernetes 1.20: Pod Impersonation and Short-lived Volumes in CSI Drivers

Third Party Device Metrics Reaches GA

Kubernetes 1.20: Search for Volume Permission

Don't Panic: Kubernetes and Docker

Wednesday, December 02, 2020

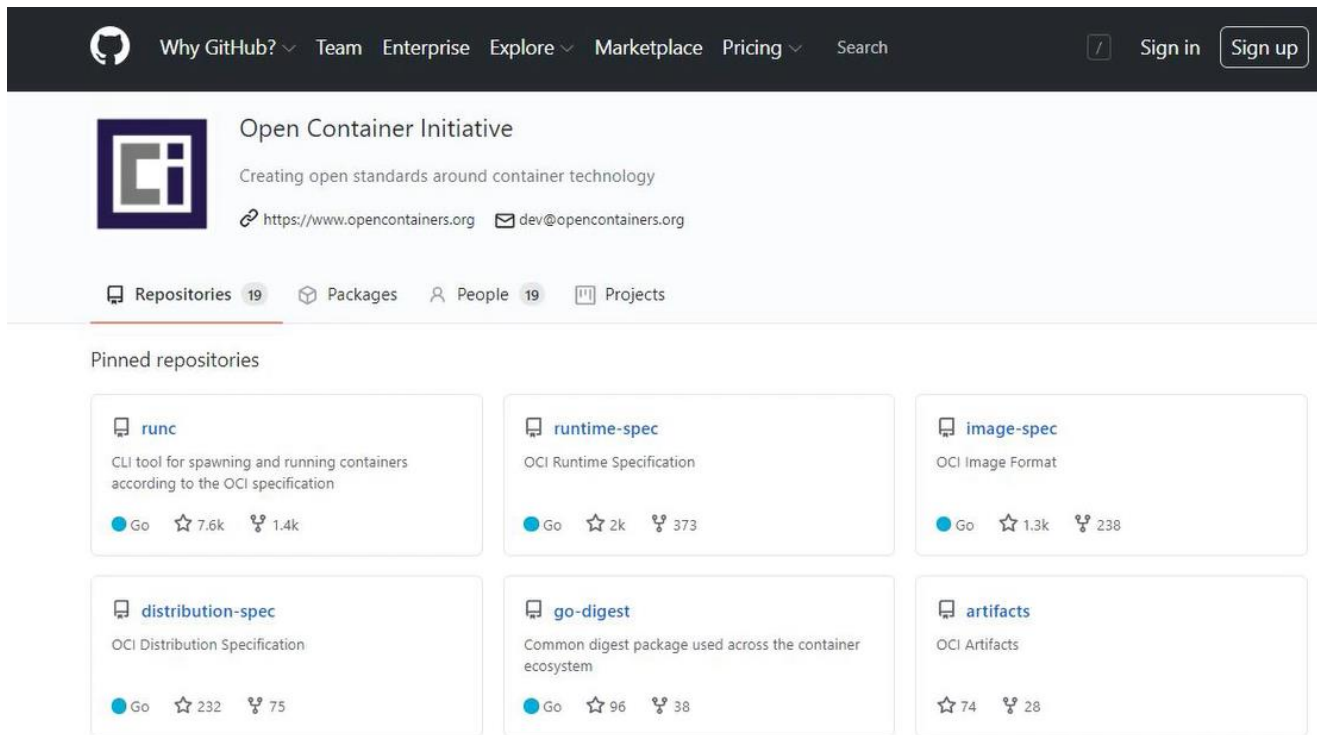
Authors: Jorge Castro, Duffie Cooley, Kat Cosgrove, Justin Garrison, Noah Kantrowitz, Bob Killen, Rey Lejano, Dan "POP" Papandrea, Jeffrey Sica, Davanum "Dims" Srinivas

Kubernetes is [deprecating Docker](#) as a container runtime after v1.20.

You do not need to panic. It's not as dramatic as it sounds.


TL;DR Docker as an underlying runtime is being deprecated in favor of runtimes that use the [Container Runtime Interface \(CRI\)](#) created for Kubernetes. Docker-

OCI provides runc – this is also used by Docker!









The screenshot shows the GitHub page for the Open Container Initiative. The header includes navigation links like 'Why GitHub?', 'Team', 'Enterprise', 'Explore', 'Marketplace', 'Pricing', and 'Search'. The main content area features the OCI logo and name, followed by a description: 'Creating open standards around container technology'. Below this are links to the website and email. A section for 'Pinned repositories' lists six key projects: runc, runtime-spec, image-spec, distribution-spec, go-digest, and artifacts, each with a brief description and statistics.

Why GitHub? ▾ Team Enterprise Explore ▾ Marketplace Pricing ▾ Search [/](#) [Sign in](#) [Sign up](#)

 **Open Container Initiative**
Creating open standards around container technology
<https://www.opencontainers.org> dev@opencontainers.org

[Repositories 19](#) [Packages](#) [People 19](#) [Projects](#)

Pinned repositories

-  **runc**
CLI tool for spawning and running containers according to the OCI specification
[Go](#) [7.6k](#) [1.4k](#)
-  **runtime-spec**
OCI Runtime Specification
[Go](#) [2k](#) [373](#)
-  **image-spec**
OCI Image Format
[Go](#) [1.3k](#) [238](#)
-  **distribution-spec**
OCI Distribution Specification
[Go](#) [232](#) [75](#)
-  **go-digest**
Common digest package used across the container ecosystem
[Go](#) [96](#) [38](#)
-  **artifacts**
OCI Artifacts
[74](#) [28](#)

Now build your own container by hand! 😊

Reinhard Kugler, SBA Research

Floragasse 7, 1040 Vienna

rkugler@sba-research.org

References

- **The Linux Programming Interface**, Michael Kerrisk (2010)
- **Container Security**, Liz Rice (2020)
- **CVE-2016-996**: On-entry container attack https://bugzilla.suse.com/show_bug.cgi?id=1012568#c6
- runc (OCI) <https://github.com/opencontainers/runc>
- **Container Networking From Scratch**, Kristen Jacobs, Oracle https://www.youtube.com/watch?v=6v_BDHlgOY
- **Bocker** - Docker implemented in around 100 lines of bash, Peter Wilmott <https://github.com/p8952/bocker>
- **Kubernetes Deconstructed**: Understanding Kubernetes by Breaking It Down, Carson Anderson <https://www.youtube.com/watch?v=90kZRyPcRZw>
- **Securing Container Runtimes** -- How Hard Can It Be?, Aleksa Sarai (2020) https://www.youtube.com/watch?v=tGseJW_uBB8
- **Rootless Containers from Scratch**, Liz Rice <https://www.youtube.com/watch?v=jeTKgAEyhsA>
- **Filesystem mounts in user namespaces**, Christian Brauner <https://www.youtube.com/watch?v=2CuWuW7AYdE>
- **What's Under the Hood of Docker?** Process Separation in the Linux kernel by Janos Pasztor <https://www.youtube.com/watch?v=8iWb71ZOZPc>