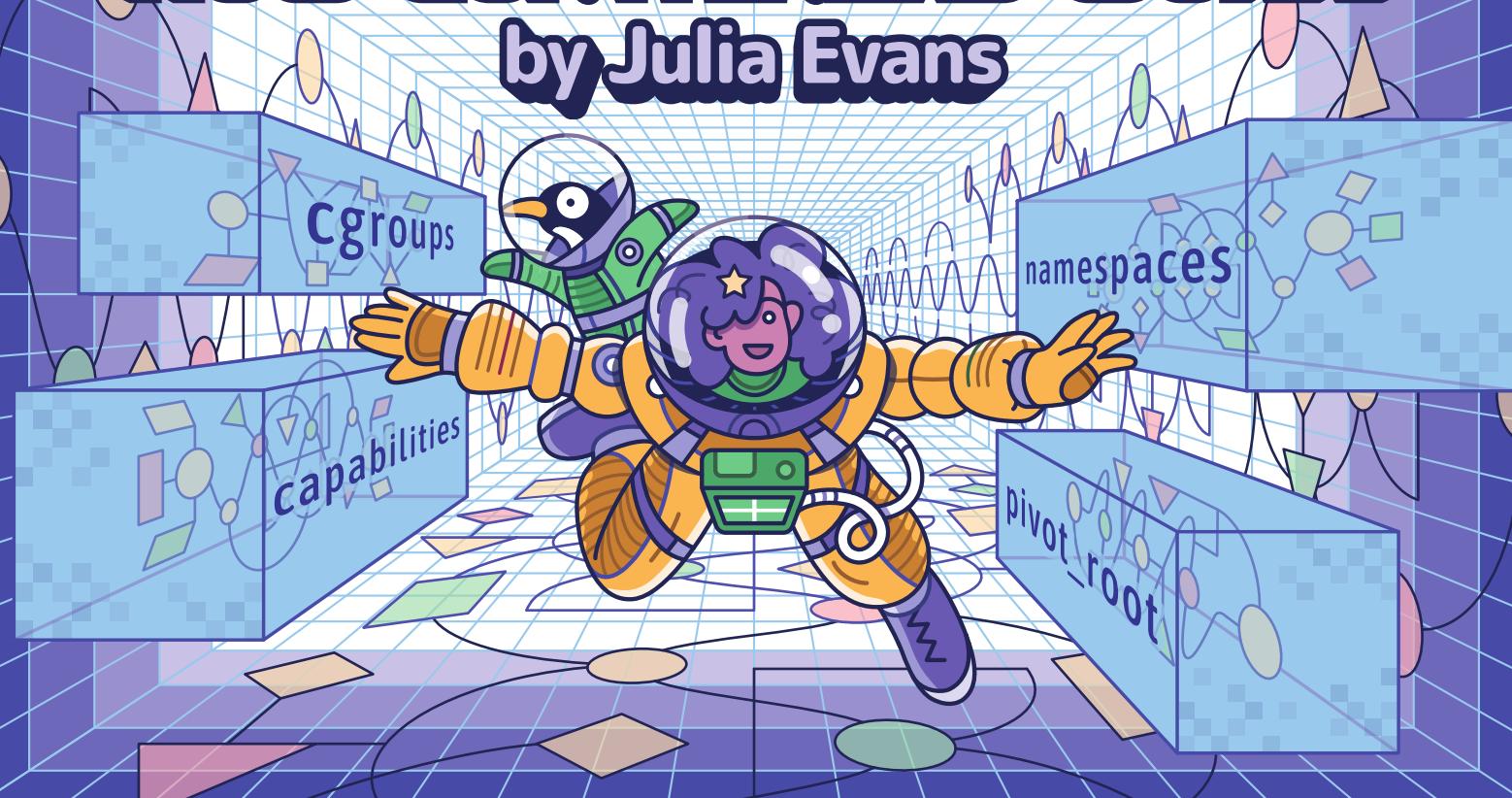


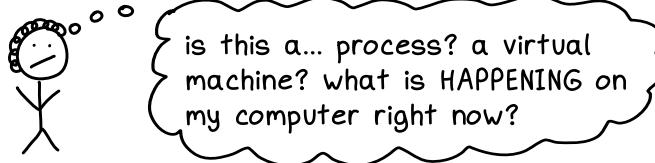
# HOW CONTAINERS WORK

by Julia Evans

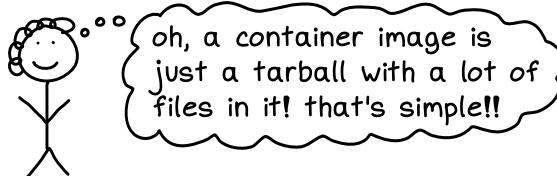


# why this zine?

When I started using containers I was SO CONFUSED.

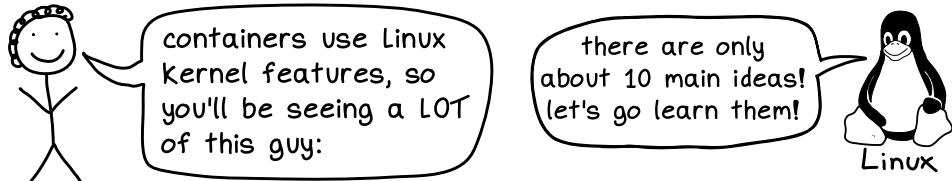


So I decided to learn how they work under the hood!



Now I feel confident that I can solve basically any problem with containers because I understand how they work.

I hope that after reading this zine, you'll feel like that too.

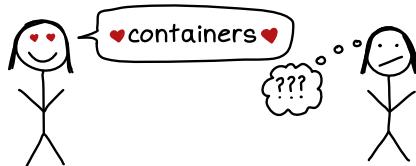


# table of contents

why containers?.....	4	cgroups.....	13
the big idea: include EVERY dependency.....	5	namespaces.....	14
containers aren't magic.....	6	how to make a namespace.....	15
containers = processes.....	7	PID namespaces.....	16
container kernel features.....	8	user namespaces.....	17
pivot_root.....	9	network namespaces.....	18
layers.....	10	container IP addresses.....	19
overlay filesystems.....	11	capabilities.....	20
container registries.....	12	seccomp-BPF.....	21
		configuration options .....	22

# why containers?

there's a lot of container hype

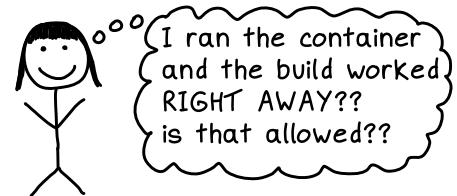


Here are 2 problems they solve...

**problem:** building software is annoying

```
$ ./configure  
$ make all  
ERROR: you have version  
2.1.1 and you need  
at least 2.2.4
```

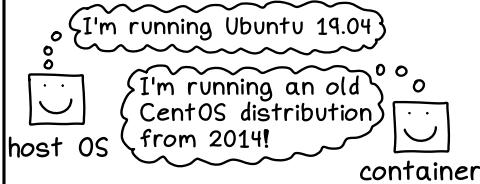
**solution:** package all dependencies in a ★ container ★



Many CI systems use containers.

containers have their own filesystem

This is the big reason containers are great.

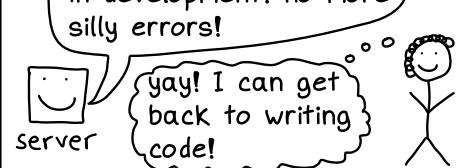


**problem:** deploying software is annoying too

ugh, my website is broken because I used a Python 3.6 feature and the server only has Python 3.5!

**solution:** deploy a container

I have the exact same version of everything as in development! no more silly errors!



# the big idea: include EVERY dependency

5

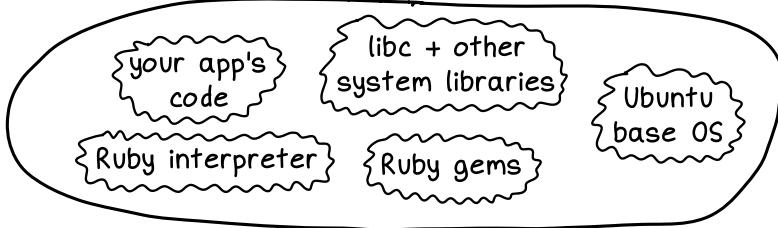
containers package EVERY dependency together



to make sure this program will run on your laptop, I'm going to send you every single file you need

a container image is a tarball of a filesystem

Here's what's in a typical Rails app's container:



how images are built

0. start with a base OS
1. install program + dependencies
2. configure it how you want
3. make a tarball of the WHOLE FILESYSTEM



this is what `docker build` does!

running an image

1. download the tarball
2. unpack it into a directory
3. run a program and pretend that directory is its whole filesystem

images let you "install" programs really easily



I can set up a Postgres test database in like 5 seconds! wow!

# containers aren't magic

6

These 15 lines of bash will start a container running the fish shell. Try it!  
(download this script at [bit.ly/containers-arent-magic](http://bit.ly/containers-arent-magic))

It only runs on Linux because these features are all Linux-only.

# containers = processes

a container is a group of Linux processes



on a Mac, all your containers are actually running in a Linux virtual machine



I started `top` in a container.  
Here's what that looks like in ps:

outside the container

```
$ ps aux | grep top
USER PID START COMMAND
root 23540 20:55 top
bork 23546 20:57 top
```

inside the container

```
$ ps aux | grep top
USER PID START COMMAND
root 25 20:55 top
```

these are the same process!

container processes can do anything a normal process can ...



I want my container to do X Y Z W!



sure! your computer, your rules!

... but usually they have restrictions

different PID

namespace cgroup memory limit

different root directory

limited capabilities

not allowed to run some system calls

the restrictions are enforced by the Linux Kernel



NO, you can't have more memory!

on the next page we'll list all the kernel features that make this work!



# container kernel features

## containers use these Linux Kernel features

"container" doesn't have a clear definition, but Docker containers use all of these features.

### ♥ pivot\_root ♥

set a process's root directory to a directory with the contents of the the container image

### ★ cgroups ★

limit memory/CPU usage for a group of processes



only 500 MB of RAM for you!

Linux

### ♥ namespaces ♥

allow processes to have their own:

- network
- PIDs
- hostname
- mounts
- users
- + more

### ★ capabilities ★

security: give specific permissions

### ★ overlay filesystems ★

this is what makes layers work! Sharing layers saves disk space & helps containers start faster

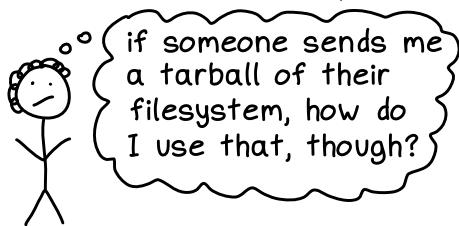
### ♥ seccomp-bpf ♥

security: prevent dangerous system calls

# pivot\_root

a container image is a  
tarball of a filesystem

(or several tarballs: 1 per layer)



**chroot:** change a process's root directory

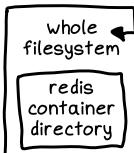
If you chroot to /fake/root, when it opens the file /usr/bin/redis, it'll get /fake/root/usr/bin/redis instead.

You can "run" a container just by using chroot, like this:

```
$ mkdir redis; cd redis
$ tar -xzf redis.tar
$ chroot $PWD /usr/bin/redis
# done! redis is running!
```

programs can break  
out of a chroot

chroot



all these files are  
still there! A root  
process can access  
them if it wants.

pivot\_root



you can unmount  
the old filesystem  
so it's impossible to  
access it.

Containers use pivot\_root instead of chroot.

to have a "container" you  
need more than pivot\_root

pivot\_root alone won't let you:

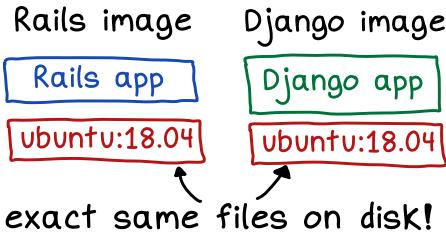
- set CPU/memory limits
- hide other running processes
- use the same port as another process
- restrict dangerous system calls

# layers

different images  
have similar files



reusing layers  
saves disk space



a layer is a directory

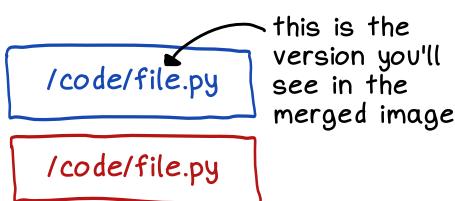
```
$ ls 8891378eb*
bin/ home/ mnt/ run/ tmp/
boot/ lib/ opt sbin/ usr/
dev/ lib64/ proc/ srv/ var/
etc/ media/ root/ sys/
files in an
ubuntu:18.04 layer
```

every layer has an ID

usually the ID is a  
sha256 hash of the  
layer's contents

example: 8e99fae2..

if a file is in 2 layers,  
you'll see the version  
from the top layer



by default, writes go  
to a temporary layer



these files  
might be deleted  
after the  
container exits

To keep your changes, write  
to a directory that's mounted  
from outside the container

# overlay filesystems

## how layers work:

`mount -t overlay`

can you combine these 37 layers into one filesystem?



lowerdir:  
the layers. read only.

```
you can run  
$ mount -t overlay  
inside a container to  
see all the lowerdirs  
that were combined to  
create its filesystem!
```

mount -t overlay  
has 4 parameters

lowerdir:

list of read-only directories

**upperdir:**

directory where writes should go

workdir:

empty directory for internal use

target:

the merged result

upperdir:  
where all writes go

when you create, change, or delete a file, it's recorded in the upperdir.

usually this starts out empty  
and is deleted when the  
container exits

**lowerdir:**

the layers. read only.

here's an example!

```
$ mount -t overlay overlay -o  
    lowerdir=/lower,upperdir=/upper,workdir=/work /merged
```

```
$ ls /upper
```

```
cat txt dog txt
```

```
$ ls /lower
```

dog.txt bird.txt

\$ ls /merged

cat.txt dog.txt bird.txt

the merged version of dog.txt is the one from the upper directory

# container registries

12

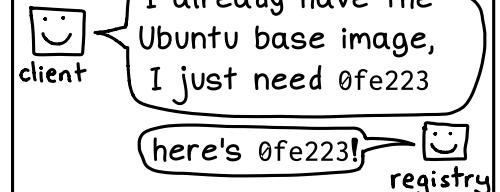
sharing container images is useful



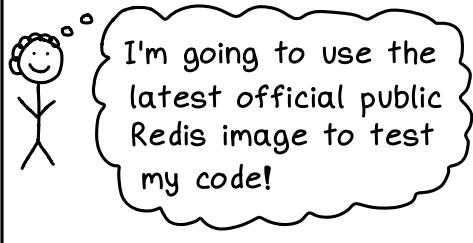
a registry is a server that serves images

images have an ID <sup>like "leff92"</sup>  
and sometimes a tag  
like "18.04" or "latest"

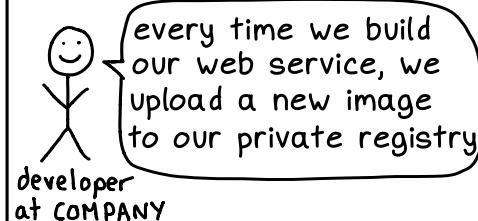
registries let you download just the layers you need



there are **public** container registries...



... and **private** registries

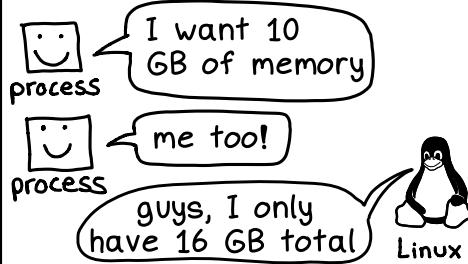


be **careful** where your container images come from

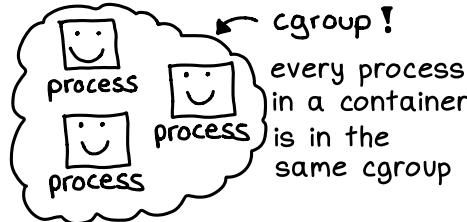


# cgroups

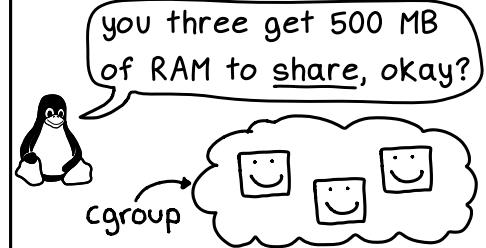
processes can use a lot of memory



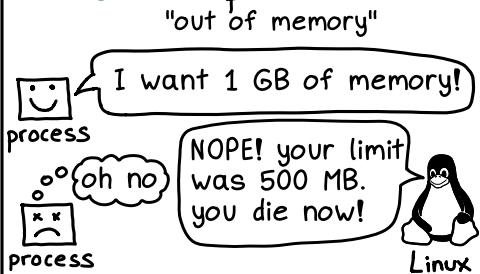
a cgroup is a group of processes



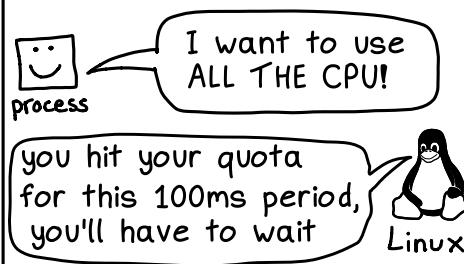
cgroups have memory/CPU limits



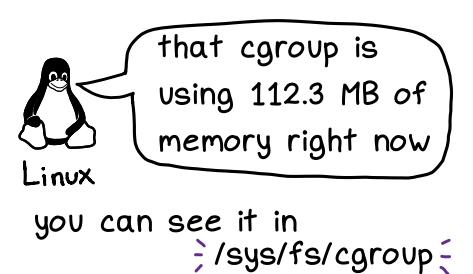
use too much memory: get OOM killed



use too much CPU: get slowed down



cgroups track memory & CPU usage



# namespaces

14

inside a container,  
things look different



I only see 4  
processes in  
ps aux, that's  
weird...

why things look different:  
↳ namespaces ↳



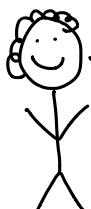
I'm in a different  
PID namespace so  
ps aux shows  
different processes!

every process has  
7 namespaces

```
$ lsns -p 273
    PID          NS TYPE
4026531835  cgroup
4026531836  pid
4026531837  user
4026531838  uts
4026531839  ipc
4026531840  mnt
4026532009  net
```

↑ namespace ID

there's a default  
("host" namespace)



"outside a  
container" just  
means "using the  
default namespace"

processes can have  
any combination  
of namespaces



I'm using the host  
network namespace  
but my own mount  
namespace!

you can also see a  
process's namespace with:  
\$ ls -l /proc/273/ns

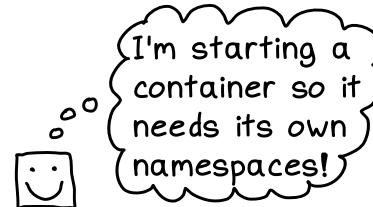
# how to make a namespace

15

processes use their parent's namespaces by default



but you can switch namespaces at any time



command line tools

\$ unshare --net COMMAND

run COMMAND in a new network namespace

\$ sudo lsns

list all namespaces

\$ nsenter -t PID --all COMMAND

run COMMAND in the same namespaces as PID

## namespace system calls

### ★ clone ★

make a new process

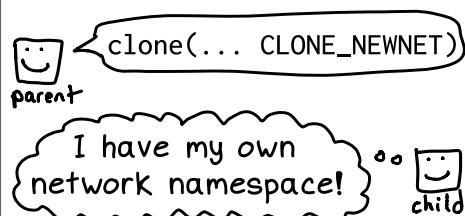
### ★ unshare ★

make + use a namespace

### ★ setns ★

use an existing namespace

★clone★ lets you create new namespaces for a child process



each namespace type has a ❤ man page ❤

\$ man network\_namespaces

...

A physical network device can live in exactly one network namespace

...

# PID namespaces

16

the same process has different PIDs in different namespaces

PID in host	PID in container
23512	4
23513	12
23518	PID 1 is special

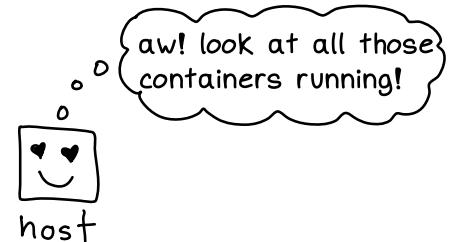
PID namespaces are in a tree

host PID namespace  
(the root)

child child child

Often the tree is just 1 level deep (every child is a container)

you can see processes in child PID namespaces



if PID 1 exits, everyone gets killed



Killing PID 1 accidentally would be bad



rules for signaling PID 1

from same container:

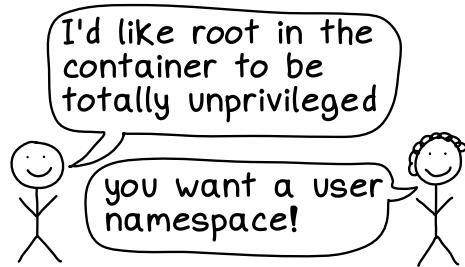
only works if the process has set a signal handler

from the host:

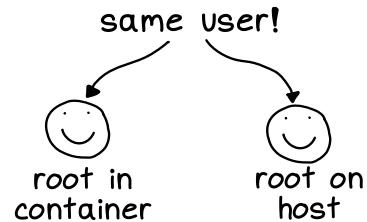
only SIGKILL and SIGSTOP are ok, or if there's a signal handler

# User namespaces

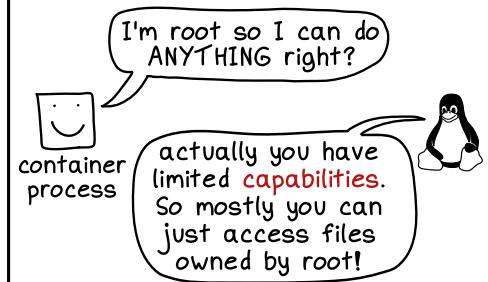
user namespaces are a security feature...



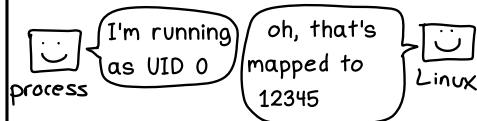
... but not all container runtimes use them



"root" doesn't always have admin access



in a user namespace, UIDs are mapped to host UIDs



The mapping is in  
/proc/self/uid\_map

unmapped users show up as "nobody"

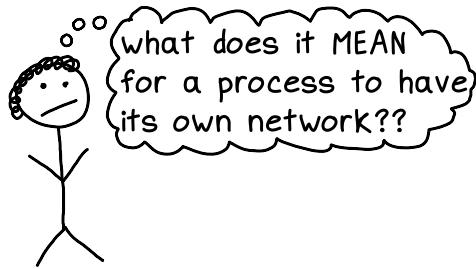
create user namespace  
\$ unshare --user bash  
\$ ls -l /usr/bin  
.. nobody nogroup apropos  
.. nobody nogroup apt  
these are "actually" owned by root  
but we didn't map any users

how to find out if you have a separate user namespace

compare the results of  
\$ ls /proc/PID/ns  
between a container process and a host process

# network namespaces

network namespaces  
are kinda confusing



namespaces usually  
have 2 interfaces

(+ sometimes more)

- the loopback interface  
(127.0.0.1/8, for connections  
inside the namespace)
- another interface  
(for connections from  
outside)

every server listens  
on a port and network  
interface(s)

0.0.0.0:8080

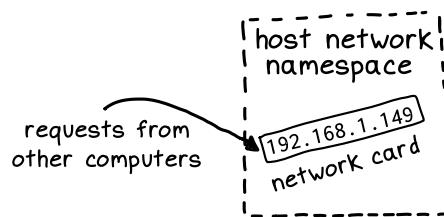
means

"port 8080 on every network  
interface in my namespace"

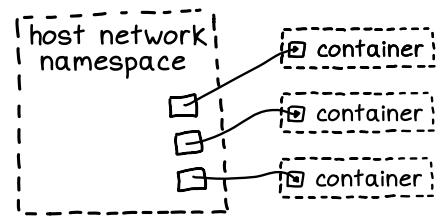
127.0.0.1 stays  
inside your namespace



your physical network  
card is in the host  
network namespace

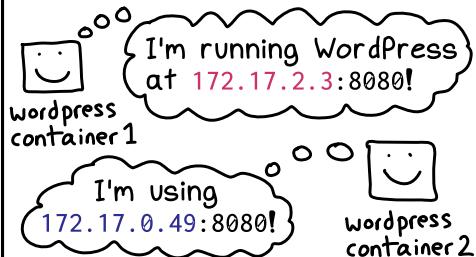


other namespaces are  
connected to the host  
namespace with a bridge



# container IP addresses

containers often get their own IP address



containers use private IP addresses

192.168.\*.\*  
10.\*.\*.\*  
172.16.\*.\*  
-> 172.32.\*.\*

} reserved for private networks (RFC 1918)

This is because they're not directly on the public internet

for a packet to get to the right place, it needs a route



inside the same computer, you'll have the right routes

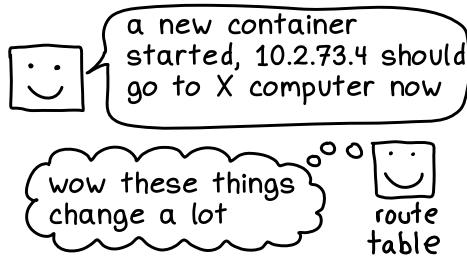
same computer:

```
$ curl 172.16.2.3:8080
<html>....
```

different computer:

```
$ curl 172.16.2.3:8080
.... no reply ....
```

distributing the right routes is complicated



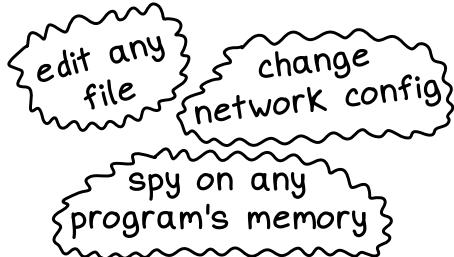
cloud providers have systems to make container IPs work

In AWS, this is called an "elastic network interface"

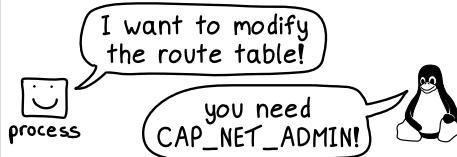
# capabilities

20

we think of root as being all-powerful...



... but actually to do "root" things, a process needs the right ★capabilities★



there are dozens of capabilities

\$ man capabilities explains all of them. But let's go over 2 important ones!

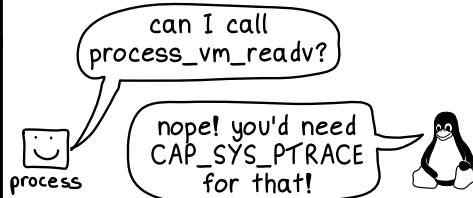
## CAP\_SYS\_ADMIN

lets you do a LOT of things.  
avoid giving this if you can!

## CAP\_NET\_ADMIN

allow changing network settings

by default containers have limited capabilities



## \$ getpcaps PID

print capabilities that PID has

## getcap / setcap

system calls:  
get and set capabilities!

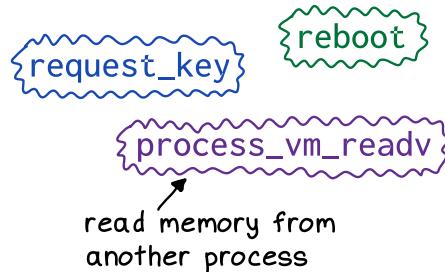
# seccomp-bpf

21

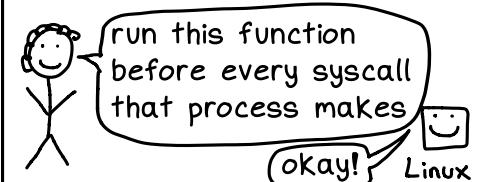
all programs use system calls



rarely-used system calls can help an attacker



seccomp-BPF lets you run a function before every system call

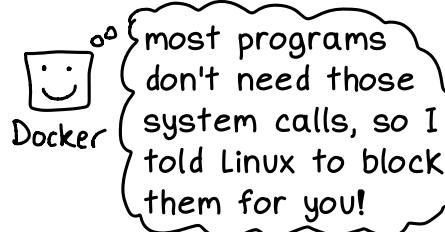


the function decides if that syscall is allowed

example function:

```
if name in allowed_list {  
    return true;  
}  
return false; this means the syscall doesn't happen!
```

Docker blocks dozens of syscalls by default



2 ways to block scary system calls

1. limit the container's capabilities
2. set a seccomp-bpf whitelist

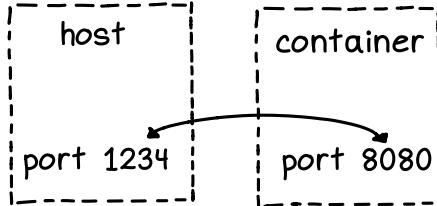
You should do both!

# configuration options

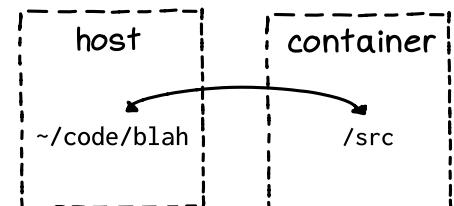


here are the 6 most important things you can configure when starting a container!

map a port to the host



mount directories from the host



set capabilities

set memory and CPU limits



add seccomp-bpf filters

only 200 MB RAM for you!

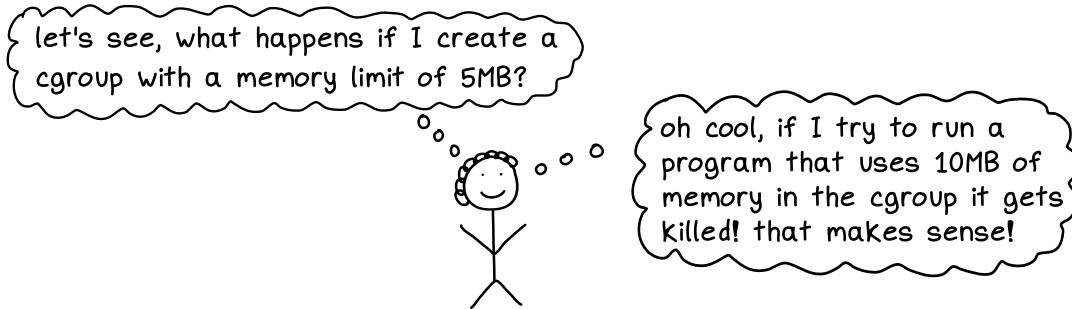
use the host network namespace

usually the default is to use a new network namespace!

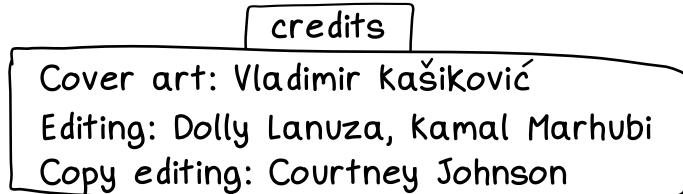
# ♥ thanks for reading ♥

23

I did a bunch of the research for this zine by reading the man pages.  
But, much more importantly, I experimented -- a lot!



So, if you have access to a Linux machine, try things out!  
Mount an overlay filesystem! Create a namespace! See what happens!



♡ this?  
more at  
★wizardzines.com★