SYS1 - Filesystem

man hier.7

Jules Aubert

Filesystem

Just a set of directories. . .

Filesystem

Just a set of directories. . .

Or is it?

hierarchy

```
man 7 hier
```

Device is nice

/dev/ looks like fun

```
cat /dev/input/mouse0
```

NOW MOVE YOUR MOUSE!!!

Device is nice

```
dd if=/dev/sda of=/tmp/data bs=10K count=1 status=progress
...
file /tmp/data
```

The entry point to your disk is a simple file.

Hey Manu, tu mount?

What about those mountpoints?

```
mount
```

Your filesystem is not simple directories, but an amazing maze of files!

Back to the fs

We can download rootfs to learn and play

Kernels? Where we're we going we don't need kernels

- Alpine is one of the lightest
- It works with a lot of distributions actually
- (We could take it from LXC but shush)

Double V guette

Wget us to try:

https://dl-cdn.alpinelinux.org/alpine/v3.16/releases/x86_64/alpine-minirootfs-3.16.2-x86_64.tar.gz

http://cdimage.ubuntu.com/ubuntu-base/releases/14.04/release/ubuntu-base-14.04-core-amd64.tar.gz

Then the untar

```
mkdir rootfs
sudo tar xf rootfs.tar.gz -C rootfs
```

Change root

~~man 1 croute~~

man 1 chroot

Which is a wrapper around pivot-root

But wait

What's a container?

But wait

What's a container?

For the sake of this course, we will say that it is just an isolated filesystem.

It is so much more in reality, be we won't dive into that. Let's say we're gonna take a look under the hood of the containers.

Demo

It's demo time!

```
sudo chroot rootfs /bin/sh
```

The executed sh is the one inside the downloaded rootfs

Inside our container

Inside the container

```
# some commands...
ps
```

Wait a minute

Wait… why is ps failing?

ps is a parsing tool going to some files from /proc and /sys

/proc/ and /sys/ are virtual fs

See them as kernel's APIs exposing information

So. . . can we access to the parent filesystem?

In the parent filesystem

```
vlc&
# Get the processus ID printed immediatly
```

In the container

```
kill VLC_PROCESSUS_ID
```

ISOLATION IS NOT WORKING OMG ALERT

You just killed a process from a container... Wait what?

Don't worry, there's a simple (and historical) explanation

initramfs

But how does a rootfs get mounted at first on the boot?

initramfs (a very tiny OS) loads the drivers then pivots into the real rootfs

And that's it. The tool was only used to this, so there was not a need to this kind of security. Now the devs should either recode it all or just configure it

Anyway, let's go back to our rootfs

Using the namespaces(7)

Because we inherited the father's namespace

Let's isolate our rootfs!

Syscalls, again

- clone(2)
- setns(2)
- unshare(2)

Jules Aubert presents: Fun with syscalls

clone(2) is actually the syscall hidden behind fork(2)

Let’s go back to our business

So, about those wrappers. . .

man 1 unshare

All right chums let's do this

```
sudo mount -t proc /proc rootfs/proc
unshare -f -p -n --mount-proc=rootfs/proc/ chroot rootfs /bin/sh
```

- -f = fork the executed program, chroot here
- -p = unshare the PID namespace
- -n = unshare the network namespace ; if you don't add it, you can see your network interfaces with ip(1), but if you add it, you won't see your network interfaces

About ps

Inside the container

```
ps
...
kill ANY_PARENT_PROCESSUS_ID
```

(Hey... /bin/sh has now the '1' pid and we can not kill parent's process... We're isolated!)

cgroup

Back to /sys (the directory, not the class, you fools)

By the way, replace $MYCGROUP with any name, it's up to you to name your own cgroup

```
sudo mkdir /sys/fs/cgroup/$MYCGROUP
cd /sys/fs/cgroup/$MYCGROUP
ls
```

It populated itself

Let's play a bit. . .

cgroup - Give it a try

Let's limit a processus max memory to 100MB

```
echo $((100 * 1024 * 1024)) | sudo tee memory.max
```

cgroup - Calling Python2

```python
#!/usr/bin/env python2

f = open("/dev/urandom", "r")
data = ""
i=0

while True:
    data += str(f.read(10000000))
    i += 1
    print '%dmb' % (i*10,)
```

cgroup - Calling Python3

```
#!/usr/bin/env python3

f = open("/dev/urandom", "rb")
data = ""
i=0

while True:
    data += str(f.read(10000000))
    i += 1
    print(f"{i*10}mb")
```

Ce système est trop petit pour nous deux

On shell number 1 that will executes the awesome Python script

```
echo $$
```

That is the shell PID

cgroup - Limit the ram to shell number 1

On shell number 2, using cgroup files.

```
echo PID_SHELL_1 | sudo tee cgroup.procs
```

or

```
sudo cgexec -g memory:$MYCGROUP ./memory.py
```

Now shell number 1 has a limited ram amount when executing

cgroup - Back to shell 1

Execute the Python script from shell number one... wait, what's hapenning? THE SCRIPT IS STILL EXECUTING! Y0U L1E T0 ME !!!!!!11!!1

Well no... the script is actually swapping its limited memory into a file.

If you look closely, there's a display slowdown when getting to 100MB

cgroup - Back to /sys/fs/cgroup/$MYCGROUP

On shell number 2

```
echo $((10 * 1024 * 1024)) | sudo tee memory.swap.max
```

That is 10MB max to be swapped, no more

Execute the Python script again

It works! The process receives a segfault when getting higher the limited memory

Docker = unshare and chrooting

With a bit of parsing on config files

Ok maybe not totally that, but anyway, you now got a more precise idea of what's going on with LXC, Docker, and other container solutions

T’es même pas cap!

man 7 capabilities

root is not superuser... its capabilities make them superuser

Back to the server

```
./server 0.0.0.0 80
# error...
sudo setcap cap_net_bind_service+ep ./server
./server 0.0.0.0 80
```

?

Questions ?