```
----------------     /------|                                ----------------
| Roll your    |    /       |                    |-------|   | 8d 35 fa ff  |
|  own chroot  |   /   /----| ------\       /----|--+ +--|   |       ff ff  |
|   container  |   |  /       | ---\ \/----\| /\ |  | |      | 83 c6 11     |
|              |   |  |  |-| || |  | || /\ || || |  | |      | 31 c9        |
|              |   |  \  | | || ---/ /| || || \/ |  | |      | 83 c1 13     |
|              |   \   \-| | || ---\ \| \/ |\----/  | |      | f3 a4        |
| for          |    \    | +-+| |  | |\----/        ---      | c3           |
|  - reversing |     \---| +-+---  ---                       | by Orion     |
|  - analysis  |         | | | |                             | <lawlor@     |
|  - archival  |         |-| |-|                             |  alaska.edu> |
----------------                                             ----------------
```

A chroot container lets you run a binary inside a custom-built filesystem, and is a good way to constrain code execution, and to understand how a binary actually runs.

UNIX's 'everything is a file' concept means modern file systems expose a huge attack surface with many suid executables, named pipes, and sensitive temp files. A chroot container denies this access by default, but isn't bloated like docker.

```
----------------------------------------------------------------------------
|Main UNIX Filesystem| Set up by your distribution, bloated with crap      |
----------------------                                                     |
|                                                                          |
| /                -------------------------------------------------------  |
| /bin             |chroot filesystem| Set up by you, tiny and light     | |
| /lib             -------------------                                   | |
| /etc             |                                                     | |
| /home/your/chroot| /                                                   | |
| /usr             | /bin    Only contains the utility programs you want | |
| /dev             | /lib    Shared libraries you decide to include      | |
| /proc            | /etc    Sanitized or customized config files        | |
| ...              -------------------------------------------------------  |
----------------------------------------------------------------------------
```

```
;;;;;;;;;;;
chroot.0: ; ---- ch'ing the root filesystem ----
```

Syntax: sudo chroot <path to new root directory> <command to run there>

Start by making a directory with the binary you want to run:

```
  $ mkdir -p /home/your/chroot
  $ cd /home/your/chroot
  $ mkdir bin
  $ cp /bin/bash bin/sh
```

The chroot command just takes the path to the new filesystem:

```
  $ sudo chroot /home/your/chroot /bin/sh
```

This will basically always fail with:

```
  chroot: failed to run command '/bin/sh': No such file or directory
```

If the binary exists, it's missing a shared library loaded by that binary. Check the shared libraries used with the ldd script:

```
  $ ldd bin/sh
```

The kernel provides linux-vdso.so.1, but you need to make everything else. This degree of shared library control can be very handy to run ancient binaries, or if you need to gdb a particular combo of lib versions without bricking your host system.

On a recent arm64 linux machine, I needed:

```
  $ mkdir lib
  $ cp /lib/ld-linux-aarch64.so.1 lib
  $ cp /lib/aarch64-linux-gnu/libtinfo.so.6 lib
  $ cp /lib/aarch64-linux-gnu/libc.so.6 lib
```

That's the dynamic linker ld-linux, ncurses, and the C standard library. We're dumping them all into lib/ wherever they came from.

On x86_64 linux, binaries have /lib64/ld-linux-x86-64.so.2 hardcoded, but will look for all their other libs in /lib.

Run your ld-linux .so with "--help" (it's a runnable ELF binary!) to get the full list of lib paths it will search in. (ldd is ld-linux.so --list).

Once the libraries are in place, try the chroot again:

```
  $ sudo chroot /home/your/chroot /bin/sh
bash-5.2# echo It Works
It Works
bash-5.2# ls
bash: ls: command not found
bash-5.2# echo *
bin lib
bash-5.2# cd bin
bash-5.2# echo *
sh
^D
```

Shell builtins work fine, like cd or echo or pwd, but not ls.

Let's fix that!

```
  $ cp /bin/ls bin/ls
  $ sudo chroot /home/your/chroot /bin/sh
sh-5.2# ls
ls: error while loading shared libraries: libselinux.so.1:
    cannot open shared object file: No such file or directory
```

ldd on bin/ls shows I need libselinux.so.1 and libpcre2-8.so.0, and then ls works ... ish?

```
sh-5.2# ls -l
total 8
drwxrwxr-x 2 1000 1000 4096 Dec 19 20:35 bin
drwxrwxr-x 2 1000 1000 4096 Dec 19 20:36 lib
             ^^^^ ^^^^
```

File owner and group are shown numerically, since we don't have an /etc yet.

;;;;;;;;;;;
chroot.1: ; ---- strace all the syscalls ----

Usually when a program misbehaves in a chroot, it's because it needs some random files, and the hard part is figuring out *which* files it wants where.

Syntax: strace <command to run>
Output: every kernel syscall made by that command as it runs

```
Let's use strace to watch exactly what syscalls `ls` makes in our chroot:
  $ cp /usr/bin/strace bin/
(Do the ldd dance to get strace running in the chroot)
  $ sudo chroot /home/your/chroot /bin/sh
sh-5.2# strace ls -l
execve("/bin/ls", ["ls", "-l"], 0xfffff79532c8 /* 17 vars */) = 0
... 100+ lines of shared libraries thrashing around ...
openat(AT_FDCWD, "/etc/passwd", O_RDONLY|O_CLOEXEC) = -1 ENOENT
   (No such file or directory)

Trapped in the huge spew of library bloat is the one file we need to add,
the famous /etc/passwd. We can just make up a username for this file:
  $ mkdir etc
  $ cat > etc/passwd
lol:x:1000:1000:never:/gonna/give/you:/bin/up
^D

Trying this from inside the chroot, our fake username works!
sh-5.2# ls -l
total 12
drwxrwxr-x 2 lol 1000 4096 Dec 19 20:50 bin
drwxrwxr-x 2 lol 1000 4096 Dec 19 20:56 etc
drwxrwxr-x 2 lol 1000 4096 Dec 19 20:36 lib
                 ^^^^
But the group is still listed numerically. Checking strace again, we see
another ENOENT when ls tries to open /etc/group, so we just make one:
  $ cat > etc/group
nope:x:1000:
^D
sh-5.2# ls -l
total 12
drwxrwxr-x 2 lol nope 4096 Dec 19 20:50 bin
drwxrwxr-x 2 lol nope 4096 Dec 19 23:24 etc
drwxrwxr-x 2 lol nope 4096 Dec 19 20:36 lib


Most programs don't check things very closely, so you can fake things
in /proc or /dev with just flat files: `echo predictable > dev/random`
will silently backdoor most crypto inside the chroot!

Some programs require access to /proc or /sys, so if you can tolerate the
attack surface you can just bind mount the real thing into your chroot:
  $ mount -o bind /dev dev
  $ mount -o bind /proc proc
  $ mount -o bind /sys sys
(But try faking it, it's more controllable and surprisingly effective.)


;;;;;;;;;;
chroot.2: ; ---- chroot jailbreak ----

In a complicated system chroot has a lot of escape opportunities:
   https://github.com/earthquake/chw00t

Everything the kernel touches except the filesystem is still accessible:
   - process lists and kill(), so `kill -9 -1` will still nuke the box
   - network access (the attacker is coming from 127.0.0.1 or ::1/128!)
   - device access (in the chroot, mknod /dev/sda and mount escape)
```

```
True container systems are quite an evolution from a basic chroot:
   - Podman or Docker or LXC isolate the network, PIDs, and cgroups
   - FreeBSD jails allow syscall translation and network isolation


;;;;;;;;;;;
chroot.3: ; ---- architectural chroot ----

A working chroot is a fully encapsulated system, with binaries and libs, so
you can move it between machines easily. An "architectural chroot"
can help you run binaries from other CPUs like x86/arm/risc-v/mips.

On modern linux, "sudo apt install qemu-user-static" makes chroot
automagically run binaries from any of the 34(!) supported architectures.

On older linux, you can register the ELF header and emulator into
/proc/sys/fs/binfmt_misc/register via a binary mask of the ELF bits.

Chroot is under-rated for cross-platform reversing and analysis: you can grab
an old MIPS or ARM32 firmware image, run its binaries in a chroot, and try
even GOT/PLT/ROP vulns using your desktop CPU but the old libs and binaries.
(Also useful for running your favorite old tool/game on new hardware!)


;;;;;;;;;;;
chroot.FF: ; ---- bonus challenges ----
Easy:
  - Build a chroot from one of your boxes.
  - Copy your chroot to another CPU arch (x86, arm64, risc-v) and run it.

Hard:
  - Use binwalk to pull a filesystem image from a firmware update file,
    and use (architectural) chroot to get it running on your machine.
  - Get a CUDA program running inside a chroot.
```

--[PREV | HOME | NEXT]--