jambit Abendvortrag – "Containers unplugged"

Docker unplugged

Michael Kerrisk, man7.org © 2019

mtk@man7.org

2 July 2019, Munich

Outline

Who am I?

- Contributor to Linux *man-pages* project since 2000
  - Maintainer since 2004
    - *https://www.kernel.org/doc/man-pages/contributing.html*
  - Project provides ≈1050 manual pages, primarily documenting system calls and C library functions
    - *https://www.kernel.org/doc/man-pages/*
- Author of a book on the Linux programming interface
  - *http://man7.org/tlpi/*
- Trainer/writer/engineer
  - Lots of courses at *http://man7.org/training/*
- Email: `mtk@man7.org`
  Twitter: `@mkerrisk`

Outline

Docker unplugged

- Let's explore the use of namespaces, capabilities, cgroups, and seccomp in *docker*
  - *https://docs.docker.com/engine/docker-overview/*, *https://docs.docker.com/get-started/*
- We'll look at the set-up when running *docker* as an unprivileged user

Some set-up: add unprivileged user to `docker` group

- Create `docker` group and place unprivileged user in that group, in order to run *docker* as unprivileged user:

```
$ sudo groupadd docker
$ sudo gpasswd -a $USER docker
$ id
uid=1000(mtk) gid=1000(mtk) groups=1000(mtk),10(wheel),
1001(docker)
```

  - This allows access to socket that is used to communicate with the *docker* daemon
    - `/var/run/docker.sock` (UNIX domain socket)
- After this, user will need to log out and in again, to get group ID
  - Or use *newgrp(1)* to start a shell that includes the group:

```
$ newgrp docker
```

Some set-up: create subuid and subgid entries

- Enable “userns-remap” for a (valid) username in docker daemon:

```
$ sudo sh -c 'cat >> /etc/docker/daemon.json' << EOF
{
    "userns-remap": "mtk"
}
EOF
```

- Restart the daemon, so the previous step takes effect:

```
$ sudo systemctl restart docker
```

Some set-up: create subuid and subgid entries

- The preceding steps created “subuid” + “subgid”:

```
$ cat /etc/subuid
mtk:100000:65536
$ cat /etc/subgid
mtk:100000:65536
```

  - See *subuid(5)* and *subgid(5)* manual pages
- These entries define the default UID and GID maps employed in user NS for *docker* containers run by unprivileged users on this system

Some set-up: install and demo *busybox* container image

- We'll use the *docker busybox* image for some experiments:

```
$ sudo docker pull busybox
```

- As a demo, run up a container that executes a single shell command and then terminates:

```
$ docker run busybox sh -c 'echo "hello world"'
hello world
```

  - See *https://docs.docker.com/engine/reference/run/*

- For info: `/bin` in busybox container image is mostly just symlinks to *busybox* binary:

```
$ docker run busybox ls -l bin |awk '{print $2, $5, $9}'
...
395 1120520 busybox
...
395 1120520 ls
...
395 1120520 sh
...
```

Some experiments: credentials and mappings

- What are PID and credentials of first process in container:

```
$ docker run busybox sh -c 'echo "$$"; id'
1
uid=0(root) gid=0(root) groups=10(wheel)
```

  - Shell has PID 1, and is running as UID 0

- What are the UID and GID mappings:

```
$ docker run busybox sh -c 'cat /proc/self/uid_map'
         0     100000      65536
$ docker run busybox sh -c 'cat /proc/self/gid_map'
         0     100000      65536
```

  - These mappings are derived from the “subuid” and “subgid” definitions

Outline

Some experiments: capabilities and namespaces

- Run a *sleep* process inside a container

```
$ docker run busybox sh -c 'sleep 1000'
```

- From shell in initial PID NS, discover PID of *sleep* process:

```
$ ps -C sleep
  PID TTY          TIME CMD
 5484 ?        00:00:00 sleep
```

- Next, we run commands to show capabilities and namespace relationships of the *sleep* process

Some experiments: capabilities

- Outside the container, view capabilities of *sleep* process:

```
$ grep Cap /proc/5484/status
CapInh: 00000000a80425fb
CapPrm: 00000000a80425fb
CapEff: 00000000a80425fb
CapBnd: 00000000a80425fb
CapAmb: 0000000000000000
```

- *getpcaps* provides more readable output:

```
$ getpcaps 5484
Capabilities for '14792': = 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+eip
```

- Less than half of all capabilities are available by default
  - Can override defaults with *--cap-add* and *--cap-drop* options of *docker run*

Docker and capabilities

- *docker* runs container with restricted capabilities, since it considers some capabilities to be unnecessary/risky
  - https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities
- Example omissions:
  - `CAP_SYS_ADMIN`: allows way too many things!
    - Notably, *mount* and *umount*; also *clone()* with most `CLONE_NEW*` flags
  - `CAP_SYS_MODULE`, `CAP_SYS_PACCT`, `CAP_SYS_TIME`: only meaningful in initial user NS
  - `CAP_SYS_RESOURCE`: override resource limits
  - `CAP_SYS_BOOT`: reboot the system

Some experiments: namespace hierarchy

- Next, we look at NSs of a shell in initial namespaces (PID 3452) vs *sleep* process inside container (PID 5484)
  - Using my `namespaces/namespaces_of.go` program

Some experiments: namespace hierarchy

```
$ sudo go run namespaces_of.go $$ 5484
user {3 4026531837} <UID: 0>
        [ 3452 ]
    cgroup {3 4026531835}
            [ 3452 5484 ]
    ipc {3 4026531839}
            [ 3452 ]
    mnt {3 4026531840}
            [ 3452 ]
    net {3 4026532000}
            [ 3452 ]
    pid {3 4026531836}
            [ 3452 ]
    uts {3 4026531838}
            [ 3452 ]
    user {3 4026533318} <UID: 0>
            [ 5484 ]
        ipc {3 4026533393}
                [ 5484 ]
        mnt {3 4026533319}
                [ 5484 ]
        net {3 4026533323}
                [ 5484 ]
        pid {3 4026533321}
                [ 5484 ]
        uts {3 4026533320}
                [ 5484 ]
```

Some experiments: namespace hierarchy

- From preceding, we see that process inside container is in a new user NS that owns new instances of all NS types (except cgroup NS)
  - Currently, Docker seems not to make use of cgroup NSs

Some experiments: mount points

- *sleep* process is in a noninitial mount NS, where it has distinct mount points for:
  - `/proc` (for PID NS of container)
  - `/dev/mqueue` (for IPC NS of container)
- Obtain container ID, and show it has separate `/proc/PID`:

```
$ docker container ls | awk '$2 == "busybox" {print $1}'
b9babde073e3
$ docker exec b9babde073e3 \
           sh -c 'grep Name /proc/[1-9]*/status'
/proc/1/status:Name:  sleep
/proc/8/status:Name:  grep
```

  - Two PIDs: initial process (*sleep*) and process running *grep*
    - *sleep* is PID 1 because *sh –c* (program first run by PID 1) did an *execve()* (without *fork()*) to replace itself with *sleep*
  - On the server side, *docker exec* uses *setns()* to enter NSs

Some experiments: PID namespaces

- Use `namespaces/namespaces_of.go` to show PID NSs of shell in initial PID NS and *sleep* process:

```
$ sudo go run namespaces_of.go --pidns $$ 5484
pid {3 4026531836}
        [ 3452 ]
    pid {3 4026533321}
            [ 5484 ]
```

  - When run with `--pidns` option, `namespaces_of.go` uses indentation to show hierarchical relationship of PID NSs
  - *sleep* is in noninitial PID NS that is a child of initial PID NS
- From a shell in initial PID NS, `/proc/PID/status` shows PID of *sleep* process in each PID NS:

```
$ grep NStgid /proc/5484/status      # NStgid shows PIDs
NStgid: 5484  1
```

Outline

Docker and cgroups

- *docker* uses control groups (cgroups) to control distribution of resources into container
  - *http://docs.docker.com/config/containers/resource_constraints/*
- Our previously created container has its own cgroup in each of the cgroup hierarchies:

```
$ cat /proc/5484/cgroup
11:blkio:/system.slice/docker-b9b...d38.scope
10:memory:/system.slice/docker-b9b...d38.scope
9:freezer:/system.slice/docker-b9b...d38.scope
...
1:name=systemd:/system.slice/docker-b9b...d38.scope
0::/system.slice/docker-b9b...d38.scope
```

  - "`b9b...d38`" is abbreviation of 64-char *docker* container ID

Some experiments: cgroups

- Create a container where we limit CPU bandwidth to 50%:

```
$ docker run --cpu-period=2000 --cpu-quota=1000 \
        busybox sh -c 'sleep 1000'
```

- Obtain container ID and verify settings inside container:

```
$ docker container ls | awk '$2 == "busybox" {print $1}'
d2202861cade
$ cd /sys/fs/cgroup/cpu,cpuacct/system.slice/
$ cat docker-d2202861cade*/cpu.cfs_period_us
2000
$ cat docker-d2202861cade*/cpu.cfs_quota_us
1000
```

- Copy (statically linked) *cpu_burner* into container & run it:

```
$ docker cp timers/cpu_burner d2202861cade:/cpu_burner
$ docker exec -it d2202861cade /cpu_burner
[73] 1: elapsed/cpu = 1.870; %CPU = 53.479
[73] 2: elapsed/cpu = 2.000; %CPU = 50.000
[73] 3: elapsed/cpu = 2.000; %CPU = 50.000
```

Outline

Docker and seccomp

- We can specify a seccomp profile to limit syscalls that can be made in a *docker* container
- There is a default seccomp profile that denies 40+ syscalls deemed risky/unnecessary in a container
  - `/etc/docker/seccomp.json`
  - *https://docs.docker.com/engine/security/seccomp/*
  - Generated filter is more than 900(!) BPF instructions
    - Precise set of permitted/excluded syscalls depends on container set-up (e.g., which capabilities are permitted inside container)
  - Example system calls excluded/constrained by default:
    - *settimeofday()*, *ptrace()*, *init_module()*, *keyctl()*, *setns()*
    - *clone()* can be used only with limited *flags*

Docker and seccomp

- Default seccomp profile can be suppressed or replaced with user-specified profile (specified as JSON file)
  - *docker run --security-opt=seccomp=unconfined*
  - *docker run --security-opt=seccomp=<profile>.json*

Some experiments: seccomp

- By default, the *unshare(2)* system call is allowed:

```
$ docker run busybox sh -c 'unshare -U echo hi'
hi
```

- Demonstrate the effect of a profile that denies *unshare(2)*:

```
$ cat deny_unshare.json
{ "defaultAction" : "SCMP_ACT_ALLOW",
  "syscalls": [
     { "name" : "unshare", "action" : "SCMP_ACT_KILL" }
  ]
}
$ docker run --security-opt=seccomp=deny_unshare.json \
             busybox sh -c 'unshare -U echo hi'
$ echo $?
159
```

  - No output from second *docker run* command because process it started was killed (by seccomp filter)
  - *$?* == 159 ⇒ process (looked like it) was killed by `SIGSYS`

Some experiments: seccomp

- Dump filter using my `seccomp/dump_seccomp_filter`:

```
$ docker run --security-opt=seccomp=deny_unshare.json \
             busybox sh -c 'sleep 500' &
$ sudo ./dump_seccomp_filter "$(pidof sleep)" bpf.blob
Dumped 8 BPF instructions
```

  - See `PTRACE_SECCOMP_GET_FILTER` in *ptrace(2)* man page

- Disassemble BPF to reveal treatment of *unshare()* syscall:

```
$ sudo sh -c "libseccomp/tools/scmp_bpf_disasm < bpf.blob"
line  OP   JT   JF   K
=================================
0000: 0x20 0x00 0x00 0x00000004 ld  $data[4]    # load architecture
0001: 0x15 0x00 0x05 0xc000003e jeq 3221225534 true:0002 false:0007
0002: 0x20 0x00 0x00 0x00000000 ld  $data[0]    # load syscall num
0003: 0x35 0x00 0x01 0x40000000 jge 1073741824 true:0004 false:0005
0004: 0x15 0x00 0x02 0xffffffff jeq 4294967295 true:0005 false:0007
0005: 0x15 0x01 0x00 0x00000110 jeq 272  true:0007 false:0006
0006: 0x06 0x00 0x00 0x7fff0000 ret ALLOW
0007: 0x06 0x00 0x00 0x00000000 ret KILL
$ grep unshare /usr/include/asm/unistd_64.h
#define __NR_unshare 272
```

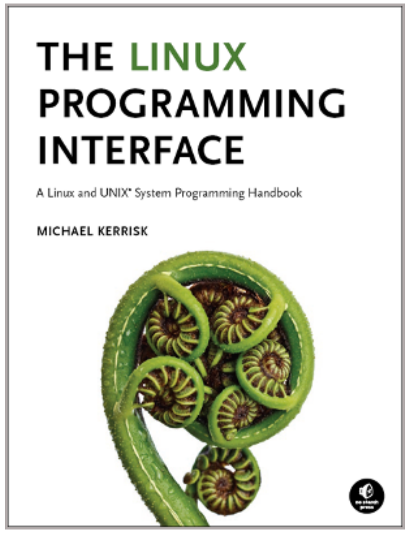
Thanks!

Michael Kerrisk   mtk@man7.org   @mkerrisk

Slides at http://man7.org/conf/
Source code at http://man7.org/tlpi/code/

Training: Linux system programming, security and isolation APIs, and more; http://man7.org/training/

The Linux Programming Interface, http://man7.org/tlpi/


THE LINUX PROGRAMMING INTERFACE
A Linux and UNIX® System Programming Handbook
MICHAEL KERRISK