Contents

1 Definitions

Or better a series of practical definitions that can help [McC18] while reading the documentation. Unfortunately they are blurry.

container is the runtime instantiation of a Container Image. A container is a standard Linux process typically created through a clone() system call instead of fork() or execvp(). Also, containers are often isolated further through the use of cgroups, SELinux or AppArmor.

container image in its simplest definition, is a file which is pulled down from a Registry Server and used locally as a mount point when starting Containers.

container engine is a piece of software that accepts user requests, including command line options, pullsimages, and from the end user's perspective runs the container. There are many container engines, including docker, podman, RKT, CRI-O[container runtime interface - OCI], LXD. Also, many cloud providers, Platforms as a Service (PaaS), and Container Platforms have their own built-in container engines which consume Docker or OCI compliant Container Images.

container runtime a lower level component typically used in a Container Engine but can also be used by hand for testing. The Open Containers Initiative (OCI) Runtime Standard reference implementation is runc. This is the most widely used container runtime, but there are others OCI compliant runtimes, such as crun, railcar, and katacontainers. Docker, CRI-O, and many other Container Engines rely on runc.

2 Namespace

> A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes. One use of namespaces is to implement containers. – `man 7 namespaces`

Every process has of a set of namespaces! But what can it do on them depends on the capabilities it has on that namespace

3 Linux capabilities

> For the purpose of performing permission checks, traditional UNIX implementations distinguish two categories of processes: privileged processes (whose effective user ID is 0, referred to as superuser or root), and unprivileged processes (whose effective UID is nonzero). Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials (usually: effective UID, effective GID, and supplementary group list).

> Starting with Linux 2.2 (1999), Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled. – `man capabilities`

How can a user perform a privileged operation? There are two possible ways: the old one, "setuid/setgid bits," and the new one, "capabilities."

3.1 setuid/setgit

If we do ls -la passwd we obtain something like: `-rwsr-xr-x. 1 root root 32760 Jan 19 2023 /usr/bin/passwd` where the x in the user line is substituted by an `s`. The consequence is that when the executable is run, the user is executing it as the file owner and not as the user running it.

What does this mean? Everything is fine if the code is written correctly and it performs all the necessary checks not to escalate user privileges. However, the user can perform privileged escalation if this is not the case.

The evident shortcoming of this approach is that while the user is running that code, it is getting all the privileges of the user who owns that executable, and this has no level of granularity. The solution to this problem is to assign to a given executable only a set of capabilities that will allow said program to act only on a specific resource. The kernel will then block the process if it asks for resources it can not access.

3.2 capabilities

Linux capabilities are per thread attributes

Technically they can be set at runtime but usually given an executable you can give it the capabilities to perform a given set of operations and when run it will automatically acquire those capabilities

(such capabilities are saved as extended attributes)

As of today there are 38 capabilities that can be assigned to 5 sets.:

To have a rough idea these 38 capabilities are responsible for controlling syscall

- `CAP_SYS_BOOT` allow rebooting
- `CAP_IPC_LOCK` allow memory allocation with huge pages via `mmap`

and so on.

Of the 5 sets we will focus on two of them:

- Effective: capabilities that a process has
- Permitted: capabilities that a process can ask to escalate, to my understanding you should only make effective those capabilities that you really need.

3.3 Why all this fuss about capabilities?

Because your user has no capabilities, but capabilities are tied to the **user** namespace, ergo if you are capable of generating a user namespace you can in principle ask for all the capabilities in such namespace. Spoiler alert: by design you are allowed to spawn user namespace as child of the current one.

For now let's define the baseline of privileges we have:

```
getpcaps $$
```

No capabilities in the current user shell.

```
getpcaps 1
```

pid 1 (the init system, systemd in this case) has all capabilities effective and permitted.

4 User namespace

Material: [Ove21b]

> User namespaces isolate security-related identifiers and attributes, in particular, user IDs and group IDs, the root directory, ..., and capabilities. A process's user and group IDs can be different inside and outside a user namespace. In particular, a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.
> – `man user_namespace`

The child process created ... in a new namespace... starts out with a complete set of capabilities in the new user namespace. A capability inside a user namespace permits a process to perform operations (that require privilege) only on resources governed by that namespace. In other words, having a capability in a user namespace permits a process to perform privileged operations on resources that are governed by (nonuser) namespaces owned by (associated with) the user namespace (see the next subsection). When a nonuser namespace is created, it is owned by the user namespace in which the creating process was a member at the time of the creation of the namespace. Privileged operations on resources governed by the nonuser namespace require that the process has the necessary capabilities in the user namespace that owns the nonuser namespace.

Each process is always associated with a set of namespaces; for example the current process has the following namespaces.

```
ls -An --time-style=+"" /proc/$$/ns
```

permission	uid	gid	file name		symlink
lrwxrwxrwx.	1000	1000	cgroup	->	cgroup:[4026531835]
lrwxrwxrwx.	1000	1000	ipc	->	ipc:[4026531839]
lrwxrwxrwx.	1000	1000	mnt	->	mnt:[4026531841]
lrwxrwxrwx.	1000	1000	net	->	net:[4026531840]
lrwxrwxrwx.	1000	1000	pid	->	pid:[4026531836]
lrwxrwxrwx.	1000	1000	$\text{pid}_{\text{forchildren}}$	->	pid:[4026531836]
lrwxrwxrwx.	1000	1000	time	->	time:[4026531834]
lrwxrwxrwx.	1000	1000	$\text{time}_{\text{forchildren}}$	->	time:[4026531834]
lrwxrwxrwx.	1000	1000	user	->	user:[4026531837]
lrwxrwxrwx.	1000	1000	uts	->	uts:[4026531838]

where the large numbers are inodes.

If we get out for our namespace:

```
unshare -U -f /usr/bin/whoami
unshare -U -f /bin/bash -c "getpcaps \$$"
```

but we can remap ourselves in root and obtain all capabilities in the child user namespace. Also the mapping of the user, it happens following the rules: `-r` map current user as root, new user are mapped to subIDs, if those are available.

```
unshare -Ur -f /usr/bin/whoami
unshare -Ur -f /bin/bash -c "getpcaps \$$"
# User mapping from outside to inside the namespace
unshare -Ur -f /bin/bash -c "cat /proc/\$$/uid_map"
```

So we escalated the capabilities, than what? can we do what we want? NO. Because we have inherited all the other namespaces from our parent namespace where we have no cap! For example:

```
unshare -Ur -f /usr/bin/whoami
unshare -Urp -f mount -t proc proc /tmp 2>&1

root
mount: /tmp: permission  denied.
dmesg(1) may have more information after failed mount system call.
```

This mark the specialty of this user namespace.

5 Mount namespace

Material: [Ove21a] [Ker13]

Mount namespace behaves differently form user namespace

In user namespace you always have a new empty one, where you might be mapped as root or nobody. But is new.

In the mount namespace you get a copy of the original namespace. This means that if you delete a file... well you delayed it!

Remember that you don't have more permissions than the process that spawned you.

shared subtree. Every mount point has its own propagation type associated (it is a metadata)

OK, so which is the use of namespace new line

The main point about rounding space is to isolate the list of mountpoints seen by a process in a name space new.

What is a mount point? mount points out there are listed in `/proc/self/mountinfo`

All files in Linux are organized hierarchically and are rooted at slash in a tree. '/' is mounted at boot time and is called root new line You can attach any file system if for example a USB port at any point in the tree structure.

Sometimes you want to share mount points, for example you want to share some file in more points and for that there is the instructional `mount -bind A B` that does the job, you cannot do it if you're not ruled, but if you're root the namespace you can do it.

There is also minus minus -rbind, that means recursive bind and propagate whatever you mount below the AB.

Why are you interested in it?

1. that it might happen that you need to change root folder and you want to bring some files with you?
2. You might want to mount a version file system, for example snapshot.
3. Or you have been hired by a tree data agency and you will need to provide all the users without custom user namespace or mount namespace to avoid them to see other people. And in that case Pam can help you. For example you can drop you when you open a new session into a new namespace dedicated for tha You can achieve whatever you want.

There are some flags that it is possible to use to modify mount point propagation:

shared A mount that belongs to a peer group. Any changes that occur will propagate through all members of the peer group.

slave One-way propagation. The master mount point will propagate events to a slave, but the master will not see any actions the slave takes.

shared and slave Indicates that the mount point has a master, but it also has its own peer group. The master will not be notified of changes to a mount point, but any peer group members downstream will.

private Does not receive or forward any propagation events.

unbindable Does not receive or forward any propagation events and cannot be bind mounted.

5.1 Hands on

5.1.1 Setup

```
$ mkdir -p fakeroot
$ wget https://dl-cdn.alpinelinux.org/alpine/v3.13/releases/x86_64/alpine-minirootfs-3.13.
$ tar xvf alpine-minirootfs-3.13.1-x86_64.tar.gz -C fakeroot
$ chown container-user. -R fakeroot
```

```
unshare -Urm -f /usr/bin/bash
# mount the fakeroot folder in the new mount namespace!
# --bind
mount --bind fakeroot fakeroot
mkdir old_root
pivot_root . old_root
PATH=/bin:/sbin:$PATH
# -l lazy
umount -l /old_root
```

Observation:

- `findmnt` command, inside the namespae find the fakeroot mount point, outside it doesn't
- after `pivot_root` have a complete different set of command
- we can not hide ourselves, run a `sleep` in the container and then `systemctl stautus user-$(id).service` in a terminal.
- proc is empty... so sad... we can not `mount -t proc proc /proc`, but we are sudo!!!! how is it possible? we ARE root! `whoami`

6 Pid namespace

- [Ove21c]

6.1 What are Process Ids?

All process in linux are identified by a Pid. command: `htop`.

All process are tracked in procfs. Proc filesystem is a pseudo-filesystem which provides an interface to kernel data structures. The standard want it mounted in the `/proc` folder, but as any file system it can be mounted everywhere. Do not expect standard tools to work if you don't mount it in `/proc`

6.2 The namespace

- There can not be 2 process with the same PID, unless they don't live in different pid name spaces
- a pid in a child namespace is always mapped in a pid the parent pid namespace.
- a new pid namespace start pid numbers from 1, one is a special pid (init process)
- you need this to:
  - convince a process to be an init system (systemd)
  - live migrate container between different machines.
- `pstree -p -N` (run on login)

- `pstree -N user` is limited since you can not access all the information on other user namespace

Running a pid namespace:

- run it after `unshare -Upmr -f /bin/bash`

Why fork (`-f`)? This is an idiosyncrasy of the API implementation.

- clone(), more general version of fork spawn a new process in the namespace. Fork does not have fine grain control over namespace creation.
- setns() -> move a process to a new namespace
- unshare() -> create the new namespace but does not place the caller in a new namespace

You can check the process tree by using `pstree -N pid -p`, if you don't fork bash is the calling process (caller), not unshare, and by construction it can not be put in the new namespace. If you fork everything is fine.

```
[4026531836]
# unshare is a process in the original namespace
bashunshare
[4026532698]
# bash is a process in the child namespace
bash
```

6.3 How to stop a process?

`kill -s signal pid`

Possible signal?

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup (*)
SIGINT	2	Term	Interrupt from keyboard
SIGKILL	9	Term	Kill signal
SIGTERM	15	Term	Termination signal
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process

- The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
- often SIGHP is used by daemon to reload config files!!

If you kill the pid 1, all process in the namespace are send sigkill!!

7 UTS

Material [Ove22a]

It govern the `hostname`, is it important... yes openssl certificates need this piece of information to be correct. We will see that we can solve this problem with an ingress, but never the less, setting an hostname (very important while handling authentication).

```
unshare -UrmpiuCn -f /bin/bash
mount --bind fakeroot fakeroot
pivot_root .
export PATH=/bin:/sbin:$PATH
mount -t proc proc /proc/
mount -t tmpfs tmpfs run/
hostname pippo
```

on another shell: `lsns` shows all the unhshare namespaces

8 IPC/Cgroup/time

8.1 IPC

> IPC namespaces isolate certain IPC resources, namely, System V IPC objects (see sysvipc(7)) and (since Linux 2.6.30) POSIX message queues (see $\text{mq}_{\text{overview}}(7)$). The common characteristic of these IPC mechanisms is that IPC objects are identified by mechanisms other than filesystem pathnames.

8.2 time

is to play with time, the only usage I see is to prank people about your uptime:

- `unshare -UrfT --boottime 2000000000 uptime`

8.3 Cgroup

Add a cgroup fs, very useful when live migrating containers, or having to convince a process that he is pid1 (systemd), or for security.

9 NET

Material [Ove22b, Ove22c]

> Network namespaces provide isolation of the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the /proc/net directory (which is a symbolic link to /proc/pid/net), the /sys/class/net directory, various files under `/proc/sys/net`, port numbers (sockets), and so on. In addition, network namespaces isolate the UNIX domain abstract socket namespace (see unix(7)). – `man 7 network_namespace`

9.1 Connecting two network namespaces

Two namespaces talking one another via a veth(virtual Ethernet) devices.

What is a veth? is a device that behaves for all purposes as an Ethernet connection, except that it is not physical and on the other side there is another veth. veth devices are always created in interconnected pairs. Packages injected on one side of this interface come out at the other side.

This is of little use if both sides reside on the same network namespace, but if different namespaces are involved is the de facto approach to move data.

The first step is to ask for a complete set of namespace `unshare -UinpmrC -f /bin/bash`, then before moving, on let's define a set of helpers variables:

```
namespace1=client
namespace2=server
ip_address1="10.10.10.10/24"
ip_address2='10.10.10.20/24'
interface1=veth-client
interface2=veth-server
command='python3 -m http.server 80'
```

Figure 1: Variables needed to simplify the exercise: the names of the two namespaces we will create are arbitrary, as are the names of the two veth interfaces. The IPs have been chosen in a specific subnet, 10.0.0.0/8, reserved for communications within a private network. Other common pools for private networks are 172.16.0.0/12 (commonly used for Calico) and 192.168.0.0/16 (commonly used at home). The command instead has a nice trick; it is the simplest HTTP server that serves the local folder but runs on port 80. Port 80 is the default port for the Internet (HTTP, to be more precise), and you can open it only if you are root.

In the exercise, we first mount the two folders `run` and `proc`. This operation substitutes the folders inherited by the father mount namespace with the `proc` Filesystem and the `run` Filesystem corresponding to the original `pid` namespace and mount namespace. `run` directory is designed to allow applications to store the data they require. Files in this directory include process IDs, socket information, lock files, and other data that is required at run-time but cannot be stored in `/tmp/` because programs such as `tmpwatch` could potentially delete it from there [crv11]. `proc` should not be strictly necessary. `run` is needed for `ip` to work and have the necessary privileges to create the files corresponding to the network namespaces we are creating.

The list of commands to achieve this result is presented in fig~2. This list of commands makes heavy use of the `ip` command. Such command is needed to show/manipulate routing, network devices, interfaces, and tunnels. It is one of the lowest levels available to manage network-related operations. In particular:

- `ip link` allows us to configure network devices
  - `add` add interfaces
  - `set` set a parameter of an interface
- `ip netns` allows us to operate with / in a network namespace

After creating the veth interfaces, before moving them to different namespaces we can observe with `ip a` the following output:

```
2: ptp-veth-server@ptp-veth-client: <BROADCAST,MULTICAST,M-DOWN>
    mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether d6:75:55:cc:c0:76 brd ff:ff:ff:ff:ff:ff
```

```
3: ptp-veth-client@ptp-veth-server: <BROADCAST,MULTICAST,M-DOWN>
    mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether f6:95:31:21:19:ea brd ff:ff:ff:ff:ff:ff
```

Let's decompose it:

- `#` the number represent the unique interface id.
- `ptp-veth-server@ptp-veth-client` is the interface name before the `@` and where it goes
- `<BROADCAST,MULTICAST,M-DOWN>` : This interface supports broad- and multicasting.
- `mtu 1500`: The maximum transfer unit this interface supports.
- `qdisc noop`: The scheduler is using a discipline called, none.
- state DOWN: The interface is not connected. To bring it up we must issue the `ip link set dev interface_name up` command on both sides of the cable.
- group default: This interface is in the "default" interface group.
- qlen 1000: The maximum length of the transmission queue.
- link/ether: The MAC address of the interface, and broadcast

An extra: the loopback interface:

```
1: lo: <LOOPBACK>
    mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

The `loopback` device is another virtual network interface, like `veth`. Its primary purpose is to allow internal communication with local services that can be hosted for diagnostics, troubleshooting, etc. A typical example is name resolution: on systems based on systemd usually the name resolution is performed by a plugin of systemd itself that runs a name server in a separate network namespace visible at IP 127.0.0.53. It is possible to test it and check that it indeed is like that by monitoring the traffic on the loopback interface `sudo tcpdump -i lo` and trying to query the DNS with a standard tool like `dig` as done in the following code where `dig` is used with two different servers. While monitoring the loopback interface, it can be observed that the first query will generate some traffic while the second one will not.

```
$ dig @127.0.0.53 google.it
.....
;; ANSWER SECTION:
google.it. 161 IN A 216.58.209.35

;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
.....
$ dig @8.8.8.8 google.it
.....
```

```
;; ANSWER SECTION:
google.it. 300 IN A 142.250.180.131
.....
```

Another common example is the `cups` server that behaves similarly.

```
# mount correct proc and run folders in our mount namespace
mount -t tmpfs tmpfs /run
mount -t proc proc /proc
# create the two name spaces
ip netns add $namespace1
ip netns add $namespace2
ip netns list
# check their status
ip netns exec $namespace1 ip ad
ip netns exec $namespace2 ip ad
# create the veth (and chek on it)
ip link add ptp-$interface1 type veth peer name ptp-$interface2
# (from)ptp-veth-client@ptp-veth-server(to)
ip link set ptp-$interface1 netns $namespace1
# (from)ptp-veth-client@ifX(to interface X)
ip link set ptp-$interface2 netns $namespace2
ip -all netns exec ip addr
# set static IPs
ip netns exec $namespace1 ip addr add $ip_address1 dev ptp-$interface1
ip netns exec $namespace2 ip addr add $ip_address2 dev ptp-$interface2
# bring interfaces up
ip netns exec $namespace1 ip link set dev ptp-$interface1 up
ip netns exec $namespace2 ip link set dev ptp-$interface2 up
```

Figure 2: First sequence of commands

Now that the set of connections are set up is possible to start our service in the secondary namespace. In this piece of code `curl` is used as a generic tool to get data from a server. On a practical prospective this should look like in fig 3

```
# start the server in background
ip netns exec $namespace2 $command &
# try to get some data out of it
# ok!
ip netns exec $namespace1 curl 10.10.10.20:80
# not ok!
ip netns exec $namespace1 curl 127.0.0.1:80
```

9.1.1 Exercise:

- Try to get data with curl from `google.com`
- You can not open port 80 on your PC without sudo, but you can do so on the virtual ethernet interface inside the network namespace we created. Can you access the http service from the Internet? If so, why?

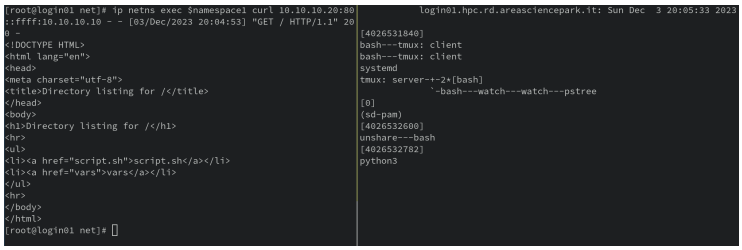


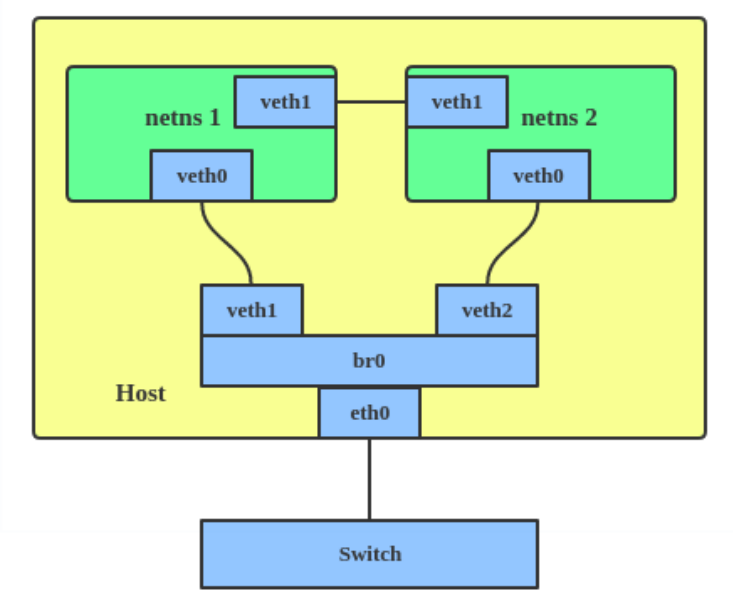
Figure 3: tmux session, on the left `pstree -N net` on the right the curl executed. From the left side we can observe how the `unshare` process lives in different naspace than the default one used by the regular user as doese the `python3` http server.

- Try the inception and run the same code inside a `podman` (`run --rm -it fedora`) container. Can you create the required namespaces? no? if so why? Tips:
  - try to check if a standard container gives you a user with all the capabilities or not. solve it with the following flags:
    - `--cap-add all --privileged`
  - small bug: `mount -t sysfs sysfs --make-private /sys`
  - extra software needed `iproute`, `procps`

9.2 Network interfaces

Source: [Liu22]

veth The VETH (virtual Ethernet) device is a local Ethernet tunnel. Devices are created in pairs, as shown in the diagram below. Packets transmitted on one device in the pair are immediately received on the other device. When either device is down, the link state of the pair is down.



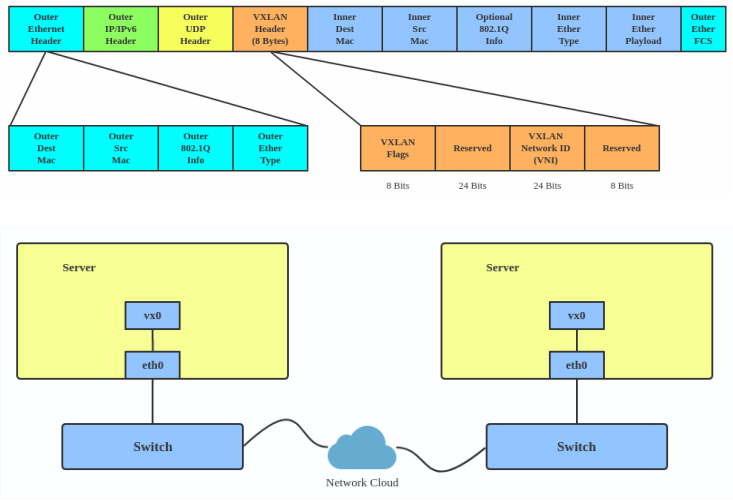
VLAN A VLAN, aka virtual LAN, separates broadcast domains by adding tags to network packets. VLANs allow network administrators to group hosts under the same switch or between different switches.

- pkgs
- topo

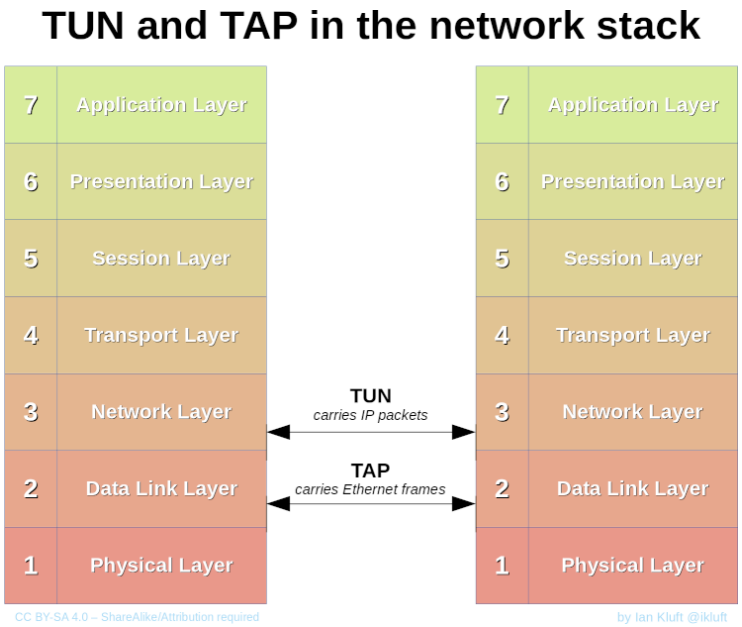
```
ip link add link eth0 name eth0.2 type vlan id 2
```

is an interface that inject tagged packets

VxLAN VXLAN encapsulates Layer 2 frames with a VXLAN header into a UDP-IP packet, which looks like this:



tun/tap In computer networking, TUN and TAP are kernel virtual network devices. Being network devices supported entirely in software, they differ from ordinary network devices which are backed by physical network adapters. On the other side of a tun/tap interface there is a software usually, not another port!!



9.3 Giving internet to the namespace

Getting access to the internet from a namespace without privileges is possible. However, with a caveat: We must mask the traffic as if a regular unprivileged user generated it. This masking is what is done by `podman`, and it achieves it with `slirp4netns`, a Swiss knife that generates a `tap` interface in the target networking namespace and intercepts all the output from that interface redirecting it to the regular connections slirp4netns repo.

To run it, first find a pid in the target namespace `pstree -N net -p`, then get a `tap0` running the main executable.

```
$ slirp4netns --configure \
    --mtu=65520 \
    $pid \
    --disable-host-loopback \
    tap0
```

From the output, we can infer that a nontrivial network infrastructure with a DNS, a DHCP, and a Gateway was generated for our namespace.

```
psent tapfd=5 for tap0
received tapfd=5
Starting slirp
MTU:             65520
```

```
Network:          10.0.2.0
Netmask:          255.255.255.0
Gateway:          10.0.2.2
DNS:              10.0.2.3
DHCP begin:       10.0.2.15
DHCP end:         10.0.2.30
Recommended IP:   10.0.2.100
```

This is the same done by podman as shown in figure 4.

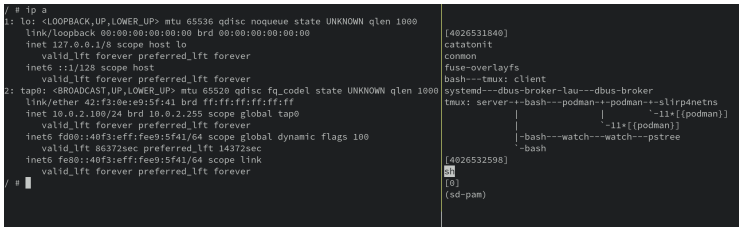


Figure 4: On the left is a shell within an Alpine container run with Podman; on the right is the process tree, structured by net namespace. Slirp4netns is running in the base net namespace as in the previous case, and in the container, we find a tap0 interface.

9.4 A complex exercise: adding a bridge

```
ping 8.8.8.8
# clean the routes
ip addr delete 10.0.2.100/24 dev tap0
# add a bridge
ip link add br0 type bridge
ip link set br0 up
# create some interfaces
ip link add int-a type veth peer name int-b
# connect to bridge
ip link set int-a master br0
ip link set int-a up
# add ip to secon interface
ip addr add 10.0.2.101/24 brd 10.0.2.255 dev int-b
ip link set int-b up
# add routing
ip route add default via 10.0.2.2
ping 8.8.8.8 # :)
```

- exercise: connect a sub namespace by using the bridge!

10 crun

Now that we have all the elements it is possible to substitute our `unashare` call with `crun`, the OCI runtime.

In the script, the complex command at line two can be decomposed as spawning a bash process that creates a new container ready for running (busy box, in this case, a standard container that contains a set of fundamental gnu tools) and exports the produced filesystem in a tar format. `tar` then decompress the filesystem. Finally, we generated a small configuration file for `crun`, and we run the container. Formally "run" performs a two-step process: first, create the container and then start it.

```
mkdir -p rootfs
bash -c 'podman export $(podman create busybox) | tar -C rootfs -xvf -'
crun spec --rootless
crun run cont1

# Change the executable of the container:
# sed -i 's/"sh"/"sh", "-c", "echo Hi, my PID is $$; sleep 10; echo Bye Bye"/' config.json
# sed -i 's/"terminal": true/"terminal": false/' config.json
```

Figure 5: How to crete and run a container with pure crun

The configuration file is a `json` where we can recognize some sections:

```
{
  "ociVersion": "1.0.0",
  "process": {
    "args": ["sh"],
    "env": [
      "PATH=/usr/sbin:/usr/bin:/sbin:/bin",
    ],
    "capabilities": {
      "effective": [
        "CAP_AUDIT_WRITE",
        "CAP_KILL",
        "CAP_NET_BIND_SERVICE"
      ],
    },
  },
  "root": {
    "path": "rootfs",
    "readonly": true
  },
  "mounts": [
    {
      "destination": "/proc",
      "type": "proc",
      "source": "proc"
    },
  ],
  "linux": {
    "namespaces": [
      {"type": "pid"},
      {"type": "network"},
```

```
      {"type": "ipc"},
      {"type": "uts"},
      {"type": "user"},
      {"type": "cgroup"},
      {"type": "mount"}
    ]
  }
}
```

`args` pass the executable, `env` the environment variable. `crun`, by default, does not propagate all the capabilities inside the container. The root folder (`/`) is mounted as read-only; we need some mechanisms to add files without ruining the base image. The layered file system will come into play to allow filesystem modifications without contaminating the base image. `crun` mounts the `proc` folder as done before by hand, and all the Linux namespaces are generated.

Notice that the runtime does not provide any networking other than the loopback interface, but it is possible to inject a `tap` interface as it was done before. This operation is usually done by other components like `podman` itself or `cri-o`

10.0.1 Exercise

- Inject a `tap` interface and connect your container to internet. Tip use `nslookup google.it 8.8.8.8` to verify the connection.

11 Conmon

Conmon is a shim, a small adapter that abstracts the container engine. It is used by podman and cri-o, and takes care of operations such as logs and container monitoring.

12 podman/cri-o

`podman` pod manager, takes care of managing pods and container lifecycle inside pods. It has an API compatible with docker and offers the same functionality of monitoring logging, and resource isolation. Network is handled in a simple enough manner that can be reproduced with little effort. It uses `conmon` for container monitoring and crun as the default runtime. `podman` also handles image lifecycle, from the build phase to the test and publication phase

In practice it makes the whole container managing work simpler.

A set of command that must be known:

command	description
build	Build an image using instructions from Containerfiles
image	Manage images
kill	Kill one or more running containers with a specific signal
network	Manage networks
pod	Manage pods
ps	List containers
rm	Remove one or more containers
rmi	Remove one or more images from local storage
run	Run a command in a new container

`cri-o` is only a Container Runtime Interface (container engine) Daemon. It does not offer any image lifecycle step other than download. Being a daemon to interact with it, we need an external program. `kubernetes` is one, but on a simpler level, `crictl` can be used even if there are more common ways to interact with `cri-o`. `crictl` has some commands that are very similar to Podman's ones.

command	description
images, image, img	List images
pods	List pods
rmp	Remove one or more pods
runp	Run a new pod
ps	List containers
rm	Remove one or more containers
rmi	Remove one or more images
run	Run a new container inside a sandbox

In order for `crictl` the `cri-o` daemon must be up and running, and such daemon must be run by root. The networking in this case is managed via a compatible CNI (discussion for another day) plugin. For reference `file:///etc/crio/` contains the `crio` configuration and `file:///etc/cni/` contains the network configuration.

Note: `cri-o` and `containerd` are container runtime daemons that show the same API that can be access thorough a socket. The fact that these APIs are similar allows `crictl` to work with both.

13 Overlay fs

An overlay filesystem combines two filesystems - an upper filesystem and a lower filesystem. When a name exists in both filesystems, the object in the upper filesystem is visible while the object in the lower filesystem is either hidden or, in the case of directories, merged with the upper object.

The lower filesystem can be any filesystem supported by Linux and does not need to be writable. The lower filesystem can even be another overlayfs. The upper filesystem will normally be writable and if it is it must support the creation of trusted.* extended attributes, and must provide a valid d_{type} in readdir responses, so NFS is not suitable.

A read-only overlay of two read-only filesystems may use any filesystem type. The options lowerdir and upperdir are combined into a merged directory by using:

```
mount -t overlay overlay\
      -olowerdir=/lower,upperdir=/upper,workdir=/work \
      /merged
```

Where:

`-t overlay` specify the overlay type

lowerdir is a directory from any filesystem, does not need to be on a writable filesystem.

upperdir is a directory, normally on a writable filesystem.

workdir an empty directory on the same filesystem as upperdir.

13.1 Exploring container image

By building a container with the following docker file:

```
FROM alpine:latest

RUN apk add curl
RUN apk add go

COPY ./testfile.txt /opt


ENTRYPOINT ["/bin/bash"]
```

Where an alpine base image is taken as a reference, 2 applications are installed a file is copied and an entry point is added. We obtain after building it with `podman build . -t my-alpine` a grand total of 4 images (output of `podman image list --all`:

REPOSITORY	TAG	IMAGE ID	SIZE	operation
localhost/my-alpine	latest	dad9e2237fec	332 MB	ENTRYPOINT
<none>	<none>	87e4cd263905	332 MB	COPY
<none>	<none>	916d6b9394c4	332 MB	RUN
<none>	<none>	b78840c89866	14 MB	RUN
remote/alpine	latest	8ca4688f4f35	7.63 MB	FROM

It is possible to investigate the size of all these images using dive, a tool created to inspect images. It is possible to see which folders are touched and how by every layer. It is essential to notice that changing a file in a layer preserves the lower layer value of the file, increasing the effective size of the container image. Some best practices are positioning changes only on top of the chain; in this way, fewer layers are affected by change.

Why is there a need for small images?

- Bandwidth Savings: moving data is expensive on scale
- Faster Deployment, especially in scenarios where there is the need to deploy thousands of pods

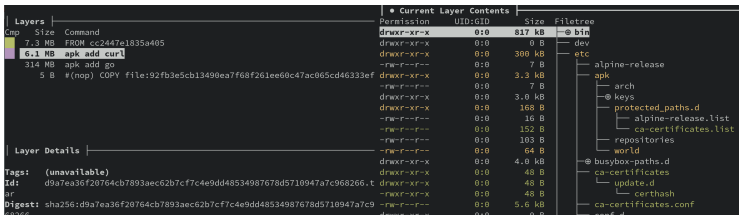


Figure 6: On the left, the different layers of the final `my-alpine` image with the occupied space. On the right, the changed files by the selected layer are in yellow, and the added files are in green. In this case, curl was installed, and some configuration files have been changed as expected. It is interesting to notice that certificates, probably those for HTTPS, have been added. Furthermore, the `apk` command updated its repositories, changing some files in the apk folder.

- Reduced Attack Surface, fewer applications, fewer vulnerabilities
- etc.

Why is there a need for larger images? Try to debug something without any tool. Jokes aside, small images are usually better, especially in large deployments.

When a container image is run, the base filesystem is read-only, and the container engine adds another temporary layer to make it writable. Once the container is deleted, the temporary filesystem is also removed; with it, all the data that was not saved elsewhere.

14 Podman

14.1 Two containers in a pod

As discussed, one of the main reasons we are working with Podman is that in it, a pod is a first-class citizen. Such behavior is only helpful if we understand where to draw the line between containers and pods. Pods are container containers; in technical terms, pods inside a container share a subset of kernel namespaces. It is possible to put more pods inside a container:

```
# get the list of pods
podman ps -a --pod
# create a pod
podman pod create my-pod
# place a container in the pod
podman run --rm -it  \
       --pod my-pod \
       alpine:latest
```

Can an empty pod exist? The answer is no; there is always a container inside a pod. Such a container is a brain-dead container running `catatonit -P` as it is possible to check in the process tree organized by namespace.

The kernel namespaces are shared according to the following table:

namespace	container in pod	pods
cgroup	no	no
mnt	no	no
pid	no	no
$\text{pid}_{\text{forchildren}}$	no	no
ipc	yes	no
net	yes	no
time	yes	yes
$\text{time}_{\text{forchildren}}$	yes	yes
user	yes	yes
uts	yes	no

It is possible to see separate containers in a pod have distinct `pid`, `mnt`, and `cgroup` namespaces. That is the minimum requirement for different operative systems in separate containers. On the other hand, they share the net namespace, making them share the same network stack. The last column reports the overlap between pods. It is unexpected to see that different containers share the same user namespace. This sharing is still in place because of security issues, but work is underway to use it proficiently [BN23].

It is possible to clean up the environment we created with the following command `podman pod rm my-pod` after deleting all the containers created inside the pod.

14.2 Communication between containers in pod

The most common container-to-container communications occur via UNIX or network socket. The latter is easily solved using the net namespace's loopback interface, but for the first, we need to share data between the mount namespaces of the two containers.

The solution to sharing data between containers is volumes. Volumes are filesystems mounted somewhere in the container where the application can store data for different purposes. Here, the emptyDir type of volume is used (or something that looks like an emptyDir volume: podman can generate emptyDir volumes on the fly, but we will do it later).

emptyDir volumes are scoped to the Pod's lifespan, but more containers can mount them simultaneously, providing an optimal facility for communication and data synchronization. Furthermore, since they survive a container restart due to a health-check failure, they can also be used for cache.

14.2.1 setup

For this operation it is possible to reuse the previously created pod.

To set up the volume:

```
podman volume create \
       --opt device=tmpfs \
       --opt type=tmpfs \
       my-vol
```

Then to start the two containers in the pod:

```
podman run -it --rm \
       --pod my-pod \
       -v my-vol:/workdir \
       -v .:/opt/:Z \
       alpine
podman run -it --rm \
       --pod my-pod \
       -v my-vol:/workdir \
       -v .:/opt/:Z \
       --entrypoint /bin/sh \
       python:3.11.6-alpine
```

To check the communications using the pod the following codes can be used:
Server:

```
import socket
import os

socket_name = "/workdir/application.socket"

s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)

try:
    os.remove(socket_name)
except OSError:
    pass

s.bind(socket_name)
s.listen(1)
conn, addr = s.accept()

while True:
    data = conn.recv(1024)
    if data:
        print('Recived form client: ', data)
        data = b'hello client!'
        conn.send(data)
    else:
        break

conn.close()
```

Client:

```
# Echo client program
import socket


socket_name = "/workdir/application.socket"

s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
```

```
s.connect(socket_name)

s.send(b'Hello, server!')
data = s.recv(1024)
s.close()

print('Received form server: ' + repr(data))
```

To check the standard localhost communication instead the http server can be used: `python3 -m http.server 8080`

14.3 Scripting everything

The basic repository to compile kuard

References

[BN23] Shay Berkovich and Arik Nemtsov. Enhancing kubernetes security with user namespaces, January 2023. [Online; accessed 4. Dec. 2023].

[crv11] crve. Linux distributions to include /run/ directory - the h open: News and features, March 2011. [Online; accessed 3. Dec. 2023].

[Ker13] Michael Kerrisk. Namespaces in operation. *lwn.net*, 2013. [Online; accessed 13. Nov. 2023].

[Liu22] Hangbin Liu. Introduction to linux interfaces for virtual networking, oct 2022. [Online; accessed 27. Oct. 2022].

[McC18] Scott McCarty. A practical introduction to container terminology, February 2018. [Online; accessed 5. Nov. 2023].

[Ove21a] Steve Ovens. Building a container by hand using namespaces: The mount namespace. *Enable Sysadmin*, March 2021.

[Ove21b] Steve Ovens. Building a linux container by hand using namespaces. *Enable Sysadmin*, March 2021.

[Ove21c] Steve Ovens. Building containers by hand: The pid namespace. *Enable Sysadmin*, April 2021.

[Ove22a] Steve Ovens. Building a container by hand using namespaces: The UTS namespace. *Enable Sysadmin*, January 2022.

[Ove22b] Steve Ovens. Building containers by hand using namespaces: The net namespace. *Enable Sysadmin*, March 2022.

[Ove22c] Steve Ovens. Building containers by hand using namespaces: Use a net namespace for VPNs. *Enable Sysadmin*, March 2022.