Copied From: https://book.hacktricks.xyz/linux-unix/privilege-escalation/docker-breakout/docker-privileged

## What Affects

When you run a container as privileged these are the protections you are disabling:

### Mount /dev

In a privileged container, all the **devices can be accessed in `/dev/`**. Therefore you can **escape** by **mounting** the disk of the host.
```
Inside default container
Inside Privileged Container
# docker run --rm -it alpine sh
ls /dev
console  fd       mqueue   ptmx     random   stderr   stdout   urandom
core     full     null     pts      shm      stdin    tty      zero

# docker run --rm --privileged -it alpine sh
ls /dev
cachefiles       mapper           port             shm              tty24            tty44            tty7
console          mem              psaux            stderr           tty25            tty45            tty8
core             mqueue           ptmx             stdin            tty26            tty46            tty9
cpu              nbd0             pts              stdout           tty27            tty47            ttyS0
```

### Read-only kernel file systems

Kernel file systems provide a mechanism for a **process to alter the way the kernel runs.** By default, we **don't want container processes to modify the kernel**, so we mount kernel file systems as read-only within the container.
```
Inside default container
Inside Privileged Container
# docker run --rm -it alpine sh
mount | grep 'ro'
sysfs on /sys type sysfs (ro,nosuid,nodev,noexec,relatime)
cpuset on /sys/fs/cgroup/cpuset type cgroup (ro,nosuid,nodev,noexec,relatime,cpuset)
cpu on /sys/fs/cgroup/cpu type cgroup (ro,nosuid,nodev,noexec,relatime,cpu)
cpuacct on /sys/fs/cgroup/cpuacct type cgroup (ro,nosuid,nodev,noexec,relatime,cpuacct)

# docker run --rm --privileged -it alpine sh
mount | grep '(ro'
```

### Masking over kernel file systems

The **/proc** file system is namespace-aware, and certain writes can be allowed, so we don't mount it read-only. However, specific directories in the /proc file system need to be **protected from writing**, and in some instances, **from reading**. In these cases, the container engines mount **tmpfs** file systems over potentially dangerous directories, preventing processes inside of the container from using them.

**tmpfs** is a file system that stores all the files in virtual memory. tmpfs doesn't create any files on your hard drive. So if you unmount a tmpfs file system, all the files residing in it are lost for ever.
```
Inside default container
Inside Privileged Container
# docker run --rm -it alpine sh
mount | grep /proc.*tmpfs
tmpfs on /proc/acpi type tmpfs (ro,relatime)
tmpfs on /proc/kcore type tmpfs (rw,nosuid,size=65536k,mode=755)
tmpfs on /proc/keys type tmpfs (rw,nosuid,size=65536k,mode=755)

# docker run --rm --privileged -it alpine sh
mount | grep /proc.*tmpfs
```

### Linux capabilities

Container engines launch the containers with a **limited number of capabilities** to control what goes on inside of the container by default. **Privileged** ones have **all** the **capabilities** accesible. To learn about capabilities read:
Linux Capabilities
```
Inside default container
Inside Privileged Container
# docker run --rm -it alpine sh
apk add -U libcap; capsh --print
Current: 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
Bounding set =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

# docker run --rm --privileged -it alpine sh
apk add -U libcap; capsh --print
Current: =eip cap_perfmon,cap_bpf,cap_checkpoint_restore-eip
Bounding set
=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_
```

You can manipulate the capabilities available to a container without running in `--privileged` mode by using the `--cap-add` and `--cap-drop` flags.

### Seccomp

**Seccomp** is useful to **limit** the **syscalls** a container can call. A default seccomp profile is enabled by default when running docker containers, but in privileged mode it is disabled. Learn more about Seccomp here:
Seccomp
```
Inside default container
Inside Privileged Container
# docker run --rm -it alpine sh
grep Seccomp /proc/1/status
Seccomp:        2
Seccomp_filters:        1

# docker run --rm --privileged -it alpine sh
grep Seccomp /proc/1/status
Seccomp:        0
Seccomp_filters:        0

# You can manually disable seccomp in docker with
--security-opt seccomp=unconfined
```

Also, note that when Docker (or other CRIs) are used in a **Kubernetes** cluster, the **seccomp filter is disabled by default**

### AppArmor

**AppArmor** is a kernel enhancement to confine **containers** to a **limited** set of **resources** with **per-program profiles**. When you run with the `--privileged` flag, this protection is disabled.
AppArmor
```
# You can manually disable seccomp in docker with
--security-opt apparmor=unconfined
```

### SELinux

When you run with the `--privileged` flag, **SELinux labels are disabled**, and the container runs with the **label that the container engine was executed with**. This label is usually `unconfined` and has **full access to the labels that the container engine does**. In rootless mode, the container runs with `container_runtime_t`. In root mode, it runs with `spc_t`.
SELinux
```
# You can manually disable selinux in docker with
--security-opt label:disable
```

## What Doesn't Affect

### Namespaces

Namespaces are **NOT affected** by the `--privileged` flag. Even though they don't have the security constraints enabled, they **do not see all of the processes on the system or the host network, for example**. Users can disable individual namespaces by using the **`--pid=host, --net=host, --ipc=host, --uts=host`** container engines flags.
```
Inside default privileged container
Inside --pid=host Container
# docker run --rm --privileged -it alpine sh
ps -ef
PID   USER     TIME  COMMAND
    1 root      0:00 sh
   18 root      0:00 ps -ef

# docker run --rm --privileged --pid=host -it alpine sh
ps -ef
PID   USER     TIME  COMMAND
    1 root      0:03 /sbin/init
    2 root      0:00 [kthreadd]
    3 root      0:00 [rcu_gp]ount | grep /proc.*tmpfs
```

### User namespace

Container engines do **NOT use user namespace by default**. However, rootless containers always use it to mount file systems and use more than a single UID. In the rootless case, user namespace can not be disabled; it is required to run rootless containers. User namespaces prevent certain privileges and add considerable security.

## References

- https://www.redhat.com/sysadmin/privileged-flag-container-engines

Abusing Docker Socket for Privilege Escalation