Container Security:

What Could Possibly Go Wrong?

Daniel Kouřil

egi CSIRT

cesnet

What is a container?

- fundamentally, a container is just **a running process** controlled by the host kernel
- it is **isolated** from the host and from other processes
- there are different containerization technologies available
  (Docker, Podman, Singularity, LXD, …)
  - in this tutorial, we will focus mainly on Docker but principles hold for other technologies

Docker Terminology

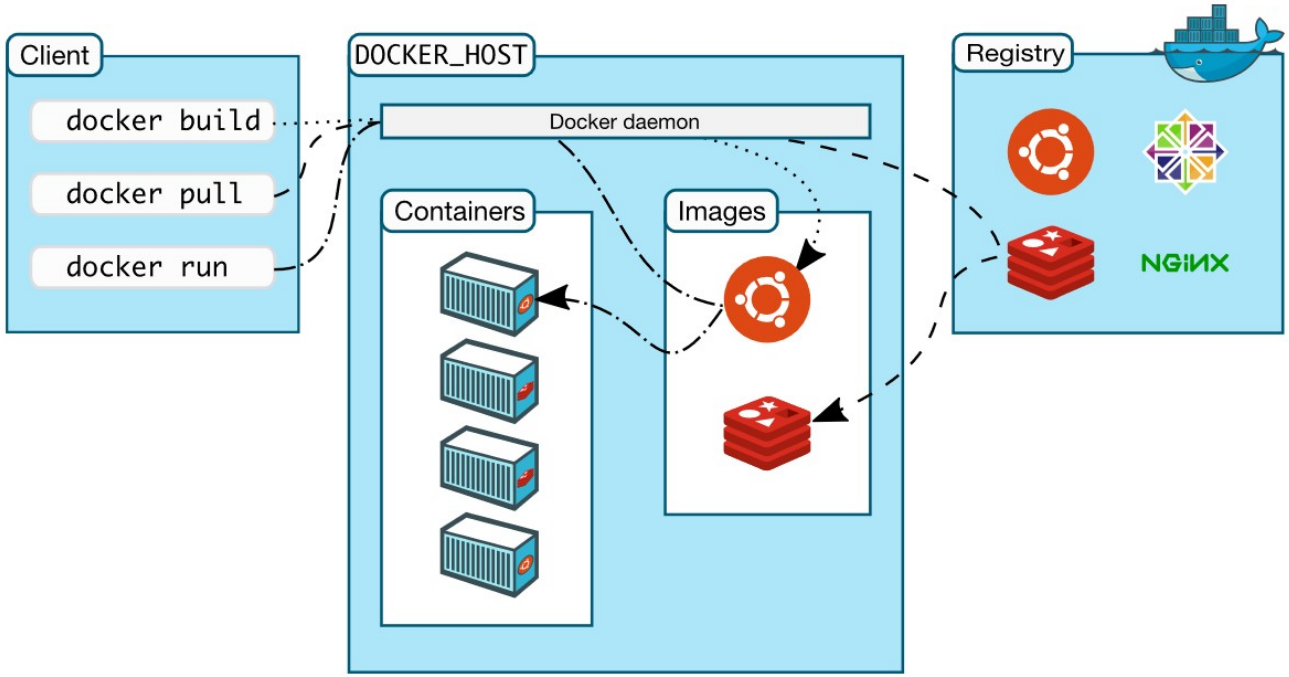
- **Docker container image** - a standalone package of files, which includes everything needed to run an application
  (code, runtime, system tools, system libraries and settings)

- an image is usually pulled from a **registry** to a host machine
  *(e.g. **DockerHub**)*

- a **Docker container** - a running instance of an image

- a host machine runs the **container engine** (**Docker Daemon**) and manages individual containers

Docker Architecture



https://docs.docker.com/get-started/overview/

Docker Container Creation

- the image is opened up and the **filesystem** of that image is copied into a **temporary directory structure** on the host
- Docker filesystem is a **stacked file system** of individual layers stacked on “mount”
- the ‘/’ root directory of the container is **mounted and available** on the host, e.g.:

  /var/lib/docker/overlay2/51415bc9cd3ab2c47d218a897516ea2bf0545595fadf4a167ed5cfd3230a5f99/

- changes to the directory **are visible** from both sides (host and container)
- when the container is removed, any changes to its state **disappear** unless “committed” via **dockerd**

```
host# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES
ba6db1f8ab7c        python:3.8          "bash"              11 days ago         Up 11 days          0.0.0.0:5000->5000/tcp   zen_wozniak
host#
host# docker exec ba6db1f8ab7c mount | grep ' / '
overlay on / type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay2/l/TL2SKWTCH7PPJ7BMWW4ZWFINPT:/var/lib/docker/overlay2/l/J7KGELYKIXB2CFDNJWL3KL
W7DB:/var/lib/docker/overlay2/l/4C5BPXBAGRAIMMKLA6T6JX6F5D:/var/lib/docker/overlay2/l/GK4LB4QKD6EQM3NHVYLV3XDIIB:/var/lib/docker/overlay2/l/7747UFBNYPA
TDJ5QKRFXOY2GPN:/var/lib/docker/overlay2/l/UKEBGZQU6VRQXAOIHYLSELSW3P:/var/lib/docker/overlay2/l/3C3IB3ANGSNZDBTLKVSNNRPT3S:/var/lib/docker/overlay2/l/
7HTG57SQVUSTQQOWQCIWNZB7TQ:/var/lib/docker/overlay2/l/FYP2GIR3DV2GQ77HFXI2IXVMRL:/var/lib/docker/overlay2/l/FJ7XGAOYKTVEBCD5ULRF53XF7S,upperdir=/var/li
b/docker/overlay2/60f4ebf44f92a37a28856a965a30f5bf5ac55f0f054222515ee8725ad7e2b2ce/diff,workdir=/var/lib/docker/overlay2/60f4ebf44f92a37a28856a965a30f5
bf5ac55f0f054222515ee8725ad7e2b2ce/work)
host# docker exec ba6db1f8ab7c ls /var/lib/docker/overlay2/60f4ebf44f92a37a28856a965a30f5bf5ac55f0f054222515ee8725ad7e2b2ce
ls: cannot access '/var/lib/docker/overlay2/60f4ebf44f92a37a28856a965a30f5bf5ac55f0f054222515ee8725ad7e2b2ce': No such file or directory
host#
host# mount | grep overlay
overlay on /var/lib/docker/overlay2/60f4ebf44f92a37a28856a965a30f5bf5ac55f0f054222515ee8725ad7e2b2ce/merged type overlay (rw,relatime,lowerdir=/var/lib
/docker/overlay2/l/TL2SKWTCH7PPJ7BMWW4ZWFINPT:/var/lib/docker/overlay2/l/J7KGELYKIXB2CFDNJWL3KLW7DB:/var/lib/docker/overlay2/l/4C5BPXBAGRAIMMKLA6T6JX6F
5D:/var/lib/docker/overlay2/l/GK4LB4QKD6EQM3NHVYLV3XDIIB:/var/lib/docker/overlay2/l/7747UFBNYPATDJ5QKRFXOY2GPN:/var/lib/docker/overlay2/l/UKEBGZQU6VRQX
AOIHYLSELSW3P:/var/lib/docker/overlay2/l/3C3IB3ANGSNZDBTLKVSNNRPT3S:/var/lib/docker/overlay2/l/7HTG57SQVUSTQQOWQCIWNZB7TQ:/var/lib/docker/overlay2/l/FY
P2GIR3DV2GQ77HFXI2IXVMRL:/var/lib/docker/overlay2/l/FJ7XGAOYKTVEBCD5ULRF53XF7S,upperdir=/var/lib/docker/overlay2/60f4ebf44f92a37a28856a965a30f5bf5ac55f
0f054222515ee8725ad7e2b2ce/diff,workdir=/var/lib/docker/overlay2/60f4ebf44f92a37a28856a965a30f5bf5ac55f0f054222515ee8725ad7e2b2ce/work)
host#
host# ls /var/lib/docker/overlay2/60f4ebf44f92a37a28856a965a30f5bf5ac55f0f054222515ee8725ad7e2b2ce/merged
bin  boot  data  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
host#
host# docker exec ba6db1f8ab7c touch /trace
host# docker exec ba6db1f8ab7c ls -lt /
total 84
drwxr-xr-x   1 root root 4096 May 28 16:36 bin
drwxr-xr-x   2 root root 4096 Mar 19 14:46 boot
drwxr-xr-x   3 1000 1000 4096 May 30 20:03 data
drwxr-xr-x   5 root root  360 May 28 16:33 dev
drwxr-xr-x   1 root root 4096 Jun  8 15:26 etc
drwxr-xr-x   2 root root 4096 Mar 19 14:46 home
drwxr-xr-x   1 root root 4096 May 11 03:50 lib
drwxr-xr-x   2 root root 4096 May  9 02:00 lib64
drwxr-xr-x   2 root root 4096 May  9 02:00 media
drwxr-xr-x   2 root root 4096 May  9 02:00 mnt
drwxr-xr-x   2 root root 4096 May  9 02:00 opt
dr-xr-xr-x 444 root root    0 May 28 16:33 proc
drwx------   1 root root 4096 Jun  8 13:53 root
drwxr-xr-x   3 root root 4096 May  9 02:00 run
drwxr-xr-x   1 root root 4096 May 28 16:36 sbin
drwxr-xr-x   2 root root 4096 May  9 02:00 srv
dr-xr-xr-x  13 root root    0 Jun  8 14:36 sys
drwxrwxrwt   1 root root 4096 May 28 17:17 tmp
-rw-r--r--   1 root root    0 Jun  8 16:47 trace
drwxr-xr-x   1 root root 4096 May  9 02:00 usr
drwxr-xr-x   1 root root 4096 May  9 02:00 var
host# date
Wed Jun  8 16:47:36 CEST 2022
host#
host# ls /var/lib/docker/overlay2/60f4ebf44f92a37a28856a965a30f5bf5ac55f0f054222515ee8725ad7e2b2ce/merged
bin  boot  data  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  trace  usr  var
host#
```

Starting Docker Container Processes

- the container processes are maintained **natively** via the host kernel
- to provide application sandboxing, Docker uses Linux **namespaces** and **cgroups**
- when you start a container with `docker run`, Docker creates **a set of namespaces** and **control groups**, which contain the process(es) started inside the container

Namespaces

- Docker Engine uses the following namespaces on Linux
  - **PID namespace** for process isolation
  - **NET namespace** for managing/separating network interfaces
  - **IPC namespace** for separating inter-process communication
  - **MNT namespace** for managing/separating filesystem mount points
  - **UTS namespace** for isolating kernel and version identifiers
    (mainly to set the hostname and domainname visible to the process)
  - **User ID** (user) namespace for privilege isolation
- user namespace **must be enabled** on purpose, it is **not** used by default

PID namespace

- allows the container to establish **separate process trees**
- the complete picture still **visible** from the **host** (running in the “system” namespace)

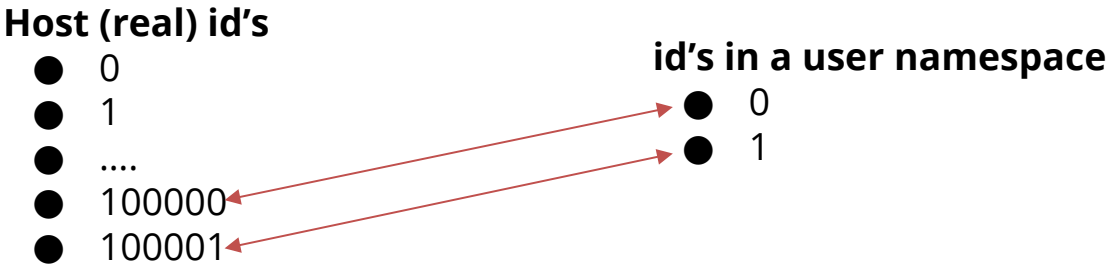
```
host# docker run -it debian bash
root@3146c2faec9b:/# dash
# ps af

  PID   TTY      STAT   TIME     COMMAND
    1   pts/0    Ss     0:00     bash
    6   pts/0    S      0:00     dash
    7   pts/0    R+     0:00      \_ ps af
```

Host displays all processes

```
 1029  ?        Ssl    7:48  /usr/bin/containerd
28834 ?        Sl     0:00  \_ containerd-shim -namespace moby  .....
28851 pts/0    Ss     0:00  \_ bash
28899 pts/0    S+     0:00          \_ dash
```

User ID (user) Namespace

- enables establishing **separated uid/gid allocations**, decoupled from real identifiers
  - A user process in a the namespace is assigned a ‘local’ identifier that is recognized only inside the namespace
- a **mapping** need to be maintained between uids/gids in the namespace and “global” (real) uids/gids



- by default, user namespace is not enabled by Docker, i.e. ***root* in the container is *root* in the host**

Cgroups I.

- short for **control groups**
- they allow Docker Engine to **maintain available system resources**
- they implement **resource limiting** for different resources (CPU, disk I/O, etc.)
- they help to ensure that a single container can be **assigned only limited resources**
- cgroups are organized in a (tree) **hierarchy** for a given cgroup type

Cgroups II.

- a process (thread) **may be assigned** one or more cgroup(s)
  - Management possible (e.g.) via the `/sys` pseudo-filesystem (`/sys/fs/cgroup`)
- Example how to set up a cgroup:

```
# create a specific cgroup:
mkdir /sys/fs/cgroup/memory/memory_eaters

# limit the memory usage to 10MB
echo 10000000 > /sys/fs/cgroup/memory/memory_eaters/memory.limit_in_bytes

# enter the new cgroup with the current shell  to apply to limit:
echo $$ > /sys/fs/cgroup/memory/memory_eaters/cgroup.procs
```

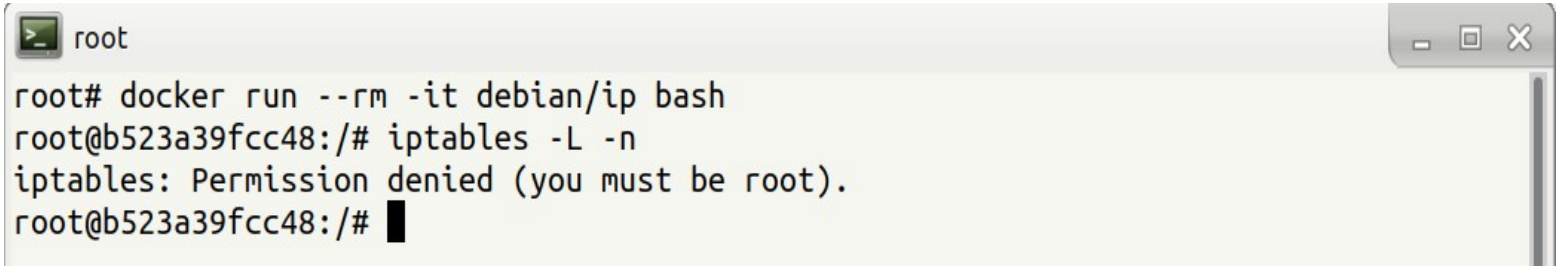
Linux Kernel Capabilities

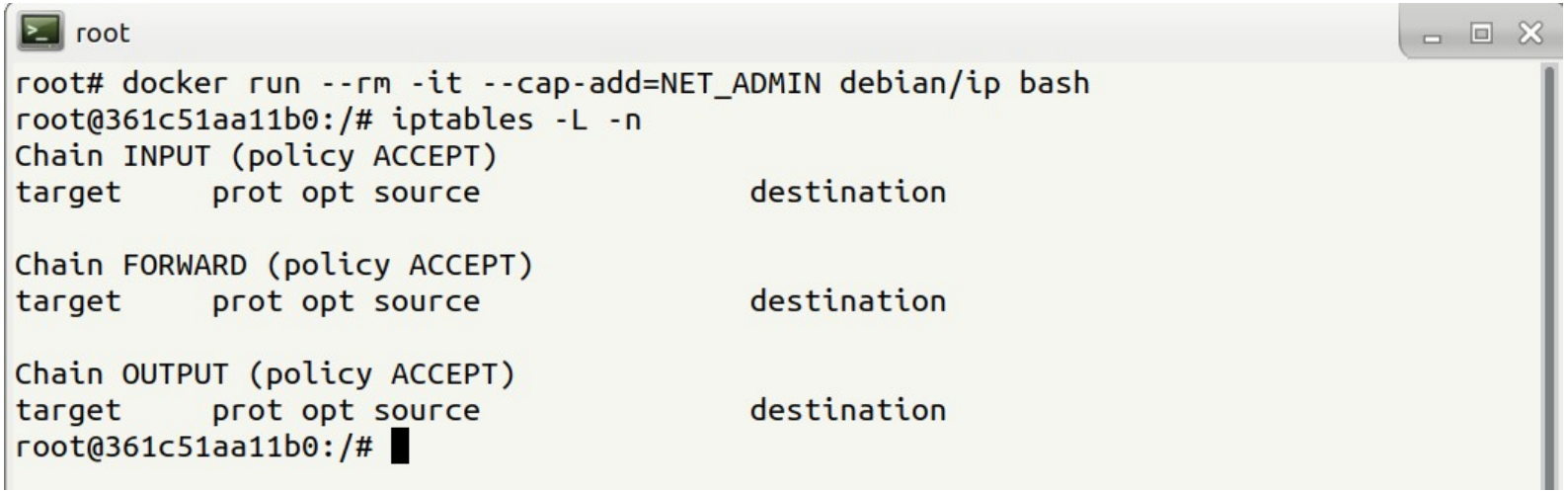
- capabilities turn the binary “root/non-root” dichotomy into a **fine-grained access control system**
- by default, Docker starts containers with **a restricted set of capabilities**
- Docker supports the **addition** and **removal** of capabilities
- additional capabilities extend the utility but have security implications, too
- a container started with **--privileged flag** obtains **all** capabilities
- running **without --privileged** doesn’t mean the container doesn’t have root privileges!

I am the root. Or not?

- multiple levels of elevated privileges, from an unprivileged user to full root rights:
  - if user namespace is **enabled**, the root inside a container has no root privileges outside in the host system
    - not available in default Docker setup
  - **by default**, the root in a container has some elevated privileges but restricted by a **set of capabilities**
  - we can **explicitly** add **extra capabilities** to a container on start
  - with the **--privileged flag**, we have full root rights granted

root may still be limited





Docker Daemon

- containers and images are maintained by Docker Daemon
- Docker daemon **needs full access** (administrative) to the system
  - Containers can be given all features they need
    - Internally (c.f. iptables before) or externally (port forwarding)
- Docker daemon **provides full access** (administrative) to the system
  - e.g., it allows you to share a directory between the Docker host and a guest container
    - we can start a container where the /host directory is the / directory on your host
- only **trusted users** should be allowed to control your Docker daemon
  - Compare with *root-less* container technology

Docker interface

- clients communicate with the daemon using an interface
- by default, Docker Daemon listens for requests at a unix domain socket created at **/var/run/docker.sock**
- it is possible to make the Docker Daemon listen on a network interface
- CLI clients have options to specify the endpoint (e.g. over the network)

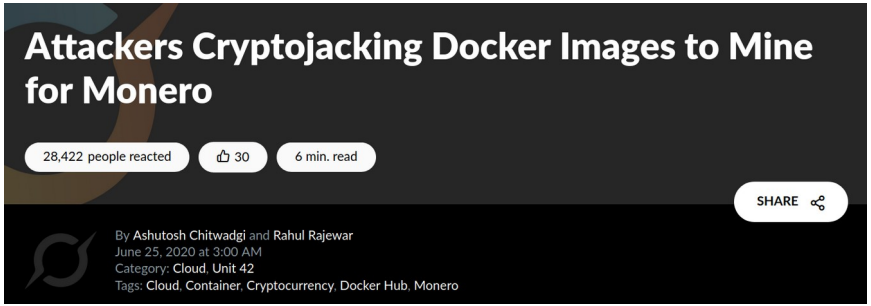
Docker vs. chroot command

- a container **isn't instantiated by the user** but the Docker daemon on behalf of the user
- anyone who is allowed to communicate with the Docker Daemon **can manage containers**
- that includes using any **configuration parameters**
  - they can play with binding/mounting files/directories
  - or decide which user id will be used in the container
  - it's not just better chroot

Container Security

Threat Landscape

- proper **deployment** and **configuration** requires understanding the technology
- **image management** (integrity and authenticity of the image)
- trust in the **image maintainer** and the **repository operator**
- **malicious images** may be found even in an official registry



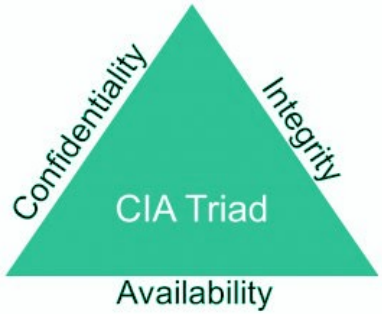
https://unit42.paloaltonetworks.com/cryptojacking-docker-images-for-mining-monero/

Usual Best Practice

- especially proper **vulnerability**/**patch management**
- it is often kernel-related and therefore requiring reboot
- proper patch management **extremely important** (couple of vulns over the past few years)
- out of scope for today

Escaping containers

- a **very general** term
- it does **not necessarily mean controlling the host** system
- severity is determined by the risks
- **data access** (according to the C.I.A triad):
  - Reading (violation of C.)
  - Modifying (violation of I.)
- **executing** code **outside** the container (assigned cgroups and namely namespaces)



Escaping with containers

- make use of technology to **bypass existing barriers**
  - e.g., mounting a directory to a container
- inject a “hook” that is invoked **by a trusted component** in the system
  - a crontab rule or a kernel “notifier” running command on certain events
  - must run outside the container - APIs (e.g. inotify) won’t help

Docker-related incidents

- **unprotected access** to Docker Daemon over the Internet
  - revealed by common Internet scans
  - instantiation of malicious containers used for dDoS activities
- **stolen credentials** providing access to the Docker Daemon
  - used to deploy a container set up in a way allowing breaking the isolation
  - the attackers escaped to the host system
  - an deployed crypto-mining software and misused the resources

Other kernel security features

- it is possible to **enhance Docker security** with systems like TOMOYO, AppArmor, SELinux, etc.
- you can also run the kernel with GRSEC and PAX
- all these extra security features require **extra configuration**

Cheat Sheets

Docker Cheat Sheet I.

start a new container
docker run IMAGE
docker run --rm IMAGE

start a new container in interactive mode (e.g. with a shell)
docker run -it IMAGE bash

start a new container from an image with a command
docker run IMAGE command

start a new container and map a local directory into the container
docker run -v HOSTDIR:TARGETDIR IMAGE

Docker Cheat Sheet II.

show a list of running containers
docker ps

show a list of all containers
docker ps -a

delete a container
docker rm CONTAINER

start a shell inside a running container
docker exec -it CONTAINER bash

stop a running container
docker stop CONTAINER

resume a stopped container
docker start CONTAINER

download an image from a repository
docker pull IMAGE

List local images
docker image ls

Delete a local image
docker image rm IMAGE

Practical Part