



Waterford Institute of Technology

NETWORK AND SYSTEM SECURITY

BACHELOR OF SCIENCE (HONS) APPLIED COMPUTING

---

# OCI Vulnerabilities and Best Practices

---

Ciaran ROCHE - 20037160

April 16, 2019

## **Plagiarism Declaration**

Unless otherwise stated all the work contained in this report is my own. I also state that I have not submitted this work for any other course of study leading to an academic award.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Description</b>	<b>4</b>
2.1	What is a Container . . . . .	4
2.2	Control Groups . . . . .	5
2.3	Namespaces . . . . .	5
2.4	Capabilities . . . . .	7
2.5	Open Container Initiative . . . . .	7
2.6	Docker . . . . .	7
2.7	Description Summary . . . . .	8
<b>3</b>	<b>Work Carried Out</b>	<b>9</b>
3.1	Docker Host and Kernel Security . . . . .	9
3.2	Docker Container Escape . . . . .	11
3.3	Docker Image Authenticity . . . . .	14
3.4	Docker Resource Abuse . . . . .	16
3.5	Docker Vulnerabilities in Static Images . . . . .	18
3.6	Docker Credentials and Secrets . . . . .	20
3.7	Container Auditing . . . . .	22
<b>4</b>	<b>Summary</b>	<b>25</b>
4.1	Immutable Containers . . . . .	25
4.2	Trusted Sources . . . . .	25
4.3	Role-based Access . . . . .	25
4.4	Host OS Precautions . . . . .	26
4.5	Automated Security Testing . . . . .	26
4.6	Utilize Monitoring Tools . . . . .	26
4.7	Utilize Orchestration Tools . . . . .	26
	<b>Appendices</b>	<b>27</b>
A	Moby Seccomp Defaults . . . . .	27
B	Docker Capabilities . . . . .	27
C	CIS Guide Extract . . . . .	28
	<b>Bibliography</b>	<b>29</b>

# 1 Introduction

Software development is ever changing, as we shift away from traditional monolithic Virtual Machine deployments to a more microservice container deployment the security aspects around applications change. Today applications consist of 'n' number of containers, this alone significantly enlarges the potential attack surface (Bryk, 2018).

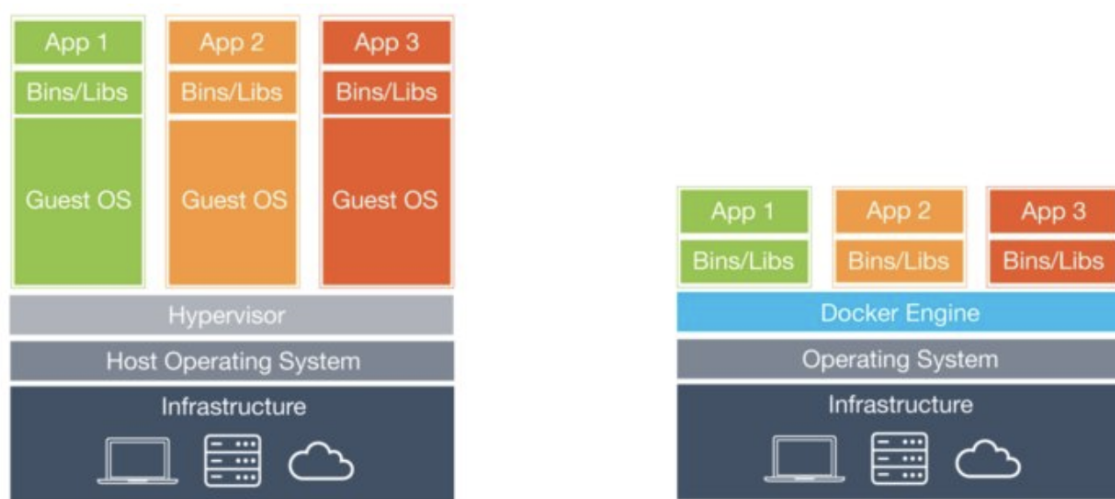
This paper looks toward this attack surface and tries to identify potential vulnerabilities and security challenges with Open Container Initiative compliant containers. Based on the findings from the report a guide will be constructed outlining a number of industry standard best practices when working with containers, this guide will be discussed in detail in Section 4.

## 2 Description

### 2.1 What is a Container

Containers are a modern version of capabilities that have been available in Unix operating system Solaris for decades, in that you can run processes in isolation to the rest of the operating system (*Oracle Solaris 11*, n.d.). When it comes to applications, traditionally one would have a virtual machine which would house the application. This would mean the applications processes would run in isolation to the operating system on its host but it does this with the cost of a massive over head in that of the entire virtual machines operating system consumers memory and resources (*Yegulalp, 2018*).

Containers are a means that looks at how to run applications and processes in isolation, and they do this by giving applications a partitioned segment of the hosts operating system. This is managed by the container run time environment, while there is many run time environments that support OCI containers for the majority of this paper the focus will be on Dockers run time called libcontainer (*What is a Container, 2018*).



**Figure 1:** *Docker Container vs Virtual Machine (image from Docker)*

Figure 1 paints a clear picture of the difference between a conventional system of applications running on a virtual machines (VMs) and applications running in containers. As can be seen from the figure, VMs are an abstraction of the physical hardware. The hypervisor allows multiple VM's run on a single hosts operating system. Each VM has is own operating system along with all the binaries, libraries and packages needed for the applications.

Where as in the Figure 1 it can be seen that containers are more of an abstraction at the application layer. As with VMs multiple containers can run on the same machine, but unlike VMs, containers share the OS kernel, all running in isolation based on a given space (*What is a Container*, 2018).

An understanding of cgroups and namespaces is needed to understand how containers share the OS kernel. It is these features that build the boundaries between containers and processes running on a host (Yegulalp, 2018).

## 2.2 Control Groups

Control Groups or Cgroups for short allow for the allocation of resources such as CPU time, system memory, network bandwidth or a combination of the resources. cgroups allow for the fine-grained control over allocating, prioritizing, denying and managing system resources. Management of these resources increase overall efficiency.

It must be noted that all processes on a Linux based system are child processes of a common parent. The *init* process, which is executed at boot time by the kernel starts all other processes. Thus the Linux process model is a single hierarchical tree.

Cgroups share similarities in that they are hierarchical and child cgroups inherit some attributes from their parent cgroup. This allows for simultaneous cgroups on a system. To paint a clearer picture if we describe the Linux process model as a single tree of processes, then the cgroup model is made up of one or many unconnected trees of processes (*Red Hat*, 2018).

## 2.3 Namespaces

Namespaces act similar to cgroups in that they deal with resource isolation, but unlike cgroups which deals with a number processes, Namespaces only isolate a single process. Linux Namespaces wrap a global system resource and makes it appear to the processes within its namespace that the global system resource is dedicated to that process (*Linux Manual*, n.d.a).

Furthermore these namespaces are broken into six different types, each namespace wraps a particular global resource. Due to this the overall nature of namespaces is to support the implementation of containers (Kerrisk, 2013).

### 2.3.1 UTS namespaces

In the space of containers, the UTS namespaces allow each container to have its own hostname and NIS domain name. This is used for the initialization and config scripts that take action based

on the namespace. The term UTS gets its name from the 'Unix Time-Sharing System" ([Kerrisk, 2013](#))

### **2.3.2 IPC namespaces**

IPC namespaces allow for its own interprocess communication resource. In other words it provides shared memory spaces for accelerated communication (POSIX message queue filesystem) ([Kerrisk, 2013](#)).

### **2.3.3 PID namespaces**

PID namespaces isolate the process ID number space. To put differently processes in different PID namespaces can have the same PID. The main benefit to this is that containers can be migrated between hosts and keep the same process IDs for all processes inside the container ([Kerrisk, 2013](#)).

### **2.3.4 Network namespaces**

Network namespaces provide isolation network resources in that each network namespace has its own network devices, IP routing tables, port numbers, IP addresses etc. Network namespaces make containers extremely powerful from a network perspective. In that you could have multiple containers on the same host all bound to port 80 in their own network namespace ([Kerrisk, 2013](#)). Given that each container has its own virtual networking, tools like Docker Swarm and Kubernetes utilize this and provide suitable networking rules on the host.

### **2.3.5 User namespaces**

User namespaces isolate the user and group ID number spaces. This means a process's user and group ID's can be different inside and outside of a user namespace. A typical use case would be a process would have full root privileges for operations within the user namespace but is unprivileged for operation outside the namespace ([Kerrisk, 2013](#)).

### **2.3.6 Mount namespaces**

Mount namespaces isolate filesystem mount points seen by a group of processes. This allows for processes in different mount namespaces to have different views of a filesystem hierarchy ([Kerrisk, 2013](#)).

## 2.4 Capabilities

As Linux capabilities are mentioned during the practical element in Section 3.2 it is worth providing some theory background here on what Linux capabilities are.

Traditional UNIX systems have two categories of processes, privileged and unprivileged. Privileged processes are identified as user ID 0, or also known as superuser or root. Whereas unprivileged processes have a UID of nonzero (*Linux Manual*, n.d.b).

## 2.5 Open Container Initiative

The Open Container Initiative was established in June 2015 by Docker, CoreOS and many other leaders in the container industry. The purpose of the initiative was to create open industry standards around container formats and runtime. Currently the OCI have two specifications, the runtime specification and the image specification (*OCI*, n.d.).

The purpose of these specifications is to ensure that all OCI compliant technologies are interoperable with each other, thus reducing the likelihood of a major company take over controlling the entire Container market.

Due to the interoperability of OCI compliant containers the rest of this paper will focus on Docker, but all topics covered are applicable to not just Docker but all OCI compliant technologies.

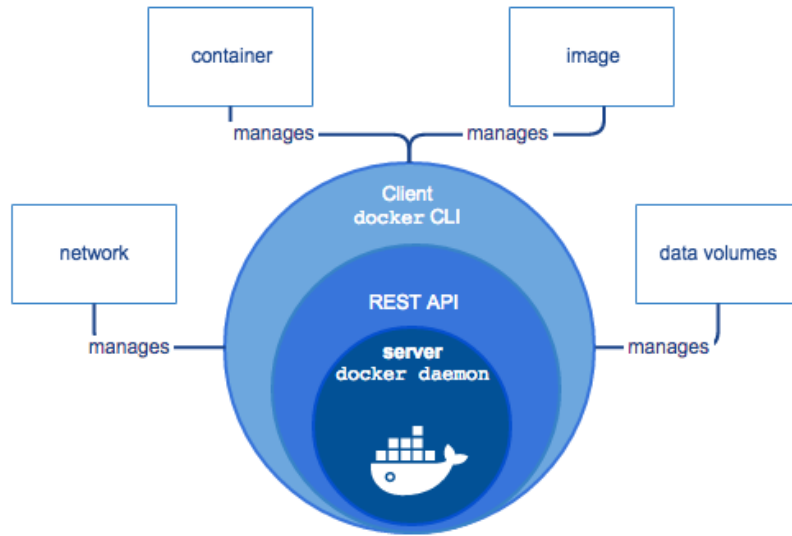
## 2.6 Docker

Docker's underlying architecture can be described as a client and server (docker daemon) based. This is shown in Figure 2, the server receives requests from the client through a RESTful API. The API along with a command line client are shipped by Docker. Due to the architecture it allows for the docker daemon and client to run on the same machine or alternatively the client can connect to a remote docker daemon (Rad et al., 2018). It is the Docker daemon which orchestrates and manages all containers and images on the host whereas the client is the control of the daemon.

Also depicted in Figure 2 is a Docker Image. A Docker image is essentially an immutable snapshot of a container. Running a Docker build command creates images. It is from this created image that a container is created by the Docker run command. Generally images are stored on a registry and will be demonstrated in Section 3.

Docker Hub would be a main source for storing Docker images, and it itself is a huge security risk, as the Hub allows anybody to push their images to be used by themselves or other people. A study carried out in 2016 showed out of 352,416 community images tested on Docker Hub the worst image contained almost 1,800 vulnerabilities, with of 3,892 official images tested the worst image contained





**Figure 2:** *Docker Architecture (image from Docker)*

almost 800 vulnerabilities (Colyer, 2017). The topic of Docker image authenticity is covered in Section 3.3.

## 2.7 Description Summary

So we have looked at what a container is and compared it to a traditional virtual machine. It was discussed how containers use control groups and namespaces to isolate processes from those of other containers and the host machine. We outlined what the Open Container Initiative is and discussed the architecture of Docker. Backed by the knowledge learned, we will look at a number of use cases based on a number of security services in Section 3.

### 3 Work Carried Out

The work carried out will look at container security threats with an emphasis on Docker. Each element will be divided into a number of sections. Out lining the threat description, what the attack could be and why it affects Docker. We will then look at the best practices in Docker to prevent this kind of security threats. Finally look at a reproducible proof of concept to showcase the threat.

Each element below will fall into one of the following security services, authentication, access control, data confidentiality, data integrity, non-repudiation and availability. Authentication is the correct identification of an entity or a source of data. Access control is who can access what and in what way. Data confidentiality is the non-disclosure to external parties. Data integrity is the correctness of data. Non-repudiation is the proof that communication actually took place. Finally availability is to ensure that a system is available when it is required.

Label 1 categorizes each element to a particular security service.

Security Service	Section
Authentication	3.2, 3.3, 3.6
Access Control	3.1
Data Confidentiality	3.6, 3.5
Data Integrity	3.2, 3.3
Non-Repudiation	3.6
Availability	3.4

**Table 1:** *Security Service Elements*

#### 3.1 Docker Host and Kernel Security

##### 3.1.1 Description

If a host system is compromised the container isolation outlined in Section 2.1 will not make any difference. From Figure 1 we saw that container run on top of the host kernel. This gives the advantages of being efficient over conventional virtual machines, but from a security perspective it can be seen as a risk.

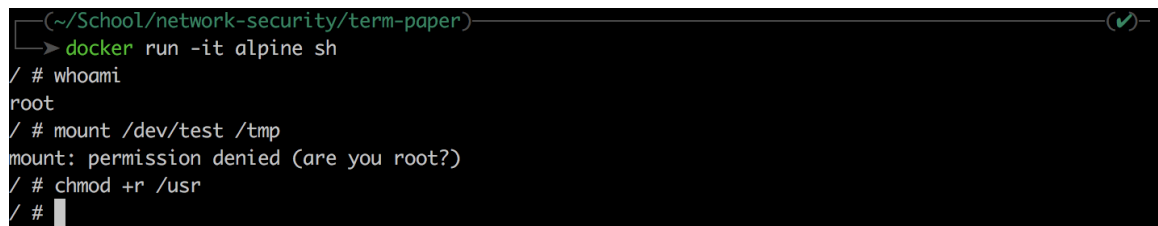
##### 3.1.2 Best Practices

Docker supply a bench auditing tool which will be discussed and demonstrated in Section 3.7.1 with a number of Open Source auditing tools available will also be discussed and demonstrated in Section 3.7.2. These tools as a whole check the configuration for best practices.

It is often the case with building containers to simply fire and forget. This is due the choice of base image which was discussed in Section 2.6 or even how Docker blocks certain behaviour on a container. It is often good practice to enforce your own Mandatory Access Control. This will resolve any unwanted operations, these controls can be placed at both the host and on the container at the kernel level and can be utilized from many tools including Seccomp, SELinux, and AppArmor. For the purpose of this paper we will look at using Seccomp. The profile used will be from Moby which is a Docker backed project and can be found at Appendix A. A simple rule of thumb for building containers if I don't need why have it there.

### 3.1.3 Proof of Concept

Seccomp can be thought of as a firewall for the kernel call interface. Regardless of this some calls are already blocked by default Docker profile:

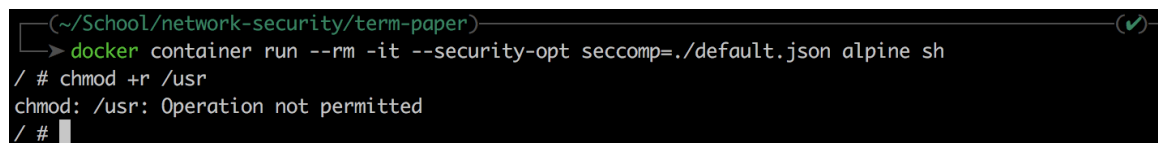


```
(~/School/network-security/term-paper)
> docker run -it alpine sh
/ # whoami
root
/ # mount /dev/test /tmp
mount: permission denied (are you root?)
/ # chmod +r /usr
/ #
```

Figure 3: Kernel Security Demo 1

As we can see from Figure 3 we have run a standard docker alpine image and started a bash session in the container. When we ran *whoami* we can see that we have *root* access. As can be seen Docker as standard blocks the *mount* call but does not block the *chmod* call. Allowing us to modify permissions on files and folders.

This can be rectified by applying our own Seccomp profile to the container. If we check Appendix A we can see a list of syscalls on line 52 which are whitelisted. If we remove *chmod* from the list and run our container with the *-security-opt* flag we can see we have successfully blocked the use of the *chmod* call.



```
(~/School/network-security/term-paper)
> docker container run --rm -it --security-opt seccomp=./default.json alpine sh
/ # chmod +r /usr
chmod: /usr: Operation not permitted
/ #
```

Figure 4: Kernel Security Demo 2

## 3.2 Docker Container Escape

### 3.2.1 Description

To escape or breakout from a container is the term used to when container isolation checks have been bypassed. Allowing an attacker to gain access to sensitive information from the host or gaining additional privileges. One of the most popular escapes in containers is the Dirty COW exploit CVE-2016-5195 (*Red Hat Portal*, 2016). The Dirty Cow is a race condition found in the way Linux kernel's memory subsystem handles the copy-on-write breakage of private read-only memory mapping. Allowing an unprivileged user to gain write access to read-only memory mappings. As this exploit is known it is often patched but the CVE affects all Linux kernels, thus enforcing the importance of maintaining an updated container.

Away from the exploit Docker daemon runs as root by default. It is good practice to create a user-level namespace or simply drop some of the container root capabilities.

### 3.2.2 Best Practices

As mentioned previous, if I don't need it why do I have it. So dropping capabilities that are not required by the software within the container is a must. For example, `CAP_SYS_ADMIN` grants a wide range of root level permissions. Micheal Kerrisk is quoted saying "CAP\_SYS\_ADMIN is the new root" (Kerrisk, 2012). Dropping these kind of capabilities is a must for a secure container.

As discussed in Section 2.4 docker container run as UID 0 to avoid this a user should be created in an isolated user namespace limiting the privileges over that of the host regular user.

If a you have to run a privileged container, always check that it is from a trusted source which is discussed in Section 3.3.

Always check your mount points from the host. For instance the Docker socket `/var/run/docker.sock`, `/proc`, `/dev` are special mounts that perform the containers core functionality. Insure you understand the why and the how to limit processes from gaining access to this privileged information. Sometimes it is enough just to expose the file system with read-only privileges.

### 3.2.3 Proof of Concept

Docker by default allows the root account to create device files, you should restrict this if your application does not require it. As you can see from Figure 5 we run an alpine image and can easily create a device file. Where as if we launch an alpine image with the flag `-cap-drop=MKNOD` the operation is not permitted. A root user will override any file permissions by default. It is good practise to restrict this in containers using different users which may contain mounts with sensitive

```
(~/School/network-security/term-paper)
➔ sudo docker run -it alpine sh
Password:
/ # mknod /dev/random3 c 1 8
/ # ls /dev
console  fd      mqueue  ptmx    random  shm      stdin   tty      zero
core     full    null     pts     random3 stderr   stdout  urandom
/ # exit
(~/School/network-security/term-paper)
➔ sudo docker run --rm -it --cap-drop=MKNOD alpine sh
/ # mknod /dev/random3 c 1 8
mknod: /dev/random3: Operation not permitted
/ # ls /dev
console  fd      mqueue  ptmx    random  stderr   stdout  urandom
core     full    null     pts     shm      stdin   tty      zero
/ #
```

Figure 5: Container Escape Demo 1

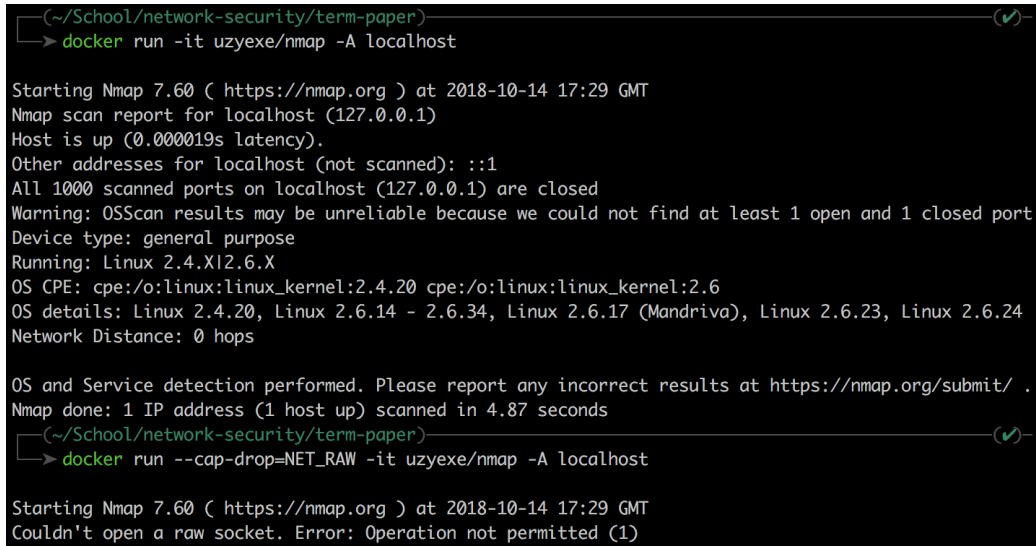
```
(~/School/network-security/term-paper)
➔ docker run -it ubuntu:14.04
root@5a8d859b88c4:/# useradd newuser
root@5a8d859b88c4:/# mkhomedir_helper newuser
root@5a8d859b88c4:/# su - newuser
newuser@5a8d859b88c4:~$ touch secret
newuser@5a8d859b88c4:~$ echo "secret info" > secret
newuser@5a8d859b88c4:~$ chmod 600 secret
newuser@5a8d859b88c4:~$ ls -l
total 4
-rw----- 1 newuser newuser 12 Oct 14 17:09 secret
newuser@5a8d859b88c4:~$ exit
logout
root@5a8d859b88c4:/# cat /home/newuser/secret
secret info
root@5a8d859b88c4:/# exit
exit
(~/School/network-security/term-paper)
➔ sudo docker run --rm -it --cap-drop=DAC_OVERRIDE ubuntu:14.04 sh
Password:
# useradd newuser
# mkhomedir_helper newuser
# su - newuser
newuser@cfd20bf61a8e:~$ touch secret
newuser@cfd20bf61a8e:~$ echo "secret info" > secret
newuser@cfd20bf61a8e:~$ chmod 600 secret
newuser@cfd20bf61a8e:~$ ls -l
total 4
-rw----- 1 newuser newuser 12 Oct 14 17:11 secret
newuser@cfd20bf61a8e:~$ exit
logout
# cat /home/newuser/secret
cat: /home/newuser/secret: Permission denied
# exit
```

Figure 6: Container Escape Demo 2

data. As can be seen from Figure 6 we launch a ubuntu image. Create a new user and user home directory. We create a file and pipe some text to it. We set the permissions to *600* so that only the

owner can read and write to the file. We exit back to root user and as can be seen root is able to *cat* the file.

Whereas when we launch a ubuntu image with the flag *-cap-drop=DAC\_OVERRIDE* if we complete the same steps the root user is denied from reading the file.



```
(~/School/network-security/term-paper)
➤ docker run -it uzyexe/nmap -A localhost

Starting Nmap 7.60 ( https://nmap.org ) at 2018-10-14 17:29 GMT
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000019s latency).
Other addresses for localhost (not scanned): ::1
All 1000 scanned ports on localhost (127.0.0.1) are closed
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
Device type: general purpose
Running: Linux 2.4.X12.6.X
OS CPE: cpe:/o:linux:linux_kernel:2.4.20 cpe:/o:linux:linux_kernel:2.6
OS details: Linux 2.4.20, Linux 2.6.14 - 2.6.34, Linux 2.6.17 (Mandriva), Linux 2.6.23, Linux 2.6.24
Network Distance: 0 hops

OS and Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 4.87 seconds

(~/School/network-security/term-paper)
➤ docker run --cap-drop=NET_RAW -it uzyexe/nmap -A localhost

Starting Nmap 7.60 ( https://nmap.org ) at 2018-10-14 17:29 GMT
Couldn't open a raw socket. Error: Operation not permitted (1)
```

**Figure 7:** *Container Escape Demo 3*

*Cap-drop* is an extremely powerful command, as if we refer to Appendix B we can see a number of Linux capabilities which are allowed by default and need to be manually dropped. For instance *NET\_RAW* is the capability to use RAW and PACKET sockets, this is common method for security scanners and malware tools.

As you can see from Figure 7 we launch an nmap container with no problems, if a container has been compromised an attacker first port of call may be to see what ports are open this is a serious vulnerability in the case if our application does not need the use of RAW and PACKET sockets, why have it as a capability in our container. Where as using the *CAP-DROP* flag we can also see from Figure 7 that the operation is not permitted.

## 3.3 Docker Image Authenticity

### 3.3.1 Description

Docker images are littered across repositories on the Internet. But if you are pulling images without using any trust and authenticity you could be running any kind of software on your host devices. A number of questions you should ask yourself before pulling an image.

- Where did the image come from?
- Do I trust the creator? What are their security policies?
- Is there any means to cryptographically prove that the author is who they say they are?
- How do I know no one has tampered with the image after I have pulled it

Docker allows you to pull and run any image from any source as standard. So even if you are using your own images you need to ensure no one else tampers with them. It often boils down to a Public Key chain of trust.

### 3.3.2 Best Practices

As with any piece of software it comes down to common sense, never run something that you do not explicitly trust the sources. To err on the side of caution it is always worth enforcing mandatory signature verification for any image that is to be pulled and or run on a host.

### 3.3.3 Proof of Concept

For this proof of concept it will show how to enable public key chain of trust on images.

If we refer to Figure 8 there is a lot going on there, so we shall break it down. We have created a project called *my-example* in it we have a *Dockerfile*. This *Dockerfile* just pulls an alpine image. Next we build this image and call it *muldoon/alpineunsigned* muldoon being my own Docker Hub user name.

Next we login to Docker using our Docker Hub credentials. Now we can push our unsigned image to our Docker Hub. Finally if we export *DOCKER\_CONTENT\_TRUST* and equal it to 1. This enables Docker trust enforcement. And as can be seen from Figure 8 if we try to pull the image we get trust error.

```

[~/School/network-security/term-paper/my-example] (✓)
> cat Dockerfile
FROM alpine:latest
[~/School/network-security/term-paper/my-example] (✓)
> docker build -t muldoon/alpineunsigned .
Sending build context to Docker daemon 2.048kB
Step 1/1 : FROM alpine:latest
--> 196d12cf6ab1
Successfully built 196d12cf6ab1
Successfully tagged muldoon/alpineunsigned:latest
[~/School/network-security/term-paper/my-example] (✓)
> docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: muldoon
Password:
Login Succeeded
[~/School/network-security/term-paper/my-example] (✓)
> docker push muldoon/alpineunsigned:latest
The push refers to repository [docker.io/muldoon/alpineunsigned]
df64d3292fd6: Mounted from library/alpine
latest: digest: sha256:02892826401a9d18f0ea01f8a2f35d328ef039db4e1edcc45c630314a0457d5b size: 528
[~/School/network-security/term-paper/my-example] (✓)
> export DOCKER_CONTENT_TRUST=1
[~/School/network-security/term-paper/my-example] (✓)
> docker pull muldoon/alpineunsigned
Using default tag: latest
Error: remote trust data does not exist for docker.io/muldoon/alpineunsigned: notary.docker.io does not have trust data for docker.io/muldoon/alpineunsigned

```

Figure 8: Docker Image Authenticity 1

Now if we build another image this time using the flag `--disable-content-trust` and set it to false it will build the container and sign it by default. This can be seen in Figure 9

```

[~/School/network-security/term-paper/my-example] (✗)
> docker build --disable-content-trust=false -t muldoon/alpine:latest .
Sending build context to Docker daemon 2.048kB
Step 1/1 : FROM alpine@sha256:621c2f39f8133acb8e64023a94dbdf0d5ca81896102b9e57c0dc184cadaf5528
--> 196d12cf6ab1
Successfully built 196d12cf6ab1
Successfully tagged muldoon/alpine:latest
Tagging alpine@sha256:621c2f39f8133acb8e64023a94dbdf0d5ca81896102b9e57c0dc184cadaf5528 as alpine:latest

```

Figure 9: Docker Image Authenticity 2

With our newly signed container built we can push the image to our Docker Hub. This time we get asked to create a two passphrases. One for our root key and the other for the repository. The root key or offline key as it is also known is only needed for the creation of new repositories associated with the account. And the repository key is used for the image we just pushed so that we can



identify it. This can be seen in Figure 10. With the image pushed to our Docker Hub it can be seen that we can pull the image with no trust errors.

```
(~/School/network-security/term-paper/my-example)
➤ docker push muldoon/alpine:latest
The push refers to repository [docker.io/muldoon/alpine]
df64d3292fd6: Mounted from muldoon/alpine:signed
latest: digest: sha256:02892826401a9d18f0ea01f8a2f35d328ef039db4e1edcc45c630314a0457d5b size: 528
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with ID e42be79:
Repeat passphrase for new root key with ID e42be79:
Enter passphrase for new repository key with ID bca5e36:
Repeat passphrase for new repository key with ID bca5e36:
Finished initializing "docker.io/muldoon/alpine"
Successfully signed docker.io/muldoon/alpine:latest
(~/School/network-security/term-paper/my-example)
➤ docker pull muldoon/alpine:latest
Pull (1 of 1): muldoon/alpine:latest@sha256:02892826401a9d18f0ea01f8a2f35d328ef039db4e1edcc45c630314a0457d5b
sha256:02892826401a9d18f0ea01f8a2f35d328ef039db4e1edcc45c630314a0457d5b: Pulling from muldoon/alpine
Digest: sha256:02892826401a9d18f0ea01f8a2f35d328ef039db4e1edcc45c630314a0457d5b
Status: Image is up to date for muldoon/alpine@sha256:02892826401a9d18f0ea01f8a2f35d328ef039db4e1edcc45c630314a0457d5b
Tagging muldoon/alpine@sha256:02892826401a9d18f0ea01f8a2f35d328ef039db4e1edcc45c630314a0457d5b as muldoon/alpine:latest
```

Figure 10: Docker Image Authenticity 3

## 3.4 Docker Resource Abuse

### 3.4.1 Description

As discussed in Section 2.1, containers are lightweight in comparison to Virtual Machines, this means we can deploy multiple containers on our hardware. But with lots of entities comes the fight for resources. Be it bugs in the applications or deliberate attacks a Denial of Service can easily happen consuming all the resources on a host leaving none for the containers.

### 3.4.2 Best Practices

Limits to resources are in fact disabled by default, so configuring of resources before deployment is a must. One method to limit resources is to utilize the resource limitation features that come with

the Linux kernel. Another method is to stress test containers as part of the CI/CD cycles. Load testing is crucial when it comes to knowing the physical limits to the operations of containers. And finally make use of tools for monitoring and alerting when resources are low.

### 3.4.3 Proof of Concept

For this proof of concept we will use cgroups that was discussed in Section 2.2 to limit resources to a container. We can do this simply by running the command seen in Figure 11. As we want to test the limitations of 2G memory has been set we need to install *stress* tool on the machine, so first need to run an *apt-get update*

```
(~/School/network-security/term-paper)
➤ docker run -it --memory=2G --memory-swap=3G ubuntu bash
root@6ea716a63f16:/# apt-get update
```

Figure 11: Docker Resource Abuse 1

With our ubuntu container updated, we can run the command seen in Figure 12 to install *stress*.

```
root@6ea716a63f16:/# apt-get install stress
```

Figure 12: Docker Resource Abuse 2

Once stress is installed we run the command seen in Figure 13 to stress test our container.

```
root@6ea716a63f16:/# stress --cpu 8 --io 4 --vm 4 --vm-bytes 8G --timeout 10s
stress: info: [260] dispatching hogs: 8 cpu, 4 io, 4 vm, 0 hdd
stress: FAIL: [260] (415) <-- worker 272 got signal 9
stress: WARN: [260] (417) now reaping child worker processes
stress: FAIL: [260] (451) failed run completed in 6s
root@6ea716a63f16:/#
```

Figure 13: Docker Resource Abuse 3

As we have set the limits to 2GB main memory, with 3GB in total to include *swap*. Swapping is a technique that allows a computer to execute programs and manipulate files that are larger than main memory (*CentOS*, n.d.). We can see from Figure 13 that we are stressing the container with over 8GB of memory usage. As can be seen this fails. If we run *Docker Stats* in a separate terminal on the host machine, we can see the container usage spike before the process is killed. As the process is looking for more resources then the cgroup will allow the process is killed.

## 3.5 Docker Vulnerabilities in Static Images

### 3.5.1 Description

As containers are essentially isolated, if they are acting as expected it is easy to forget the underlying architectures and dependencies. As vulnerabilities are found daily it is important to have the appropriate measures in place to test containers to ensure any security flaws are patched and updated as soon as possible.

### 3.5.2 Best Practices

A simple but not fully effective solution would be to rebuild and update images periodically to ensure the latest patches are applied. This does have the overhead of re-testing your containers to ensure no patches have broken the application.

While live-patching containers is considered a bad practice, Docker and tools like Kubernetes often provide the means to roll out updates without disrupting uptime of the application.

As said many times throughout this paper, if you do not need it why have it. This applies here as keeping a container simple and minimal will reduce the attack surface and also reduce the need for frequent updates.

The number one approach is to provide some sort of vulnerability scanner. This will be looked at in detail in Section 3.7.2 and Section 3.7.1. These tools check containers for all known vulnerabilities and can be incorporated in CI/CD.

### 3.5.3 Proof of Concept

As this paper looks at auditing tools for vulnerabilities in images in a later section, we will look at using a different registry service to Docker Hub.

CoreOS Quay repository uses the open source security image scanner Clair. Despite being aimed for commercial use a lot of Quay's services are free to use. With an account set up on Quay.io it is easy to incorporate it into a workflow over using the conventional *Docker push* which we saw earlier in the paper.

As can be seen from Figure 14 we have an image called *muldoon/test-server:latest* running. We just need an extra step over directly pushing it to a registry as we do with Docker Hub, this time we need to commit it to a repository that we have created on Quay. In this case the repository that was created was called *quay.io/ciaranroche/test-image*. The steps in creating the repository have been left out for brevity.

With the image commit to the repository it is only a matter of pushing to the repository. A similar pattern seen from the likes of GitHub.

```

(~)
> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
5eafeee3c5a7       muldoon/test-server:latest "node app.js"      12 seconds ago     Up 11 second
s                   8000/tcp           tender_clarke

(~)
> docker commit 5eafeee3c5a7 quay.io/ciaranroche/test-image
sha256:3a5503672ea7fc7e097daa7c7f9785154bb1edc1e699a110d646f7d37cf34df6

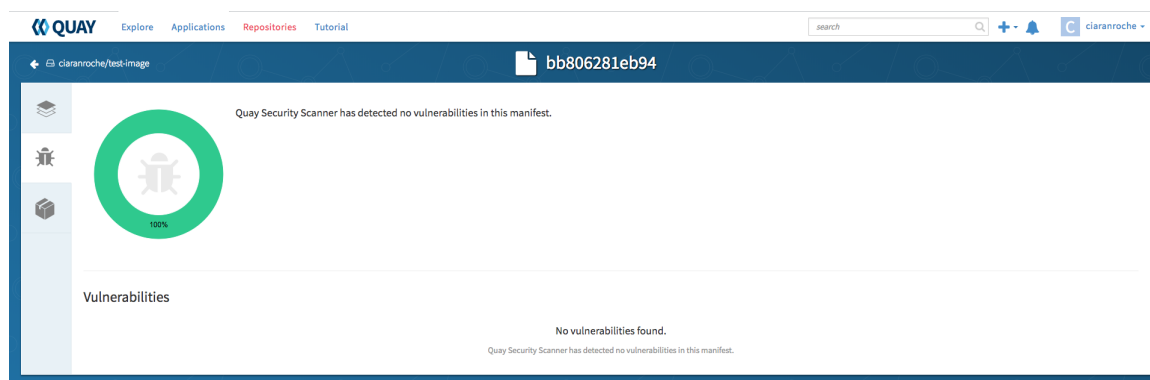
(~)
> docker push quay.io/ciaranroche/test-image
The push refers to repository [quay.io/ciaranroche/test-image]
ea7b3273e2a8: Pushed
b47d9ae7d8d4: Pushed
7f7a180d941c: Pushed
03cf0a520d88: Pushed
95a42fa95cfe: Pushed
ebf12965380b: Pushed
latest: digest: sha256:bb806281eb945bde5035b4d5f14dbce092076f10b94b93be016ff7bc68e598eb size: 8942

(~)
>

```

**Figure 14:** *Docker Vulnerabilities in Static Images 1*

Once the image has been pushed into the repository we can navigate to the Quay account, and from there inspect the image security scan. As can be seen from Image 15 no known vulnerabilities have been found. If they had been known vulnerabilities we would receive a link to the CVE and also any upstream patched package versions.



**Figure 15:** *Docker Vulnerabilities in Static Images 2*

## 3.6 Docker Credentials and Secrets

### 3.6.1 Description

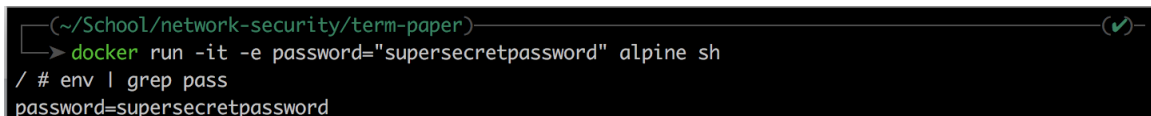
Due to the dynamic nature of containers in that you do not just set up a server so to speak, containers are constantly created, destroyed and or updated. Due to this nature a secure process is needed to share sensitive info, be it user password hashes, encryption keys etc.

### 3.6.2 Best Practices

A common practice is storing secrets in environment variables, this is very insecure. Instead use a Docker credentials management software to manage your secrets. As a general rule of thumb never leave credentials and or secrets in a container, an IBM report can be quoted in saying "it is the same as leaving a jail cell's keys inside the jail cell" (*IBM Data Science Experience*, n.d.).

### 3.6.3 Proof of Concept

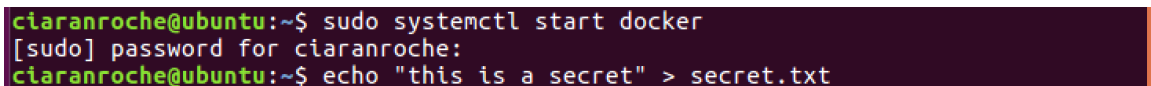
Figure 16 shows how to capture environment variables in a docker container. We launch an alpine image with a password. Once on a bash instance on the container if we simply run `env | grep pass` we can see the password is stored in plain text inside the container.



```
(~/School/network-security/term-paper)
➤ docker run -it -e password="supersecretpassword" alpine sh
/ # env | grep pass
password=supersecretpassword
```

Figure 16: Docker Credentials and Secrets

With the growth in popularity of orchestration systems such as Docker Swarm or Kubernetes. They all provide a basic secret management system. For this I switch over to a Ubuntu Virtual Machine to avoid any configuration errors with my own personal machine. From Figure 17 we launch docker and create a `secret.txt` file and pipe in a secret.



```
ciaranroche@ubuntu:~$ sudo systemctl start docker
[sudo] password for ciaranroche:
ciaranroche@ubuntu:~$ echo "this is a secret" > secret.txt
```

Figure 17: Docker Credentials and Secrets 2

With a base secret file created on our host Virtual Machine, we launch Docker Swarm, and assign an IP address to be advertised, this is crucial in initializing Docker Swarm as the address is advertised to all members of the swarm and allows member API access and overlay networking, it will be used

under the hood for us when we retrieve the key later on. The initialize command can be seen in Figure 18

```
ciaranroche@ubuntu:~$ sudo docker swarm init --advertise-addr 192.168.99.12
Swarm initialized: current node (yeejionzvcwf2r7ok0ec4rcvj1) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-2piwize0wdi9libtymw7c78khybomd09ksgh2iw99
512pf9mdt-4e9qvswyecpyihp56do55m18 192.168.99.12:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow
the instructions.
```

Figure 18: Docker Credentials and Secrets 3

Next we need to create a secret this can be seen in Figure 19 where we create a secret called somesecret and specify the *secret.txt* we created earlier.

```
ciaranroche@ubuntu:~$ sudo docker secret create somesecret secret.txt
uywxh5nxno51mp5sx1s3dp0y7
```

Figure 19: Docker Credentials and Secrets 4

With the secret created and our swarm initialised we now create an nginx container and specify the created secret as can be seen in Figure 20.

```
ciaranroche@ubuntu:~$ sudo docker service create --name nginx --secret source=somesecret,target=somesecret,mode=0400 nginx
p4mk4f418dnt8t8tjpv1iauk
overall progress: 1 out of 1 tasks
1/1: running
verify: Service converged
ciaranroche@ubuntu:~$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
17dcdf55ca83	nginx:latest	"nginx -g 'daemon of...'"	16 seconds ago
Up 13 seconds	80/tcp	nginx.1.uqh1ozbjw5u6efcluk86t7p09	

Figure 20: Docker Credentials and Secrets 5

Finally if we log into the nginx container we can retrieve the secret seen in Figure 21. This is been retrieved from Docker Swarm and is not stored in the container. If we run *ls -l* we can see only root can read the secret with no other access.

It has to be noted this is at a very minimum of security. But it does mean that secrets are stored properly and you can rotate and revoke them as you wish from Docker Swarm.

```
claranroche@ubuntu:~$ sudo docker exec -it 17dcdf55ca83 sh
# cat /run/secrets/somesecret
this is a secret
# ls /run/secrets/somesecret
/run/secrets/somesecret
# ls -l /run/secrets/somesecret
-r----- 1 root root 17 Oct 15 18:18 /run/secrets/somesecret
```

Figure 21: *Docker Credentials and Secrets 6*

## 3.7 Container Auditing

### 3.7.1 Docker Bench Audit Tool

The Docker Bench for Security is a docker container that runs a script to check for common best-practices around deploying Docker containers. It has to be noted that the container is run with a lot of privileges for instance it shares the host's filesystem along with pid and network namespaces. For this case you must insure you trust the container and its origins before running it on a host system (Docker, 2018).

For this test I ran multiple containers on a Ubuntu Virtual Machine. These images where pulled from Docker Hub and where commonly used, Ubuntu images, Alpine images, Nginx images, MongoDB images and SQL images. Figure 22 shows the initial output. Overall no containers failed a test but we did receive a lot of warnings, many of which where outlined already in Section 3.

If you note the first Warning message in Figure 22, Ensure a separate partition for containers has been created. As this tool has been inspired by the CIS Docker Community Edition Benchmark Guide, you can refer to the guide to find the appropriate action this can be seen in Appendix C in relation to our first warning.

```
# -----
# Docker Bench for Security v1.3.4
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker Community Edition Benchmark v1.1.0.
# -----

Initializing Mon Oct 15 19:19:02 UTC 2018

[INFO] 1 - Host Configuration
[WARN] 1.1 - Ensure a separate partition for containers has been created
[NOTE] 1.2 - Ensure the container host has been Hardened
[INFO] 1.3 - Ensure Docker is up to date
[INFO] * Using 18.06.1, verify is it up to date as deemed necessary
[INFO] * Your operating system vendor may provide support and security maintenance for Docker
[INFO] 1.4 - Ensure only trusted users are allowed to control Docker daemon
[INFO] * docker:x:999
[WARN] 1.5 - Ensure auditing is configured for the Docker daemon
[WARN] 1.6 - Ensure auditing is configured for Docker files and directories - /var/lib/docker
[WARN] 1.7 - Ensure auditing is configured for Docker files and directories - /etc/docker
[INFO] 1.8 - Ensure auditing is configured for Docker files and directories - docker.service
[INFO] * File not found
[INFO] 1.9 - Ensure auditing is configured for Docker files and directories - docker.socket
[INFO] * File not found
[WARN] 1.10 - Ensure auditing is configured for Docker files and directories - /etc/default/docker
[INFO] 1.11 - Ensure auditing is configured for Docker files and directories - /etc/docker/daemon.json
[INFO] * File not found
[INFO] 1.12 - Ensure auditing is configured for Docker files and directories - /usr/bin/docker-containerd
[INFO] * File not found
[INFO] 1.13 - Ensure auditing is configured for Docker files and directories - /usr/bin/docker-runc
[INFO] * File not found

[INFO] id_2 - Docker daemon configuration
[WARN] 2.1 - Ensure network traffic is restricted between containers on the default bridge
[PASS] 2.2 - Ensure the logging level is set to 'info'
[PASS] 2.3 - Ensure Docker is allowed to make changes to iptables
[PASS] 2.4 - Ensure insecure registries are not used
[PASS] 2.5 - Ensure aufs storage driver is not used
[INFO] 2.6 - Ensure TLS authentication for Docker daemon is configured
[INFO] * Docker daemon not listening on TCP
[INFO] 2.7 - Ensure the default ulimit is configured appropriately
[INFO] * Default ulimit doesn't appear to be set
[WARN] 2.8 - Enable user namespace support
```

Figure 22: Docker Benchmark Audit Tool

### 3.7.2 Open Source Audit Tool

The majority of Docker Auditing tools, including the official tool used above in Section 3.7.1 are Open Source. So to give some comparison we are going to take another Open Source tool, DockScan. This tool scans containers for security issues and vulnerabilities. It can be easily installed as it is written in Ruby, we can use `gem install` which can be seen in Figure 23.

```
(~/School/network-security/term-paper) (✓)
➤ gem install dockscan
```

Figure 23: Open Source Audit Tool

With the tool installed we launch an NGinx container locally and begin our scan this can be seen in Figure 24. As can be seen we get a human readable output outlining an flaws or vulnerabilities. In this case we can see that many of the recommendations have already been covered throughout Section 3.



```
(~/School/network-security/term-paper) (✓)
➔ docker run --name some-nginx -d -p 8080:80 nginx
786778dc376d1d51c6a0c6828268cc30843a31ef5c7dfd0dcd143eae36bf07ea
(~/School/network-security/term-paper) (✓)
➔ dockscan unix:///var/run/docker.sock
Dockscan Report

High
Container have passwordless users in shadow: It is recommended to set password for user or to lock use
r account.
Running Experimental version of Docker.: It is recommended to replace Docker version with stable and p
roduction ready one.

Medium
Docker running with IPv4 forwarding enabled: It is recommended to disable IPv4 forwarding by default.

Low
Container have higher number of changed files: It is recommended to have minimal number of changed fil
es inside container and do not store data inside container. It is recommended to use volumes.
Insecure registries in use: It is recommended to use secure registries and configuration without inse
cure registries.

W, [2018-10-16T11:05:08.126222 #33591] WARN -- : Following modules failed: ContainerSSHProcess
```

**Figure 24:** *Open Source Audit Tool*

### 3.7.3 Container Auditing Summary

As mentioned previously companies are moving away from giant monolithic applications to more microservice architectures, and with that comes the need and want to rapidly roll out updates and deployments (*NeuVector*, 2018). Implementation of a CI/CD pipeline should be a starting point for checking containers for vulnerabilities. The examples in this Section, show two tools that can be used for auditing containers. It should be noted that these tools can be easily incorporated into a CI/CD work flow. As a good practice you should implement such tools, to insure the integrity of your containers.

## 4 Summary

We have gained an understanding on the underlying architecture to Docker and OCI containers (namespaces and cgroups). But with that we have come to realize that due to Docker running containers in isolation, and having its own built in security in mind it is easy for developers to forget and not think about security when it comes to working with Containers.

We have shown some security threats to Docker Containers despite default security settings, so with that in mind below is a number of practises to follow to help secure your containers.

### 4.1 Immutable Containers

It is often the case to leave shell access to images, this is considered bad practise as it opens up to attackers exploiting the container through injection attacks. By creating immutable containers reduces this risk. The need for shell access can be replaced through the use of orchestration tools allowing for the rebuild, updating and redeployment of containers as needed. It should be noted having an immutable container may affect data persistence, it is also good practise to store data outside of containers.

### 4.2 Trusted Sources

There are many available open source containers available for an abundance of use cases, Linux server, Node.js, NGinx etc. However you should always know where these containers came from, and what the owners security practises are like, eg, how often are they updated. It is good practise to create your own trusted repository and run only trusted containers which you can verify. Even with trusted images there is always the risk of them being tampered in transit, it is always good practise to check signatures, this can easily be scripted prior to running a container.

### 4.3 Role-based Access

Due to the nature of many registries allowing anyone push and pull images it can have severe security implications. It is good practise to set up role-based access to these registries. This reduces and controls who can access and modify your images. As for pulling other users images, only ever use from trusted sources and always flag any vulnerable images you find. As with the containers its always good practice to verify signatures and utilize registries with vulnerability scanning.

## 4.4 Host OS Precautions

When in the container security head space it is often easy to over look the Host. Based off the CIS Docker Benchmark guide to which the Docker Auditing Tool is based off recommend that there is user authentication on the host, along with access roles. Have specific permissions for binary file access and enable logging on the host. To ensure full isolation between the container and host it is recommended to run the Docker engine in kernel mode and the containers in user mode. It is also recommended to utilize the Linux namespaces, secgroups and cgroups on the host machine.

## 4.5 Automated Security Testing

This paper showcased two auditing tools available. There is an abundance of tools to choose from and it show be incorporated into CI/CD pipelines choosing a tool best suited to the application you are running. This ensures no containers leave development with any known vulnerabilities in them

## 4.6 Utilize Monitoring Tools

Away from vulnerabilities monitoring tools should be utilized to reduce and help fight against unwanted behaviour and denial of service attacks on your production containers.

## 4.7 Utilize Orchestration Tools

While Docker provides some security provisions, following on with your own provisions it is recommended to use an orchestration tool to benefit from the added security provided by them along with the easier control of your containers. Tools such as Kubernetes and Docker Swarm make the redeployment and updating of containers easy, along with providing secret management services out of the box.

## Final Note

It is easy to fire and forget about security when working with Containers, the main thing to take away from this paper is be proactive with security your choices and remember, *"if my application does not need it, then why do I have it in my container?"*.

## Appendices

### A Moby Seccomp Defaults

<https://raw.githubusercontent.com/moby/moby/master/profiles/seccomp/default.json>

### B Docker Capabilities

Capability Key	Capability Description
SETPCAP	Modify process capabilities.
MKNOD	Create special files using mknod(2).
AUDIT_WRITE	Write records to kernel auditing log.
CHOWN	Make arbitrary changes to file UIDs and GIDs (see chown(2)).
NET_RAW	Use RAW and PACKET sockets.
DAC_OVERRIDE	Bypass file read, write, and execute permission checks.
FOWNER	Bypass permission checks on operations that normally require the file system UID of the process to match the UID of the file.
FSETID	Don't clear set-user-ID and set-group-ID permission bits when a file is modified.
KILL	Bypass permission checks for sending signals.
SETGID	Make arbitrary manipulations of process GIDs and supplementary GID list.
SETUID	Make arbitrary manipulations of process UIDs.
NET_BIND_SERVICE	Bind a socket to internet domain privileged ports (port numbers less than 1024).
SYS_CHROOT	Use chroot(2), change root directory.
SETFCAP	Set file capabilities.

**Figure 25:** *Docker capabilities by default*

## C CIS Guide Extract

### 1.1 Ensure a separate partition for containers has been created (Scored)

#### Profile Applicability:

Level 1 - Linux Host OS

#### Description:

All Docker containers and their data and metadata is stored under `/var/lib/docker` directory. By default, `/var/lib/docker` would be mounted under `/` or `/var` partitions based on availability.

#### Rationale:

Docker depends on `/var/lib/docker` as the default directory where all Docker related files, including the images, are stored. This directory might fill up fast and soon Docker and the host could become unusable. So, it is advisable to create a separate partition (logical volume) for storing Docker files.

#### Audit:

At the Docker host execute the below command:

```
grep /var/lib/docker /etc/fstab
```

This should return the partition details for `/var/lib/docker` mount point.

#### Remediation:

For new installations, create a separate partition for `/var/lib/docker` mount point. For systems that were previously installed, use the Logical Volume Manager (LVM) to create partitions.

#### Impact:

None.

#### Default Value:

By default, `/var/lib/docker` would be mounted under `/` or `/var` partitions based on availability.

#### References:

<https://www.projectatomic.io/docs/docker-storage-recommendation/>

#### CIS Controls:

14 Controlled Access Based on the Need to Know Controlled Access Based on the Need to Know

## Bibliography

Bryk, A. (2018), ‘Microservice and container security: 10 best practices’.

**URL:** <https://www.apriorit.com/dev-blog/558-microservice-container-security-best-practices>

*CentOS* (n.d.).

**URL:** [https://www.centos.org/docs/5/html/5.2/Deployment\\_Guide/s1-swap-what-is.html](https://www.centos.org/docs/5/html/5.2/Deployment_Guide/s1-swap-what-is.html)

Colyer, A. (2017), ‘A study of security vulnerabilities on docker hub’.

**URL:** <https://blog.acolyer.org/2017/04/03/a-study-of-security-vulnerabilities-on-docker-hub/>

Docker (2018), ‘docker/docker-bench-security’.

**URL:** <https://github.com/docker/docker-bench-security>

*IBM Data Science Experience* (n.d.).

**URL:** <https://wycd.net/posts/2017-02-21-ibm-whole-cluster-privilege-escalation-disclosure.html>

Kerrisk, M. (2012), ‘Cap\_sys.admin: the new root’.

**URL:** <https://lwn.net/Articles/486306/>

Kerrisk, M. (2013), ‘Namespaces in operation, part 1: namespaces overview’.

**URL:** <https://lwn.net/Articles/531114/>

*Linux Manual* (n.d.a).

**URL:** <http://man7.org/linux/man-pages/man7/namespaces.7.html>

*Linux Manual* (n.d.b).

**URL:** <http://man7.org/linux/man-pages/man7/capabilities.7.html>

*NeuVector* (2018).

**URL:** <https://neuvector.com/container-security/continuous-container-security/>

*OCI* (n.d.).

**URL:** <https://www.opencontainers.org/about>

*Oracle Solaris 11* (n.d.).

**URL:** <https://www.oracle.com/solaris/solaris11/>

Rad, B. B., Bhatti, H. J. and Ahmadi, M. (2018), ‘An introduction to docker and analysis of its performance’.

*Red Hat* (2018).

**URL:** [https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/6/html/resource\\_management\\_guide/](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/)

*Red Hat Portal* (2016).

**URL:** <https://access.redhat.com/security/cve/cve-2016-5195>

*What is a Container* (2018).

**URL:** <https://www.docker.com/resources/what-container>

Yegulalp, S. (2018), 'What is docker? docker containers explained'.

**URL:** <https://www.infoworld.com/article/3204171/docker/what-is-docker-docker-containers-explained.html>