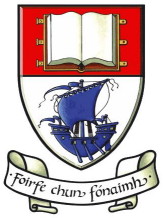

Waterford Institute of Technology

Network Security

Docker Security

Stephen Coady

20064122

BSc in Applied Computing

October 19, 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 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 tools which can help strengthen the security of these containers 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

A virtual machine will tend to be more secure than a container 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).

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 interprocess 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*b*).

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*a*).

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 is 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 some might say this is an advantage. An outline of this can be seen below in Figure 1.

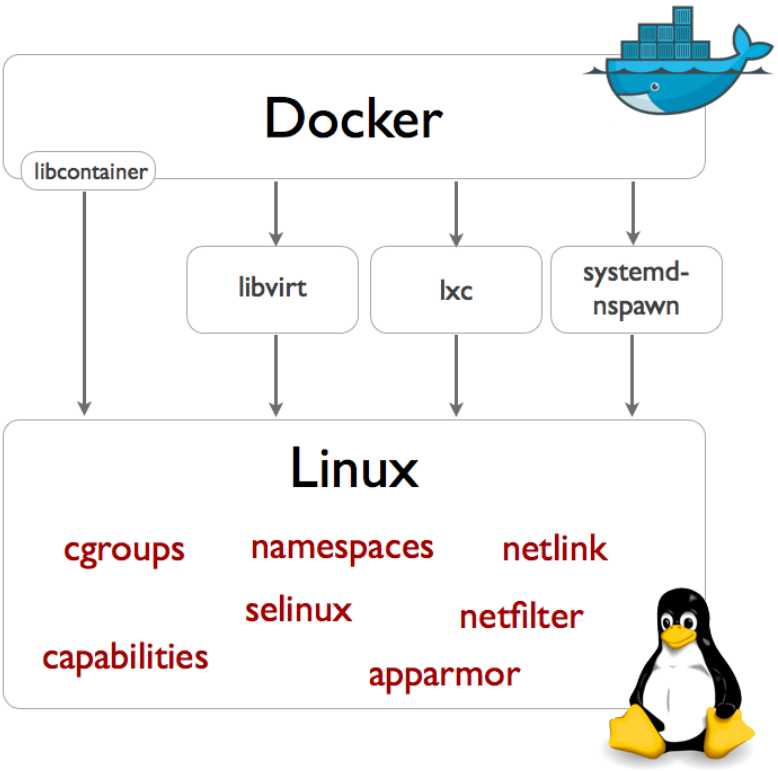

Figure 1: *How Docker Operates on the Linux Kernel. Credit: (Hykes, 2014)*

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 feature 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 these images which could have been put through security checks. We will look at this further in Section 3.

3 Work Carried Out

4 Conclusion

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*

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

Docker (2016*b*), '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* .

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/*

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