Docker ecosystem – Vulnerability Analysis

A. Martin[a], S. Raponi[b], T. Combe[a], R. Di Pietro[b,]

{*antony.martin@nokia.com/antonymartin.pro@gmail.com, sraponi@hbku.edu.qa, theo-nokia@sutell.fr, rdipietro@hbku.edu.qa*}

[a] *Nokia Bell Labs, 1, route de Villejust, 91620 Nozay, France*
[b] *HBKU-CSE, Education City, Doha, Qatar*

Abstract

Cloud based infrastructures have typically leveraged virtualization. However, the need for always shorter development cycles, continuous delivery and cost savings in infrastructures, led to the rise of containers. Indeed, containers provide faster deployment than virtual machines and near-native performance. In this paper, we study the security implications of the use of containers in typical use-cases, through a vulnerability-oriented analysis of the Docker ecosystem. Indeed, among all container solutions, Docker is currently leading the market. More than a container solution, it is a complete packaging and software delivery tool. In this paper we provide several contributions: we first provide a thorough survey on related work in the area, organizing them in security-driven categories, and later we perform an analysis of the containers security ecosystem. In particular, using a top-down approach, we identify in the different components of the Docker environment several vulnerabilities—present by design or introduced by some original use-cases. Moreover, we detail real world scenarios where these vulnerabilities could be exploited, propose possible fixes, and, finally discuss the adoption of Docker by PaaS providers.

Keywords:
Security, Containers, Docker, Virtual Machines, DevOps, Orchestration.

1. Introduction

Virtualization-rooted cloud computing is a mature market. There are both commercial and Open Source driven solutions. For the former ones, one may mention Amazon's Elastic Compute Cloud (EC2) [7], Google Compute Engine [26] [46], VMware's vCloud Air, Microsoft's Azure, while for the latter ones examples include OpenStack combined with virtualization technologies such as KVM or Xen.

Recent developments have set the focus on two main directions. First, the acceleration of the development cycle (agile methods and *devops*) and the increase in complexity of the application stack (mostly web services and their frameworks) trigger the need for a fast, easy-to-use way of pushing code into production. Further, market pressure leads to the densification of applications on servers. This means running more applications per physical machine, which can only be achieved by reducing the infrastructure overhead.

In this context, new lightweight approaches such as containers or unikernels [54] become increasingly popular, being more flexible and more resource-efficient. Containers achieve their goal of efficiency by reducing the software overhead imposed by virtual machines (VM) [64] [56] [62], thanks to a tighter integration of guest applications into the host operating system (OS). However, this tighter integration also increases the attack surface, raising security concerns.

The existing work on container security [38] [58] [51] [42] focuses mainly on the relationship between the host and the container. This focus is completely justified by the fact that, while virtualization exposes well-defined resources to the guest system (virtual hardware resources), containers expose (with restrictions) the host's resources (e.g., IPC / file-system) to the applications. However, the latter feature represents a threat to both confidentiality and availability of applications running on the same host.

Containers are now part of a complex ecosystem - from container to various repositories and orchestrators - with a high level of automation. In particular, container solutions embed automated deployment chains [17] meant to speed up code deployment processes. These deployment chains are often composed of third parties elements, running on different platforms from different providers, raising concerns about code integrity. This can introduce multiple vulnerabilities that an adversary can exploit to penetrate the system. To the best of our knowledge, while deployment chains are fundamental for the adoption of containers, the security of their ecosystem has not been fully investigated yet.

This paper has been published in ComCom (Elsevier), it can be accessed here: https://doi.org/10.1016/j.comcom.2018.03.011 and cited as per here.

The vulnerabilities we consider are classified, relatively to a hosting production system, from the most remote ones to the most local ones, using Docker as a case study. We actually focus on Docker's ecosystem for three reasons. First, Docker successfully became the reference on the market of container and associated *DevOps* ecosystem. Indeed, with more than 29 billions of Docker container downloads, 900.000 Dockerized apps in the Docker Hub and 3.000 community contributors [21], Docker is the world's leading software container platform. Second, data protection is still one of the biggest barriers for container adoption in the enterprise for production workloads [1]. Finally, Docker is already running in some environments which enable experiments and exploring the practicality of some attacks.

Contributions. In this paper, we provide several contributions. First, we provide a thorough survey of related work, classifying them into security-driven categories. Later, we make a detailed list of security issues related to the Docker ecosystem, and run some experiments on both local (host-related) and remote (deployment-related) aspects of this ecosystem. Further on, we show that the design of this ecosystem triggers behaviours (captured in three use-cases) that lower security when compared to the adoption of a VM based solution, such as automated deployment of untrusted code. This is the consequence of both the close integration of containers into the host system and of the incentive to scatter the deployment pipeline at multiple cloud providers. Finally, we argument on the fact that these use-cases trigger and increase specific vulnerabilities, which is a factor rarely taken into account in vulnerability assessment.

Roadmap. This paper is organized as follows: in Section 2, we provide the background information for virtualization and its alternatives. In Section 3 we survey the related work in the area, organizing them in security-driven categories. We then focus on Docker's architecture in Section 4, including its ecosystem. In Section 5 we outline Docker's security architecture. In Section 6, we present Docker's use cases from several points of view and show how they differ from VM and other containers (LinuxVServer, OpenVZ, etc.) use cases. From these typical use cases we build, in Section 7, a vulnerability-oriented risk analysis, classifying vulnerabilities into five categories. Finally, in Section 8, we discuss the implications of these vulnerabilities in a cloud-based infrastructure and the issues they trigger. Conclusions are drawn in Section 9.

2. Technology background

Cloud applications have typically leveraged virtualization, which can also be used as a security component [50] (e.g., to provide monitoring of VMs, allowing easier management of the security of complex cluster, server farms,

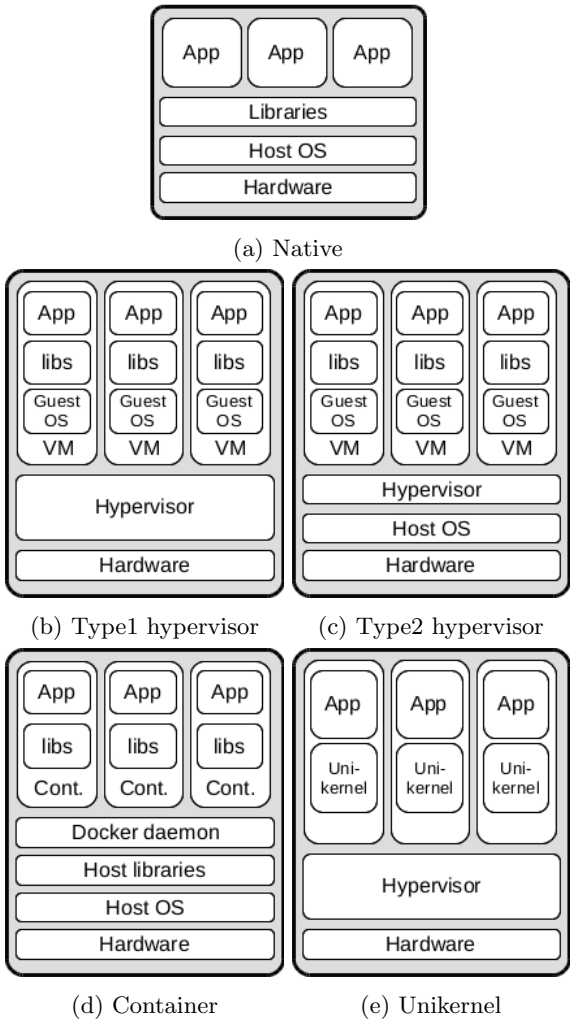

Figure 1: Application runtime models

and cloud computing infrastructure). Given some fundamental constraints such as performance overhead, flexibility, and scalability, alternatives to virtualization have emerged: unikernels as well as containers. All these approaches are summarized in Fig. 1, and detailed below.

2.1. Virtual machines (VM)

Virtual machines are the most common way of performing cloud computing: they are fully functional OS, running on top of an emulated hardware layer, provided by the underlying hypervisor. The hypervisor can either be running directly on hardware (Fig. 1b) (Xen) or on a host OS (Fig. 1c) (for instance KVM). VM can be cloned, installed within minutes, and booted within seconds, hence allowing to stack them with centralized tools. However, the presence of two operating systems (host and guest) along with an additional virtual hardware layer introduces

significant overhead in performance. Hardware support for virtualization dramatically reduces the cited overhead, but performance is far from bare metal, especially for I/O operations [64] [56] [62].

2.2. Containers

Containers (Fig. 1d) provide near bare metal performance as opposed to virtualization (fig. 1a and 1b) [64] [56] [62] with the further possibility to run seamlessly multiple versions of applications on the same machine. For instance, new instances of containers can be created quasi-instantly to face a customer demand peak.

Containers have existed for a long time under various forms, which differ by the level of isolation they provide. For example, BSD jails [48] and chroot can be considered as an early form of container technology. As for recent Linux-based container solutions, they rely on a kernel support, a userspace library to provide an interface to syscalls, and front-end applications. There are two main kernel implementations: LXC-based implementation, using cgroups and namespaces, and the OpenVZ patch. The most popular implementations and their dependencies are shown in Table 1.

Containers may be integrated in a multi-tenant environment, thus leveraging resource-sharing to increase average hardware use. This goal is achieved by sharing the kernel with the host machine. Indeed, in opposition to VM, containers do not embed their own kernel but run directly on the host kernel. This shortens the syscalls execution path by removing the guest kernel and the virtual hardware layer. Additionally, containers can share software resources (e.g., libraries) with the host, avoiding code duplication. The absence of kernel and some system libraries (provided by the host) makes containers very lightweight (image sizes can shrink to a few megabytes). It makes the boot process very fast (about one second [65]). This short startup time is convenient to spawn containers on-demand or to quickly move a service, for instance when implementing Network Function Virtualization (NFV). The deployment of such containers —agnostic of each other even though running on the same shared kernel— requires isolation.

In sections 4, 5, 6 and 7, we will discuss the cgroups and namespaces based containers, and especially Docker's containers. Indeed, Docker popularity, coupled with the extended privileges on the machines it is run, make it a target with a high payoff for any adversary, and that is why we concentrate on the vulnerabilities it is subject to, and possible countermeasures.

2.3. Unikernels

Unikernels consist of very lightweight operating systems (OS), specifically designed to run in VM (Fig. 1e). Unikernels were originally designed to run on the Xen hypervisor [34]. They are more performant than classical VM by shortening the syscalls execution path: they do not embed drivers and the hypervisor does not emulate a hardware layer. The interaction is achieved through a specific API, enabling optimizations that were impossible with the legacy model where an unmodified kernel was running on top of emulated hardware [37]. However, these modifications make unikernels dependent on the hypervisor they run on. Indeed, currently most unikernels development are bound to the Xen hypervisor. Furthermore, unikernels are usually designed to run programs written in a specific programming language. Thus, they only embed the libraries needed by this specific language and are optimized for this specific language (e.g., HaLVM for Haskell, OSv for Java). It decreases the induced overhead relatively to a VM. A detailed performance comparison of OSv against Docker and KVM is currently available [56]. A study on this topic [37] shows that unikernels achieve better performance than VM, and addresses some of the security concerns containers suffer from. This is possible since applications running in unikernels do not share the host OS kernel. The study concludes that unikernels implementations are not mature enough for widespread deployment in production. However, latest developments consider unikernels as a serious concurrent to containers in the longer term [54] [53], both for security reasons and for their significantly short startup time.

2.4. Comparison

Each of these alternatives provides a different trade-off between multiple factors, including performance, isolation, boot time, storage, OS adherence, density and maturity. Most of these factors are performance-related, but security and maturity of the solutions are also at stake. According to the relative importance of these factors, and driven by the intended use, one specific solution may be preferred.

On the one hand, traditional VM are quite resource-consuming and slow to boot and deploy. However, they provide a strong isolation that has been experienced in production for many years. On the other hand, containers are lightweight, very fast to deploy and boot, impose low performance overhead —at the price of less isolation— and provide a new environment with almost no feedback from production uses. Unikernels try to achieve a compromise between VMs and containers, providing a fast and lightweight —but still experimental— execution environment running in an hypervisor.

3. Related work

In this section we provide a thorough survey on related work in the area. We first describe the studies that provide a comparison between containers and virtualization techniques. Then we organize the related work into security-driven groups according to the type of security contribution provided (i.e., security aspects of containers, defense against specific attacks, vulnerability analysis, and use-case driven vulnerability analysis).

Table 1: Container solutions

Base	Container	Library	Kernel dependence	Other dependencies
LXC	Docker	libcontainer	cgroups + namespaces + capabilities + kernel version 3.10 or above	iptables, perl, Apparmor, sqlite, GO
	LXC	liblxc	cgroups + namespaces + capabilities	GO
	LXD	liblxc	cgroups + namespaces + capabilities	LXC, GO
	Rocket	AppContainer	cgroups + namespaces + capabilities + kernel version 3.8 or above	cpio, GO, squashfs, gpg
	Warden	custom tools	cgroups + namespaces	debootstrap, rake
OpenVZ	OpenVZ	libCT	patched kernel	specific components: CRIU, ploop, VCMMD

Container and virtualization comparison

A comparison between containers and virtualization is provided in [43]. This article is a general-purpose survey of containers, showing containers concepts, pros and cons, and detailing features of some container solutions. It concludes on the prediction (the paper dates back to 2014) that containers will be a central technology for future PaaS solutions. However, the comparison is essentially made on a performance basis, and the only security concern mentioned is the need for a better isolation between containers.

A thorough performance evaluation of virtualization and containerization technologies —Docker, LXC, KVM, OSv (unikernel)— is provided in [56]. The authors use multiple benchmark tests (Y-Cruncher, NBENCH, Noploop, Linpack, Bonnie++, STREAM, Netperf) to assess CPU, memory and network throughput and disk I/O in different conditions. They show that containers are significantly better than KVM in network and disk I/O, with performance almost equal to native applications. They conclude by mentioning security as the trade-off of performance for containers. A similar performance evaluation is made in [44].

Security aspects of containers

The security aspect of containers is discussed in more detail in [38]. This article details Docker's interaction with the underlying system, e.g., on the one hand internal security relying on namespaces and cgroups, intended and achieved isolation, and per-namespace isolation features. On the other hand, it details operating system-level security, including host hardening (with a short presentation of Apparmor and SELinux) and capabilities. The authors insist on the need to run containers with the least privilege level (e.g non-root) and conclude that with this use and the default Docker configuration, containers are fairly safe, providing a high level of isolation.

A security-driven comparison among Operating System-level virtualization systems is provided in [58]. In the approach of OS-level virtualization, a number of distinct user space instances (often referred to as containers) are executed on top of a shared operating system kernel. The authors propose a generic model for a typical OS-level virtualization setup with the associated security requirements and compare a selection of OS-level virtualization solutions with respect to this model.

Bacis et al. in [35] focuses on SELinux profiles management for containers. The authors propose an extension to the Dockerfile specification to let developers include the SELinux profile of their container in the built image, to simplify their installation on the host. This solution attempts to address the problem that the default SELinux Docker profile gives all containers the same type, and all Docker objects the same label, so that it does not protect containers from other containers [55].

Defense against specific attacks

With the wide spread usage of containerization technology, numerous articles related to specific defenses have come to light. Given that containers can directly communicate with the kernel of the host, an attacker can perform several escape attacks in order to compromise both the container and the host environment. Once out of the container, she has the potential to cause serious damages by obtaining rights through privilege escalation techniques or by causing the block of the system (and therefore of all the related containers hosted) making use of DoS attacks. In [47], the authors analyze Docker escape attacks and propose a defense method that exploits the status namespaces. This method provides for a dynamic detection of namespace status at runtime (i.e., during the execution of the processes). The resulting monitoring is able to detect anomalous processes and can prevent their escape behaviours, increasing the security and the reliability of the container operations. In [40] the authors proposes a technique based on the limitation of container memory to reduce the Docker attack surface and protects the container technology from DoS attacks.

Vulnerability Analysis

In [45], the authors provide an analysis of the content of the images available to download on the Docker Hub, from a security point of view. The authors show that a significant amount of official and unofficial images on the Docker Hub embed packages with known security vulnerabilities, that can therefore be exploited by an attacker. Detailed results show that 36% of official images

contain high-priority CVE vulnerabilities, and 64% contain medium or high-priority vulnerabilities. Another recent work related to the vulnerabilities inside the images of Docker Hub is described in [61]. The authors of the paper analyze the Docker Hub images by using the framework DIVA (Docker Image Vulnerability Analysis). With the analysis of exactly 356.218 images they show that both official and unofficial images have more than 180 vulnerabilities on average. Furthermore, many images have not been updated for hundreds of days and the vulnerabilities commonly tend to propagate from parent images to child ones. Lu et al. in [52] the authors study the typical penetration testing process under Docker environment related to common attacks such as DoS, container escape and side channel. A. Mouat, in [57], provides an overview of some container vulnerabilities, such as Kernel exploits, DoS attacks, container breakouts, poisoned images, and compromising secrets. The study describes the Docker technology and provides security tips in order to limit the related attack surface. Although some vulnerabilities in our paper are common to those of the book, our work faces them from a different point of view (e.g., both works analyze the vulnerabilities inside the images but only ours considers the vulnerabilities due to the image automatic construction starting from the software development platform GitHub). Besides, in our work we study the security implications of the use of containers taking into account the typical use-cases.

Use-cases driven vulnerability analysis

To the best of our knowledge, our work is the first one that provides a use-cases driven vulnerability analysis. We evaluate the impact of vulnerabilities in the three most used use-cases: Docker recommended use-case, wide-spread use-case (e.g., casting containers as Virtual Machines), and Cloud Provider CaaS use-case. The approach is new since it does not directly concern the Docker software but the code distribution process.

The same authors of this paper have produced a magazine article [41] addressing (at a magazine level) just a subset of the topics discussed here.

4. Docker

In this section, Docker's ecosystem and main components, i.e., specification, kernel support, daemon, Docker Hub and dedicated OS that emerged around Docker, are presented.

The term Docker is overloaded with a few meanings. It is first a specification for container images and runtime, including the Dockerfiles allowing a reproducible building process (Fig. 2 - component a). It is also a software that implements this specification (the Docker daemon, named Docker Engine: Fig. 2 - component b), a central repository where developers can upload and share their images (the Docker Hub: Fig. 2 - component c), and other unofficial repositories (Fig. 2 - component d), along with a trademark (Docker Inc.) and bindings with third parties applications (Fig. 2 - component e). The build process implies fetching code from external repositories (containing the packages that will be embedded in the images: Fig. 2 - component g). An orchestrator (Fig. 2 - component f) can be used for managing the lifecycle of the operational infrastructure.

The Docker project is written in *Go* language and was first released in March 2013. Since then, it has experienced an explosive diffusion and widespread adoption [21].

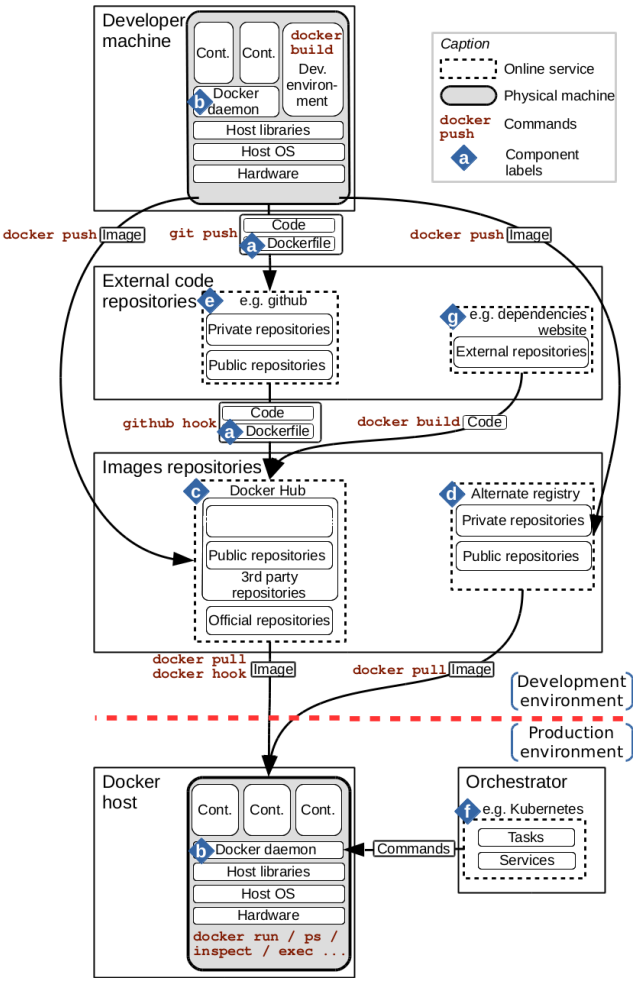

Figure 2: Overview of the Docker ecosystem. Arrows show code path, with associated commands on them (`docker <action>`)

4.1. Docker specification

The specification's scope is container images and runtime.

Docker images are composed of a set of layers along with metadata in *JSON* format. They are stored at `/var/lib/docker/<driver>/` where `<driver>` stands for the storage driver used (e.g., AUFS, BTRFS, VFS, Device Mapper, OverlayFS). Each layer contains the modifications done to the file-system relatively to the previous

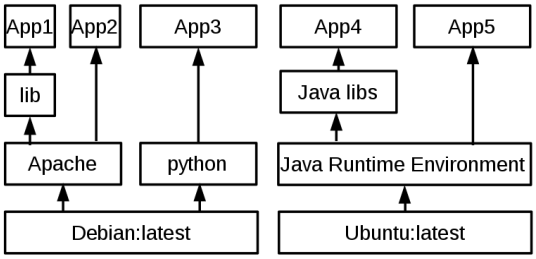

Figure 3: Example of image inheritance trees

layer, starting from a base image (generally a lightweight Linux distribution). This way, images are organized in trees and each image has a parent, except from base images that are roots of the trees (Fig. 3). This structure allows to ship in an image only the modifications specifically related to that image (app payload). Therefore, if many images on a host inherit from the same base image, or have the same dependencies, they will be fetched only once from the repositories. Additionally, if the local storage driver allows it (with a union file-system, i.e., a read-only file system, and some sort of writable overlay on top [63]), it will be stored only once on the disk, leading to substantial resource savings. The detailed specification for Docker images and containers can be found at [20].

Images metadata contain information about the image itself (e.g., ID, checksum, tags, repository, author...), about its parent (ID) along with (optional) default runtime parameters (e.g., port re-directions, cgroups configuration). These parameters can be overridden at launch time by the `docker run` command.

The build of images can be done in two ways. It is possible to launch a container from an existing image (`docker run`), perform modifications and installations inside the container, stop the container and then save the state of the container as a new image (`docker commit`). This process is close to a classical VM installation, but has to be performed at each image rebuild (e.g., for an update); since the base image is standardized, the sequence of commands is exactly the same. To automate this process, Dockerfiles allow to specify a base image and a sequence of commands to be performed to build the image, along with other options specific to the image (e.g., exposed ports, entry point...). The image is then built with the `docker build` command, resulting in another standardized tagged image that can be either run or used as a base image for another build. The Dockerfile reference is available in [25].

4.2. Docker internals

Docker containers rely on creating a wrapped and controlled environment on the host machine in which arbitrary code could (ideally) be run safely. This isolation is achieved by two main kernel features, kernel namespaces [36] and control groups (cgroups). Note that these features were merged starting from the Linux kernel version 2.6.24 [2]. There are currently 7 different namespaces in the kernel, each one addressing a specific aspect of the system [29]:

- PID: provides a separate process tree with PID numbers relative to the namespace (two processes in different namespaces can have the same local PID). Each PID in a namespace is mapped to a unique global PID.
- IPC (inter-process communication): provides POSIX message queues, SystemV IPC, shared memory, etc..
- NET: provides network resources — each NET namespace contains its own network stack, including interfaces, routing tables, iptables rules, network sockets, etc..
- MNT: provides file-system mountpoints: each container has its own view of the file-system and mount points —like an enhanced chroot— in order to avoid path traversals, chroot escapes, or information leak / injection through /proc, /sys and /dev directories.
- UTS: provides hostname and domain isolation.
- USER: provides a separate view of users and groups, including UIDs, GIDs, file permissions, capabilities...
- CGROUP: provides a virtualization of the process' cgroups view — each cgroup namespace has its own set of cgroup root directories, that represents its base points.

Each of these namespaces has its own kernel internal objects related to its type, and provides to processes a local instance of some paths in `/proc` and `/sys` file-systems. For instance, NET namespaces have their own `/proc/net` directory. A thorough list of per-namespace isolated paths is provided by [29] and their isolation role is detailed in [58].

New namespaces can be created by the `clone()` and `unshare()` syscalls, and processes can change their current namespaces using `setns()`. Processes inherit namespaces from their parent. Each container is created within its own namespaces. Hence, when the main process (the container entry point) is launched, all container's children processes are restricted to the container's view of the host.

cgroups are a kernel mechanism to restrict the resource usage of a process or group of processes. They prevent a process from taking all available resources and starving other processes and containers on the host. Controlled resources include CPU shares, RAM, network bandwidth, and disk I/O.

4.3. The Docker daemon

The Docker software itself (Fig. 2b) runs as a daemon on the host machine. It can launch containers, control their level of isolation (cgroups, namespaces, capabilities restrictions and SELinux / Apparmor profiles), monitor them to trigger actions (e.g restart) and spawn shells into running

containers for administration purposes. It can change iptables rules on the host and create network interfaces. It is also responsible for the management of container images: pull and push images on a remote registry (e.g the Docker Hub), build images from Dockerfiles, sign them, etc.. The daemon itself runs as root (with full capabilities) on the host, and is remotely controlled through a UNIX socket. The ownership of this socket determines which users can manage containers on the host using the `docker` command. Alternatively, the daemon can listen on a classical TCP socket, enabling remote container administration without requiring a shell on the host.

4.4. The Docker Hub

The Docker Hub (Fig. 2c) is an online repository that allows developers to upload their Docker images and let users download them. Developers can sign up for a free account, in which all repositories are public, or for a pay account, allowing the creation of private repositories. Repositories from a developer are namespaced, i.e., their name is "developer/repository". There also exist official repositories, directly provided by Docker Inc, whose name is "repository". These official repositories stand for most used base images to build containers. They are "a curated set of Docker repositories that are promoted on Docker Hub" [19].

The Docker daemon, along with the Docker Hub and the repositories are similar to a package manager, with a local daemon installing software on both the host and the remote repositories. Some of the repositories are official while others are unofficial, provided by third parties. From this point of view, the Docker Hub security can be compared to that of a classical package manager [39]. This similarity guided our vulnerability analysis study in Section 7.

4.5. The Docker Store

The Docker Store [23] is an online self-service portal that allows Docker developers to sell and distribute their Docker images to Docker users. The Store contains free and open-source images, as well as software directly sold by publishers. Every user can sign for a free account and can both download freeware images, or purchase premium ones. Users can also become publishers to share their Docker images, hence getting several benefits. Among the many, publishers have greater visibility, the possibility to achieve the Docker certified quality marks, the possibility to use the Docker Store licensing support (to limit access to their software based on the type of users), and a communication channel with the customers, i.e., every customer is notified in case of Docker image updates or upgrades. Although Docker has just its own registry for containers (Docker Hub), the Docker Store is specifically oriented to the needs of enterprises. It offers enterprises with commercially supported software from trusted and verified publishers.

4.6. Docker dedicated operating systems

In addition to the Docker package in mainstream distributions, a number of dedicated distributions have been developed specifically to run Docker or other container solutions. They allow running Docker on host OS other than Linux when run inside a VM, without the complexity of a full Linux distribution. We experimented three of these distributions:

- Boot2docker [10], a distribution based on TinyCore-Linux, meant to be very lightweight (the bootable .iso weights 27MiB). It is mainly used to run Docker containers on OS other than Linux (e.g., running in VirtualBox on Windows Server). The security advantage, when compared to mainstream distributions, is the reduced attack surface due to the minimal installation.

- CoreOS [12], a distribution dedicated to containers. It can run Docker, along with Rocket, for which it was designed. Rocket is a fork of Docker that only runs containers: in opposition to the monolithic design of Docker, interaction with the ecosystem and image builds are managed by other tools in CoreOS. The OS integrates with Kubernetes [28] to orchestrate container clusters on multiple hosts.

- RancherOS [31], an OS entirely based on Docker, meant to run Docker containers. The init process is a Docker daemon (`system-docker`) and system services run in (privileged) containers. One of these services is another Docker daemon (`user-docker`) that spawns itself user-level containers. All installed applications on the system run in Docker containers, so that Docker commands are used to install and update software on the host. No external package manager is required.

5. Docker containers security architecture

By construction, Docker security relies on three components: 1.) isolation of processes at userspace level managed by the Docker daemon; 2.) enforcement of the cited mechanism at kernel level; and, 3.) network operations security. In the following section, we develop each component.

5.1. Isolation

Docker containers rely exclusively on Linux kernel features, including namespaces, cgroups, hardening and capabilities. Namespace isolation and capabilities drop are enabled by default, but cgroup limitations are not, and must be enabled on a per-container basis through `-a -c` options on container launch. The default isolation configuration is relatively strict, the only flaw is that all containers share the same network bridge, enabling ARP poisoning attacks between containers on the same host.

However, the global security can be lowered by options, triggered at container launch, giving (to containers) extended access to some parts of the host (`--uts=host`, `--net=host`, `--ipc=host`, `--privileged`, `--cap-add=<CAP>`, etc.). These features enhance containers convenience, enabling containers to interact with the host system at the price of introducing possible vulnerabilities. For example, when given the option `--net=host` at container launch, Docker does not place the container into a separate NET namespace and therefore gives the container full access to the host's network stack (enabling network sniffing, reconfiguration, etc.).

Additionally, security configuration can be set globally through options passed to the Docker daemon. This includes options lowering security, like the `--insecure-registry` option, disabling TLS certificate check on a particular registry. Options increasing security are available, such as the `--icc=false` parameter, that forbids network communications between containers and mitigates the ARP poisoning attacked described before. However, they prevent multi-container applications from operating properly, hence are rarely used.

5.2. Host hardening

Host hardening through Linux Security Modules is a means to enforce security related limitations constraints imposed to containers (e.g., compromise of a container and escape to the host OS). Currently, SELinux, Apparmor and Seccomp are supported, with available default profiles. These profiles are generic and not restrictive (for instance, the `docker-default` Apparmor profile [5] allows full access to file-system, network and all capabilities to Docker containers). Similarly, the default SELinux policy puts all Docker objects in the same domain. Therefore, default hardening does protect the host from containers, but not containers from other containers. This must be addressed by writing specific profiles, that depend individually on the containers.

SELinux experiments were conducted in 2012 on LXC containers [55], showing that the default SELinux profile gives all containers the same type, and all objects the same label. This configuration does not protect containers from other containers.

5.3. Network security

Network resources are used by Docker for image distribution and remote control of the Docker daemon.

Concerning image distribution, images downloaded from a remote repository are verified with a hash, while the connection to the registry is made over TLS (except if explicitly specified otherwise). Moreover, starting from version 1.8 issued in August 2015, the Docker Content Trust [16] architecture allows developers to sign their images before pushing them to a repository. Content Trust relies on TUF (The Update Framework [60]). It is specifically designed to address package manager flaws [39]. It can recover from a key compromise, mitigate replay attacks by embedding expiration timestamps in signed images, etc.. The trade-off is a complex management of keys. It actually implements a PKI where each developer owns a root key ("offline key") that is used to sign ("signing keys") Docker images. The signing keys are shared among every entity needing to issue an image (possibly including automated signatures in an automated code pipeline, meaning that third-parties have access to the keys). The distribution of the (numerous) root public keys is also an issue.

The daemon is remote-controlled through a socket. By default, the socket used to control the daemon is a UNIX socket, located at `/var/run/docker.sock` and owned by `root:docker`. Access to this socket allows to pull and run any container in privileged mode, therefore giving root access to the host. In case of a UNIX socket, a user member of the `docker` group can gain root privileges, and in case of a TCP socket, any connection to this socket can give root privileges to the host. Therefore the connection must be secured with TLS (`--tlsverify`). This enables both encryption and authentication of the two sides of the connection (while adding additional certificate management).

6. Typical Docker use-cases

Most of the security discussions about containers compare them to VMs, thus assuming both technologies are equivalent in terms of design. Although VMs equivalence is the aim of some container technologies (e.g., OpenVZ used to spawn Virtual Private Servers), recent "lightweight" container solutions such as Docker were designed to achieve completely different objectives than the ones achieved by VMs [4]. Therefore, it is a key point to develop the Docker typical use-cases here, as an introduction to the vulnerability analysis and to put the vulnerabilities (Section 7) in perspective of each context.

We can distinguish three types of Docker usages:

- Recommended use-case, i.e., the usages Docker was designed for, as explained in the official documentation;
- Wide-spread use-case, i.e., the common usages done by application developers and system administrators;
- Cloud provider's CaaS use-case, i.e., the usages guided by the Cloud providers implementations to cope with both security and integration within their infrastructure.

6.1. Recommended use-case

Docker developers recommend a micro-services approach [15], meaning that a container must host a single service, in a single process (or a daemon spawning children). Therefore a Docker container is not considered as a VM: there is no package manager, no init process, no sshd to manage it. All administration tasks (container

stop, restart, backups, updates, builds...) have to be performed via the host machine, which implies that the legitimate containers admin has root access to the host. Indeed, Docker was designed to isolate applications that would otherwise run on the same host, so this root access is assumed to be granted. From a security point of view, isolation of processes (through namespaces) and resources management (through cgroups) makes it safer to deploy Docker applications compared to not using container technology but rather usual processes on the host.

The main advantage of Docker is the ease of application deployment. It was designed to completely separate the code plane from the data plane: Docker images can be built anywhere through a generic build file (Dockerfile) which specifies the steps to build the image from a base image. This generic way of building images makes the image generation process and the resulting images almost host-agnostic, only depending on the kernel and not on the installed libraries. The considerable effort and associated benefits of adopting the micro-services approach are developed in [49]. Airpair [24] lists eight proven real world Docker use cases, that fit in the official recommendations:

- Simplifying configuration;
- Code pipeline management;
- Developer productivity;
- App isolation;
- Server consolidation;
- Debugging capabilities;
- Multi-tenancy; and,
- Rapid deployment.

6.2. Wide-spread use-case

According to Forrester consulting [1], one of the main reasons for adopting containers is to increase developers efficiency rather than favoring micro-service architectures (i.e., the recommended use-case). In fact, two of the most popular images on Docker Hub are Ubuntu (by far) and CentOS [18], two VM-oriented container images. Some sysadmins or developers use Docker as a way of shipping complete virtual environments and updating them on a regular basis, casting containers' usage as VM [6]. Although this is convenient since it limits system administration tasks to the bare minimum (e.g., `docker pull`), it has several security implications. First, embedding more software than the payload the container was designed for, increases the attack surface of resulting container images. Additional packages and libraries could lead to vulnerabilities that would otherwise be avoided. Moreover, this software bloat makes containers management more complex and leads to wasted resources (e.g., larger images, more bandwidth and storage needed do deploy them, more processes in the containers). Then, with containers embedding enough software to run a full system (logging daemon, ssh server, even sometimes an init process), it is tempting to perform administration tasks from within the container itself. This is completely opposed to Docker's design. Indeed, some of these administration tasks need root access to the container. Some other administration actions (e.g., mounting a volume in a container) may need extra capabilities that are dropped by Docker by default. This kind of usage tends to increase the attack surface. Indeed, it enables more communication channels between host and containers, and between co-located containers, increasing the risk of attacks, such as privilege escalation.

Eventually, with the acceleration of software development cycles allowed by Docker, developers cannot maintain each version of their product and only maintain the latest one (tag "latest" on Docker repositories). As a consequence, old images are still available for downloading, but they have not been updated for hundreds of days and can introduce several vulnerabilities [61]. A study [45] has shown that more than 30% of images on the Docker Hub contain high severity CVE vulnerabilities, and up to 70% contain high or medium severity vulnerabilities.

Note that these images cannot always be reproducibly built: although the Dockerfile is public on the Docker Hub, it often includes a statement `ADD start.sh /start.sh` or similar, that copies an install script from the maintainer's computer (and not available on the Dockerhub) to the image and runs it, without appearing in the Dockerfile. Some maintainers even remove this script after execution.

6.3. Cloud providers CaaS use-case

In this section, we present the integration of Docker as provided by the main Cloud Providers. We focused on Amazon Web services, Google Container Engines, and Microsoft Azure as they are three market leaders. Further, we experimented on them as per the provided level of security, and results are reported in the following.

Amazon provides a recent support for Docker containers in its Elastic Compute Cloud (EC2) orchestration tool (generally available since April 2015). It allows users to launch containers in EC2 VM instances and to control them with a dedicated interface (EC2 Container Service (ECS)). Users must first create VM from their EC2 account, install Docker and the ECS daemon on them, register them to their ECS cluster, and, finally, they can launch Docker containers via the web interface [7] or via a command-line tool. Concerning Docker security, the host configuration is up to the users since the host is an EC2 instance, i.e., a VM. Standard images do not provide either Apparmor or SELinux (but they can be installed) and the host network configuration is dependent on the EC2 private cloud [7].

Similarly, Google provides support for Docker in its Compute Engine infrastructure. It has recently issued its Kubernetes orchestrator [28]. It allows to automatically create a cluster of VM on which Docker is installed and

configured, running on a private overlay network. Containers are grouped in "pods": sets of containers sharing the same NET namespace (and so interfaces and IP addresses) and optionally cgroups, enabling direct communication between them. Highly coupled containers (multiple micro-services composing the same application) typically run in the same pod. Pods are the base entity of the Kubernetes orchestrator, just as VM are for classical cloud infrastructures. Pods are automatically instantiated and placed on VMs in the cluster, according to redundancy and availability constraints defined by "replication controllers". These replication controllers are themselves part of "services": entities that define the global parameters of an application (external port mappings, etc.). All nodes in a cluster run a daemon (kubelet) that controls the local Docker daemon, and a central node performs the orchestration. Cluster administration is performed via the `kubectl` command, and a web interface is expected soon.

Kubernetes installation is native in Google Container Engine, which automatically creates the VMs of the cluster. Users can only chose a template for their VMs. `kubectl` commands can be run from any machine with credentials to access the API on the central node. We experimented this setup, choosing the template "n1-standard-1" for the VMs. Installation is also possible on a number of other commercial platforms [27] (including Amazon EC2) via distribution and platform-specific scripts. Installation from scratch on a custom platform is also possible.

Microsoft, instead, provides support for Docker both with Azure Container Instances, and with Azure Container Services (AKS). The former is a solution for any scenario that can operate in a single isolated container, including simple applications and build jobs, while the latter is a solution for a full container orchestration, including service discovery across multiple containers, automatic scaling, and coordinated application upgrades. Azure Container Instances allows a user to download or build a container image. These images can be pushed in an Azure-based private registry called Azure Container Registry. Users can interact with both the images using the Docker CLI tools, and with containers using the Azure CLI [8]. AKS allows to rapidly distribute Docker Swarm, DC/OS, and Kubernetes clusters [9]. AKS has the sole task of installing and deploying the cluster. The orchestrators, in turn, have the goal of managing the containers and services itself. A swarm represents a cluster of nodes that can be either physical or virtual machines. There are two types of nodes: manager and worker nodes. The manager nodes maintain the state of the swarm, while the worker ones have the burden to execute containers [32]. An application image can be deployed by creating a service. When a user deploys the service to the swarm, the swarm manager has the responsibility to schedule it as one o more tasks that run independently of each other on nodes in the swarm. A task is the atomic unit of scheduling within a swarm which is instantiated inside a container. Docker Swarm provides two types of service deployments: replicated and global. Using the replicated service, the user must specify the number of identical tasks she wants to run. A global service instead runs one task of every node, each time the user adds a node to the swarm the orchestrator creates a task and the scheduler assigns the task to the new node [32]. Concerning security, Docker Swarm uses the swarm mode PKI. The manager node generates a new root CA which are used to secure communication with other nodes that join the swarm. Each time a node joins the swarm, the manager issues a certificate to the node [32].

Table 2: Correspondence between Amazon ECS, Kubernetes and Docker Swarm main components

Amazon ECS	**Kubernetes**	**Docker Swarm**
Container instance (VM)	Node	Node
ECS agent	Kubelet	Manager Swarm
Task	Pod	Task
Service	Replication controller	N. A.
Task definition	Service	Service
Cluster	Cluster	Swarm

The aforementioned Cloud Provider approaches are similar and their orchestrators have corresponding main components (Table 2). The main differences lie between Tasks and Pods. Indeed, while tasks are groups of containers that are launched together on the same host, the Kubernetes pod approach differs from Docker's micro-service approach since containers of a same pod share some resources. A pod is closer to a VM than to a container from a functional point of view, albeit with no kernel. Another difference concerns the scaling of the services. In fact, Docker Swarm uses Docker Compose to scale an application [33] — being Docker Compose a tool for defining and running multi-container Docker applications [30].

7. Docker vulnerability-oriented analysis

For each of the three typical Docker use-cases detailed in the previous section, the chosen approach is to define first an adversary model, and then to perform a vulnerability-oriented security analysis [59].

7.1. Adversary model

Given the ecosystem and use-cases description, we consider two main categories of adversaries, i.e., direct and indirect.

A direct adversary is able to sniff, block, inject, or modify network and system communications. She targets directly the production machines. Locally or remotely, she can compromise:

- in-production containers (e.g., from an Internet facing container service, she gains root privileges on the related container; from a compromised container, she makes a DoS on co-located containers, i.e., containers on the same host OS);
- in-production host OS (e.g., from a compromised container, she gains access to critical host OS files, i.e., a container's escape);
- in-production Docker daemons (e.g., from a compromised host OS, she lowers the default security parameters to launch Docker containers);
- the production network (e.g., from a compromised host OS, she redirects network traffic).

An indirect adversary has the same capabilities of a direct one, but she leverages the Docker ecosystem (e.g., the code and images repositories) to reach the production environment.

Depending on the attack phase, we identified the following targets: containers, host OS, co-located containers, code repositories, images repositories, management network.

To subvert a *dockerized* environment, we consider here a subset of all the potential attack vectors, i.e., Docker containers, code repositories, and images repositories. We consider primarily these attack vectors as they are associated with services and interfaces publicly available. Other attack vectors may include host OS, management network or physical access to systems.

7.2. Vulnerabilities identification

In this section we analyze separately the main components of the Docker ecosystem, revealing (part of) their attack surface.

Following a top-down approach, we identified five categories of vulnerabilities, each one related to a different layer of the ecosystem. Typical use-cases (see Section 6), already observed vulnerabilities (e.g., CVE [14]), and scope of the mitigations (e.g., SELinux, seccomp) enriched this iterative process.

These levels are classified from the most remote to the most local one, relatively to a production system hosting Docker containers:

- Insecure production system configuration;
- Vulnerabilities in the image distribution, verification, decompression, storage process;
- Vulnerabilities inside the images;
- Vulnerabilities directly linked to Docker or libcontainer; and,
- Vulnerabilities of the Linux kernel.

We discuss below these five categories, highlighting the limits of the protections offered by Docker. We assume minimal configuration (at least the default config) is applied.

7.3. Insecure configuration

Docker's default configuration is relatively secure as it provides isolation between containers and restricts the containers' access to the host. A container is placed in its own namespaces and own cgroup, and only owns the following capabilities: `CAP_CHOWN`, `CAP_DAC_OVERRIDE`, `CAP_FSETID`, `CAP_FOWNER`, `CAP_MKNOD`, `CAP_NET_RAW`, `CAP_SETGID`, `CAP_SETUID`, `CAP_SETFCAP`, `CAP_SETPCAP`, `CAP_NET_BIND_SERVICE`, `CAP_SYS_CHROOT`, `CAP_KILL` and `CAP_AUDIT_WRITE`.

Vulnerabilities. The use of some options, either given to the Docker daemon on startup, or given to the command launching a container, can provide containers an extended access to the host. We have, for instance:

- Mounting of sensitive host directories into containers
- TLS configuration of remote image registries
- Permissions on the Docker control socket
- Cgroups activation (disabled by default)
- Options directly providing to containers an extended access to the host (`--net=host`, `--uts=host`, `--privileged`, additional capabilities)

Exploitation. For example, the option `--uts=host` allocates the container in the same UTS namespace of the host. This, in turn, allows the container to see and change the host's name and domain. The option `--cap-add=<CAP>` gives to the container the specified capability, thus making it potentially more harmful to the host. With `--cap-add=SYS_ADMIN`, a container can, for instance, remount /proc and /sys sub-directories in read-/write mode, and change the host's kernel parameters, leading to potential vulnerabilities, such as data leakage or denial of service.

Along with these runtime container options, several settings on the host can pave the way to attacks. Even basic properties can at least trigger denial of service. For instance, with some storage drivers (e.g., AUFS), Docker does not limit containers disk usage. A container with a storage volume can fill this volume and affect other containers on the same host, or even the host itself if the Docker storage, located at `/var/lib/docker`, is not mounted on a separate partition.

Mitigation. In order to limit these harmful options which can lead to have the host being accessed by the container, the Center for Internet Security realized a Docker Benchmark [11]. It provides two lists of options: the ones that should be used and the ones that should not be used when running containers as isolated applications with Docker. These options are sorted in six categories: Host configuration, Docker daemon configuration, Docker daemon configuration files, Container images and build file, Container runtime, and Docker security operations. They include the ones mentioned above, along with other host configuration, hardening configuration, file permissions on the host, and TLS configuration for Docker registries and the control socket.

A Docker host running containers that fulfill both the good practices [15] and the CIS recommendations [11], and that further embed a single userspace application, should not need any of the options cited beforehand. Using these options one can break the isolation property. Hence, these options should be used only with trusted containers, reducing the container to an application packaging tool on the production host, and not to an isolation tool.

7.4. Vulnerabilities in the image distribution process

The distribution of images through the Docker Hub and other registries is a source of vulnerabilities in Docker, since the code within these images will be executed on the host. We first discuss vulnerabilities and attacks on the Docker Hub and other registries. Then, we study vulnerabilities that are more specific to Docker, such as vulnerabilities in the extraction process, and related to the automated build chain.

7.4.1. Docker as a package manager

Vulnerabilities. The architecture of the Docker Hub is similar to a package repository, with the Docker daemon acting as a package manager on the host. Therefore it is vulnerable to the same vulnerabilities of package managers. These vulnerabilities include processing, storage and uncompression of potentially untrusted code, performed by the Docker daemon with root privileges. This code can be either tampered at the source (malicious image) or during the transfer (for instance as a consequence of the `--insecure-registry` option given to the Docker daemon, that makes possible a Man-in-the-Middle attack between the registry and the host).

Exploitation. Attacks on package managers are possible [39] if an attacker controls a part of the network between the Docker host and the repository. A successful attack would allow her to make her image downloaded on docker hosts. This leads to compromised images that can exploit vulnerabilities in the extraction process. First, since images are compressed, a specifically crafted image containing a huge file filled with gibberish data (e.g., zeros) would at least fill the host storage device causing denial of service (zipbomb-like attack). Then, since images are extracted on the host file-system, path traversals have been possible in the past (CVE-2014-9356 [13], fixed in Docker 1.3.3). Exploitation of this vulnerability made the uncompression of the images (which is performed as root) follow absolute symlinks on the host, making possible to replace binaries on the host with binaries from the image. Other possible attacks include code injection in images or replay of old images containing known vulnerabilities.

Mitigation. Before release 1.8, the only protection was the use of TLS on the connection to the registry, which could be disabled. With version 1.8, Docker introduced Content Trust [16], an architecture for signing images the same way packages are signed with package managers. This raises two issues. First, Content Trust —and so image signature check— can be disabled passing the `--disable-content-trust` option to the Docker daemon. This looks convenient for a private registry on a local network, but constitutes a security vulnerability. Then, the image signature process requires to trust the developers, which is only possible by unifying the image signature. Moreover, this solution does not scale with thousands of developers signing their repositories with their own key. Beyond the technical challenge, this is a problem of trust and scale.

7.4.2. Automated deployment pipeline vulnerabilities

Vulnerabilities. Automated builds and webhooks proposed by the Docker hub are a key element in this distribution process. They lead to a pipeline where each element has full access to the code that will end up in production. Each element is also increasingly hosted in the cloud. For instance, to automate this deployment, Docker proposes automated builds on the Docker Hub, triggered by an event from an external code repository (GitHub, BitBucket, etc.) — Fig. 4, step 2. Docker then proposes to send an HTTP request to a Docker host reachable on the Internet to notify it that a new image is available; this event triggers an image pull and the container restarts on the new image (Docker hooks [17]) — Fig. 4, steps 9 and 10. With this deployment pipeline, a commit on GitHub (Fig. 4, step 1) will trigger a build of a new image (Fig. 4, step 2); this image will be automatically launched it in production (Fig. 4, steps 9 and 10). Optional test steps can be added before production, themselves potentially being hosted at another provider. In this last case, the Docker Hub makes a first call to a test machine (Fig. 4, steps 3 and 5), that will then pull the image, run the tests, and send the results to the Docker Hub using a callback URL (Fig. 4, steps 4 and 6). The build process itself often downloads dependencies from other third-parties repositories (Fig. 4, steps 7 and 8), sometimes over an unsecure channel (that could be tampered with). The whole code pipeline is exposed in Fig. 4.

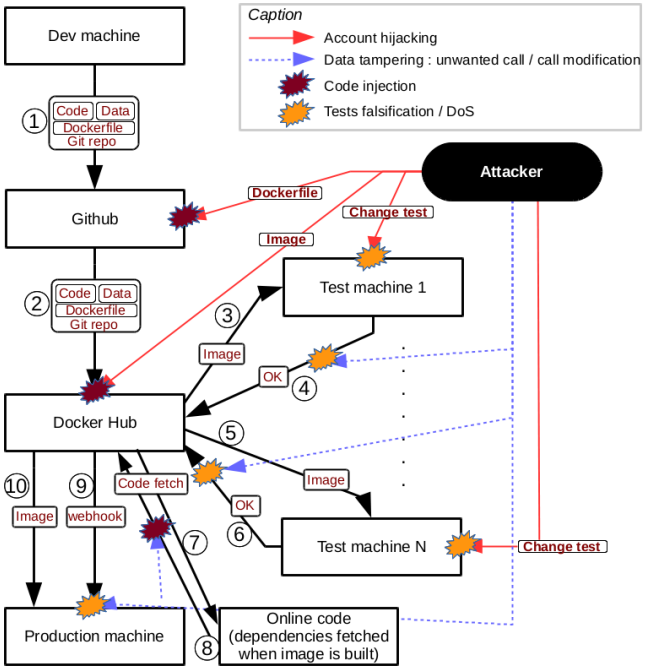

Figure 4: Automated deployment setup in the public cloud using GitHub, the Docker Hub, external test machines and repositories from where code is downloaded during build process.

Exploitation. In this architecture, compromise approaches include account hijacking, tampering with network communications (depending on the use of TLS), and insider attacks. This setup adds several external intermediary steps to the code path, each of them having its own authentication and attack surface, overall increasing the global attack surface.

For instance, we had the intuition that a compromised GitHub account could lead to the execution of malicious code on a large set of production machines within minutes. We therefore tested a scenario including a Docker Hub account, a GitHub account, a development machine and a production machine. The assumption was that the adversary will use the Docker ecosystem to put in production a backdoored Docker container. More precisely, we assumed that the adversary had successfully compromised some code on the code repository (for instance, via a successful phishing attack).

Due to network restrictions (corporate proxy) our servers could not be reached by webhooks, so we wrote a script to monitor our repository on the Docker Hub as well as downloads of new images (Fig. 5). Our initial intuition was confirmed: adversary's code was put in production 5 minutes and 30s after the adversary's commit on GitHub. This attack is particularly dreadful, since it scales to an arbitrary number of machines watching the same Docker Hub repository.

Note that while compromising a code repository is independent of Docker, automatically pushing it in production dramatically increases the number of compromised machines, even if the malicious code is removed within

```
#!/bin/bash
while [ 1 -eq 1 ]
do
status=‘docker pull docker_hub_acc/dockerhook-auto \
     | grep "Image is up to date"‘
if  [ "$status" = "" ]
then  # Updating and restarting container
 docker ps | grep docker_hub_acc/dockerhook-auto \
 | awk ’{print $1 }’ | xargs docker stop
 docker rm dockerhook
 docker run -d -p 82:80 --name dockerhook \
 docker_hub_acc/dockerhook-auto
fi
sleep 5
done
```

Figure 5: Script monitoring the remote repository and updating the container if a new version is available

minutes. Compromise could also happen at the Docker Hub account level, with the same consequences. Account hijacking is not a new problem, but it should be an increasing concern with the multiplication of accounts at different providers. With source code tampered at the victim machine, TLS is useless. Actually, the tampered code is "securely" distributed over TLS to the various repositories. Furthermore, with TLS, IDS and IPS monitoring solutions are blind — unless legitimate man-in-the-middle setups are done.

Moreover, while code path is usually (and always with Docker) secured using TLS communications, it is not the case of API calls that trigger builds and callbacks. Tampering with these data can lead to erroneous test results, unwanted restarts of containers, etc.. Additionally, such a setup is not compatible with the Content Trust scheme, since code is processed by external entities between the developer and the production environment. Content Trust provides an environment in which a single entity is trusted (the person or organization that signed the images) while in the present case trust is split over several external entities, each of them being capable of compromising the images.

7.5. Vulnerabilities inside the images

The code available for download on the Docker Hub (used to build images) is directly exposed to attackers when in production.

Vulnerabilities. in [45] it has been showed from crawling the Docker Hub that 36% of official images contain high-priority CVE vulnerabilities, and 64% contain medium or high-priority vulnerabilities. These figures lower to 23% and 47% respectively for images tagged "latest". Although these images are the most downloaded on the Docker Hub, they contain a significantly high amount of vulnerabilities, including some recent well known vulnerabilities (e.g., shellshock and heartbleed).

The *DevOps* movement, promoted by Docker, lets developers package themselves their applications, thus mixing

the development and production environments hence possibly introducing vulnerabilities. Development versions of packages or dev tools can remain in the final version of the Docker image, increasing its attack surface (e.g., a debugging tool installed in the image).

Then, the provided images often contain outdated versions of packages, either because their base image (e.g., Ubuntu or Debian) is old or because their build process pulls outdated code from some remote repository. The multiplication of image builds —virtually one for each commit in a project— leads to a persistence of outdated images still available on the repositories, while fast development cycles generally focus on latest versions.

Exploitation. Exploitation of such vulnerabilities is relevant in a context where the attack comes from outside (i.e., not a malicious image). Classical application vulnerabilities exploitation methods are possible, provided that the container exposes an entry point (network port, input data, etc.). Additionally, images built from external code repositories (i.e., images that pull data from some repository during the build process —Online code on Fig. 4— as stated in their Dockerfile) are dependent on this repository and on the (in)security of the connection used to fetch these data. These repositories, not always official, are another entry point for code injection.

Mitigation. Both Docker Hub and Docker Cloud make use of the Docker Security Scanning. Users can scan images in private repositories to verify that they are free from kwown security vulnerabilities. This feature is available as a free preview for private repository subscribers for a limited time. The scan traverses each layer of the image, identifying the software components and indexing their SHAs. These SHAs are compared against the CVE database in order to obtain information about the well-known security vulnerabilities [22]. The whole scan phase takes from 1 to 24 hours, depending on the size of the evaluating images. The support provided is limited due to both the availability only for private repositories and the cost of the service. Moreover, if a vulnerability is not part of the database the scan cannot reveal it, making the service unresponsive to new attacks.

7.6. Vulnerabilities directly linked to Docker or libcontainer

Vulnerabilities. Vulnerabilities found into Docker and libcontainer [14] mostly concern file-system isolation: chroot escapes (CVE-2014-9357, CVE-2015-3627), path traversals (CVE-2014-6407, CVE-2014-9356, CVE-2014-9358), access to special file systems on the host (CVE-2015-3630, CVE-2016-8867, CVE-2016-8887), container escalation (CVE-2014-6408), and privilege escalation (CVE-2016-3697). Most of these specific vulnerabilities are all patched as of Docker version 1.6.1. Further, CVE-2016-3697 is patched as of Docker version 1.11.0 while CVE-2016-3697 and CVE-2016-8867 are patched with Docker version 1.12.3. Since container processes often run with PID 0 (i.e., root privileges), they have read and write access on the whole host file-system when they escape. Thus, they are allowed to overwrite host binaries, which leads to a delayed arbitrary code execution with root privileges.

Exploitation. Beyond the kernel namespaces, cgroups, Docker dropping capabilities and mount restrictions, Mandatory Access Control (MAC) can enforce constraints in case the normal execution flow is not respected. This approach is visible in the `docker-default` Apparmor policy. However, there is room for improvements in the MAC profiles for containers. In particular, Apparmor profiles normally behave as whitelists [3], explicitly declaring the resources any process can access, while denying any other access when the profile is in enforce mode. However, the `docker-default` profile installed with the `docker.io` package gives containers full access to network devices, file-systems along with a full set of capabilities, and contains just a small list of `deny` directives, consisting *de facto* in a blacklist.

7.7. Vulnerabilities of the Linux kernel

Since containers run on the same kernel of the host, they are vulnerable to kernel exploits. An exploit giving full root privileges into a container can allow an attacker to break out this container and compromise the host (triggering for instance isolation and integrity breach, as well as data exposure).

8. Discussion

8.1. Vulnerability assessment

We conducted an assessment of the severity of the explained vulnerabilities (Table 3) according to each use-case. This is coherent with the choice we made to analyze the Docker ecosystem and typical use-cases rather than focusing on a specific use-case with a specific application. This is also consistent with the NIST methodology that defines the context as a dimension that has to be considered when performing a vulnerability assessment [59]. Therefore, we defined three use-cases to base our comparison upon and we also proved experimentally some of the highlighted vulnerabilities.

Our assessment is exclusively focused on the difference between use-cases, all other dimensions (i.e., threats and remediations) being equal. As a general comment, the wide-spread use-case (i.e., casting containers as VM) exposes vulnerabilities more than the other use-cases. Moreover, we can see that, whatever the considered use-cases, the kernel vulnerabilities are of a similar severity level, as well as the vulnerabilities directly linked to Docker. We believe that this analysis of the different layers of the Docker ecosystem provides a valuable vulnerability assessment state of the art and a sound basis for the research community upcoming contributions.

Table 3: Relative assessment scale of vulnerabilities severity on the three developed usages.

Vulnerability categories	Docker recommended use-case	Wide-spread use-case (e.g., casting containers as VM)	Cloud Provider CaaS use-case
Insecure configuration	**Moderate.** Docker's default configuration on local systems is relatively secure — see Section 7.3. Lowering of security configuration possible by the sysadmin or containers placement.	**Very high** Very likely insecure configuration.	**High** CIS benchmark on EC2 by default score 62% of compliance. Containers in pods sharing the same NET namespace with Kubernetes.
Vulnerabilities in the image distribution, verification, decompression and storage process	**Very high** Usage promoted extensively be the *DevOps* approach. Automation at all layers to bring shorter development cycles and continuous delivery.	**Moderate** Containers used as VMs, followed by less continuous delivery	**High** Automation at all layers to bring shorter development cycles and continuous delivery.
Vulnerabilities inside the images	**Moderate** By default exposing a limited attack surface.	**Very high** Very likely usage of heavyweight Linux distribution images with an attack surface bigger than micro-services-oriented images.	**Moderate** Depending on what images and where they are retrieved from: both micro-server and VM like usages are possibly used here.
Vulnerabilities directly linked to Docker or libcontainer	**Similar level across the use-cases**	**N. A.**	**N. A.**
Vulnerabilities in the kernel	**Similar level across the use-cases**	**N. A.**	**N. A.**

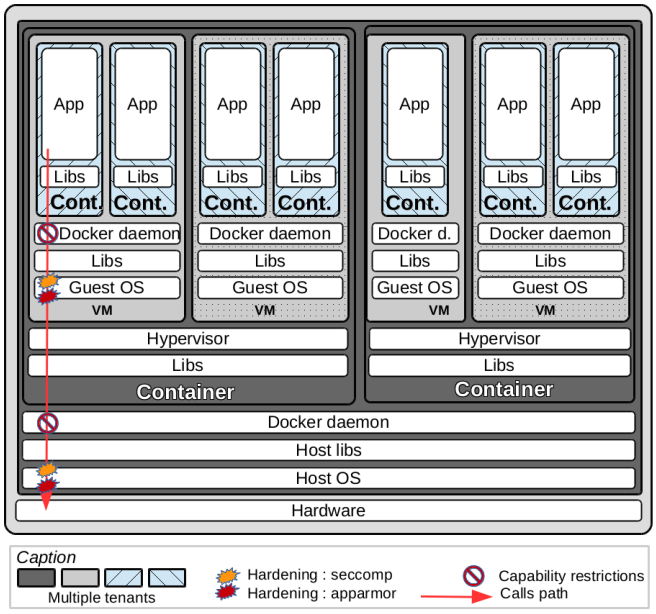

Figure 6: Containers integration in a multi-tenant cloud system. There can be up to three different tenants vertically, and for each vertical layer an arbitrary number of tenants horizontally.

8.2. Multi-tenant implementation

Container execution in PaaS providers data-centers is becoming a standard for cloud applications [7] [46]. Ensuring isolation is of paramount importance in such multi-tenant installations where users can run their own code in containers. We showed in Section 6.3 that main public cloud providers run these containers inside VM, and allocate a VM to a single user, creating a vertical multi-tenancy. These VMs are sometimes themselves running in containers (e.g., Google Container Engine [46]) — Fig. 6.

Users may also delegate accounts to other users in their cluster, creating an horizontal multi-tenancy, where containers belonging to different users run concurrently in the same VM. For instance, Kubernetes allows creating user accounts within a cluster, each user being able to run containers. As illustrated in Fig. 6, the security profiles (e.g., SELinux and Apparmor profiles) are stacked at multiple layers. Therefore, the configuration of these security profiles may be challenging. In particular, enforcing cumulative and independently managed security profiles could lead to a situation where legitimate actions could be pre-

vented from executing. This can become a complex task, since even with basic Apparmor profile, we observed unexpected behaviours, as described below.

```
#!/bin/bash
mount -o remount,rw /proc/sys
mount -o remount,rw /proc/sysrq-trigger
echo 1 > /proc/sys/kernel/sysrq
echo p > /proc/sysrq-trigger
```

Figure 7: Script assessing effective application of Apparmor restrictions

A script (Fig. 7) was run in a Docker container trying to perform actions prohibited by the `apparmor-default` profile while it was enforced. This container was run with the `--cap-add=SYS_ADMIN` option, so that it could perform the `mount` command. We ran our tests on two machines: a Debian 8 (kernel 3.16) with the Docker package from the distribution (version 1.6.2) and an Ubuntu 14.04 (kernel 3.19) with Docker installed from `get.docker.com` (version 1.8.3). The default Apparmor profiles for both distributions forbid mounts and write access to `/proc/sys/kernel/sysrq` and `/proc/sysrq-trigger`, so that even with CAP_SYS_ADMIN none of these commands should succeed. While on Ubuntu the commands were blocked, the mounts were successfully performed on the Debian 8, even though the profile contained en explicit "deny mount".

This example shows that, even with a regular installation from repositories and a rudimentary script, the Apparmor default profile for Docker may be applied differently. This situation can only worsen when several layers of containers and hardening are nested, as illustrated in Fig. 6.

Additionally, this architecture does not replace VM, so the rapidity of execution of containers is not exploited as it could be. Code execution paths of containers (left arrow on Fig. 6) are even longer than with a single VM, since they add two Docker processes, with their libraries and the associated hardening.

8.3. Synoptic survey

According to the categories introduced above, we map in Table 4 the contributions of the most representative work on containers, as well as our own.

The topic of containers has been addressed in the literature from different perspectives. Some work make a comparison with virtual machines, either from a performance point of view [43, 56, 44] or from a security one [52]. Other work evaluate the security aspects of containers [38, 58, 35, 55, 57] or propose defenses related to specific attacks [40, 47]. The analysis of Docker vulnerabilities (generic or specific) are dealt with in various work [40, 45, 61, 52, 57] but in this paper we introduce new elements (e.g., the analysis and exploitation of the vulnerabilities arisen from the automated construction of images triggered by events from external code repository, such as Docker Hub and BitBucket), as well as a taxonomy that can be used by other scientists and practitioners to evolve the mapping between solutions and the related vulnerabilities contributing to the identification of possible countermeasures.

8.4. Alternatives to Docker

In parallel to containers, Unikernels have been around for a few years. Although they are not used in production yet, they address the isolation issue by embedding their own kernel, specifically optimized for one application. They achieve performance close to or even better than containers, and their very fast boot could allow launching them on the fly to serve a specific request. Latest technology developments promote unikernels to be a serious concurrent to containers [54] [53] for the coming years.

9. Conclusion

In this paper, we performed a review of the components of the whole Docker ecosystem—the containers, the code delivery, and related orchestration. Leveraging an extensive literature review, coupled with a security-driven classification, we have identified key elements to assess the security of each component. In particular, we highlighted some new classes of vulnerabilities. To substantiate our findings, we performed some experiments to demonstrate how tangible were the security risks. We showed that the usual comparison "VM vs. containers" is inappropriate since the use cases are different and so is the underlying architecture.

We also showed that from a local point of view, many vulnerabilities of Docker are due to casting containers as VM. From the point of view of the ecosystem, the multiplication of external intermediaries, providing code that will end up in production, widely increases the attack surface.

Further, we discussed the current usage of orchestrators, and pointed out that these orchestrators could be a means of limiting misuse of Docker — leading to host and containers weak isolation. In particular, enforcing isolation could be achieved introducing higher levels of abstraction (tasks, replication controllers, remote persistent storage, etc.) that completely remove host dependence, enabling better isolation. We are currently working on the analysis of the orchestrators security.

Table 4: Synoptic survey of the major current contributions in the container domain

	Container and virtualization comparison	Security aspects of containers	Defense against DoS attacks	Defense against Escape attacks	Vulnerability analysis	Use-cases driven vulnerability analysis
[43]	✓					
[56]	✓					
[44]	✓					
[38]		✓				
[58]		✓				
[35]		✓				
[55]		✓				
[40]			✓		✓	
[47]		✓		✓		
[45]					✓	
[61]					✓	
[52]	✓	✓			✓	
[57]		✓	✓		✓	
This work	✓	✓			✓	✓

References

[1] Containers: Real adoption and use cases in 2017. http://en.community.dell.com/techcenter/cloud/m/dell_cloud_resources/20443801.

[2] Notes from a container, October 2007. https://lwn.net/Articles/256389.

[3] Novell apparmor administration guide, September 2007. https://www.suse.com/documentation/apparmor/pdfdoc/book_apparmor21_admin/book_apparmor21_admin.pdf.

[4] Docker and ssh, June 2014. https://blog.docker.com/2014/06/why-you-dont-need-to-run-sshd-in-docker.

[5] docker-default apparmor profile, June 2014. https://wikitech.wikimedia.org/wiki/Docker/apparmor.

[6] Containers are not vms, March 2016. https://blog.docker.com/2016/03/containers-are-not-vms/.

[7] Amazon ec2 container service reference, Last checked January 2018. http://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_instances.html.

[8] Azure container instances, Last checked January 2018. https://docs.microsoft.com/en-us/azure/container-instances/.

[9] Azure container services, Last checked January 2018. https://docs.microsoft.com/en-us/azure/container-service/.

[10] Boot2docker project, Last checked January 2018. http://boot2docker.io/.

[11] Cis docker benchmark. Tech. rep., Center for Internet Security, January 2018.

[12] Coreos project, Last checked January 2018. https://coreos.com/docs/.

[13] Cve-2014-9356, Last checked January 2018. https://security-tracker.debian.org/tracker/CVE-2014-9356.

[14] Cve vulnerability statistics on docker, Last checked January 2018. http://www.cvedetails.com/product/28125/Docker-Docker.html.

[15] Docker best practices, Last checked January 2018. https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices.

[16] Docker content trust, official documentation, Last checked January 2018. https://docs.docker.com/engine/security/trust/content_trust.

[17] Docker hub: Automated builds and webhooks, Last checked January 2018. https://docs.docker.com/docker-hub/builds.

[18] Docker hub images, Last checked January 2018. https://hub.docker.com/explore.

[19] Docker hub official repositories, Last checked January 2018. https://docs.docker.com/docker-hub/official_repos.

[20] Docker image specification, Last checked January 2018. https://github.com/docker/docker/blob/master/image/spec/v1.md.

[21] Docker overview, Last checked January 2018. https://www.docker.com/company.

[22] Docker security scanning, Last checked January 2018. https://docs.docker.com/docker-cloud/builds/image-scan.

[23] Docker store, Last checked January 2018. https://docs.docker.com/docker-store.

[24] Docker use-cases, Last checked January 2018. https://www.airpair.com/docker/posts/8-proven-real-world-ways-to-use-docker.

[25] Dockerfile reference, Last checked January 2018. https://docs.docker.com/engine/reference/builder.

[26] Google compute engine reference, Last checked January 2018. https://cloud.google.com/compute/docs/.

[27] Kubernetes installation advices, Last checked January 2018. https://kubernetes.io/docs/setup/pick-right-solution/.

[28] Kubernetes orchestrator, Last checked January 2018. http://kubernetes.io/.

[29] Linux kernel namespaces man page, Last checked January 2018. http://man7.org/linux/man-pages/man7/namespaces.7.html.

[30] Overview of docker compose, Last checked January 2018. https://docs.docker.com/compose/overview/.

[31] Rancheros project, Last checked January 2018. http://rancher.com/docs/os/v1.1/en/.

[32] Swarm mode, Last checked January 2018. https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/.

[33] Use compose with swarm, Last checked January 2018. https://docs.docker.com/compose/swarm/.

[34] Xen wiki page about unikernels, Last checked January 2018. http://wiki.xenproject.org/wiki/Unikernels.

[35] Bacis, E., Mutti, S., Capelli, S., and Paraboschi, S. Dockerpolicymodules: mandatory access control for docker containers. In *Communications and Network Security (CNS), 2015 IEEE Conference on* (2015), IEEE, pp. 749–750.

[36] Biederman, E. Multiple instances of the global linux namespaces. In *Proceedings of the 2006 Linux Symposium* (2006). https://www.kernel.org/doc/ols/2006/ols2006v1-pages-101-112.pdf.

[37] Briggs, I., Day, M., Guo, Y., Marheine, P., and Eide, E. A performance evaluation of unikernels. Tech. rep., 2014.

[38] Bui, T. Analysis of Docker Security, 2015. arXiv:1501.02967v1.

[39] Cappos, J., Samuel, J., Baker, S. M., and Hartman, J. H. A look in the mirror: attacks on package managers. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pp. 565–574.

[40] Chelladhurai, J., Chelliah, P. R., and Kumar, S. A. Securing docker containers from denial of service (dos) attacks. In *Services Computing (SCC), 2016 IEEE International Conference on* (2016), IEEE, pp. 856–859.

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

[42] Di Pietro, R., and Lombardi, F. *Security for Cloud Computing*. Artec House, Boston, 2015. ISBN 978-1-60807-989-6.

[43] Dua, R., Raja, A., and Kakadia, D. Virtualization vs containerization to support paas. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering (IC2E)* (March 2014), pp. 610–614.

[44] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers. Tech. rep., IBM Research Report, July 2014. `http://www.cs.nyu.edu/courses/fall14/CSCI-GA.3033-010/vmVcontainers.pdf`.

[45] Gummaraju, J., Desikan, T., and Turner, Y. Over 30% of official images in docker hub contain high priority security vulnerabilities. Tech. rep., BanyanOps, May 2015.

[46] Intel. Linux* containers streamline virtualization and complement hypervisor-based virtual machines.

[47] Jian, Z., and Chen, L. A defense method against docker escape attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy* (2017), ACM, pp. 142–146.

[48] Kamp, P.-H., and Watson, R. N. M. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE)* (May 2010). `http://therbelot.free.fr/Install_FreeBSD/jail/jail.pdf`.

[49] Killalea, T. The hidden dividends of microservices. *Communications of the ACM 59*, 8 (2016), 42–45.

[50] Lombardi, F., and Di Pietro, R. Secure virtualization for cloud computing. *Journal of Network and Computer Applications 34*, 4 (2011), 1113–1122.

[51] Lombardi, F., and Di Pietro, R. Virtualization and cloud security: Benefits, caveats, and future developments. In *Cloud Computing*, Z. Mahmood, Ed., Computer Communications and Networks. Springer International Publishing, 2014, pp. 237–255.

[52] Lu, T., and Chen, J. Research of penetration testing technology in docker environment.

[53] Madhavapeddy, A., Leonard, T., Skjegstad, M., Gazagnaire, T., Sheets, D., Scott, D., Mortier, R., Chaudhry, A., Singh, B., Ludlam, J., Crowcroft, J., and Leslie, I. Jitsu: Just-in-time summoning of unikernels. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 559–573.

[54] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. vol. 48, ACM, pp. 461–472.

[55] Miller, A., and Chen, L. Securing your containers - an exercise in secure high performance virtual containers. In *Proceedings of the International Conference on Security and Management (SAM)* (2012), The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), p. 1. `http://worldcomp-proceedings.com/proc/p2012/SAM9702.pdf`.

[56] Morabito, R., Kjallman, J., and Komu, M. Hypervisors vs. lightweight virtualization: A performance comparison. In *Proceedings of the 2015 IEEE International Conference on Cloud Engineering* (2015), pp. 386–393.

[57] Mouat, A. Docker security using containers safely in production, 2015.

[58] Reshetova, E., Karhunen, J., Nyman, T., and Asokan, N. Security of os-level virtualization technologies: Technical report. *CoRR abs/1407.4245* (2014).

[59] Ross, R. S. Guide for conducting risk assessments (nist sp-800-30rev1). *The National Institute of Standards and Technology (NIST), Gaithersburg* (2012).

[60] Samuel, J., Mathewson, N., Cappos, J., and Dingledine, R. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 61–72.

[61] Shu, R., Gu, X., and Enck, W. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (2017), ACM, pp. 269–280.

[62] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 275–287.

[63] Wright, C. P., and Zadok, E. Kernel korner: Unionfs: Bringing filesystems together. *Linux J. 2004*, 128 (Dec. 2004), 8–.

[64] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., and De Rose, C. A. F. Performance evaluation of container-based virtualization for high performance computing environments. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (Washington, DC, USA, 2013), PDP '13, IEEE Computer Society, pp. 233–240.

[65] Zheng, C., and Thain, D. Integrating containers into workflows: A case study using makeflow, work queue, and docker. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing* (New York, NY, USA, 2015), VTDC '15, ACM, pp. 31–38.