To Docker or Not to Docker: A Security Perspective

Article *in* IEEE Cloud Computing · September 2016

DOI: 10.1109/MCC.2016.100

Roberto Di Pietro
Hamad bin Khalifa University

To Docker or not to Docker: a security perspective

Theo Combe Antony Martin Roberto Di Pietro

The need for always shorter development cycles, continuous delivery and cost savings in cloud based infrastructures, led to the rise of containers, that provide much more flexibility than virtual machines and near-native performance. Among all container solutions, Docker is currently leading the market. In particular, Docker is a complete packaging and software delivery tool. In this work, we first provide a comprehensive view on containers ecosystem; later, we discuss through realistic use-cases the security implications of Docker environment. Further, we define an adversary model, point out several vulnerabilities affecting current usages of Docker, and finally we discuss further research directions.

KEYWORDS

Security, Containers, Docker, Linux, Orchestration.

I. Introduction

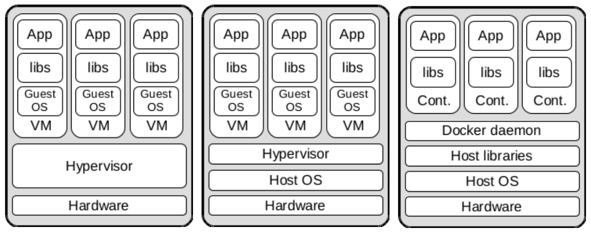
Cloud computing is inherently rooted on virtualization technologies. Recently, lightweight technologies such as containers have emerged and become also increasingly popular and integrated into the Cloud offers. Actually, containers tight integration into the host OS allows reducing the software overhead imposed by virtual machines (VM) [1].

However, this tighter integration increases the attack surface, raising security concerns. The existing work on container security [2] [3] [4] [5] focuses mainly on the relationship between the host and the container. However, containers are now part of a complex ecosystem - from container to various repositories and orchestrators - put together with a high level of automation. In particular, container solutions embed automated deployment chains [6] meant to speed up code deployment processes. These chains are often composed of third parties elements, running on different platforms from different providers, raising concerns about code integrity. This can cause multiple vulnerabilities that an adversary exploit to penetrate the system. To the best of our knowledge, while being fundamental for the adoption of containers, the security of their ecosystem has not been fully investigated yet.

We actually focus on Docker's ecosystem for three reasons. First, Docker successfully became the reference on the market of containers and associated *DevOps* ecosystem. In particular, 92% of surveyed people by ClusterHQ and DevOps.com [7] are using or planning to use Docker in a container solution. Second, security is the first barrier to container adoption in production environment [7]. Finally, Docker is already running in some environments which enable experiments and exploring the practicality of some attacks.

In this paper we first review Linux containers and Docker technologies in Section II, and then Docker's security in Section III. We then show that the design of the containers/Docker ecosystem triggers behaviours that lower security compared to the adoption of virtual machine based solutions. We argue in Section IV on the fact that these usages trigger specific vulnerabilities, which is a factor rarely taken into account, and we further point out vulnerabilities affecting current usages of Docker. Conclusions and future work are drawn in Section VI.

II. Containerisation and Dockerisation in a growing ecosystem



(a) Type1 hypervisor (b) Type2 hypervisor (c) Container

Fig. 1: Application runtime models

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

TABLE I: Container solutions

Cloud applications have typically leveraged virtualization. However, the acceleration of the development cycle (agile methods and *DevOps*), the increase in complexity of the application stack (mostly web services and their frameworks), and the market pressure in favor of the densification of applications on servers have triggered the need for a fast, easy-to-use way of pushing code into production.

A. Linux Containers

Containers (Figure 1c) provide near bare metal performance as opposed to virtualization (Figures 1a and 1b) [1] with the further possibility to run seamlessly multiple versions of applications on the same machine. New instances of containers can be created quasi-instantly to face a customer demand peak — which is convenient to spawn applications on-demand or to quickly move a service, for instance when implementing Network Function Virtualization (NFV).

Containers have existed for a long time under various forms, which differ by the level of isolation they provide. For example, BSD jails and chroot can be considered as an early form of container technology. Recent Linux-based container solutions 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 dependences are shown in Table I.

Containers may be integrated in a multi-tenant environment, thus making profit of resource-sharing to increase average hardware use. This goal is achieved by sharing the kernel with the host machine. Indeed, in opposition to virtual machines, 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, hence avoiding code duplication. The absence of kernel and some system libraries makes containers very lightweight (image sizes can shrink to a few megabytes), which makes the boot process very fast.

In the next sections we will discuss the Docker based containers. Indeed Docker popularity and extended privileges on the machines it runs on, make it a target of choice.

B. Docker

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 (Figure 2a). It is also a software that implements this specification (the Docker daemon, named Docker Engine: Figure 2b), a central repository where developers can upload and share their images (the Docker Hub: Figure 2c) and other unofficial repositories (Figure 2d), along with a trademark (Docker Inc.) and bindings with third parties applications (Figure 2e). The build process implies fetching code from external repositories (containing the packages that will be embedded in the images: Figure 2g).

The Docker project is written in *Go* language and was first released in March 2013. Figure 2 summarizes the Docker ecosystem. Docker's main components, i.e.

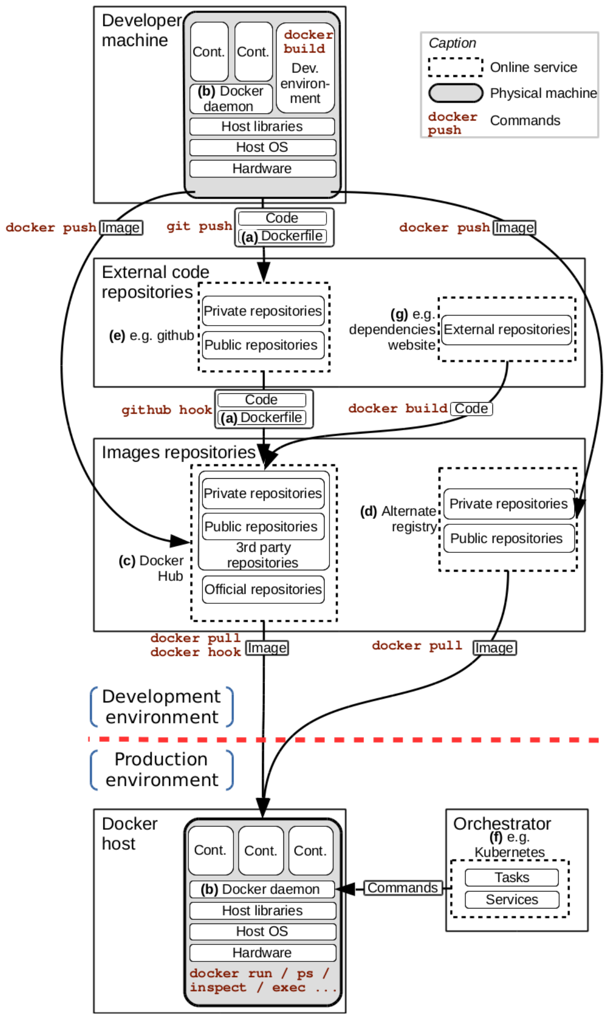


Fig. 2: Overview of the Docker ecosystem. Arrows show code path, with associated commands on them (`docker <action>`)

specification, kernel support, daemon and Docker Hub are presented below.

1) Docker specification

The specification's scope is container images and runtime.

Docker disk 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 filesystem relatively to the previous 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 which are roots of the trees. This structure allows to ship in an image only the modifications specifically related to this image.

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 classical virtual machine 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 (Figure 2a) 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). The image is then built with the `docker build` command.

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 [8] and control groups (cgroups), that were merged starting from the Linux kernel version 2.6.24.

Namespaces split the view processes have of the system. There are currently 6 different namespaces in the kernel, isolating various aspects of the system: PID, IPC, NET, MNT, UTS and USER. Each of these namespaces has its own kernel internal objects related to its type, and provides processes a local instance of some paths in `/proc` and `/sys` filesystems. The Linux namespaces' isolation role is detailed in [3].

Cgroups are a kernel mechanism to restrict the resource usage of a process or group of processes. It aims at preventing 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.

3) The Docker daemon

The Docker software (Figure 2b) itself 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. Alternatively, the daemon can listen on a classical TCP socket.

4) The Docker Hub

The Docker Hub (Figure 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 are "repository".

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 them are official while others are unofficial repositories provided by third parties.

III. Docker security overview

Docker's security relies on three components: 1.) isolation of processes at userspace level managed by the Docker daemon; 2.) enforcement of this isolation by the kernel; and, 3.) network operations security.

A. 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 extended access on some parts of the host to containers — see Section IV-C1.

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, such as the `--icc=false` parameter, which forbids network communications between containers and mitigates the ARP poisoning attacked described before, are available but prevent multi-container applications from operating properly, and hence are rarely used.

B. Host hardening

Host hardening through Linux kernel 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, not restrictive (for instance the `docker-default` Apparmor profile [9] allows full access to filesystem, 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.

C. 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 and the connection to the registry is made over TLS (except if explicitly specified otherwise). Moreover, since version 1.8 (issued in August 2015) the Docker Content Trust [10] architecture allows developers to sign their images before pushing them to a repository. Content Trust relies on TUF (The Update Framework [11]). It is specifically designed to address package manager flaws [12]. 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", themselves used to sign Docker images.

The daemon is remote-controlled through a socket, making it possible to perform any Docker command from another host. By default the socket used to control the daemon is a UNIX socket, located at `/var/run/docker.sock` and owned by `root:docker`, but it can be changed for a TCP socket. 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 on the host. Therefore the connection must be secured with TLS (`--tlsverify`). This enables both encryption and authentication of the two sides of the connection (and adds additional certificate management).

IV. Docker usages security challenges

Most of the security discussions about containers compare them to virtual machines, thus assuming both technologies are equivalent in terms of design. Although this 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. Therefore, it is a key point to develop the Docker typical usages to discuss the security implications of such usages and how they affect Docker's security (Section IV-C).

A. *Docker usages*

We can distinguish three types of Docker usages:

a) Recommended usages, i.e. the usages Docker was designed for, as explained in the official documentation.
b) Wide-spread usages, i.e. the common usages done by application developers and system administrators.
c) PaaS providers usages, i.e. the usages guided by the PaaS providers implementations to cope with both security and integration within their infrastructure.

a) Recommended usages: Docker developers recommend a microservices approach [13], 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 virtual machine: there is no package manager, no init process, no sshd to manage it. All administration tasks (container stop, restart, backups, updates, builds, etc...) have to be performed via the host machine, which implies that the legitimate containers admin has root access to the host.

Docker developers also recommend a reproducible and automated deployment of applications. Docker images should 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 process and the resulting images almost host-agnostic, only depending on the kernel and not on the installed libraries.

b) Wide-spread usages: Some system administrators or developers use Docker as a way of shipping complete virtual environments and updating them on a regular basis, turning their containers into virtual machines. Although this is convenient since it limits system administration tasks to the bare minimum (e.g. `docker pull`), it has several security implications as we will demonstrate below. 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, which is completely opposed to Docker's design. Indeed some of these administration tasks need root access on 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.

c) PaaS providers usages: The integration of Docker provided by the main PaaS providers as of end of 2015 is presented here. The focus is on Amazon Web Services and Google Container Engine as they are two market leaders, and we experimented on them. Both solutions provide similar approaches: virtual machines or cluster of virtual machines are created first, an orchestrator tool is also available to manage the containers inside virtual machines. In this model, the container's admin has full rights on the orchestrator that in turn manages the containers. The microservice approach promoted by *DevOps* and Docker currently needs manual configuration to launch appropriate images on appropriate nodes. This task is automated by orchestrators that manage clusters of virtual machines, themselves running on multiple physical hosts.

B. *Adversary model*

Given the ecosystem and usages 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, they gain root privileges on the related container; from a compromised container, they make a DoS on co-located containers, i.e. containers on the same host OS);
- in-production host OS (e.g. from a compromised container, they gain access to critical host OS files, i.e. a container's escape);
- in-production Docker daemons (e.g. from a compromised host OS, they lower the default security parameters to launch Docker containers);
- the production network (e.g. from a compromised host OS, they redirect 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.

CVE records illustrate that these are relevant targets. Vulnerabilities found in Docker and the libcontainer mostly concern filesystem isolation: chroot escapes (CVE-2014-9357, CVE-2015-3627), path traversals (CVE-2014-6407, CVE-2014-9356, CVE-2014-9358) and access to special file systems on the host (CVE-2015-3630). These specific vulnerabilities are all patched as of Docker 1.6.2. Since container processes often run with UID 0, they have read and write access on the whole host filesystem when they escape, allowing them to overwrite host binaries, which leads to a delayed arbitrary code execution with root privileges.

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.

C. Vulnerabilities affecting Docker's usages

The Docker attack surface encompasses the whole deployment toolchain proposed by Docker, from the build to the execution of the images, including image conception, image distribution process, automated builds, image signature, host configuration and third-party components. In this section, we analyze some Docker vulnerabilities in the light of the usages described in the previous section.

1) Insecure local configuration

Docker's default configuration on local systems following recommended usages is relatively secure as it provides isolation between containers and restricts containers' access to the host. Actually, assuming these isolations mechanisms are working as expected (no implementation vulnerabilities or CVEs), in both the "Recommended" and "PaaS providers" usages, a privilege boundary between the containers and the host machine has been designed. Technical controls supporting the boundary include the isolation of processes through namespaces, resources management through cgroups and by default limited communication capabilities between the containers and the host.

However, the "Wide-spread" usages take advantage of options, either given to the Docker daemon on startup, or given to the command launching a container, that can give extended access to the host to containers. These options trigger security concerns as a side-effect when used with untrusted containers (a candidate attack vector, see Section IV-B). They include:

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

For instance, 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...). The option `--uts=host` lets the container in the same UTS namespace as the host which 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` subdirectories in read/write mode and change the host's kernel parameters, leading to potential vulnerabilities, data leakage, or denial of service.

Along with these runtime container options, several settings on the host have influence on potential attacks. Even basic properties can at least trigger denial of service. For instance, with some storage drivers (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.

As mentioned in the Section IV-B, whatever the usages, containers are an attack vector and therefore represent a potential threat for the host. This is even more relevant in the "Wide-spread" usages, as containers are used as virtual machines and have therefore a bigger attack surface than microservice containers, and are prone to more vulnerabilities leading to attacks such as container's escapes, etc... — see Section IV-B.

2) Weak local access control

Beyond the kernel namespaces, cgroups, Docker dropping capabilities and mount restrictions, Mandatory Access Control 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 [14], explicitly allowing 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 a full access on network devices, filesystems along with a full set of capabilities, and contains a small list of `deny` directives, consisting *de facto* in a blacklist.

These vulnerabilities are relevant to all usages. These vulnerabilities would lead to similar attacks as the ones mentioned in Section IV-C1, e.g. containers's escapes or DoS.

3) Vulnerabilities in the image distribution process

As part of the Docker ecosystem, the distribution of images through the Docker Hub and other registries is a source of vulnerabilities. These are similar to the classical package managers ones [12], therefore we consider here only the automated deployment pipeline perspective.

Automated builds and webhooks proposed by the Docker hub are a key element in the image distribution process. They lead to a pipeline in which each element has full access to the code that will end up in production, and is 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 (e.g. github). 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, which triggers an image pull and container restart on the new image (Docker hooks [6]). With this deployment pipeline, a commit on github will trigger a build of a new image and automatically launch it in production. Optional test steps can be added before production, themselves potentially hosted at yet another provider. In this last case, the Docker Hub makes a first call to a test machine, that will then pull the image, run the tests and send results to the Docker Hub using a callback URL. The build process itself often downloads dependencies from other third-parties repositories, sometimes over an unsecure channel prone to tampering.

This setup adds several external intermediary steps to the code path, and each of them has its own authentication and attack surface, 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 and downloads new images. Our initial intuition was confirmed: adversary's code was put in production 5 minutes and 30s after the adversary's commit on github. This scales to an arbitrary number of machines watching the same Docker Hub repository. Due to space limitations, detailed results will be reported in a different work [15].

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 minutes. Compromise could also happen at the

Docker Hub account level, with the same consequences. Account hijacking is not a new problem, but should be an increasing concern with the multiplication of accounts at different providers.

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 external entities, each of them being capable of compromising the images.

These vulnerabilities are especially relevant in the ”Recommended” and ”PaaS providers” usages. In particular, these usages aim at making an extensive use of automation at all layers to deliver shorter development cycles and continuous delivery.

V. Summary

In the previous sections, we have identified key vulnerabilities:

- insecure configuration;
- weak local access control; and,
- vulnerabilities in the image distribution process.

We have shown that these vulnerabilities are to be assessed according to the usage:

- Docker recommended usages;
- Wide-spread (i.e. casting containers as VM) usages; and,
- PaaS provider usages.

In particular, many vulnerabilities of Docker are due to casting 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.

VI. Conclusion

In this paper, we have shown that the Docker attack surface encompasses the whole deployment toolchain proposed by Docker, from the build phase to the execution of the images, including image conception, image distribution process, automated builds, image signature, host configuration, and third-party configuration. In particular, the analysis we conducted revealed that Docker usages have security implications for both containers and their hosts, and repositories.

We have highlighted that a possible solution to the discussed security issues can be represented by an orchestrator: it could help limiting misuses of Docker through higher levels of abstraction (tasks, replication controllers, remote persistent storage, etc) that completely remove host dependence, enabling better isolation. We are currently investigating orchestrator security issues.

References

[1] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, “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*, PDP ’13, (Washington, DC, USA), pp. 233–240, IEEE Computer Society, 2013.

[2] T. Bui, “Analysis of Docker Security,” 2015. arXiv:1501.02967v1.

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

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

[5] F. Lombardi and R. Di Pietro, “Virtualization and cloud security: Benefits, caveats, and future developments,” in *Cloud Computing* (Z. Mahmood, ed.), Computer Communications and Networks, pp. 237–255, Springer International Publishing, 2014.

[6] “Docker hub: Automated builds and webhooks,” September 2015. https://docs.docker.com/docker-hub/builds.

[7] ClusterHQ and DevOps.com, “The current state of container usage. identifying and eliminating barriers to adoption,” June 2015. https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2015.pdf.

[8] E. Biederman, “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.

[9] “`docker-default` apparmor profile,” June 2014. https://wikitech.wikimedia.org/wiki/Docker/apparmor.

[10] “Docker content trust, official documentation,” September 2015. https://docs.docker.com/security/trust/content_trust.

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

[12] J. Cappos, J. Samuel, S. M. Baker, and J. H. Hartman, “A look in the mirror: attacks on package managers,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security* (P. Ning, P. F. Syverson, and S. Jha, eds.), pp. 565–574, ACM, 2008.

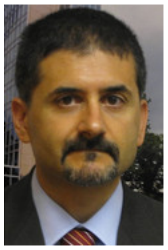
[13] “Docker best practices,” September 2015. https://docs.docker.com/articles/dockerfile_best-practices.

[14] “Novell apparmor administration guide,” October 2007. https://www.suse.com/documentation/apparmor/pdfdoc/book_apparmor21_admin/book_apparmor21_admin.pdf.

[15] T. Combe, A. Martin, and R. Di Pietro, "Containers - vulnerability analysis," tech. rep. http://ricerca.mat.uniroma3.it/users/dipietro/containers_security.pdf.

Theo Combe is a graduate student from the Ecole polytechnique, France, currently following a double degree at Telecom ParisTech, where he studies networks and cybersecurity. In 2015, he worked at Thales Communications & Security in the SDN-NFV research group, as a part-time project along with his studies, and later at Bell Labs, on Docker and containers security. Email: theo-nokia@sutell.fr

Antony Martin is a Member of the Technical Staff in the Security department at Nokia Bell Labs, Nozay, France. His research interests include network security, virtualization, cloud computing and NFV. He holds CISSP, GCIH, CCDA, CCNA, Snort professional certifications and 14 active patents. Email: antony.martin@nokia.com



Prof. Dr. Roberto Di Pietro is Security Research Group Head and PI at Nokia Bell Labs, Nozay, France. His research interests include security, privacy, distributed systems, computer forensics, and analytics. He has 160+ scientific publications on these topics, totaling 4600+ citations. Email: roberto.di_pietro@nokia.com