

To Docker or Not to Docker: A Security Perspective

Theo Combe, Telecom Paris-Tech

Antony Martin and Roberto Di Pietro, Nokia Bell Labs

Container solutions such as the popular Docker environment provide more flexibility than virtual machines and offer near-native performance in cloud-based infrastructures. However, Docker and its current usage scenarios entail security vulnerabilities that must be addressed.

Cloud computing is inherently rooted in virtualization technologies. Recently, new lightweight virtualization technologies such as containers have become increasingly popular and are nowadays an essential part of cloud offerings. Containers also tightly integrate into the host operating system, reducing the software overhead imposed by virtual machines (VMs).¹ However, this tighter integration also increases the attack surface, raising security concerns.

Existing work on container security focuses mainly on the relationship between the host and the container.^{2–5} However, containers are now part of a complex ecosystem, which includes containers and various repositories and orchestrators, that is highly automated. In particular, container solutions embed automated deployment chains⁶ that are meant to speed up the code deployment processes. These chains are often composed of third-party elements running on different platforms provided by different providers, raising concerns about code integrity. This can cause multiple vulnerabilities that an adversary could exploit to penetrate the system.

To the best of our knowledge, container ecosystem security has yet to be fully investigated, despite being fundamental to container adoption. Here, we address that gap and focus our investigation on the Docker ecosystem for three reasons. First, Docker successfully became the reference on both the market of containers and the associated DevOps ecosystem. In particular, 92 percent of people surveyed by ClusterHQ and DevOps.com are using or planning to use Docker in a container solution.⁷ Second, security is the first barrier to container adoption in the production environment,⁷ Docker being no exception

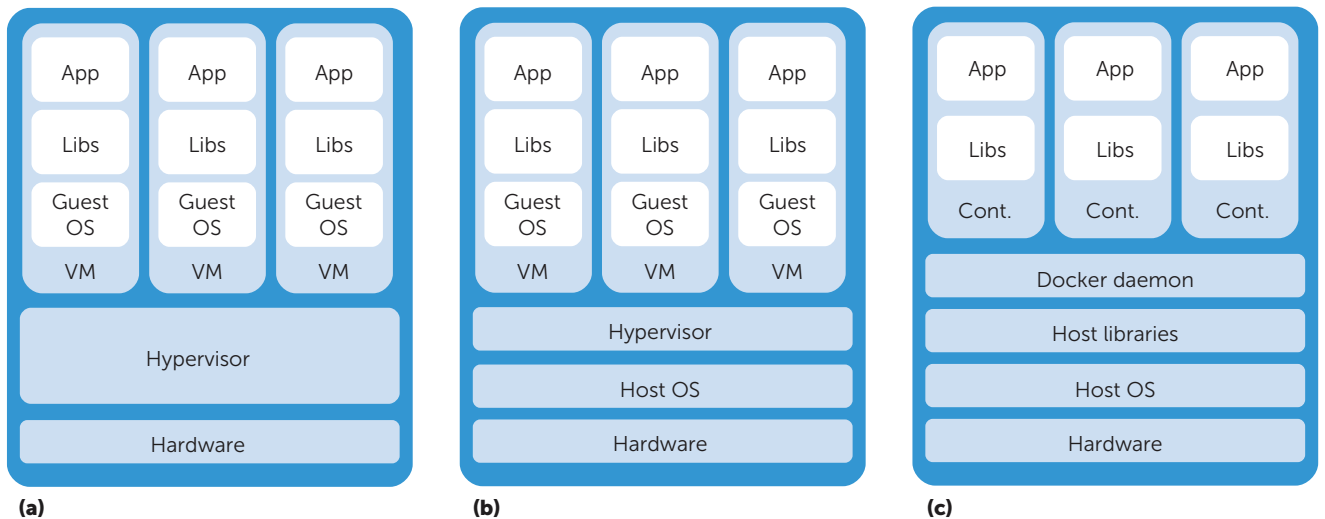


FIGURE 1. Comparing various application runtime models: (a) a type 1 hypervisor, (b) a type 2 hypervisor, and (c) a container.

in this. Finally, Docker is already running in some environments, making it possible to run experiments and explore the practicality of some attacks.

Containerization and Dockerization in a Growing Ecosystem

Cloud applications have typically leveraged virtualization. However, several factors—including acceleration of the development cycle (such as agile methods and DevOps), an increasingly complex application stack (mostly Web services and their frameworks), and market pressure to densify applications on servers—have triggered the need for a fast, easy-to-use way of pushing code into production.

Linux Containers

Figure 1 shows how virtualization hypervisors (Figures 1a and 1b) compare to a container (Figure 1c), which provides near-bare-metal performance¹ and offers the possibility of seamlessly running 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 for spawning applications on-demand or quickly moving a service, such as when implementing network function virtualization (NFV).

Containers have long existed in various forms that differ by the level of isolation they provide. For

example, Berkeley Software Distribution (BSD) jails and chroot can be considered an early form of container technology. Recent Linux-based container solutions rely on kernel support—that is, a userspace library to provide an interface to syscalls and front-end applications. There are two main kernel implementations: Linux container (LXC) implementations using cgroups and namespaces, and the OpenVZ patch. Table 1 shows the most popular implementations and their dependences.

Containers can be integrated in a multitenant environment, thus profiting from resource sharing to increase average hardware use. This is achieved by sharing the kernel with the host machine. Indeed, unlike VMs, containers don't embed their own kernel, but rather 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 (such as libraries) with the host, hence avoiding code duplication. The absence of kernel and some system libraries make containers very lightweight (image sizes can shrink to a few megabytes), which enables a quick boot process.

Docker

As Figure 2 shows, the Docker ecosystem includes various components. Docker provides a specification

Table 1. Container solutions.

Base	Container	Library	Kernel dependence	Other dependencies
Linux containers (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: checkpoint/restore in userspace (CRIU), ploop, Virtuozzo container memory management daemon (VCMMMD)

for container images and runtime, including Dockerfiles that allow a reproducible building process (Figure 2a). Docker software implements this specification using the Docker daemon, known as the *Docker engine*. The repositories include a central repository, the *Docker hub*, which lets developers upload and share their images, along with a trademark and bindings with third-party applications (Figure 2b). Finally, the build process fetches code from external repositories and holds the packages that will be embedded in the images (Figure 2c). Docker is written in the Go language and was first released in March 2013.

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 the JavaScript Object Notation (JSON) format. The images are stored at `/var/lib/docker/<driver>/`, where `<driver>` stands for the storage driver being used—such as advanced multi-layered unification filesystem (aufs), B-tree file system (Btrfs), *virtual filesystem switch* (VFS), device mapper, or OverlayFS. Each layer contains the filesystem modifications relative to the previous layer, starting from a base image (typically, a lightweight Linux distribution). This lightweight Linux distribution organizes the images in trees; each image has a parent, except for the base images, which are the roots of the trees. This structure allows Docker to ship in an image only the modifications specifically related to it.

Docker can build images in two ways. It can launch a container from an existing image (`docker run`), perform modifications and installations inside the container, and then stop the container and save its state as a new image (`docker commit`). This

process is close to the classical VM installation, but must be performed at each image rebuild (such as for updates); because the base image is standardized, the sequence of commands is exactly the same. To automate this process, Dockerfiles (Figure 2a) let users specify a base image and a sequence of commands to be performed to build the image, along with other options—such as exposed ports—specific to the image. The image is then built with the `docker build` command.

Docker internals. Docker containers create a wrapped, controlled environment on the host machine in which arbitrary code can be (ideally) run safely. This isolation is achieved through two main kernel features—kernel namespaces⁸ and control groups (cgroups)—that were merged starting from the Linux kernel version 2.6.24. Namespaces are used to split the view that processes have of the system. Currently, the kernel has six different namespaces—PID, IPC, NET, MNT, UTS, and USER—that isolate various aspects of the system. Each of these namespaces has its own kernel internal objects related to its type, and each gives processes a local instance of some paths in the `/proc` and `/sys` filesystems. The Linux namespaces' isolation role is detailed elsewhere.³ The cgroups are a kernel mechanism to restrict the resource usage of a process or group of processes. Their goal is to 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.

The Docker daemon. The Docker software runs as a daemon on the host machine. It can launch contain-

ers, control their isolation level (cgroups, namespaces, capabilities restrictions, and SELinux/Apparmor profiles), monitor them to trigger actions (such as restart), and spawn shells into running containers for administration purposes. The software can change iptables rules on the host and create network interfaces. It's also responsible for managing container images, including pulling and pushing images on a remote registry (such as the Docker hub), building images from Dockerfiles, and signing images. 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.

The Docker hub. The Docker hub online repository lets developers upload their Docker images and lets users download them. Developers can sign up for a free account, in which all repositories are public, or for a paid account, which lets them create private repositories. Developer repositories are namespaced—that is, their name is “developer/repository.” Official repositories also exist, directly provided by Docker Inc.; these are called “repository.” The Docker daemon, hub, and repositories are similar to a package manager, with a local daemon installing software on both the host and the remote repositories. Some of these repositories are official, while others are unofficial and provided by third parties.

Docker Security Overview

Docker security relies on three factors: isolation of processes at the userspace level managed by the Docker daemon, enforcement of this isolation by the kernel, and network operations security.

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 cgroups limitations aren't; they 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 Address Resolution Protocol (ARP) poisoning attacks between containers on the same host.

However, as we describe in more detail later, Docker's global security can be lowered by options, triggered at container launch, that give extended access on some parts of the host to containers. Additionally, security configuration can be set globally through options passed to the Docker daemon.

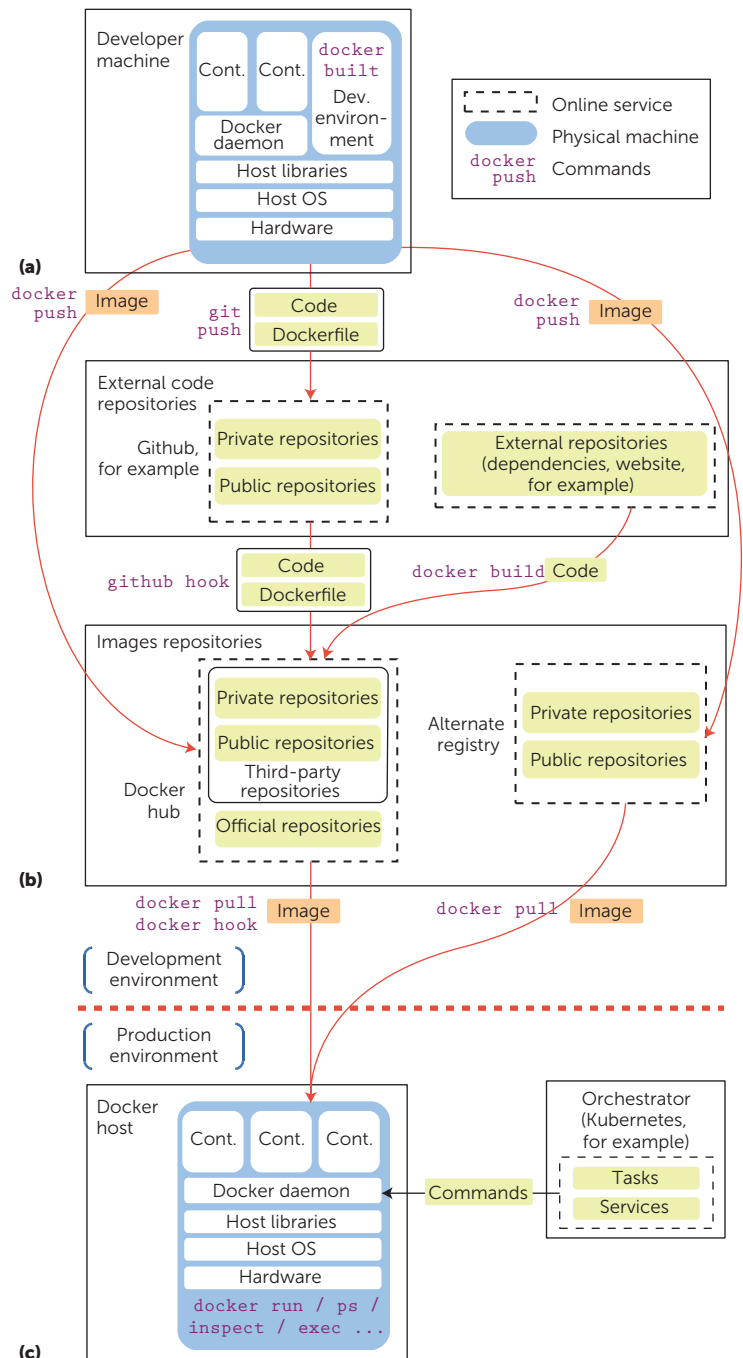


FIGURE 2. The Docker ecosystem. (a) Docker specifies container images and runtime, including Dockerfiles that enable a reproducible building process. (b) The Docker repositories. (c) The build process. Arrows show the code path and associated commands (docker <action>).

This includes options lowering security, such as the `-insecure-registry` option, which disables the Transport Layer Security (TLS) certificate check on a particular registry. Options that increase security—such as the `-icc=false` parameter, which forbids

network communications between containers and mitigates the ARP poisoning attack—are available, but they prevent multicontainer applications from operating properly, and hence are rarely used.

Host Hardening

Host hardening through Linux kernel security modules enforces security-related limitation constraints imposed on containers (such as compromising a container and escaping to the host operating system). Currently SELinux, Apparmor, and Seccomp are supported with available default profiles. These profiles are generic, not restrictive. The `docker-default Apparmor profile`⁹ (<https://wikitech.wikimedia.org/wiki/Docker/apparmor>), for example, allows full access to the filesystem, network, and all capabilities of Docker containers. Similarly, the default SELinux policy puts all Docker objects in the same domain. Therefore, while default hardening protects the host from containers, it doesn't protect containers from other containers. This security aspect can be addressed by writing specific profiles that depend individually on the containers.

Network Security

Docker uses network resources for image distribution and remote control of the Docker daemon.

To distribute images, Docker verifies images downloaded from a remote repository with a hash and the connection to the registry is made over TLS (unless explicitly specified otherwise). Moreover, the Docker Content Trust architecture now lets developers sign their images before pushing them to a repository.¹⁰ Content Trust relies on the update framework (TUF),¹¹ which was specifically designed to address package manager flaws.¹² TUF can recover from a key compromise, mitigate replay attacks by embedding expiration timestamps in signed images, and so on. The tradeoff is complex key management; TUF actually implements a public-key infrastructure in which each developer owns a root key (“offline key”) that is used to sign “signing keys” that are used to sign Docker images.

The Docker 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 to a TCP socket. Access to this socket lets attackers pull and run any container in privileged mode, thereby giving them root access to the host. In case of a Unix socket, a user member of the `docker` group can gain root privileges; when a TCP socket is used, any con-

nection to this socket can give root privileges on the host. Therefore, the connection must be secured with TLS (`-tlsverify`), which enables both encryption and authentication of the two sides of the connection (and requires additional certificate management).

Docker Usages: Security Challenges

Most of the security discussions about containers compare them to VMs, thus assuming both technologies are equivalent in terms of design. Although this is the aim of some container technologies (such as OpenVZ, which is used to spawn virtual private servers), recent lightweight container solutions such as Docker were designed to achieve completely different objectives than those of VMs. Therefore, it's important to develop Docker's typical usages to discuss their security implications and how they affect Docker's security.

Docker Usages

We can distinguish three types of Docker usages.

Recommended usages are those that Docker was designed for, as explained in the official documentation. Docker developers recommend a microservices approach¹³—that is, a container must host a single service, in a single process or in a daemon spawning children. Therefore, a Docker container isn't considered a VM: there's no package manager, no `init` process, no `sshd` to manage it. All administration tasks (container stop, restart, backups, updates, builds, and so on) must be performed via the host machine, which implies that the legitimate container's 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, depending only on the kernel and not on the installed libraries.

Widespread usages include common usages of Docker by application developers and system administrators. Some system administrators or developers use Docker as a way to ship complete virtual environments and update them regularly, turning their containers into VMs. Although this is convenient because it limits system administration tasks to the bare minimum (such as `docker pull`), as we describe later, it has several security implications. With containers embedding enough software to run a full system (logging daemon, `ssh` server, and even sometimes an `init` process), it's tempting to perform administration tasks from within the

container, which is completely opposed to Docker's design. Indeed, some of these administration tasks require root access on the container, while other administration actions—such as mounting a volume in a container—could require extra capabilities that Docker drops by default.

Platform as a service (PaaS) usages are guided by PaaS implementations to cope with security and infrastructure integration issues. As of the end of 2015, the main PaaS integrated Docker. We focus here on Amazon Web Services and Google Container Engine, the two market leaders that we experimented on. Both solutions provide similar approaches: a VM or cluster of VMs is created, with an orchestrator tool available to manage the containers inside the VMs. In this model, the container's admin has full rights on the orchestrator. The microservices approach promoted by DevOps and Docker currently requires manual configuration to launch appropriate images on appropriate nodes. This task is automated by orchestrators that manage clusters of VMs, which themselves run on multiple physical hosts.

Adversary Model

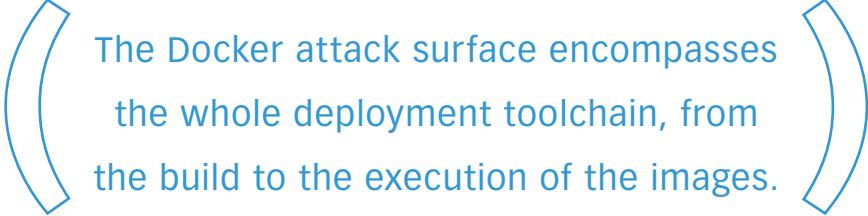
Given the ecosystem and usages description, we consider two main categories of adversaries: direct and indirect.

Direct adversaries can sniff, block, inject, or modify network and system communications, and they directly target the production machines. Locally or remotely, direct adversaries can compromise several system components:

- *In-production containers.* With containers from an Internet-facing container service, for example, attackers can gain root privileges on a related container. Then, from the compromised container, they can make a denial-of-service (DoS) attack on containers located on the same host operating system.
- *In-production host operating system.* From a compromised container, for example, attackers can gain access to critical host operating system files—that is, launch a container escape attack.
- *In-production Docker daemons.* In this case, for example, attackers might lower the default security parameters to launch Docker containers from a compromised host operating system.
- *The production network.* From a compromised host operating system, attackers can redirect network traffic and so on.

Indirect adversaries have the same capabilities as direct ones, but they leverage the Docker ecosystem (such as the code and images repositories) to reach the production environment.

Depending on the attack phase, we identified the following targets: containers, host operating system, collocated containers, code repositories, images repositories, and the management network. MITRE's Common Vulnerabilities and Exposures (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, and



The Docker attack surface encompasses the whole deployment toolchain, from the build to the execution of the images.

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. Because container processes often run with user ID 0, they have read and write access on the whole host filesystem when they escape, which lets them overwrite host binaries, leading to a delayed arbitrary code execution with root privileges.

To subvert a Dockerized environment, we consider a subset of all the potential attack vectors—the Docker containers, code repositories, and images repositories—primarily because they're associated with publicly available services and interfaces. Other attack vectors might include the host operating system, management network, or physical access to systems.

Vulnerabilities Affecting Docker 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, the image distribution process, automated builds, image signature, host configuration, and third-party components.

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. Assuming these isolation mechanisms

are working as expected in both the recommended and PaaS usages—that is, there are no implementation vulnerabilities or CVEs—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.

In contrast, widespread usages take advantage of options—given either to the Docker daemon on startup or to the command launching a container—that give containers extended access to the host. When used with untrusted containers, these options trigger many security concerns, including the following:

- options giving extended access to the host to containers (`-net=host`, `-uts=host`, `-privileged`, and so on);
- the mounting of sensitive host directories in containers;
- TLS configuration of remote image registries;
- permissions on the Docker control socket; and
- cgroups activation (disabled by default).

For instance, when given the option `-net=host` at container launch, Docker doesn't place the container into a separate NET namespace; it therefore gives the container full access to the host's network stack (enabling network sniffing, reconfiguration, and so on). The option `-uts=host` lets the container in the same UTS namespace as the host, which lets the container see and change the host's name and domain. The option `-cap-add=<CAP>` gives the container the specified capability, thus making it potentially more harmful to the host. With `-cap-add=SYS_ADMIN`, a container can, for example, remount `/proc` and `/sys` subdirectories in read/write mode and change the host's kernel parameters, leading to potential vulnerabilities, data leakage, or DoS.

Along with these runtime container options, several settings on the host can influence potential attacks. Even basic properties can at a minimum trigger DoS. For instance, when using some storage drivers (aufs), Docker doesn't limit containers' disk usage. A container with a storage volume can fill up this volume and affect other containers on the same host—or even the host itself—if the Docker storage located at `/var/lib/docker` isn't mounted on a separate partition.

As mentioned earlier, whatever the usages are, containers are an attack vector and therefore represent a potential threat for the host. This is even

more relevant in widespread usages, where containers are used as VMs and thus have a bigger attack surface than microservice containers. They also have more vulnerabilities, leading to attacks such as container escapes.

Weak local access control. Beyond the kernel namespaces, cgroups, Docker dropping capabilities, and mount restrictions, mandatory access control (MAC) enforces constraints if the normal execution flow isn't respected. This approach is visible in the `docker-default` Apparmor policy. However, the MAC profiles for containers have room for improvement. In particular, Apparmor profiles typically behave as whitelists,¹⁴ explicitly identifying which 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 complete access on network devices and filesystems with a full set of capabilities, and contains a small list of deny directives, which constitute a de facto blacklist.

These vulnerabilities are relevant to all usages and could lead to the attacks mentioned earlier, such as DoS or container escapes.

Image distribution vulnerabilities. The distribution of images through the Docker hub and other registries in the Docker ecosystem is a source of vulnerabilities. Because these vulnerabilities are similar to classical package managers,¹² we consider only the automated deployment pipeline perspective here.

Automated builds and webhooks proposed by the Docker hub are key elements 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 are 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 (such as 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. This triggers an image pull and a container restart on the new image (through Docker hooks; see <https://docs.docker.com/docker-hub/webhooks>).

In this deployment pipeline, a commit on github will trigger a build of a new image and automatically launch it into production. Optional test steps can be added before production, which might themselves be hosted at yet another provider. In this case, the Docker hub makes a first call to a test machine that will then pull the image, run the tests, and send re-

sults to the Docker hub using a callback URL. The build process itself often downloads dependencies from other third-party repositories, sometimes over an insecure channel prone to tampering.

This setup adds several external intermediary steps to the code path, each of which 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 that included a Docker hub account, a github account, a development machine, and a production machine. The assumption was that adversaries would use the Docker ecosystem to put a backdoored Docker container in production. More precisely, we assumed that adversaries had successfully compromised some code on the code repository (for instance, via a successful phishing attack).

Because of network restrictions (our server was behind a corporate proxy that did not allow us to be reached directly from the Internet on a public IP address and a port) our servers couldn't be reached by webhooks, so we wrote a script to monitor both our repository on the Docker hub and downloads of new images. Our initial intuition was confirmed: the adversaries' code was put in production five-and-a-half minutes after their commit on github. It's worth noting that this attack can scale to an arbitrary number of machines watching the same Docker hub repository. Given space limitations, we report more detailed results elsewhere.¹⁵

Although 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 isn't a new problem, but it should be an increasing concern with the multiplication of accounts at different providers.

Moreover, although the code path is usually secured using TLS communications (and always is with Docker), it's not the case with API calls that trigger builds and callbacks. Tampering with these data can lead to erroneous test results, unwanted restarts of containers, and so on. Additionally, such a setup isn't compatible with the Content Trust scheme, because 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 this case

trust is split across external entities, each of which is capable of compromising the images.

These vulnerabilities are especially relevant in the recommended and PaaS usages, which aim at an extensive use of automation at all layers to deliver shorter development cycles and continuous delivery.

An orchestrator could solve some of the security issues raised here, helping limit misuses of Docker through higher levels of abstraction—including tasks, replication controllers, and remote persistent storage—that completely remove host dependence and thus enable better isolation. Orchestrator also brings automation as a key enabler to repeatable, predictable, auditable security and continuously improved security. We are currently investigating orchestrator security issues through experiments.

Acknowledgments

We thank the anonymous reviewers for their useful comments.

References

1. M.G. Xavier et al., "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," *Proc. 21st Euromicro Int'l Conf. Parallel, Distributed, and Network-Based Processing (PDP 13)*, 2013, pp. 233–240.
2. T. Bui, "Analysis of Docker Security," 2015, arXiv:1501.02967v1.
3. E. Reshetova et al., "Security of OS-Level Virtualization Technologies," *Proc. Nordic Conf. Secure IT Systems*, 2014, pp. 77–93.
4. R. Di Pietro and F. Lombardi, *Security for Cloud Computing*, Artec House, 2015.
5. F. Lombardi and R. Di Pietro, "Virtualization and Cloud Security: Benefits, Caveats, and Future Developments," *Cloud Computing*, Z. Mahmood, ed., Springer Int'l Publishing, 2014, pp. 237–255.
6. Docker, "Automated Builds on Docker Hub," *Docker User Guide*, 2016; <https://docs.docker.com/docker-hub/builds>.
7. ClusterHQ and DevOps.com, *The Current State of Container Usage: Identifying and Eliminating Barriers to Adoption*, survey, June 2015; <https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2015.pdf>.
8. E.W. Biederman, "Multiple Instances of the Global Linux Namespaces," *Proc. Linux Symp.*, vol. 1, 2006, pp. 101–112.

10. Docker, "Content Trust in Docker," *Docker User Guide*, 2016; https://docs.docker.com/engine/security/trust/content_trust.
11. J. Samuel et al., "Survivable Key Compromise in Software Update Systems," *Proc. 17th ACM Conf. Computer and Comm. Security (CCS 10)*, 2010, pp. 61–72.
12. J. Capps et al., "A Look in the Mirror: Attacks on Package Managers," *Proc. 15th ACM Conf. Computer and Comm. Security*, P. Ning, P.F. Syverson, and S. Jha, eds., 2008, pp. 565–574.
13. Docker, "Best Practices for Writing Dockerfiles," *Docker User Guide*, 2016; https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices.
14. Novell, *Novell AppArmor Administration Guide*, Oct. 2007; 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. report, Nokia Bell Labs; http://ricerca.mat.uniroma3.it/users/dipietro/containers_security.pdf.

THEO COMBE is a graduate student at the Ecole Polytechnique, France, and is pursuing a double degree in networks and cybersecurity at Telecom Paris-Tech. His research interests include networked systems

security, telecommunications, and programming. Contact him at theo-nokia@sutell.fr.

ANTONY MARTIN is a security analyst and 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 network function virtualization. Martin has an engineering degree in telecommunications from Telecom Lille engineering school and holds a number of technical certifications. Contact him at antonymartin.pro@gmail.com.

ROBERTO DI PIETRO is security research global head at Nokia Bell Labs, Paris-Saclay, France, and a part-time professor of computer science (security) at the University of Padova, Italy. His research interests include security, privacy, distributed systems, computer forensics, and analytics. Di Pietro has a PhD in computer science from the University of Roma "La Sapienza." Contact him at roberto.di_pietro@nokia-bell-labs.com.

myCS

Read your subscriptions through the myCS publications portal at <http://mycs.computer.org>.

IEEE  computer society

CELEBRATING
70
YEARS

PURPOSE: The IEEE Computer Society is the world's largest association of computing professionals and is the leading provider of technical information in the field.

MEMBERSHIP: Members receive the monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.

OMBUDSMAN: Email ombudsman@computer.org.

COMPUTER SOCIETY WEBSITE: www.computer.org

Next Board Meeting: 13–14 November 2016, New Brunswick, NJ, USA

EXECUTIVE COMMITTEE

President: Roger U. Fujii

President-Elect: Jean-Luc Gaudiot; **Past President:** Thomas M. Conte;

Secretary: Gregory T. Byrd; **Treasurer:** Forrest Shull; **VP, Professional and Educational**

Activities: Andy T. Chen; **VP, Member & Geographic Activities:** Nita K. Patel;

VP, Publications: David S. Ebert; **VP, Standards Activities:** Mark Paulk;

VP, Technical & Conference Activities: Hausi A. Müller; **2016 IEEE Director & Delegate**

Division VIII: John W. Walz; **2016 IEEE Director & Delegate Division V:** Harold Javid;

2017 IEEE Director-Elect & Delegate Division VIII: Dejan S. Milošević

BOARD OF GOVERNORS

Term Expiring 2016: David A. Bader, Pierre Bourque, Dennis J. Frailey, Jill I. Gostin, Atsuhiko Goto, Rob Reilly, Christina M. Schober

Term Expiring 2017: David Lomet, Ming C. Lin, Gregory T. Byrd, Alfredo Benso, Forrest Shull, Fabrizio Lombardi, Hausi A. Müller

Term Expiring 2018: Ann DeMarle, Fred Douglass, Vladimir Getov, Bruce M. McMillin, Cecilia Metra, Kunio Uchiyama, Stefano Zanero

EXECUTIVE STAFF

Executive Director: Angela R. Burgess; **Director, Governance & Associate Executive**

Director: Anne Marie Kelly; **Director, Finance & Accounting:** Sunny Hwang;

Director, Information Technology & Services: Sumit Kacker; **Director, Membership**

Development: Eric Berkowitz; **Director, Products & Services:** Evan M. Butterfield;

Director, Sales & Marketing: Chris Jensen

COMPUTER SOCIETY OFFICES

Washington, D.C.: 2001 L St., Ste. 700, Washington, D.C. 20036-4928

Phone: +1 202 371 0101 • **Fax:** +1 202 728 9614 • **Email:** hq.ofc@computer.org

Los Alamitos: 10662 Los Vaqueros Circle, Los Alamitos, CA 90720

Phone: +1 714 821 8380 • **Email:** help@computer.org

MEMBERSHIP & PUBLICATION ORDERS

Phone: +1 800 272 6657 • **Fax:** +1 714 821 4641 • **Email:** help@computer.org

Asia/Pacific: Watanabe Building, 1-4-2 Minami-Aoyama, Minato-ku, Tokyo 107-0062,

Japan • **Phone:** +81 3 3408 3118 • **Fax:** +81 3 3408 3553 • **Email:** tokyo.ofc@computer.org

IEEE BOARD OF DIRECTORS

President & CEO: Barry L. Shoop; **President-Elect:** Karen Bartleson; **Past President:**

Howard E. Michel; **Secretary:** Parviz Famouri; **Treasurer:** Jerry L. Hudgins; **Director**

& President, IEEE-USA: Peter Alan Eckstein; **Director & President, Standards**

Association: Bruce P. Kraemer; **Director & VP, Educational Activities:** S.K. Ramesh;

Director & VP, Membership and Geographic Activities: Wai-Choong (Lawrence) Wong;

Director & VP, Publication Services and Products: Sheila Hemami; **Director & VP,**

Technical Activities: Jose M.F. Moura; **Director & Delegate Division V:** Harold Javid;

Director & Delegate Division VIII: John W. Walz

revised 10 June 2016

