Analysis of Docker Security

Article · January 2015

Source: arXiv

CITATIONS
126

READS
5,516

1 author:

Thanh Bui
Aalto University
14 PUBLICATIONS **156** CITATIONS

SEE PROFILE

Analysis of Docker Security

Thanh Bui
Aalto University School of Science
thanh.bui@aalto.fi

Abstract

Over the last few years, the use of virtualization technologies has increased dramatically. This makes the demand for efficient and secure virtualization solutions become more obvious. Container-based virtualization and hypervisor-based virtualization are two main types of virtualization technologies that have emerged to the market. Of these two classes, container-based virtualization is able to provide a more lightweight and efficient virtual environment, but not without security concerns. In this paper, we analyze the security level of Docker, a well-known representative of container-based approaches. The analysis considers two areas: (1) the internal security of Docker, and (2) how Docker interacts with the security features of the Linux kernel, such as SELinux and AppArmor, in order to harden the host system. Furthermore, the paper also discusses and identifies what could be done when using Docker to increase its level of security.

KEYWORDS: Containers, Docker, Security

1 Introduction

The last decade has seen an explosion of development in the area of virtualization technologies, which allow the partitioning of a computer system into multiple isolated virtual environments. The technologies offer substantial benefits that have been driving their development rapidly. One of the most common reasons for adopting virtualization technologies is *server virtualization* in data centers. With server virtualization, an administrator can create one or more virtual system instances on a single server. These virtual systems operate as real physical servers and can be rented out on a subscription basis. Amazon EC2, Rackspace, and DreamHost are some popular instances of such data center service providers. Another common use is for *desktop virtualization*, where one computer can run several OS instances. Desktop virtualization provides support for applications that can run only on a specific OS.

The growth in the use of virtualization technologies promotes the demand for a virtualization solution which can provide dense, scalable, and secure user environments. A large number of virtualization solutions have emerged to the market. They can be classified into two major classes: container-based virtualization and hypervisor-based virtualization. Of these two classes, container-based virtualization is able to provide a more lightweight and efficient virtual environment. It allows ten times more virtual environments to run on a physical server compared to hypervisor-based virtualization [19]. However, container-based virtualization also comes with security concerns.

In this paper, we analyze the security level of Docker [17], a well-known representative of container-based virtualization approach. We consider two areas: (1) the internal security of Docker, and (2) how Docker interacts with the security features of the Linux kernel, such as SELinux and AppArmor, in order to harden the host system. The analysis examined the internal security of Docker based on the level of isolation Docker can provide to its virtual environments. The interaction between Docker and the security features of the kernel was estimated based on how the features are supported by Docker. To the best of our knowledge, Docker is a relatively new technology, and this is one of the first analyses of this kind that focus on its security aspects.

The paper is structured as follows: Section 2 provides a high-level view of the two classes of virtualization solutions. Section 3 gives an overview of Docker and its underlying technologies. Section 4 presents our analysis of Docker security, and then in Section 5, we discuss the security level of Docker and what could be done to raise its level of security. The paper concludes with a summary in Section 6.

2 Virtualization Approaches

Most of the virtualization technologies can be classified into two major approaches: container-based virtualization and hypervisor-based virtualization. The former provides virtualization at the operating system level, while the latter provides virtualization at the hardware level. Each of the approaches has its own advantages and disadvantages, which are described in this section.

Container-based virtualization is a lightweight virtualization approach using the host kernel to run multiple virtual environments. These virtual environments are often referred to as *containers*. Linux-VServer [31], OpenVZ [11], and Linux Container (LXC) [10] are the three main representatives of this approach. The general architecture of a container-based virtualization solution is illustrated in Fig. 1. Container-based virtualization virtualizes at the operating system level, thus allowing multiple applications to operate without redundantly running other operating system kernels on the host. Its containers look like normal processes from outside, which run on top of the kernel shared with the host machine. They provide isolated environments with necessary resources to execute applications. These resources can be either shared with the host or installed separately inside the container.

Hypervisor-based virtualization solutions provide virtu-

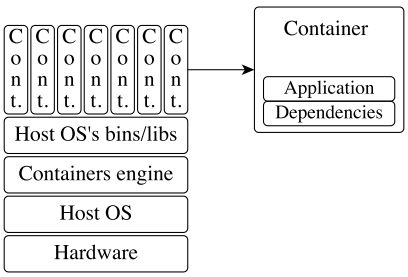

Figure 1: Architecture of Container-based Virtualization

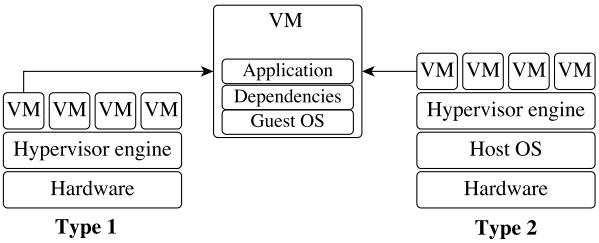

Figure 2: Architecture of Hypervisor-based Virtualization

alization at the hardware level. In contrast to container-based virtualization, a hypervisor establishes complete virtual machines (VMs) on top of the host operating system (Fig. 2). Each virtual machine comprises of not only an application and its dependencies, but also an entire guest OS along with a separate kernel. There are two classes of hypervisors: the *Type 1* hypervisor, also known as the bare metal hypervisor, which works directly on top of the underlying hardware of the host, and the *Type 2* hypervisor, also known as the hosted hypervisor, which works on top of the host operating system [26]. Xen [18] is an example of the former, while KVM [25] is of the latter. Since the *Type 1* hypervisor does not include an extra layer of the host OS, it provides better performance than the *Type 2* hypervisor.

The differences in the architecture bring some benefits to container-based virtualization over hypervisor-based virtualization. First, container-based virtualization can provide *higherer density* of virtual environments. Since a container does not include an entire OS, the size and the required resources to run an application in a container are less than that of a VM running the same application. As a result, more containers than traditional virtual machines can be deployed on the same host. Secondly, container-based virtualization also offers *better performance*. This has been demonstrated by experiments in some studies [32, 28, 27, 21]. These studies show that the performance of container-based virtualization is better than with hypervisor-based virtualization in most cases, and it is almost as good as native applications.

However, despite all of the mentioned advantages, container-based virtualization is unable to support a variety of environments in the way hypervisor-based virtualization does since all the environments of the containers must be of the same type as that of the host. For example, Windows containers cannot be run on top of a Linux host.

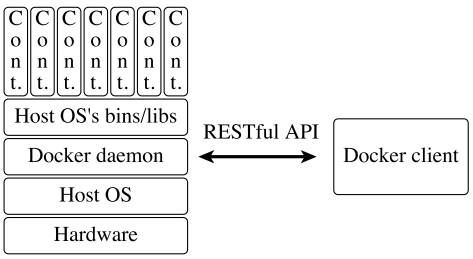

Figure 3: Architecture of Docker engine

3 Docker Overview

Docker is an open source container technology with the ability "to build, ship, and run distributed applications" [17]. It has been used in some popular applications, such as Spotify, Yelp, and Ebay.

Although container technologies have been around for more than a decade, Docker - a relatively new candidate - is currently one of the most successful technologies since it comes with new abilities that earlier technologies did not possess. First, it provides interfaces to simply and safely create and control containers. Secondly, developers can pack applications into lightweight Docker containers which can operate on almost anywhere without modification. Furthermore, Docker can deploy more virtual environments than other technologies can on the same hardware [19]. Last but not least, Docker cooperates well with third-party tools, which simplify the management and deployment process of Docker containers. DevOps tools, such as Puppet [13], Ansible [1], and Vagrant [16] can integrate with Docker, thus making Docker containers to be easily deployed to a cloud. Moreover, many orchestration tools, such as Mesos [22], Shipyard [15], and Kubernetes [7], also support Docker containers. These tools provide an abstract layer of resources management and scheduling over Docker.

Docker consists of two major components: *Docker engine* and *Docker Hub*. The former is an open source virtualization solution, while the latter is a Software-as-a-Service platform for sharing Docker images. The following sections describe in details these two components.

3.1 Docker Engine

Docker engine is a lightweight and portable packaging tool [17] which relies on container-based virtualization. Therefore, the architecture of the Docker engine (Fig. 3) is similar to that of container-based virtualization in general. The Docker containers run on top of the *Docker daemon* which is in charge of executing and managing all of the Docker containers. The *Docker client*, which provides an user interface for interacting with containers to Docker users, accepts commands from the users and then sends it to the Docker daemon through RESTful APIs. Using this method of communication enables the Docker client to run on the same host as the containers, or even on different hosts.

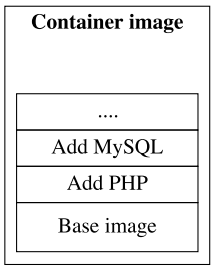

Figure 4: Container image

Docker Container

Docker used to commoditize LXC to create Docker containers. Since version 0.9, Docker has replaced LXC with libcontainer [8] - their own virtualization format - as the default container environment since Docker community desires not to depend on a third-party package. However, with either LXC or libcontainer, namespaces, cgroups, union file system, and Docker images are still the major underlying technologies to implement Docker containers.

Docker takes advantages of two Linux features, *namespaces* and *cgroups*, to safely create virtual environment for its containers. The cgroups, or control groups, provide mechanism for accounting and limiting the resources which the processes in each container can access. The namespaces wrap the operating system resources into different instances. The use of these instances gives the processes running inside a container the illusion that they have their own resources. Currently, Docker uses five namespaces to provide each container with a private view of the underlying host system [23]: mount, hostname, inter-process communication (IPC), process identifiers (PID), and network. Each of them works on specific types of system resources. The network namespaces, for example, isolate the networking resources, such as IP addresses, and IP routing tables, in order to provide each container with a separated network stack.

Docker launches its containers from *Docker images*. A Docker image is a series of data layers on top of a base image (Fig. 4). Every Docker image starts from a base image, such as Ubuntu base image or OpenSuse base image. When users make changes to a container, instead of directly writing the changes to the image of the container, Docker adds an additional layer containing the changes to the image. For example, if the user installs MySQL to an Ubuntu image, Docker creates a data layer containing MySQL and then adds to the image. This process makes the image distribution process more efficiently since only the update needs to be distributed.

In order to work with multiple layers of an image as it were a single file system layer, Docker uses a special file system called *Union File System (UnionFS)*. It allows files and directories in different file systems to be combined into a single consistent file system.

3.2 Docker Hub

Docker hub [4] is a central repository of images (both public and private), via which users can share their customized images. Users can also search for published images and download them with the Docker client. Furthermore, users can verify the authenticity and integrity of the downloaded images since Docker signed and verified the images when their owner submitted them to the hub.

4 Docker Security Analysis

Security is one of the major challenges when running services in virtual environments, especially in a multi-tenant cloud system. Virtual machines provided by hypervisor-based virtualization techniques are claimed to be more secure than containers as they add an extra layer of isolation between the applications and the host. An application running inside a VM is only able to communicate with the VM kernel, not the host kernel. Consequently, in order for the application to escalate out of a VM, it must bypass the VM kernel and the hypervisor before it can attack the host kernel. On the other hand, containers can directly communicate with the host kernel, thus allowing an attacker to save a great amount of effort when breaking into the host system. This raises a security concern over containers.

Docker is also a container-based virtualization technologies, thus having the same issue. Our analysis aims to discover whether Docker provides a safe environment to run applications. The analysis considers two areas: the internal security of Docker containers and how Docker containers interact with the additional security systems of the kernel.

4.1 Docker Internal Security

We examine the internal security of Docker based on the system and attacker model and security requirements as described by Reshetova et al. [29] for comparing a number of the OS-level virtualization technologies.

The system and attacker model is as follows: A single host machine is running a number of Docker containers c_1 ... c_n, on which a subset $\overline{C}$ of the containers are compromised and the attacker has full control over those, but the remaining subset of containers C is still under the control of the legitimate users. In this model, the attacker can perform various types of attacks, such as Denial-of-Service and Privilege escalation.

In order to encounter with these attacks, the authors stated that an OS-level virtualization solution should satisfy the following requirements: process isolation, filesystem isolation, device isolation, IPC isolation, network isolation and limiting of resources. The next sections present our analysis on how Docker fulfills the requirements.

Process Isolation

The main goal of process isolation is to prevent compromised containers from using process management interfaces to interfere with other containers. Docker achieves isolation of processes by wrapping the processes running in containers into namespaces and limiting their permissions and visibility to processes running in the other containers and the underlying host.

This mechanism operates with the support of the *PID namespaces*, which isolate the process ID number space of a container from that of the host. Since PID namespaces are hierarchical [12], a process can only see the other processes in its own namespace or in its "children" namespaces. As a consequence, once a new namespace is created and assigned to a container, the host can observe and affect the processes inside the new PID namespace of the container, but the processes inside the container cannot observe or do anything to the other processes running in the host or in other containers. If the attacker cannot observe other processes, it is harder to attack them.

The PID namespaces also allow each container to have its own init-like process (PID 1), which causes all the processes in a namespace to be terminated if it is terminated. This process assists the administrator in completely shutting down a container when something suspicious is detected.

Filesystem Isolation

In order to achieve filesystem isolation, the filesystems of the host and containers must be protected from illegitimate access and modification.

Docker uses the *mount namespaces*, also called the *filesystem namespaces*, to isolate the filesystem hierarchy associated with different containers. The mount namespaces provide the processes of each container a different view of the filesystem tree and restrict all the mount events occurring inside the container to only have impact inside the container. However, some of the kernel filesystems are not namespaced; for example, those under $/sys$, $/proc/sys$, $/proc/sysrq-trigger$, $/proc/irq$, and $/proc/bus$, and a Docker container needs to mount them in order to operate. This causes the issue that a container inherits the view of these filesystems from the host and are able to access them directly. Docker limits the threats that a compromised container could make to the host via these filesystems with the two filesystems protection mechanisms: (1) removing the write permission to these filesystems from containers and (2) not allowing any process of a container to remount any filesystem within the container [24]. The second mechanism is achieved by removing the CAP_SYS_ADMIN capability from containers.

Docker also employs a mechanism called *copy-on-write* file system [24]. As mentioned before, Docker creates containers based on file system images, and a container can write content to its own base image. When multiple containers are created on the same image, the copy-on-write file system allows each container to write content to its specific file system, thus preventing other containers from discovering the changes occurring inside the container.

Device Isolation

In Unix, the kernel and applications access the hardware through device nodes which basically are special files acting as the interfaces to the device drivers. If a container can access some important device nodes, such as $/dev/mem$ (the physical memory), $/dev/sd*$ (the storage) or $/dev/tty$ (the terminal), it can make serious damage on the host system. Thus, it is crucial to limit the set of device nodes that a container can access.

The *Device Whitelist Controller* feature [3] of cgroups provides means to limit the set of devices that Docker allows a container to access. It also prevents the processes in containers from creating new device nodes. Furthermore, Docker mounts container images with *nodev*, meaning that even if a device node was pre-created inside the image, the processes in the container using the image cannot use it to communicate with the kernel. By default, Docker does not give extended privileges to its containers. Therefore, they cannot access any devices. However, if the operator executes a container as "privileged", Docker grants access to all devices to the container.

IPC Isolation

The IPC (inter-process communication) is a set of objects for exchanging data among processes, such as semaphores, message queues, and shared memory segments. The processes running in containers must be restricted so that they can communicate only via a certain set of IPC resources and are disallowed to interfere with those in other containers and the host machine.

Docker achieves IPC isolation by using the *IPC namespaces*, which allows the creation of separated IPC namespaces. The processes in an IPC namespace cannot read or write the IPC resources in other IPC namespaces. Docker assigns an IPC namespace to each container, thus preventing the processes in a container from interfering with those in other containers.

Network Isolation

Network isolation is important to prevent network-based attacks, such as Man-in-the-Middle (MitM) and ARP spoofing. Containers must be configured in such a way that they are unable to eavesdrop on or manipulate the network traffic of the other containers nor the host.

For each container, Docker creates an independent networking stack by using *network namespaces*. Therefore, each container has its own IP addresses, IP routing tables, network devices, etc. This allows containers to interact with each other through their respective network interfaces, which is the same as how they interact with external hosts.

By default, connectivity between containers as well as to the host machine is provided using Virtual Ethernet bridge [5] (Fig. 5). With this approach, Docker creates a virtual ethernet bridge in the host machine, named $docker0$, that automatically forwards packets between its network interfaces. When Docker creates a new container, it also establishes a new virtual ethernet interface with a unique name and then connects this interface to the bridge. The interface is also connected to the $eth0$ interface of the container, thus allowing the container to send packets to the bridge.

We note here that the default connectivity model of Docker is vulnerable to ARP spoofing and Mac flooding attacks since the bridge forwards all of its incoming packets without any filtering.

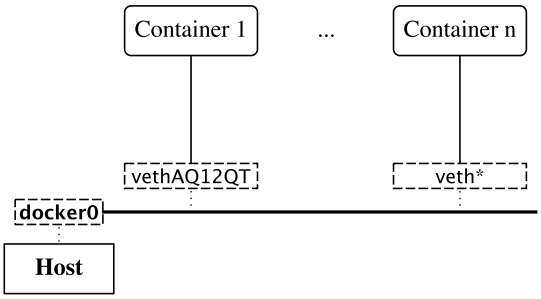

Figure 5: The default networking model of Docker

$CAP\_SETPCAP$	Modify process capabilities
CAP_SYS_MODULE	Insert/Remove kernel modules
CAP_SYS_RAWIO	Modify Kernel Memory
CAP_SYS_PACCT	Configure process accounting
CAP_SYS_NICE	Modify Priority of processes
$CAP\_SYS\_RESOURCE$	Override Resource Limits
CAP_SYS_TIME	Modify the system clock
$CAP\_SYS\_TTY\_CONFIG$	Configure tty devices
CAP_AUDIT_WRITE	Write the audit log
$CAP\_AUDIT\_CONTROL$	Configure Audit Subsystem
$CAP\_MAC\_OVERRIDE$	Ignore Kernel MAC Policy
CAP_MAC_ADMIN	Configure MAC Configuration
CAP_SYSLOG	Modify Kernel printk behavior
CAP_NET_ADMIN	Configure the network
CAP_SYS_ADMIN	Catch all

Table 1: Some capabilities disallowed in Docker containers [24]

Limiting of Resources

Denial-of-Service (DoS) is one of the most common attacks on a multi-tenant system, where a process or a group of processes attempt to consume all of the resources of the system, thus disrupting normal operation of the other processes. In order to prevent this kind of attack, it should be possible to limit the resources that are allocated to each container.

Cgroups are the key component that Docker employs to deal with this issue. They control the amount of resources, such as CPU, memory, and disk I/O, that any Docker container can use, ensuring that each container obtains its fair share of the resources and preventing any container from consuming all resources. They also allow Docker to configure the limits and constraints related to the resources allocated to each container. For example, one such constraint is limiting the available CPUs available to a specific container.

4.2 Docker and Kernel Security Systems

Some kernel security systems exist in order to harden the security of a Linux host system, including Linux capabilities and Linux Security Module (LSM). Linux capabilities restricts the privileges assigned to each process. LSM provides a framework which allows the Linux kernel to support different security models. The LSMs that have been integrated into the official Linux kernel include AppArmor [20], SELinux [30], and Seccomp [14].

This paper surveys Linux capabilities and two LSMs, AppArmor [20] and SELinux [30], which Docker currently supports. Docker can also collaborate with Seccomp but only if LXC is used; thus, we do not include it in the survey. Even though Docker does not support other security systems at the moment, it does not interfere with them. Therefore, these systems can run independently of Docker containers to protect the host [2].

Linux Capabilities

As stated in Linux capabilities man page [9], traditionally, Unix systems classified processes into two categories: *privileged processes* (owned by superuser or root) and *unprivileged processes* (owned by normal users). The kernel skipped all permission checks on the privileged processes but conducted full permission checking on unprivileged processes. However, the Linux kernel, since version 2.2, divides the privileges of the superuser into *capabilities*, which the kernel can independently enable or disable.

Docker containers run on a kernel shared with the host system, so most of their tasks can be handled by the host. As a result, in most cases, it is unnecessary to provide full root privileges to a container, thus removing some of the root capabilities from a container does not affect the usability or functionality of the container but effectively improves the security of the system. For example, the CAP_NET_ADMIN capability, which provides the ability to modify the system network, can be removed from a container since all networking configuration can be handled by the Docker daemon before starting the container.

Docker allows configuration of the capabilities that a container can use. By default, Docker disables a large number of Linux capabilities from its containers in order to prevent an intruder to damage the host system even when the intruder has obtained root access within a container. Some of the capabilities are presented in table 1, and their detailed description can be found in the Linux capabilities man page [9].

SELinux

SELinux is a security enhancement to the Linux system. Linux comes with the standard *Discretionary Access Controls (DAC)* mechanism (i.e., owner/group and permission flags of an object) to control the access to an object. SELinux provides an additional layer of permission checking, called *Mandatory Access Control*, after the standard DAC is performed. In SELinux, everything is controlled by labels. Every file/directory, process, and system object has a label. The administrator of the system uses these labels to write rules to control access between processes and system objects. These rules are called *policies*. The SELinux policies can be divided into three classes: Type enforcement, Multi-level security (MLS) enforcement, and Multi-category security (MCS) enforcement.

With the DAC mechanism, owners have full discretion over their objects, meaning that if the owners are compromised, the attacker has control over all of their objects. In SELinux model, in contrast, the kernel manages and enforces

all of the access controls over objects, not their owners. This provides a secure separation for containers as it can prevent processes, even with root privileges, within a container to illegitimately access objects outside the containers.

Docker uses two classes of policy enforcement: *Type enforcement* and *MCS enforcement* [24]. The Type enforcement protects the host from the processes in containers, and the MCS enforcement protects a container from another container.

With Type enforcement, Docker labels all container processes with $svirt\_lxc\_net\_t$ type and all content within a container with $svirt\_sandbox\_file\_t$ type. The processes running with $svirt\_lxc\_net\_t$ type can only access/write to the content labeled with $svirt\_sandbox\_file\_t$ type, but not to any other label on the system. Therefore, the processes running within containers can only use the content inside containers. However, only with this policy enforcement, Docker allows the processes in one container to have access to the content of other containers. MCS enforcement is necessary to solve this issue. When a container is launched, the Docker daemon picks a random MCS label and then puts this label on all of the processes and content of the container. The kernel only allow processes to access content with the same MCS label, thus preventing a compromised process in one container from attacking other containers.

AppArmor

AppArmor is also a security enhancement model to Linux based on Mandatory Access Control like SELinux, but restricting its scope to individual programs. It permits the administrator to load a security profile into each program, which limits the capabilities of the program. AppArmor supports two modes: enforcement mode and complain/learning mode. The enforcement mode enforces the policies defined in the profile. However, in the complain/learning mode, the violations of profile policies are permitted, but also logged. This log can be useful for developing new profiles later.

On systems that support AppArmor, Docker provides an interface for loading a pre-defined AppArmor profile when launching a new container. This profile is loaded into the container in enforcement mode in order to ensure that the processes in the container are restricted according to the profile. If the administrator does not specify a profile when launching a container, the Docker daemon automatically loads a default profile to the container, which denies access to important filesystems on the host, such as $/sys/fs/cgroups/$ and $/sys/kernel/security/$.

5 Discussion

The analysis shows that Docker provides a high level of isolation and resource limiting for its containers using namespaces, cgroups, and its copy-on-write file system, even with the default configuration. It also supports several kernel security features, which help to hardening the security of the host. The only problem we found with Docker was related to its default networking model. The virtual ethernet bridge which Docker uses as its default networking model, is vulnerable to ARP spoofing and MAC flooding attacks since it does not provide any filter on the network traffic passing through the bridge. However, this problem can be solved if the administrator manually adds filtering, such as ebtables [6], to the bridge, or changes the networking connectivity to a more secure one, such as virtual network.

It is also worth highlighting that if the operator runs a container as "privileged", Docker grants full access permissions to the container, which is nearly the same as that of processes running natively on the host. Therefore, it is more secure to operate containers as "non-privileged".

Furthermore, even though containers can provide higher density of virtual environments and better performance, they have a bigger attack surface than virtual machines since containers can directly communicate with the host kernel. However, it is possible to reduce the attack surface while maintaining these advantages. For example, this can be achieved by placing containers inside virtual machines.

6 Conclusion and Future work

Container-based virtualization can provide higher density virtual environments and better performance than hypervisor-based virtualization. However, the latter is argued to be more secure than the former. In this paper, we conducted an analysis on Docker, which is one of the most popular container-based virtualization technologies, to discover how safe its containers are. Our analysis shows that Docker containers are fairly secure, even with the default configuration. The security level of Docker containers could also be increased if the operator runs them as "non-privileged" and enables additional hardening solutions in Linux kernel, such as AppArmor or SELinux.

The future work after this paper could be to compare the security of Docker containers with that of other containerization systems or with virtual machines. Such studies could lead to e.g. a detailed static analysis Docker or a broader view of security in containers in general.

Acknowledgement

This research paper is made possible through the help and support of Miika Komu, Roberto Morabito, Jimmy Kjällman, and Tero Kauppinen from Nomadiclab.

References

[1] Ansible. `http://www.ansible.com/home/`. [Accessed 25 October 2014].

[2] Containers & docker: How secure are they? `https://blog.docker.com/2013/08/containers-docker-how-secure-are-they`. [Accessed 25 October 2014].

[3] Device whitelist controller. `https://www.kernel.org/doc/Documentation/cgroups/devices.txt`. [Accessed 12 October 2014].

[4] Docker hub. https://hub.docker.com/. [Accessed 30 September 2014].

[5] Docker: Network configuration. https://docs.docker.com/articles/networking/. [Accessed 24 September 2014].

[6] Ebtables. http://ebtables.netfilter.org/. [Accessed 25 October 2014].

[7] Kubernetes project. https://github.com/googlecloudplatform/kubernetes. [Accessed 10 November 2014].

[8] Libcontainer project. https://github.com/docker/libcontainer. [Accessed 25 October 2014].

[9] Linux capabilities. http://linux.die.net/man/7/capabilities. [Accessed 12 October 2014].

[10] LXC. https://linuxcontainers.org/. [Accessed 30 September 2014].

[11] OpenVZ. http://openvz.org/. [Accessed 30 September 2014].

[12] PID namespaces in the 2.6.24 kernel. http://lwn.net/Articles/259217/. [Accessed 30 September 2014].

[13] Puppet. http://puppetlabs.com/. [Accessed 18 October 2014].

[14] SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt. Available at: [Accessed 02 November 2014].

[15] Shipyard project. https://github.com/shipyard/shipyard. [Accessed 12 November 2014].

[16] Vagrant. https://www.vagrantup.com/. [Accessed 15 November 2014].

[17] What is docker? https://docker.com/whatisdocker/. [Accessed 15 November 2014].

[18] B. R. Anderson, A. K. Joines, and T. E. Daniels. Xen worlds: Leveraging virtualization in distance education. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '09, pages 293–297, New York, NY, USA, 2009. ACM.

[19] C. Burniske. Containers: The next generation of virtualization? http://ark-invest.com/webx0/containers-next-generation-virtualization. [Accessed 22 November 2014].

[20] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious server security. In *Proceedings of the 14th USENIX Conference on System Administration*, LISA '00, pages 355–368, Berkeley, CA, USA, 2000. USENIX Association.

[21] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. Technical Report RC25482 (AUS1407-001), IBM Research Division, July 2014.

[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[23] D. J. Walsh. Are docker containers really secure? http://opensource.com/business/14/7/docker-security-selinux. [Accessed 25 October 2014].

[24] D. J. Walsh. Bringing new security features to docker. https://opensource.com/business/14/9/security-for-docker. Available at: [Accessed 25 October 2014].

[25] A. Kivity, Y. Kamay, D. Laor, and U. Lublin. KVM: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230. 2007.

[26] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

[27] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance evaluation of virtualization technologies for server consolidation. *HP Laboratories*, 2007.

[28] N. Regola and J.-C. Ducom. Recommendations for virtualization technologies in high performance computing. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 409–416, Nov. 2010.

[29] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan. Security of OS-level virtualization technologies. In *Proceedings of the 2014 NordSec Conference*, pages 77–93, Norway, 2014.

[30] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a linux security module. NAI Labs Report #01-043, NAI Labs, Dec. 2001. Revised May 2002.

[31] S. Soltesz, H. Potzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. 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*, pages 275–287, USA, 2007. ACM.

[32] 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 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240, Washington, DC, USA, 2013. IEEE Computer Society.