International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Docker and analysis of its security

Amol Bhalerao

Keraleeya Samajam (Regd.) Dombivli's Model College, Thakurli (East), Maharashtra, India
Email id: -amolbhalerao.model@gmail.com

ABSTRACT

In recent years, the use of virtualization technologies has increased significantly. This makes the demand for efficient and secure virtualization solutions clearer.Container-based virtualization and hypervisor-based virtualization are two main types of virtualization technologies that have emerged in the market. Of these two classes, container-based virtualization is capable of providing a lighter and more efficient virtual environment, but it is not without security issues for. In this document, we discuss security level from Docker, a well-known proponent from Docker the container-based approaches.The analysis considers two areas: (1) Docker's internal security and (2) how Docker interacts with Linux kernel security features such as SELinux and AppArmor to harden the host system. In addition, the document also discusses and identifies what could be done when using Docker to increase security levels.

Keywords— Security, Containers, Docker

INTRODUCTION

In the last decade there has been an explosion of virtualization technologies that allow a computer system to be partitioned into multiple isolated virtual environments. The technologies offer significant benefits that has rapidly advanced. One of the most common reasons for using virtualization technologies is the virtualization of servers in data centers. With server virtualization, an administrator can create one or more instances of the virtual system on a single server.These virtual systems work like real physical servers and can be rented from on a subscription basis. Amazon EC2, Rackspace, and Dream Host are some popular instances of data center service providers of this type. Another common application is desktop virtualization, where a machine can run multiple instances of the operating system. Desktop virtualization provides support for applications that only run on a specific operating system.The increasing use of virtualization technologies is driving the demand for a virtualization solution that can deliver dense, scalable, and secure user environments.

Many virtualization solutions have appeared on the market. They can be divided into two main classes: container-based virtualization and hypervisor-based virtualization. Of these two classes, container-based virtualization can provide a lighter and more efficient virtual environment.This means that ten times more virtual environments can be operated on a physical server compared to hypervisor-based virtualization [1].

However, container-based virtualization also comes with security issues.In this article, we discuss the security level of Docker [2], a well-known representative of the container-based virtualization approach. We looked at two areas: (1) Docker's internal security, and (2) how Docker interacts with Linux kernel security features such as SELinux and AppArmor to harden the host system. The Review examined Docker's internal securitybased on the level of isolation Docker can provide to its virtual environments. The interaction between Docker and kernel security features was estimated based on how Docker supports the features. Thedocument is structured as follows:Section 2provides a general 1017 overviews of the twoclasses ofvirtualization solutions. Section 3 of provides an overview of Docker and's underlying technologies. Section 4 presents our discussion of Docker security, and then in Section 5 we discuss the security level of Docker and what could be done to increase your security level.

Virtualization Approaches

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

A container-based virtualization is a lightweight approach to virtualization that uses the host kernel to run multiple virtual environments. These virtual environments are often called containers. LinuxVServer [3], OpenVZ [4] and Linux Container (LXC) [5] are the three main representatives of this approach. The general architecture of a container-based virtualization solution is shown in Figure.Container-based virtualization is virtualized at the OS layer, allowing multiple applications to run the without redundantly running other OS kernels on the host. Your containers look like regular external processes running on the kernel shared with the host machine. They provide isolated environments with the resources needed to run applications. These resources can be shared with the host or installed separately within the Container.

Hypervisor-based virtualization solutions provide hardware-level virtualization. In the contrast to container-based virtualization, a hypervisor sets up entire virtual machines (VMs) on the host operating system – Fig. 2;

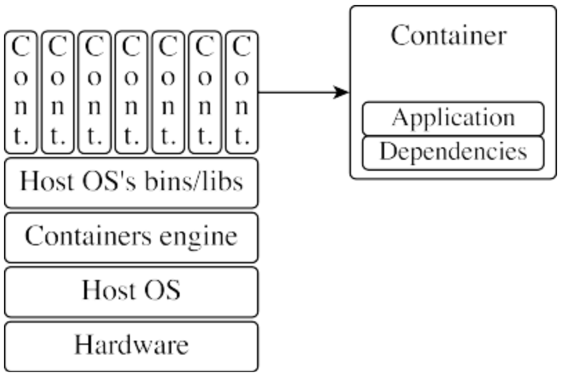

Figure 1: Architecture of Container-based Virtualization

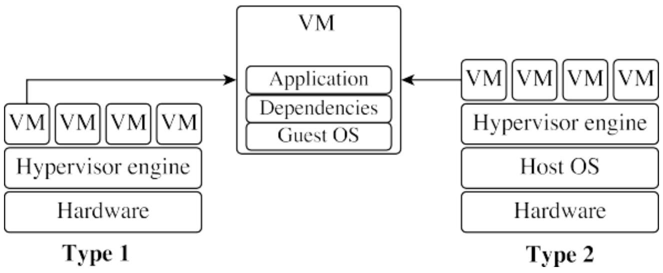

Figure 2: Architecture of Hypervisor-based Virtualization

Each virtual machine consists not only of an application and its dependencies, but also of a complete guest operating system along with a separate kernel. There are two classes of hypervisors: the Type 1 hypervisor, also known as running directly on the underlying host hardware, and the Type 2 hypervisor, also known as the hosted hypervisor and running on the host operating system [6]. Xen [7] is an example of the former, KVM [8] of the latter. Since the Type 1 hypervisor does not contain an additional host operating system layer, it offers better performance than the Type 2 hypervisor.The architectural differences give container-based virtualization some advantages over hypervisor-based virtualization. First, container-based virtualization can provide higher density virtual environments. Because a container does not contain a full operating system, the size and resources required to run an application in a container are less than the of a virtual machine running the same application. This allows more containers than traditional virtual machines to be deployed on the same host. Second, container-based virtualization also offers better performance.This was shown by experiments in some studies [9, 10, 11,12]. These studies show that container-based virtualization outperforms hypervisor-based virtualization in most cases and isalmost as good as native applications.However, despite all the benefits mentioned, container-based virtualization cannot support a variety of environments in the same way as hypervisor-based virtualization, since all container environments must be of the same environment type as the host. For example, Windows containers cannot run on a Linux host.

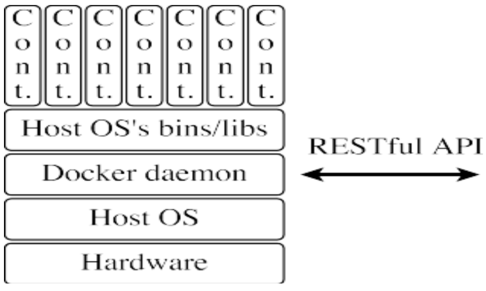

Figure 3: Architecture of Docker engine

Docker Overview

Docker is an OS container technology with the ability to "elevate, ship, and run round-robin applications. It has been used in some popular apps like Spotify, Yelp, and eBay.

Although container technologies have been around for over a decade, Docker is a relatively new candidate. It is currently one of the most successful technologies as it is endowed with new abilities that previous technologies did not possess. First, it provides interfaces to easily and securely create and control containers. Second, developers canpackage applications in lightweight Docker containers that can run almost anywhere without modification. In addition, Docker can implement more virtual environments on the same hardware than other technologies. Last but not least, Docker works well with third-party tools and simplifies the process ofmanaging and deploying Docker containers. It is easy to implement in a cloud. Many orchestration tools like Mesos , Shipyard, andKubernetes also support Docker containers. These tools provide an abstract level of resource management andscheduling over Docker. Docker consists of two main 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 these two components in detail. Docker Engine:

a) Docker Engine:-

Docker Engine is a lightweight and portable packaging tool based on container-based virtualization. Therefore, the architecture of the Docker engine (Fig. 3) is fundamentally container-based virtualization. Docker containers run in the Docker daemon, which is responsible for running and managing all Docker containers. The Docker client, which provides Docker users with a user interface for interacting with containers, accepts commands from users. And then sends them to the Docker daemon via the RESTful API. By using this communication method, the Docker client can run on the same host as the containers or even on different hosts.

Docker Container:-

Docker used to commoditize LXC to build Docker containers. Since version 0.9, Docker has replaced LXC with its own virtualization format libcontainer as the standard container environment, since the Docker community does not want to be dependent on a third-party package. LXC or libcontainer, namespaces, cgroups, Union File System and Docker images are still the main underlying technologies for implementing Docker containers.Docker uses two Linux features, namespaces and cgroups, to securely create a virtual environment for your containers.
The cgroups or control groups resources in different instances. Using these instances gives processes running in a container the illusion that they have their own resources. Currently, Docker uses five namespaces to provide all containers with a private view of the underlying host system: mount, hostname, interprocess communication (IPC), process identifiers (PIDs), and network. Each of them works with specific types of system resources. Network namespaces, such as, isolate network resources such as IP addresses and IP routing tables to provide each container with a separate network stack.Docker starts its containers from Docker images.
A Docker image is a set of layers of data over a base image (Figure 4). Every Docker image starts with a base image, such as the Ubuntu base image or the OpenSUSE base image. When users make changes to a container, Docker does not write the changes directly to the container image, but adds an extra layercontaining the changes to the image. For example, if the user installs MySQL on an Ubuntu image, Docker will create a data layer containing MySQL and will then be added to the image. This process makes the image distribution process more efficient since only update needs to be distributed.To work with multiple layers of an image as if it were a single layer file system, Docker uses a special file system called Union File System .It allows combining files and directories on different file systems into a single consistent file system.

b)Docker Hub:-

Docker hub is a central repository of images (both publicand 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 canverify the authenticity and integrity of the downloaded images since Docker signed and verified the images when theirowner submitted them to the hub.

DOCKER SECURITY ANALYSIS

Security is one of the biggest challenges when running services in virtual environments, especially in a multi-tenant cloud system. Virtual machines provided by hypervisor-based virtualization techniques are considered more secure than containers because they add an additional layer of isolation between applications and the host. An application running in a VM can only communicate with the VM kernel, not the host kernel before it can attack the host kernel. On the other hand, containers can communicate directly with the host kernel, which saves an attacker a lot of effort to break into the host system.This raises container security concerns.
Docker is also a container-based virtualization technology, so it has the same problem. Our analysis aims to find out whether Docker provides a secure environment for running applications. The examination considers two areas the internal security of Docker containers and how Docker containers interact with additional kernel security systems.

a) **Docker Internal Security:-**

We examine Docker's internal security using the system and the Reshetova et al. described attacker model and security requirements. [13] to compare a

set of OS-level virtualization technologies.
The attacker's system and model are as follows: A single host machine is running multiple Docker containers C1...CN, in which a subset C of the containers is compromised, andthe attacker has full control over them, but the remaining subset of C containers are still under the control of legitimate users.
In this model, the attacker can perform different types of attacks, such as: B. DoS and Privilege Escalation.To counteract these attacks, the authors claimed that an OS-level virtualization solution must meet the following requirements: process isolation, file system isolation, device isolation, IPC isolation, network isolation, and resource throttling. The following sections present our analysis of how Docker meets the requirements.

Process Isolation:-

The main purpose of the separation is to prevent compromised containers from using a process management combine to combine with other containers. Docker achieves process isolation by enclosing processes running in containers in namespaces and restricting their permissions and visibility to processes running in the other containers and the necessary host.
This mechanism works with support for PID namespaces, which isolates a container's process ID range from that of the host.Because PID namespaces are hierarchical, a process can only see other processes in its own namespaceor in their "child" namespaces. Thecontainercannot observe or do anything with the other processes running on the host or in other containers. If the attacker cannot observe other processes, it is more difficult to attack them. PID namespaces also allow each container to have its own startup process (PID). 1), causing all processes in a namespace to terminate when it terminates.This process helps the admin to shut down a container completely if anything suspicious is detected.

Filesystem Isolation:-

PID namespaces also allow all containers to have their self .To achieve file system isolation, the host and container file systems must be protected from unauthorized access and modification.
Docker uses mount namespaces, also called filesystem namespaces, to isolate the filesystem hierarchy associated with different containers. The mount namespaces provide the processes in each container with a different view of the file system structure and restrict any mount events that occur within the container. to affect it only inside the container. However, some of the kernel file systems do not have a namespace; for example those under /sys, /proc/sys, /proc/SysRq − trigger, /proc/IRQ and /proc/bus, and a Docker containers must mount them in order for them to work. This produces the problem that a container inherits the host's view of these file systems and can access them directly. Docker limits the threats that a compromised container could pose to the host through these filesystems with two filesystem protection mechanisms:(1) revokes the container and permissions to write to these filesystems. (2) prohibits any process in a container that remounts a file system inside the container [14].The second mechanism is achieved by removing the CAP_SY S_ADMIN capability from containers Docker also uses a mechanism called copy on write file system [14].
As mentioned above, Docker builds containers based on filesystem images, and a container can write contents to its self base image. When multiple containers are built on the same image, the copy on write filesystem allows each container to write content to its specific filesystem, preventing other containers from discovering changes occurring within the container.

Device Isolation:-

InUnix, the kernel and applications access hardware through device nodes, which are basically special files that act as interfaces to device drivers. If a container can access some important device nodes like /dev/mem (the memory), /dev/SD∗ (the memory), or /dev/TTY (the terminal), it can cause serious damage to that host system. Therefore, it is important to limit the set of device nodes that a container can access.

The cgroups Device Whitelist Driverfeature provides means to restrict the group of devices that Docker grants access to the container. It also prevents the containerized process from creating new device nodes. Also, Docker mounts container images with nodes, which means that even if a device node is previously created inside the image, processes in the container that uses the image cannot use to communicate with the kernel. By default, Docker does not grant elevated privileges to its containers. Therefore, they cannot access any device. However, if the operator operates a container as "privileged", Docker grants access to all devices in the container.

IPC Isolation:-

The IPC (Inter-Process Communication) is a set of objects for exchanging data between processes, such as semaphores, message queues, and shared memory segments. Of IPC resources and cannot interfere with those ofother containers and with the host machine.

Docker achieves IPC isolation through the use of IPC namespaces,which allows the creation of separateIPC namespaces. Processes in one IPC namespacecannot read orwrite IPC resources in other IPC namespaces. Docker assigns an IPC namespace to each container, preventing processes in one container from interfering with processes in other containers.

Network Isolation:-

Network isolation is powerful to prevent network-based attacks such as ManintheMiddle (MitM) and ARP spoofing. Containers must be configured so that they cannot spy on or tamper with network traffic from other containers or from the host .Docker creates a separate network stack for each container by using network namespaces.

So each container has its own IP addresses, IP routing tables, network devices, and so on. This allows the containers to interact with each other through their respective network. Interfaces that are identical to the way they interact with external hosts.

By default, connectivity between the containers as well as to the host machine is provided via the Virtual Ethernet Bridge(Fig. 5). With this approach, Docker creates a virtual Ethernet bridge on the host machine called docker0, which automatically forwards packets between its network interfaces. When Docker creates a new container, it also creates a new virtual Ethernet interface with an eccentric name, and then connects that interface to the bridge. The interface is also connected to the container's eth0 interface, which allows the container to send packets to the bridge.

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

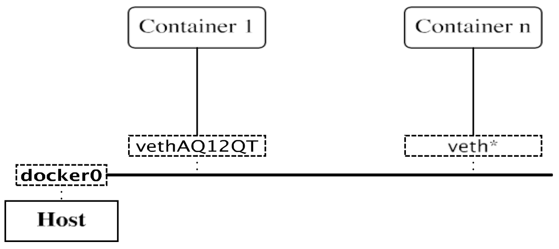

Figure 5: The default networking model of Docker

Limiting Of Resources:-

DoS is one of the most common attacks on a multi-tenant system, in which a process or a group of processes tries to consume all system resources, thereby disrupting the normal operation of the other processes. To prevent this type of attack, it should be possible to limit the resources allocated to each container. The cgroups are the key component that Docker uses to deal with this problem. They control the amount of resources like CPU, memory, and disk I/O that each Docker container can use, ensure that each container gets its fair share of the resources, and prevent one container from using all consumes resources. They also allow Docker to configure limits and restrictions on the resources allocated to each container. For example, one of these restrictions is to limit the available CPUs to a specific container.

b)Docker and Kernel Security Systems:-

There are several kernel security schemes to strengthen security of a Linux host system, including Linux functions and the Linux Security Module (LSM). Linux features limit the privileges assigned to each process. LSM provides a framework that enables the Linux kernel to support different security models. LSMs built into the official Linux kernel include AppArmor, SELinux and Seccomp.

This document examines the capabilities of Linux and two LSMs, AppArmor and SELinux, that Docker currently supports. Docker can also work with Seccomp, but only when using LXC; Therefore we did not include it in the survey.Although Docker doesn't currently support other security systems, it doesn't interfere with them. Therefore, these systems can run independently of Docker containers to guard the host .

Linux Capabilities:-

There are many kernel security schemes to strengthen security of a Linux host system, including Linux functions and the Linux Security Module (LSM). Linux features limit the privileges assigned to each process.

Table 1: Some capabilities disallowed in Docker containers [14]

CAP _SETPCAP	Modify process capabilities
CAP _SY S_MODULE	Insert/Remove kernel modules
CAP _SY S_RAWIO	Modify Kernel Memory
CAP _SY S_PACCT	Configure process accounting
CAP _SY S_NICE	Modify Priority of processes
CAP _SY S_RESOURCE	Override Resource Limits
CAP _SY S_TIME	Modify the system clock
CAP _SY S_TTY _CONFIG	Configure tty devices
CAP _AUDIT _WRITE	Write the audit log
CAP _AUDIT _CONTROL	Configure Audit Subsystem
CAP _MAC_OV ERRIDE	Ignore Kernel MAC Policy
CAP _MAC_ADMIN	Configure MAC Configuration
CAP _SY SLOG	Modify Kernel printk behavior
CAP _NET _ADMIN	Configure the network
CAP _SY S_ADMIN	Catch all

LSM provides a framework that enables the Linux kernel to support different security models. LSMs built into the official Linux kernel include AppArmor, SELinux and Seccomp. This document examines the capabilities of Linux and two LSMs, AppArmor and SELinux, that Docker currently supports. Docker can also work with Seccomp, but only when using LXC; therefore we did not include it in the survey.Although Docker doesn't currently support other security systems, it doesn't interfere with them. Therefore, these systems can run independently of Docker containers to guard the host.

SELinux:-

SELinuxis a security extensions for the Linux system. Linux comes with the standard Discretionary Access Controls (DAC) mechanism (i.e. owner/group and object permission flags) to control access to an object. SELinux provides an additional level of authorization checking, called Mandatory Access Control, after the standard DAC has run. In SELinux everything is controlled by tags. Each file/directory, process and system object has a label. The system administrator uses these tags to write rules to control access between processes and system objects. These rules are stated guidelines. SELinux policies can be further divided into three classes: Type Enforcement, Multi-Level Security (MLS) Enforcement, and Multi-Category Security (MCS) Enforcement.

With the DAC mechanism, owners have full agency over their objects, which means that if the owners are compromised, the attacker has control over all of their objects controls the objects, not their owners. This provides a secure separation for containers because even with root privileges it can prevent processes inside a container from illegitimately accessing objects outside of containers.

Docker uses two policy enforcement classes: Type Enforcement and MCS Enforcement. Type conformance protects the host from containerized processes, and -MCS conformance protects one container from another container with the type svirt_sandbox_file_t. processes running with type svirt_lxc_net_t can only access/write content tagged with all other tags in the system.

Therefore, processes running in the containers can only use the content in the containers. However, with this policy enforcement alone, Docker allows processes in one container to accesscontent in other containers. The MCS application is required to resolve this issue.When a container is started, the Docker daemon picks a random MCS tag and then places that tag on all processes and contents of the container. The kernel only allows processes to access content with the similar
MCS tag, preventing a compromised process in a container from attacking other containers.

AppArmor:-

AppArmor is also a safety enhancement model forLinux , based on mandatory access control like SELinux, but limiting its scope to individual programs. Permits the administrator to load a safetyprofile into any program, thereby limiting the program's capabilities. AppArmor supports two modes: Enforcement Mode and Learn/Complaint Mode. Enforcement Mode apply the policies defined in the profile. However, in the complaint/learn mode, violations of the profile policy are allowed but also logged. his data set can be useful for later development of new profiles.

On systems that helps AppArmor, Docker provides an interface to load a predefined AppArmor profile when a new container is started. This profile is loaded into the container in application mode to ensure that processes in the container are restricted according to the profile. If the administrator does not specify a profile when starting a container, the Docker daemon will automatically load a defaultprofile into the container, which will deny access to important filesystems on the host, such as /sys/fs/cgroups/ and /sys/kernel/security /.

DISCUSSION

Analysis shows that even with as the default, Docker provides a high level of isolation and resource throttling for its containers by using namespaces, cgroups, and its copyonwrite filesystem.

It also supports several core security features that help increase the security of the host. The only problem we encountered with Docker was related to with its default network model. The virtual Ethernet bridge, which Docker uses as the default network model, is vulnerable to ARP spoofing and MAC flooding attacks becauseit doesn't offer any filtering. in network traffic going through the bridge.

However, this problem can be solved if the administrator manually adds filters such as ebtables to the bridge or changes the network connection to a more secure connection, such as a virtual network. It's also worth noting that when the operator runs a container as "privileged", Docker grants full access permissions to the container, which is roughly equivalent to the processes running natively on the host.Therefore, it is safer to run containers as "unprivileged". In addition, while containers can provide higher density of virtual environments and better performance, they have a larger attack surface than virtual machines because containers can communicate directly with the host kernel. However, it is possible to reduce the attack surface while retaining these benefits. This can be achieved, for example, by placing containers in virtual machines.

Conclusion and Future Work

Container-based virtualization can supply higher-density virtual environments and high performance than hypervisor-based virtualization. However, it is argued that the latter is more secure than the former. Container-based virtualization technologies to find out how secure your containers are. Our analysis shows that even with default settings, Docker containers are fairly secure. The safety level of Docker containers could also be increased if the operator runs them as "unprivileged" and activates additional hardening solutions in the Linux kernel, such as AppArmor or SELinux.

Future work after this article could consist of comparing the security of Docker containers with that of other containerization systems or with virtual machines. Such studies could result in e.g. detailed static Docker analysis or a broader view of container security in general.

References

1. C. Burniske. Containers: The next generation of virtualization? http://ark-invest.com/webx0/ containers-next-generation-virtualization. [Accessed 22 November 2014]
2. What is docker? https://docker.com/ whatisdocker/. [Accessed 15 November 2014]
3. 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.
4. OpenVZ. http://openvz.org/. [Accessed 30 September 2014].
5. LXC. https://linuxcontainers.org/. [Accessed 30 September 2014].
6. D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. Linux J., 2014(239), Mar. 2014.
7. 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.
8. 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
9. 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
10. 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
11. P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance evaluation of virtualization technologies for server consolidation. HP Laboratories, 2007.
12. 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.
13. 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.
14. D. J. Walsh. Bringing new security features to docker. https://opensource.com/business/ 14/9/security-for-docker. Available at: [Accessed 25 October 2014].