RC25625 (WAT1607-027) July 15, 2016
Computer Science

IBM Research Report

Docker and Container Security White Paper

Salman Baset, Stefan Berger, James Bottomley, Canturk Isci, Nataraj Nagaratnam[1], Dimitrios Pendarakis, J. R. Rao, Gosia Steinder, Jayashree Ramanatham[1]

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

[1]IBM Cloud

Research Division
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

Docker and Container Security White Paper

Contributors: Salman Baset, Stefan Berger, James Bottomley, Canturk Isci, Nataraj Nagaratnam, Dimitrios Pendarakis, JR Rao, Gosia Steinder, Jayashree Ramanatham

Introduction

This paper presents IBM's comprehensive point of view of security and privacy for Cloud Computing services based on container technologies, in particular Docker containers. The objective is to highlight benefits as well as security challenges for Docker containers, highlight ongoing efforts that address these challenges and to motivate additional work that the Docker community and IBM are undertaking to further strengthen the security of the Docker container ecosystem. Potential users of Docker container based cloud services can use this paper to evaluate the benefits and risks associated with deploying various workloads on Docker containers, understand the evolution of Docker containers and decide what additional security mechanisms and tools to employ to further reduce security risks. The paper starts with an overview of the applicable threat model and then compares the security properties of base technologies such as Linux containers, Docker, as well hypervisors, which are the basis of Infrastructure as a Service (IaaS) offerings. Next we describe some of the gaps in security for Docker containers and how IBM has helped and continues to help the community to address them. Finally we describe some new and innovative security technologies in Docker and the Linux Kernel to further strengthen container security.

Benefits of Cloud Services Based on Linux/Docker Containers

Since Linux containers are implemented through virtualization at the system call level, applications running inside containers share the same underlying Linux kernel. Therefore, cloud services built using containers offer several benefits compared to virtual machines, specifically:

- An application running inside a container can be expected to have near bare metal performance while the same application running inside a virtual machine will not be able to reach that performance level. The reason for this lies in the fact that containers do not emulate devices but access system resources directly.

- The startup delay of containers is much shorter than that of a virtual machines since containers typically only start a few applications while a virtual machine may first run the firmware before booting an entire operating system.

- Since containers start only a few applications, they use resources, such as memory, more efficiently and can therefore be deployed with much higher density than virtual machines.

- Containers provide simplified management. The cloud operator takes responsibility for life cycle management of the operating system (optimization, updates, patching, security scans) allowing users to focus on application development and management.

- Containers provide better portability. Standardized and light-weight image formats such as Docker enable nearly perfect transfer of application across environments: from development to production and from on-premise and off-premise deployments.

- The reduced size of containers leads to a smaller attack surface for cloud customers' workloads.

- Access to a common Operating System Kernel provides higher visibility to the behavior of individual applications. Similarly, access to critical data and events may reveal anomalies and mis-configuration before they become evident through other means.

- Containers encourage microservice-based application architectures, which delegate persistent data to backend datastores and away from compute instances. This reduces the problems of unguarded proliferation of confidential content, which is a common side effect of image clone and copy in the virtual machine world.

Container Deployment Models in Cloud Computing Environments

How can the benefits of containers outlined above be realized in cloud computing environments? There are several deployment models for containers in a cloud. One of them is to enable tenants to deploy containers inside their virtual machines running in an infrastructure as a service (IaaS) cloud. In this case, the cloud provider will provide a security architecture for virtual machines, which includes the isolation of virtual machines from other tenants on the compute, network, and storage layers. Several cloud providers, have chosen this model of VMs hosting containers.

A second deployment model is to run containers directly on a shared host. In this case the cloud management stack treats containers similarly to virtual machines and applies isolation techniques directly to the containers. These techniques may include network isolation using security groups, or hiding some containers' IP addresses from the public Internet by only applying public IP addresses (NAT) to a subset of them.

Background: Container Isolation Technologies

Linux container technologies are implemented through virtualization at the system call layer and the Linux kernel is shared between all applications running in containers on the same host. When running containers directly on the shared host kernel, it is imperative to ensure proper isolation of containers from each other as well as to~~he~~ protect the host from potentially malicious containers. This can be achieved through a number of Linux isolation technologies

and features of container management stacks. The following is a list of Linux technologies employed by containers:

- Namespaces
- Control groups (cgroups)
- Linux Capabilities
- Seccomp
- Linux Security Modules (LSM): SELinux, AppArmor
- User namespace for de-privileging the container root user

The most important isolation technology for containers is **Linux namespaces**. Namespaces help to provide an isolated view of the system to each container. Namespace support exists for the areas of networking, mount points, process IDs, user IDs, inter process communication, and the setting of the hostname. Containers can be regarded as running inside a collection of these namespaces. Resource visibility is governed by namespaces which can be used to limit access to those resources by processes within each container. In the networking namespace, for example, container processes access different network interfaces that typically have different IP addresses assigned to them compared to the ones on the host or in other containers, thus providing the basic architecture for isolating containers' network traffic. Some aspects of namespace support are still work in progress and some areas, particularly those related to security subsystems, require the implementation of further namespace support for them to be independently usable by containers. Typically subsystems for which no namespace support is implemented are not accessible from within containers.

With containers running on the same host, they necessarily share its limited resources. These include resources related to compute, networking, and storage, as well as usage of devices. It is important that access to the shared resources can be controlled and containers are prevented from starving other containers or the host of time slices to access them. The Linux technology for resource control and prioritization are **control groups (Cgroups)** and container management stacks typically set them up as part of starting a container. Cgroups help to limit access to CPU shares (time slices), as well as storage and networking bandwidth and can prevent access to devices. Note that to limit the network *bandwidth* available to a container one also has to make use of the Linux traffic control (tc) system for shaping and policing of containers' network traffic.

For network bandwidth control it is important to differentiate between traffic that is occurring exclusively inside the cloud, for example between different containers, and traffic between

containers and endpoints that reside outside the cloud. The previously mentioned control groups can primarily address the former. The latter needs to be addressed on networking equipment (routers, load balancers, etc.) at the entrance of the cloud where incoming traffic volume is shaped on a per--destination IP address (container) basis so that high volumes of traffic do not reach deep into the cloud infrastructure.

Linux implements more than 300 different system calls, of which some are typically only accessible to the hosts' privileged root user. An example for this is the setting of the system clock. With multiple containers running on the same host, each container can have its own root user and invoke privileged syscalls. Since namespaces for example do not isolate the system clock, it is necessary to prevent container users from modifying the system clock via limiting access to the syscall interface. One technology that allows to achieve this is **Linux Capabilities**. The Linux Kernel currently provides 37 different capabilities; individual capabilities can prevent access to individual syscalls, or syscalls with certain parameters, or collection of syscalls. Docker for example by default drops 24 of the 37 capabilities for processes it starts in a container and thus de-privileges container applications. The consequence of dropping these capabilities is that applications running inside containers cannot set the system clock as well as perform other privileged operations such as activating or deactivating swap memory, among many others.

Another technology that can be used to limit access to syscalls is **seccomp** (mode 2). Seccomp allows for creation of (Berkeley packet) filters that can filter by syscall number and the parameters passed to syscalls. This technology allows for fine-grained access control to the kernel syscall interface, for example to catch a specific syscall and depending on the call number and arguments passed to "allow", "deny", "trap", "kill" or "trace" it. Limiting system calls can be used to further reduce the system call attack surface. This capability was added to Docker Engine 1.10, which allows the passing of a profile defining the syscalls and the filters for them, as well as a defining a default profile [2].

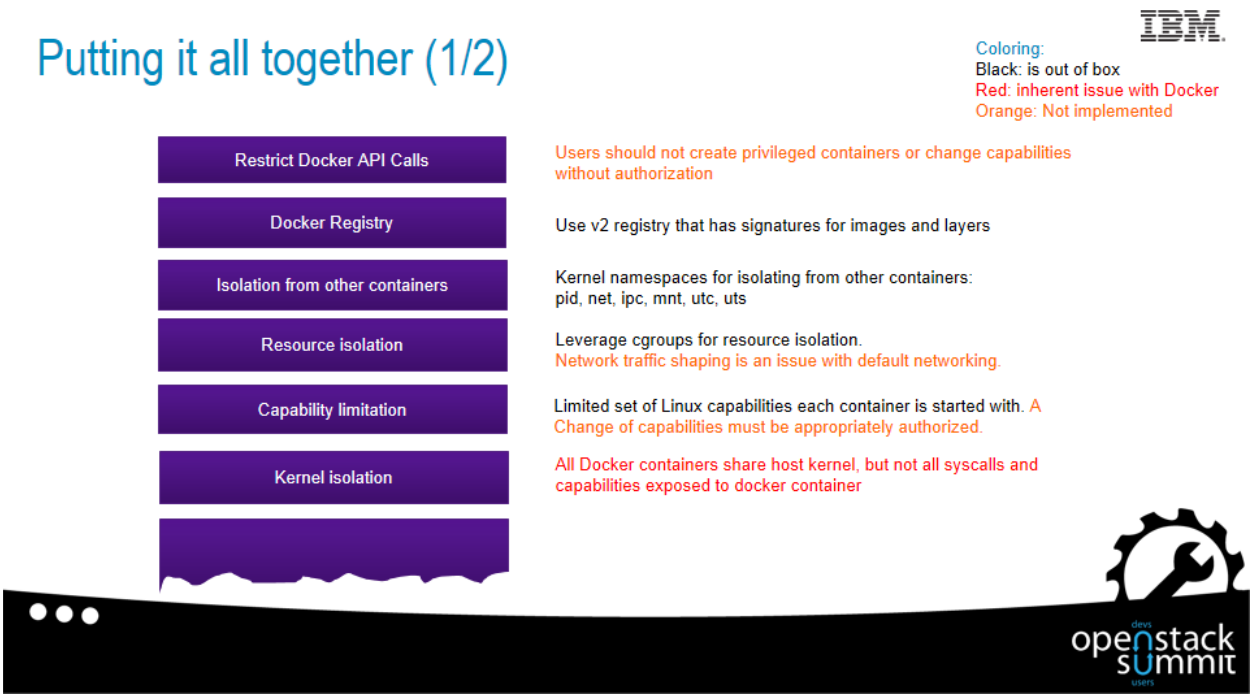
Another Linux technology for confining container processes is the **Linux Security Module (LSM)**. Two prominent LSM implementations are SELinux and AppArmor. The SELinux-based sVirt technology can be used to label container filesystems as well as the container processes. If a container process succeeded in a "jail break", sVirt would prevent it from accessing files on the host or in other containers. An AppArmor policy prevents access to certain critical files in the container's filesystems, such as proc and sysfs. It can also control access to the network as well as control Linux capabilities given to processes.

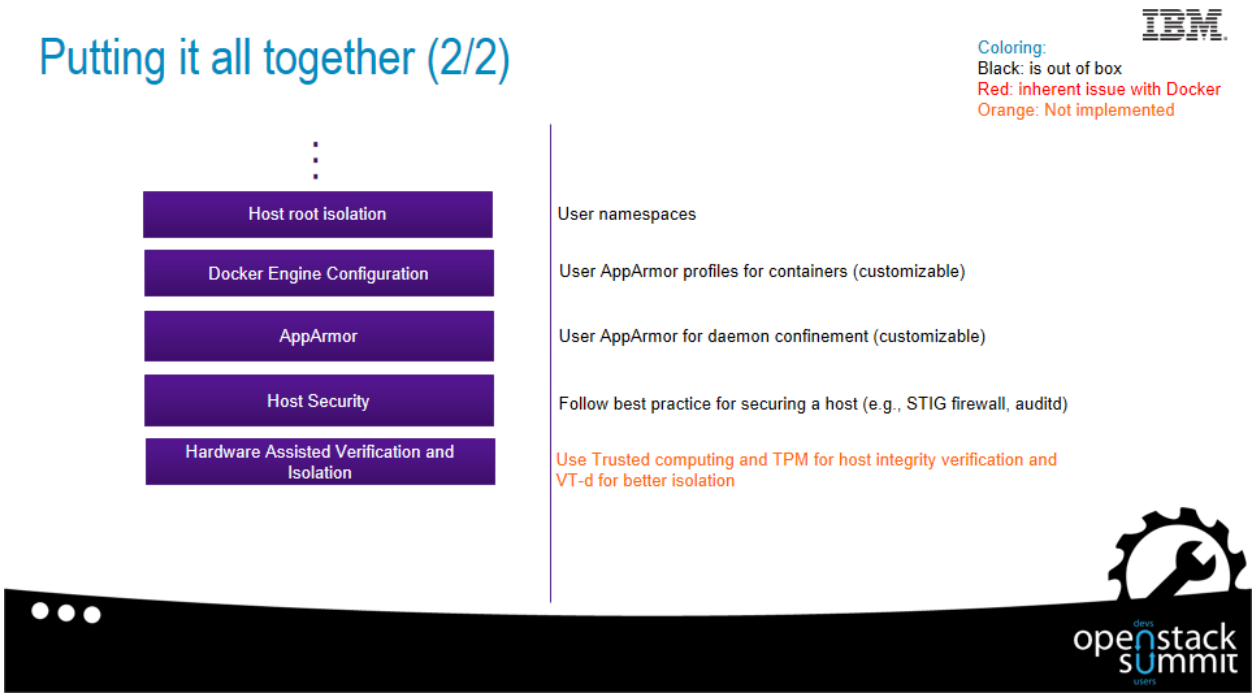
Since containers may require root privileges for various operations, such as for example adding users or updating installed packages, the danger exists that a root user inside a container may succeed in a jail break and gain root privileges on the host. Therefore, it is desirable to further deprivilege the container root user for any operations she/he may succeed in performing outside the container. **User namespace support** for containers helps to achieve this by remapping the user IDs inside a container to deprivileged user IDs (non-zero, non-root) on the host. With it the root user inside the container, identified by ID 0, becomes an arbitrary non-root user outside the container. This is a critical capability that greatly reduces the potential for

host compromise; it was initially identified by IBM which worked with the community to ensure a solution was introduced in Docker 1.10 [3].

Leveraging Container Isolation Technologies in the Cloud

As mentioned, container isolation on the host level is essential but not sufficient to achieve security for containers in a cloud environment. In a cloud environment it is important that the mentioned isolation and security technologies are applied to all containers and that the API of the container management stack, such as the docker management stack, is used appropriately. This means that API calls and parameters are filtered and the creation of containers with high privileges is prevented. The **management stack for containers** may allow much control over parameters and privileges a container is created with, exercising the capabilities for container isolation described above.
Not every container management stack fulfills this requirement. IBM has used containers in an OpenStack environment running on bare metal machines with Nova and Nova-docker drivers and has ensured that the above mentioned security and isolation technologies are applied appropriately. Other management stacks for Docker exist, such as Docker Swarm or Kubernetes, that need to be investigated individually to determine their suitability for serving as the management stacks in a multi-tenant cloud. OpenStack Magnum is an example of a management stack that was not designed for a multi-tenant container cloud environment but is currently most suitable for running containers inside VMs in a cloud where VMs of different tenants are isolated.





Related Virtualization Technologies

Linux container technologies are implemented through virtualization on the system call layer and make use of various Linux technologies to achieve isolation. Examples of Linux container technologies are Docker, LXC, Warden, and Parallel's Virtuozzo. The difference in the container technologies amounts to how they make use of the available namespace support. Some technologies may make use of all available namespaces, others only use a subset of them. They also generally differ on the level of the management stack where different features are supported, different command line utilities and APIs are implemented, and a different ecosystem exists. The performance, however, of all containers is expected to be the same.

Besides syscall virtualization, other prominent virtualization techniques include paravirtualization and full virtualization. Paravirtualization is a technique by which the kernel and device drivers are modified to work with the underlying hypervisor and hypervisor calls are implemented for transitioning from the OS kernel to the hypervisor. An example of such an implementation is the Xen hypervisor and an example for a paravirtualized kernel is implemented by Linux's Xen support. The main advantage of paravirtualization over full virtualization is the performance gain. Paravirtualized systems typically do not require the boot process to start with firmware, but launch an instance of a kernel directly.

Full virtualization on the other hand is primarily based on emulation of devices and generally requires support for hardware virtualization in the CPU to achieve good performance. The

emulation may range from emulation of systems and their original devices to the implementation of synthetic devices that are optimized for performance and that have no equivalent in form of a physical device. Also mapping of physical devices' addresses into the address space of a virtual machine (SRIOV) is a possibility to provide maximum performance. Full virtualization generally requires no changes to the operating system and kernel except for the installation of special device drivers, e.g. to support SRIOV devices, for the purpose of performance gain. Fully virtualized systems also typically start by invoking a firmware running inside the virtual machine and are capable of running any operating system without modifications. Full virtualization is implemented through a hypervisor that may either run directly on the underlying hardware (type 1) or a hypervisor that is implemented by an operating system kernel (type 2). Examples for type 1 hypervisors are Xen and VMWare ESX, while KVM/QEMU and VMWare Workstation are type 2 hypervisors. Both types require a privileged operating system that has access to several important hardware devices, such as disk controllers and network interfaces. Xen's privileged domain is domain-0 and for QEMU/KVM the equivalent is the Linux host.

Threat Model and Security Risks of Related Virtualization Technologies

A security comparison of containers versus virtual machines can be done along a few lines. For the following it should be noted that various security subsystems can be used to confine containers as well as type 2 hypervisors like QEMU/KVM. Supported Linux security subsystems are Linux Capabilities (sets of syscalls), Seccomp (syscalls), AppArmor (syscalls, file access, Linux Capabilties, networking), and SELinux (file and device access). For confining paravirtualized guest operating system, Xen implements the Xen Security Module (XSM) restricting access to hypercalls.

A clear advantage of containers can be shown when they run a well known application whose syscalls requirements are static and therefore can be subject to maximum confinement by minimizing the number of syscalls accessible while keeping the application functioning properly. However, a generic container that is for example used in a cloud setting and where a user may install any application into, cannot as easily be restricted since the syscall requirements are typically not known in advance. Here a generic policy is required that fits all containers' needs, thus leaving rather broad access to the syscall interface. A similarly restricted syscall interface is generally not possible to implement for type 2 hypervisors since they require a rather wide set of syscalls for proper functioning.

Another line of comparison may be the restriction of access of applications to syscalls. Linux currently implements more than 300 syscalls, of which a significant subset is typically available to applications running inside containers (though privileged ones may be disabled using Linux Capabilities for example). The QEMU/KVM type 2 hypervisor needs to interact with the host through syscalls and for that purpose requires access to around 216 syscalls (following its seccomp policy). While applications in containers can make direct usage of the host's syscall interface, while being limited by namespaces, applications running in a QEMU/KVM virtual machine need to find vulnerabilities in the QEMU emulator and device models to be able to

access them. Once an application has broken out of QEMU, it will be subject to restrictions imposed on the QEMU process, such as for example by an SELinux policy implemented by the sVirt policy. sVirt would in this case be able to prevent the process from accessing privileged user files if the QEMU process was subverted. The same type of SELinux policy can be applied to containers, restricting processes from accessing the host's and other containers' files in case of a jail-break. However, in all cases vulnerabilities may exist in the syscall implementations that provide access to kernel functions and lead to exploits or denial of service attacks (DOS), such as host crashes.

A quantification of the security risk of containers versus hypervisors is difficult. Containers share the same kernel with the host operating system and thus have the possibility to directly exploit system call vulnerabilities or bugs in the underlying kernel, though may be subject to confinement by security policies (sVirt, AppArmor) and namespace isolation. Virtual machines provide a higher level of isolation to the host's system call interface and, particularly if security policies are properly applied and high privileges are dropped. Besides that VMs require that two levels of vulnerabilities be exploited, one on the hypervisor (QEMU/KVM) and one in the underlying host kernel. Overall one can maybe say that the exploitation of vulnerabilities of the host is more difficult with virtual machines than with containers based on the fact that a type 2 hypervisor requires exploitable vulnerabilities on the emulator (QEMU) and host kernel level.

To help quantify the security risks one can investigate the frequency of CVEs found in the Linux kernel. To escape a secure Docker, an attacker in a container would need to find a privilege escalation attack on the shared kernel. Such kernel vulnerabilities occur roughly once a year. The last was discovered in May 2015. To escape a type 2 hypervisor, such as KVM, an attacker in a guest VM would need to become root in the VM and would need to find a vulnerability in QEMU. If security policies are applied to QEMU, file and device accesses on the host's filesystem can be prevented. Then the attacker must also find a privilege escalation attack on the native host kernel that is exploitable. QEMU vulnerabilities are discovered roughly once a year and the last was found in May 2016~~5~~. Therefore, statistically, over the past few years, it is roughly half as likely to find both a QEMU and kernel vulnerability at the same time, as just finding the kernel one, and this combination occurs roughly every 2 years.

To escape a type 1 hypervisor, such as Xen or VMware ESX, an attacker in a guest VM would need to become root in the VM, and would then need to find a vulnerability in the hypervisor. The last such vulnerability in ESX was found in February 2013, and this occurs roughly once every 2 years.

Further Strengthen Container Security

To further strengthen the security of containers, IBM Research is working on extending the Linux Integrity subsystem to work inside Linux containers and allow each tenant to individually manage its associated policy. Using the Integrity subsystem's appraisal capabilities, container user will be able to sign the files in their container's filesystem and enforce that only signed files can be executed. This effectively locks out any intruder from starting their malware in

containers, thus greatly reducing its attack surface. To achieve this, IBM Research is currently working on namespace support for the Linux integrity subsystem, which is a prerequisite to making it available to each container. Following support on the Linux level, the management stack of the container technology will be extended. The intention is to share the resulting implementation with the respective communities.

Conclusions and Summary

We have given an introduction on Linux containers and examined the technologies used for providing isolation security in Linux containers. Of particular importance are namespaces, cgroups, AppArmor and SELinux, Linux capabilities, and seccomp. It is important that these technologies are properly applied to achieve the required security level for allowing tenants to share hosts in a cloud. Therefore one must carefully examine and restrict the parameters that may be passed to a Linux container management stack since those parameters may influence the isolation and confinement properties of containers. Following this, it is important to closely examine cloud management stacks for how they interact with the underlying Linux container management stack, which in turn determines their suitability for allowing containers to share the same host. Further, we gave background on paravirtualization and full virtualization for a susequent comparison of the risk associated with running containers on bare metal machines versus virtual machines. The conclusion we reached is that there is certainly less risk running containers inside virtual machines, though the risk quantification is so that we believe that it is possible to allow different tenants containers to share the same host.

We also pointed out that IBM Research is working on novel technologies to further strengthen Linux container security.

References

1. X-Force Research and Development. " IBM X-Force Threat Intelligence Quarterly 4Q 2014," Doc # WGL03062USEN, Publish Date: Nov 2014. http://www.ibm.com/security/xforce/downloads.html
2. Docker Blog, Security: https://blog.docker.com/tag/docker-security/
3. Docker Engine 1.0 Security Improvements: https://blog.docker.com/2016/02/docker-engine-1-10-security/
4. https://insights.ubuntu.com/2015/05/18/lxd-crushes-kvm-in-density-and-speed/
5. Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio, An Updated Performance Comparison of Virtual Machines and Linux Containers, IBM Research Report, RC25482 (AUS1407-001), 2014

For More information

To learn more about IBM BlueMix please visit: http://www.ibm.com/cloud-computing/bluemix/