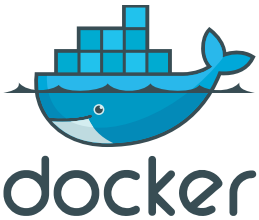
Introduction to Container Security

Understanding the isolation properties of Docker

August 2016



Executive Summary

Enterprises today are looking to transform software development practices to be agile to deliver more software faster. Container technology is arising as the preferred means of packaging and deploying applications. Docker brings a complete set of isolation capabilities to containerized applications with strong defaults out of the box to the ability for IT admins to granularly customize policies down to the exact syscall allowed on a host.

Out of the box, Docker allows you to:

- Isolate applications from each other
- Isolate applications from the host
- Improve the security of your application by restricting its capabilities
- Encourage adoption of the principle of least privilege

Regardless of which kind of application you intend to deploy in your infrastructure (off the shelf, legacy monolith or microservices), containers provide greater isolation at runtime and application integrity as it travels throughout the software development lifecycle. Docker runs on physical, cloud or virtual infrastructure allowing applications to be secured by container technology regardless of deployment.
This white paper discusses the architecture and isolation properties of Docker, containers and how to make effective use of them to secure your enterprise applications.

Key takeaways include:

- Containers provide an additional layer of protection by isolating applications from their host and from each other, while minimizing use of resources of the underlying infrastructure and reducing the surface area of the host itself.
- Containers and Virtual Machines (VMs) can be deployed together to provide additional layers of isolation and security for selected services.
- Docker provides the most complete set of security capabilities and ships with strong defaults in container technology.
- Applications packaged in containers are fundamentally more secure by default.

Introduction

Enterprises today are looking to transform business through software innovation. What is often discussed as "digital transformation" considers practices like DevOps, adoption of cloud technology and the modernization of applications. Evolving software development practices are transforming applications into collections of many small services, loosely coupled together into what are called microservices architecture. These new applications join an already vast portfolio of enterprise applications that have different architecture and processes surrounding them. Some legacy applications will also eventually be refactored to microservices while others will remain unchanged until they are retired.

In this context, Docker containers have become the de facto choice for packaging microservices applications. However, Docker container technology is not limited to microservices and can also be used to package off-the-shelf commercial and monolithic apps to provide the same isolation and portability properties. By adopting Docker, enterprises can increase agility, portability and control for all applications with a modern application platform.
Best practices for application security have long recommended a strategy that creates layers of protection to increase the overall resilience of a system. Reducing vulnerable surface area with a common OS is one such important layer. This paper demonstrates how Linux technology and Docker containers can be used to implement this strategy further, providing enhanced security, greater resource efficiency and simplified workflows throughout the application lifecycle.

Docker Overview

Docker is a platform used to build, ship, and run any applications anywhere. Organizations look to Docker to simplify and accelerate their application development and deployment process. Docker provides an easily composable and lightweight container that can change dynamically without disrupting the application as a whole. Docker containers are frictionlessly portable across development, test and production environments running locally on physical or virtual machines, in data centers, or across different cloud service providers.

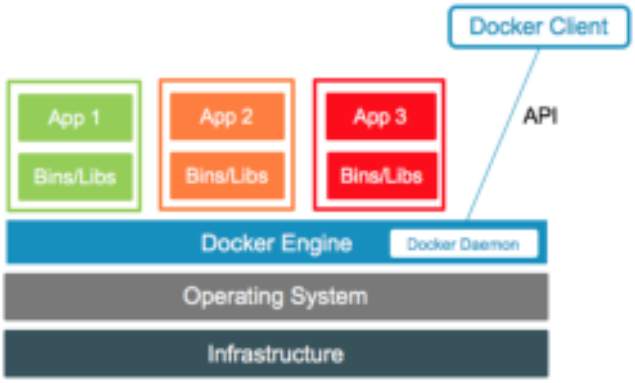
At the core is the Docker Engine, a lightweight application runtime with built in features for orchestration, scheduling, networking and security features to build and deploy single or multi-container applications. Docker Engines can be installed on any physical or virtual host running a Linux OS in a private datacenter or cloud. Docker containers are then deployed to run across the collection of Docker Engines. Containers allow developers to package large or small amounts of code and their dependencies together into an isolated package. This model then allows multiple isolated containers to run on the same host, resulting in better usage of

hardware resources, and decreasing the impact of misbehaving applications on each other and their host system. Docker containers are built from container images using layered file systems ensuring that the container itself contains only the elements needed to run the application. Images are managed and distributed from registries like Docker Cloud and Docker Trusted Registry with teams building and deploying applications. At runtime, the application containers are orchestrated, scheduled and managed from Docker Universal Control Plane. These technologies are available as an integrated platform, Docker Datacenter to power a modern application environment based on container technology.

Docker Engine Architecture

The Docker Engine uses a client-server architecture. A Docker *client* talks to the Engine's *daemon*, which does the heavy lifting of building, shipping, and running the Docker containers for a specific application service.

The Docker client can be run locally with the Docker for Mac or Windows desktop app. Both the client and daemon can run on the same system. Both the *client* and *daemon* can run on the same system, but clients can also access Docker Engines remotely. All communications between the client and daemon take place via a RESTful API and can be secured with TLS. Docker is written in Go, and the daemon uses several libraries and kernel features to deliver its functionality.



Linux Technology Best Practices and Docker Default Security

The Docker Engine supports the isolation features available in the Linux operating system and makes them available to the end user via the Docker Client. The isolation features are available out of the box with secure default settings while still allowing the user to adjust the configurations at any time.

Docker container technology increases the default security by creating isolation layers between applications and between the application and host and reducing the host surface area which protects both the host and the co-located containers by restricting access to the host.

This is in keeping with best practice recommendations for Linux systems administration as it applies the *principle of least privilege* to provide isolation and reduce vulnerable surface area. More specifically, system administrators have long been advised to chroot processes[1] and create resource restrictions around deployed applications. The Docker container model supports and enforces these restrictions by running applications in their own root filesystem, and by allowing the use of separate user accounts.

Docker goes a step further in approaching isolation as controlling what the containers can see or gain access to, what resources they can use and what they can do in relation to the host system and other containers. Docker containers provide application sandboxing and resource constraints with Linux *namespaces* and *cgroups*. While these powerful isolation mechanisms have been available in the Linux kernel for years, Docker provides simplified access to these capabilities, allowing administrators to create and manage the constraints on distributed applications containers as independent and isolated units.

Namespaces

Docker takes advantage of Linux *namespaces*[1] to provide the isolated workspace we call a *container*. When a container is deployed, Docker creates a set of namespaces for that specific container, isolating it from all the other running containers. The various namespaces created for a container include:

- PID Namespace: Anytime a program starts, a unique ID number is assigned to the namespace that is different than the host system. Each container has its own set of PID namespaces for its processes
- MNT Namespace: Each container is provided its own namespace for mount directory paths.
- NET Namespace: Each container is provided its own view of the network stack avoiding privileged access to the sockets or interfaces of another container.
- UTS Namespace: This provides isolation between the system identifiers; the hostname and the NIS domain name.
- IPC Namespace: The inter-process communication (IPC) namespace creates a grouping where containers can only see and communicate with other processes in the same IPC namespace

Cgroups

Docker also leverages Linux control groups. Control groups[2] (or *cgroups*) are a kernel level functionality that lets Docker control which resources each container can access, ensuring good container multi-tenancy. Control groups allow Docker to share available hardware resources and, if required, set up limits and constraints for containers. A good example of this applying limits to the amount of memory available to a specific container so it can't exhaust the host's resources.

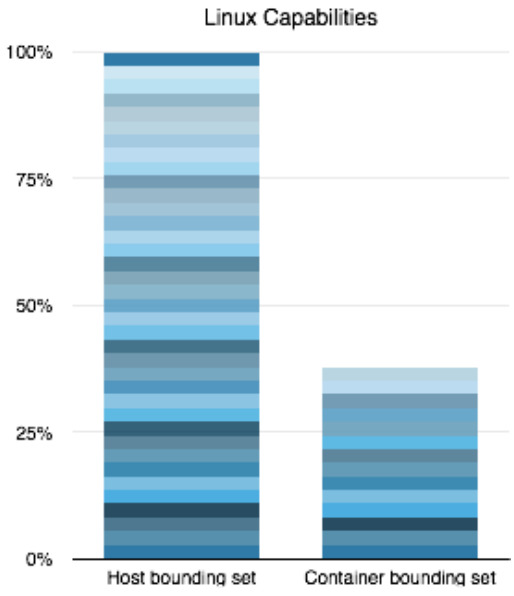
Seccomp

Docker Engine supports the use of secure computing mode (seccomp), a feature in the Linux kernel. This allows the administrator to restrict the actions available within a container down to the granularity of a single system call. This capability restricts the access that your application container has to the host system to perform actions. Enterprises can configure seccomp profiles that adhere to the different policies at your organization and apply them to the Docker environment.

Docker's default seccomp profile is a whitelist of calls that are allowed and blocks over 50 different syscalls. The vast majority of applications will be able to operate without issue with the default profile. In fact, the default profile has been able to proactively protect Dockerized applications from several previously unknown bugs in Linux. This is what we call a security non-event. Both the list of blocked syscalls and list of security non-events can be found in Docker documentation.

Process Restrictions

While the traditional view of Linux considers OS security in terms of root privileges versus user privileges, modern Linux has expanded to support a more nuanced privilege model: capabilities. Restricting both access and capabilities reduces the amount of surface area potentially vulnerable to attack.



Linux capabilities allow granular specification of user access. Traditionally, the root user has access to every capability; non-root users have a more restricted capability set, but have the option to elevate their access

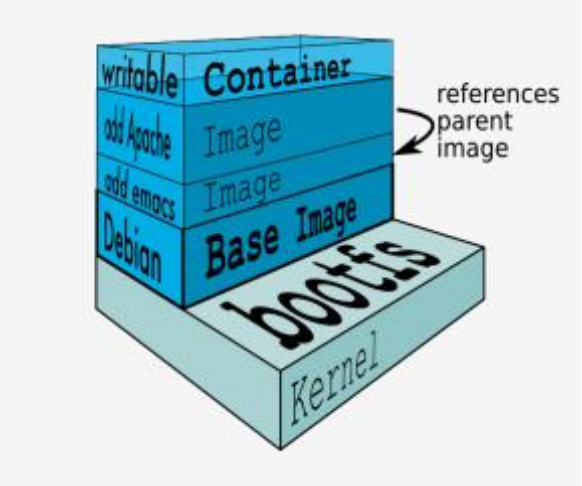
to root level through the use of *sudo* or *setuid* binaries. Doing this may constitute a security risk. Docker's default settings are designed to limit Linux capabilities and reduce this risk.

The default bounding set of capabilities inside a Docker container is less than half the total capabilities assigned to a Linux process (see figure). This reduces the possibility that application-level vulnerabilities could be exploited to allow escalation to a fully-privileged root user. Docker also employs an extra degree of granularity which dramatically expands on the traditional root/non-root dichotomy. In most cases, the application containers do not need all the capabilities attributed to the root user since a large majority of tasks requiring this level of privilege are handled by the OS environment outside the container. Consequently, containers can run with a reduced capability set without negatively impacting the application. This raises overall system security levels and makes running applications more secure by default. Because the container capabilities are fundamentally restricted, this also makes it difficult to provoke system level damages during intrusion, even if the intruder manages to escalate to root within a container.

Device and File Restrictions

In addition to process restrictions, Docker further reduces the attack surface by restricting access containerized applications have to the physical devices on a host. This is done with the device resource control groups (*cgroups*) mechanism. Containers have no default device access and have to be explicitly granted device access. These restrictions protect a container's host kernel and its hardware, whether physical or virtual, from the running applications.

Further, Docker containers use copy-on-write file systems, which allow use of the same file system image as a base layer for multiple containers. Even when writing to the same file system image, containers do not register the changes made by another container, thus effectively isolating running processes in independent containers.



Any changes made to containers are lost if you destroy the container, unless you commit your changes. Commits are added as a new layer to a base image, which allows a user to track and audit changes and, if desired, push up as a new image for storage in an image registry. This audit trail provides information which is important

for documenting and maintaining compliance. It also allows for fast and easy rollback to previous versions if a container has been compromised or a vulnerability introduced.

There are a few core Linux kernel system files that have to be in the container environment in order for applications to run. The majority of these mandatory files, such as */sys* and other files under */proc*, come mounted as *read-only.* This further limits the possibility of access, even by privileged container processes, which could potentially write to them.

Other Linux Kernel Security Features

Modern Linux kernels have many additional security constructs in addition to the aforementioned concepts of capabilities, namespaces and cgroups. Docker can leverage existing systems like TOMOYO, AppArmor, SELinux and GRSEC, some of which come with security model templates which are available out of the box for Docker containers. You can further define custom policies using any of these access control mechanisms.

Linux hosts can be hardened in many other ways and while deploying Docker enhances the host security, it also does not preclude the use of additional security tools. Specifically, we recommend users run Linux kernels with GRSEC and PAX. These patch sets add several kernel-level safety checks, both at compile-time and run-time, that attempt to defeat or make some common exploitation techniques more difficult. While not Docker-specific, these configurations can provide system-wide benefits without conflicting with Docker.

Secure by Default

Docker believes security should be inherent in the application platform and not a separate tool that then needs to be installed and configured to work with the system. Additional tools, systems and manual configuration introduces complexity and the opportunity for misconfiguration in addition to adding overhead to ongoing operations. With that in mind, Docker takes a “secure by default” approach to the security features in the Docker Engine. Not only are all the isolation properties of Linux supported in Docker with a simple user experience, they come out of the box with default configurations that provide greater protection for the applications.

NCC Group, an independent security firm, recently contrasted the security features and defaults of container platforms and published the findings in the paper “Understanding and Hardening Linux Containers.” Included is an examination of attack surfaces, threats, related

hardening features, a contrast of different defaults and recommendations across different container platforms. A key takeaway from this examination is the recommendation that applications are more secure by running in some form of Linux container than without. Container defaults provide a layer of protection while also providing enterprises a way to gain consistency and standardization without the addition of complicated configuration tooling.

This paper contrasted the security capabilities of three container technology providers and NCC found Docker Engine to provide the most complete set of security capabilities with the strongest defaults.

> *"In this modern age, I believe that there is little excuse for not running a Linux application in some form of a Linux container, MAC or lightweight sandbox."*
>
> *– Aaron Grattafiori, NCC Group*

Available Container Security Features, Requirements and Defaults			
Security Feature	**LXC 2.0**	**Docker 1.11**	**CoreOS Rkt 1.3**
User Namespaces	Default	Optional	Experimental
Root Capability Dropping	Weak Defaults	Strong Defaults	Weak Defaults
Procfs and Sysfs Limits	Default	Default	Weak Defaults
Cgroup Defaults	Default	Default	Weak Defaults
Seccomp Filtering	Weak Defaults	Strong Defaults	Optional
Custom Seccomp Filters	Optional	Optional	Optional
Bridge Networking	Default	Default	Default
Hypervisor Isolation	Coming Soon	Coming Soon	Optional
MAC: AppArmor	Strong Defaults	Strong Defaults	Not Possible
MAC: SELinux	Optional	Optional	Optional
No New Privileges	Not Possible	Optional	Not Possible
Container Image Signing	Default	Strong Defaults	Default
Root Interation Optional	True	False	Mostly False

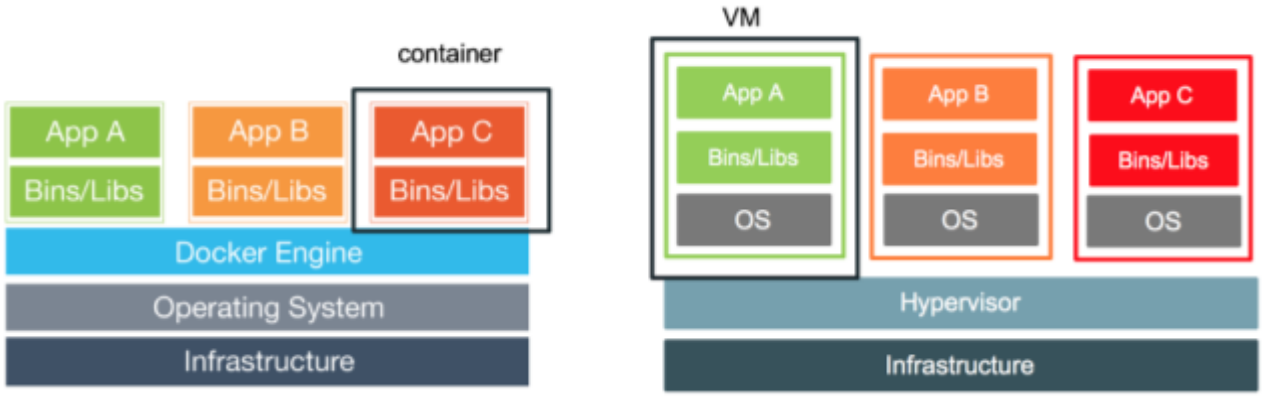
Source: Understanding and Hardening Linux Containers

Deployment Considerations: Physical or Virtual Infrastructure

Both containers and VMs provide isolated environments for running applications on a shared host, albeit from different technical perspectives. Containers and VMs can be used successfully together or separately, depending on the needs of the application environment.

Docker containers share a single host OS kernel across all of the application containers running on that machine. Isolation is provided on a per-container level by the Docker Engine. Using containers, multiple applications can be deployed to a single bare metal server without any conflict between the applications. This also provides improved utilization of the resources on that physical server. The lightweight nature of the container makes them faster, and easier to scale up or down. This approach is only possible for application services that share a common OS.

In contrast, virtual machines have a complete OS, including memory management and virtual device drivers. Isolation is provided at the virtual machine level with resources being emulated for the guest OS. This lets VMs provide a barrier between application processes and bare-metal hosts because the hypervisor prevents a VM from executing instructions which could compromise the integrity of the host platform. Protecting the host relies upon providing a safe virtual hardware environment on which to run an OS. This architecture makes considerable demands on host resources, but allows for applications with different OSs to run on a single host.



Note: Bare metal deployment does not provide ring-1 hardware isolation, given that it cannot take full advantage of hardware-assisted virtualization primitives such as Intel's VT-d and VT-x technologies. In this scenario, containerization cannot provide the same level of host isolation that it can in tandem with virtualization.

Containerization does provide isolation for running applications on bare-metal, which protects the machine from a large array of threats and is sufficient for a wide range of use cases. Users in the following scenarios may not be good candidates to use VMs, so using containers alone may be the best choice:

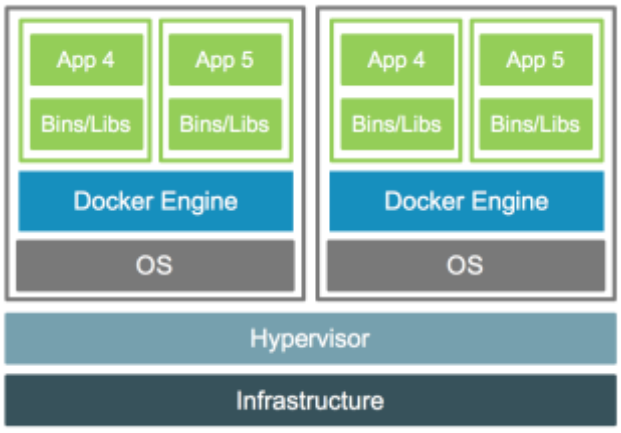
- Running performance-critical applications on a single-tenant private cloud, where cross-tenant or cross-application attacks are not as much of a concern;
- Using specialized hardware which cannot be passed through to a VM,
- Using hardware that offers direct-memory-access, thus nullifying the isolation benefits of virtualization. Many users of GPU computing are in this position.

Docker containers running on bare-metal have the same high-level restrictions applied to them as they would if running on virtual machines. In neither case would a container normally be allowed to modify devices or hardware, neither physical nor virtual.

Container and VMs Together

The nature of standard VMs means that they cannot be efficiently scaled down to the level of running a single application service. While a VM can support a relatively rich set of applications, running multiple microservices in a single VM without containers creates conflicts and running one microservice per VM is inefficient from a cost and resource utilization standpoint. Deploying Docker containers in conjunction with VMs allows an entire group of services to be isolated from each other and then grouped inside of a virtual machine host.

This provides two layers of isolation, containers and VMs, to the application. This lends itself well for multi-tenant style environments where the origin and contents of the other workloads are unknown. Additionally, reducing the number of VMs and OS instances fundamentally decreases the amount of vulnerable surface area. Docker containers pair well with virtualization technologies by protecting the virtual machine itself and providing defense-in-depth for the host.



Conclusion

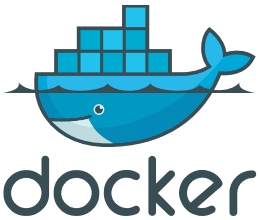
Microservices based architecture introduces different requirements for how they are developed, packaged, deployed, secured and managed across their lifecycle. These microservices are joining a portfolio of applications that include vendor off-the-self packages and custom built software. Security needs to be approached in layers that address the entire infrastructure to diverse application stack and may call for a combination of technologies to ensure the required level of security for each layer.

Docker containers simplify the deployment of the *defense-in-depth* concept in organizations for all applications. Using Docker brings immediate benefits, not only in terms of speed and ease of application development and deployment, but also in terms of security. The simple deployment of Docker increases the overall system security levels by default, through isolation, confinement, and by implicitly implementing a number of best-practices that would otherwise require explicit configuration in every OS used within the organization.

To summarize, organizations can enhance security with the use of Docker containers without adding incremental overhead to their application infrastructure because:

- Containers provide isolation for applications from their host and from each other, while minimizing use of resources of the underlying infrastructure and reducing the surface area of the host itself.
- Containers and Virtual Machines (VMs) can be deployed together to provide additional layers of isolation and security for selected services.
- Docker provides the most complete set of security capabilities with strong defaults in container technology.
- Applications packaged in containers are fundamentally more secure by default.

[1] https://wiki.debian.org/chroot
[2] http://man7.org/linux/man-pages/man7/namespaces.7.html
[3] https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt



www.docker.com