Introduction to Container Security

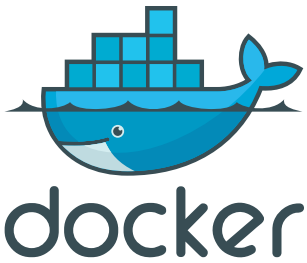


Table of Contents

Executive Summary

In recent years evolving software development practices have fundamentally changed applications. These changes have impacted the requirements to the underlying infrastructure, tools, and processes to manage applications properly throughout the lifecycle. Applications transformed from large monolithic code bases to collections of many small services, loosely coupled together into what is called a microservices architecture. Driven by the desire to ship more software, faster in a way that is portable across infrastructure, the resulting benefits of more agile organizations can be seen all around us, from the smallest start ups to the largest of enterprises.

These new applications not only behave differently but their architecture fundamentally changes how they are built, deployed, managed and secured over time. Instead of provisioning large servers to process a few large workloads in virtual machines (VM) or bare metal, collections of small applications are being run across a collection of commodity hardware. With applications sharing the same OS, containers have risen as the model for packaging these new applications. Fewer OS instances provide significant benefits to the application infrastructure with respect to host resources, costs and ongoing maintenance.

The best practices around application security have long recommended the strategy of layers in order to increase the overall resilience of a system. In this paper we will introduce the security concepts around Linux technology and Docker containers.

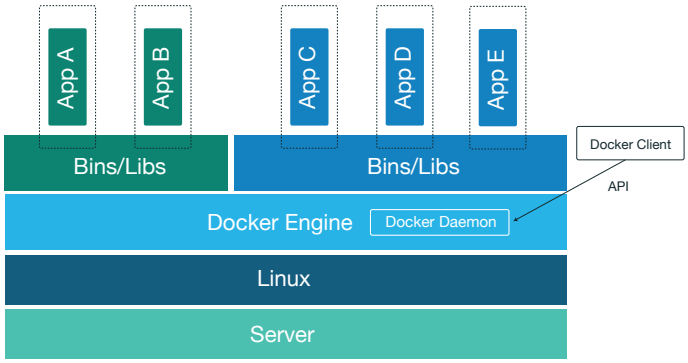
Key Takeaways Include:

- Containers provide an additional layer of protection by isolating between the applications and the host, and between the applications themselves without using incremental resources of the underlying infrastructure and by 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 application services.
- The nature of containers facilitates fast and easy application of patches and updates to the OS, application and infrastructure layers, helping maintain overall security compliance.

Docker Overview

Docker is an open platform to build, ship, and run distributed applications. Organizations approach Docker to simplify and accelerate their application development and deployment process. Docker allows distributed applications to be easily composable as a lightweight application container that can change dynamically yet non-disruptively to the application and ultimately be frictionlessly portable across development, test and production environments running on physical or virtual machines locally, in data centers or across different cloud service providers.

At the core of the platform is the Docker Engine, a lightweight application runtime which also provides robust tooling to build, ship and run Docker application containers. Docker Engines are installed on any host running a Linux OS like physical and virtual machines or servers on laptops, in data centers or cloud service providers. Docker application containers are then deployed to run on the Docker Engine. The Docker container model allows multiple isolated application containers to run on the same server, 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 application images that are stored, managed and distributed from a registry like Docker Hub. There are public, private and official image repositories available on Docker Hub from the community and ecosystem for use by Docker users.



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 may be run as a command line utility, or integrated with third-party applications via the Docker API. 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 can be secured with TLS and performed via a RESTful API. Docker is written in Go, and the daemon uses several libraries and kernel features to deliver it's functionality.

Linux Technology Best Practices and Docker Default Security

Container technology increases the default security for applications in two ways. First by applying an isolation layer between applications and between the application and the host. Secondly, it reduces the host surface area to protect both the host and the co-located containers by restricting access to the host. Docker containers build upon Linux best practices to provide stronger defaults and configurable settings to further reduce risk.

Best practice recommendations for Linux systems administration include the application of the *principle of least privilege*. System administrators have long been advised to chroot processes[1] and

[1] https://wiki.debian.org/chroot

create resource restrictions around deployed applications. The Docker container model supports and enforces these restrictions by running applications in their own root filesystem, allows the use of separate user accounts, and goes a step further to provide application sandboxing using Linux *namespaces* and *cgroups* to mandate resource constraints. While these powerful isolation mechanisms have been available in the Linux kernel for years, Docker brings forward and greatly simplifies the capabilities to create and manage the constraints around distributed applications containers as independent and isolated units.

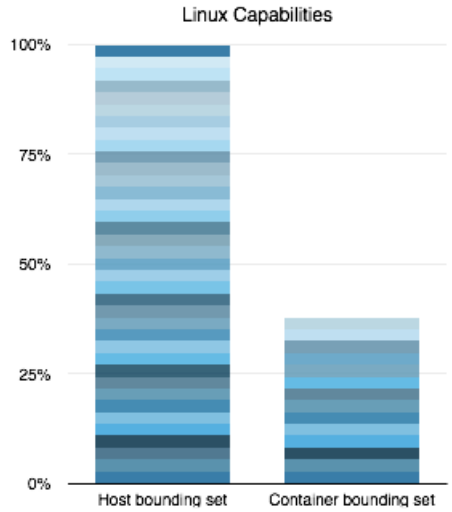
Docker takes advantage of a Linux technology called *namespaces*[2], to provide the isolated workspace we call *container*. When a container is deployed, Docker creates a set of namespaces for that specific container, isolating it from all the other running applications.

Docker also leverages Linux control groups. Control groups[3] (or *cgroups* for short), are the kernel level functionality that allows Docker to control what resources each container has access to, 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 is limiting the amount of memory available to a specific container, so it doesn't completely exhaust the resources of the host.

Process Restrictions

Restricting access and capabilities reduces the amount of surface area potentially vulnerable to attack. Docker's default settings are designed to limit Linux capabilities. While the traditional view of Linux considers OS security in terms of root privileges versus user privileges, modern Linux has evolved to support a more nuanced privilege model: capabilities.

Linux capabilities allow granular specification of user capabilities and traditionally, the root user has access to every capability. Typical non-root users have a more restricted capability set, but are usually given the option to elevate their access to root level through the use of *sudo* or *setuid* binaries. This may constitute a security risk.

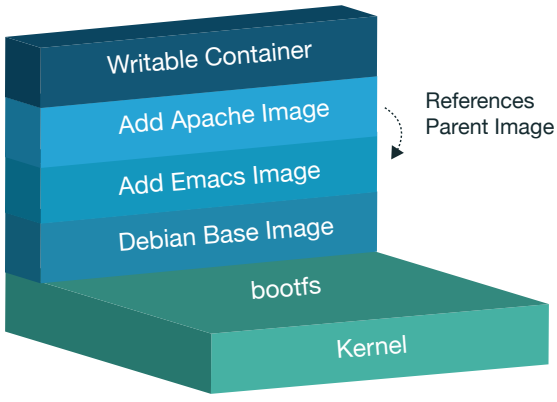


The default bounding set of capabilities inside a Docker container is less than half the total capabilities assigned to a Linux process (see Linux Capabilities figure). This reduces the possibility of escalation to a fully privileged root user through application-level vulnerabilities. Docker employs 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 the large majority of the tasks requiring this level of privilege are handled by the OS environment external to the container. Containers can run with a reduced capability set that does not negatively impact the application and yet improves the overall security system levels and makes running applications more secure by default. This makes it difficult to provoke system level damages during intrusion, even if the intruder manages to escalate to root within a container because the container capabilities are fundamentally restricted.

Device & File Restrictions

Docker further reduces the attack surface by restricting access by containerized applications to the physical devices on a host, through the use of 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 host kernel and its hardware, whether physical or virtual, from the running applications.

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 notice the changes made by another container, thus effectively isolating the processes running in independent containers.



Any changes made to containers are lost if you destroy the container, unless you commit your changes. Committing changes tracks and audits changes made to base images as a new layer which can then be pushed as a new image for storage in Docker Hub and run in a container. This audit trail is important in providing information to maintain 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 file systems that have to be in the container environment

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

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 ability of access, even by privileged container processes, to potentially write to them.

Application Image Security

Docker allows container images to originate from disk or from a remote registry. The Docker Hub is a SaaS registry that contains both public images and a number of official image repositories that are publically searchable and downloadable by Docker users. Official repositories are certified repositories from Independent Software Vendors (ISV) and Docker contributors like Canonical, Oracle, RedHat and many more. These images are maintained and supported by these upstream partners to ensure up-to-date images. Docker is agnostic to which remote registry is in use, allowing companies to run any combination of homegrown registries, public cloud-based registries or Docker Hub.

Proper tooling around application images are critical to sound security practices. IT Admins need to quickly and easily apply application updates whether planned or unplanned and get those changes deployed to the relevant applications. The easily composable and dynamic nature of Docker containers make it suitable to allow for constant change with little disruption to the rest of the composed application.

Secure communications is also vital to building and shipping applications, as container images are in constant change and need to be pushed and pulled through your infrastructure. All communications with the registries use TLS, to ensure both confidentiality and content integrity. By default, the use of certificates trusted by the public PKI infrastructure is mandatory, but Docker allows the addition of a company internal CA root certificate to the truststore.

Beyond Defaults: Docker Enhanced Controls

Docker users can expand upon the default configuration to further restrict the access an application can have to use Linux capabilities or modify files. Operators can deploy containers with a read-only file system, significantly restricting the potential of container born processes from compromising the system through implicit device access.

The bounding capabilities of Docker containers may also be restricted beyond the Docker default through the 'cap_drop' feature. Docker is also compatible with seccomp (secure computing mode), and an example is included in the '*contrib*' directory of the Engine source tree, demonstrating this integration. Using *seccomp*, selected syscalls may even be entirely disabled for container processes.

These configuration controls described above are provided as a run-time facility, where isolation, orchestration, scheduling and application management mechanisms are concentrated. This allows users to tune and configure the policies of their containers, based on the user´s operational and organizational needs.

Open-Source Security

Docker Engine is an open-source project within the Docker project, and can be obtained at https://github.com/docker/docker. As an open source project we work closely with the community for innovation and to solve issues as they arise. The published contribution guidelines are strict with regards to using code-reviews for ensuring an ongoing high-quality code-base. Docker Inc. also contracts with external security firms to engage in quarterly audits and/or penetration tests of both our code and infrastructure. Docker supports responsible disclosure of security vulnerabilities and allows security researchers to submit potential vulnerabilities for remediation online: https://www.docker.com/resources/security/

Other Linux Kernel Security Features

Modern Linux kernels have many additional security constructs in addition to the referred concept of capabilities, namespaces and cgroups. Docker can leverage existing systems like TOMOYO, AppArmor, SELinux and GRSEC, some of which come with security model templates 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 deploying Docker enhances the host security but also does not preclude the use of additional security tools. We recommend you run your Linux kernels with GRSEC and PAX. These sets of patches 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. This is not a Docker-specific configuration but can be applied and be beneficial system-wide.

Deploying Docker on your Infrastructure

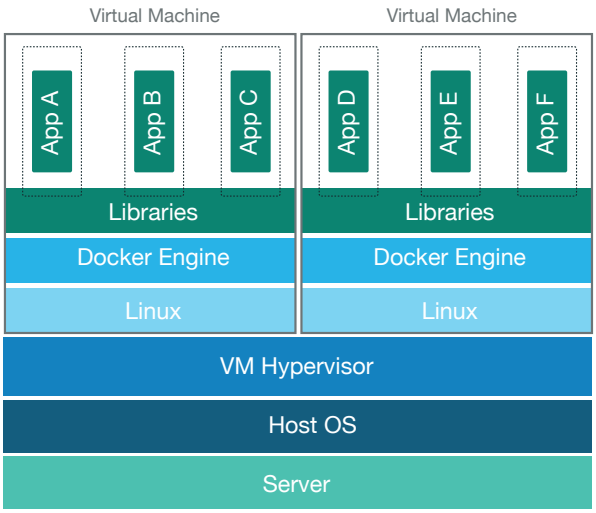
Combining Containers and Virtual Machines

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

Virtual machines have a full OS, including its own memory management and virtual device drivers. Isolation is provided at the virtual machine level with resources being emulated for the guest OS, allowing for one or more parallel (and, eventually, different) OS to run on a single host. VMs provide a barrier between application processes and bare-metal systems. The hypervisor denies 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 for which to run an OS. This architecture has different implications for host resource utilization, but allows for applications with different OS's to run on a single host.

Docker containers share a single host OS across all of the application containers running on that same host. Isolation is provided on a per application level by the Docker engine. Using containers reduces the overhead used per application because the multiple OS instances are avoided. This makes containers lighter weight, faster and easier to scale up or down and can gain higher density levels than full VMs. This approach is only possible for applications that share a common OS, like Linux is used for distributed applications.

Regular VMs function in a way that does not allow them to be efficiently scaled down to the level of running a single application service. A VM can support a relatively rich set of applications but running multiple microservices in a single VM without containers creates conflict issues while running one microservice per VM may not be financially feasible for some organizations. 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. This approach increases security by introducing two layers, containers and VMs, to the distributed application. Additionally this method employs a more efficient use of resources and can increase the density of containers while decrease the number of VMs required for the defined isolation and security goals.

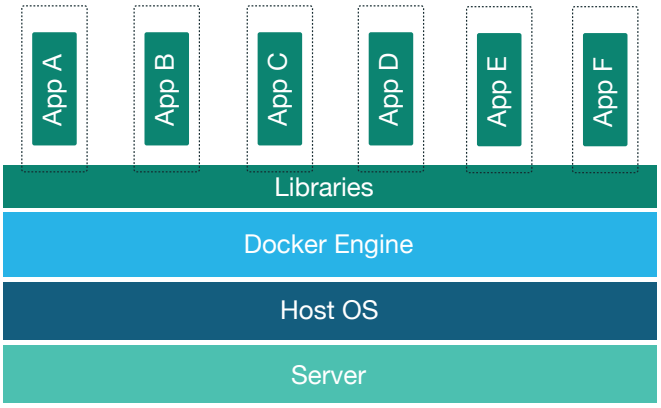


Therefore stronger application isolation can be achieved by combining virtualization and containerization than is cost and resource efficient with virtualization alone. Docker containers pair well with virtualization technologies, by protecting the virtual machine itself, and providing defense-in-depth for the host.

Running Containers on Bare-Metal

Containers provide a layer of protection between the host and its applications, isolating between application and the host. This makes it safer to deploy applications on bare-metal when compared to not using any virtualization or container technology. With containers, many application services can be deployed on a single host enabling organizations to gain higher levels of resource utilization out of their infrastructure.

However, bare metal deployment does not provide ring-1 hardware isolation, given that it cannot take full advantage of Intel's VT-d and VT-x technologies. In this scenario containerization is not a complete replacement of virtualization for host isolation levels.



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 and can instead use containers; performance-critical applications running on a single-tenant private cloud, where cross-tenant or cross-application attacks are not as much of a concern; or they are using specialized hardware which cannot be passed through to a VM, or which 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, either physical or virtual.

Lifecycle Management

Software development is just one phase of the application lifecycle. The security requirements of the test, run and manage stages are just as critical and also introduce different teams, tools and processes. Ongoing maintenance and updates whether planned or unplanned are required to keep the operations running safe and smoothly. The process of updating components of an application is often non-trivial and even worse, cumbersome even with tools that automate release of patches. Questions arise on whether the applied patch breaks something else, how difficult it is to even apply the patch and then how to cleanly roll back should the update fail. The convenience of building images is extended to updating these images and applying security updates. Operations engineers can safely and easily apply planned security updates to their images in an atomic fashion, by leveraging image tags and applying updates to prior versions. Unplanned patches and hotfixes may be applied similarly by committing, updating, and respawning container images. Updated images can be pushed to Docker Hub or any other registry to be easily shared and distributed with other teams.

These characteristics of Docker containers make it trivial for system administrators to update system utilities, packages, and even the kernel of the base hosts. OS upgrade cycles are dramatically reduced and easier to apply therefore, significantly reducing the overall exposure of the infrastructure to attacks. Docker environments also have one OS per host requiring maintenance compared to the multiples of OS's per host in the virtualization model. Reducing the total number of OS instances to patch and update reduces the operational overhead associated with security compliance.

Ensuring security compliance is equal parts applying the latest updates and keeping track of the systems, what they contain and their change history. Being able to audit a production container's change history can prove that no malicious alterations have been done to production systems, and provide the ability to roll-back the containers to a known good state after a compromise. Finally, the use of layers make it easy to check OS packages and application dependencies against public Common Vulnerabilities and Exposures (CVE) lists.

Conclusion

Microservices based architecture creates different requirements into how applications are developed, deployed and managed across their lifecycle and changes the packaging tools and security models needed to support them. Security needs to be approached in layers that address the entire infrastructure to application stack and may often be a combination of technologies to ensure the required level of security for each layer.

Docker simplifies the deployment of the *defense-in-depth* concept in organizations. Using Docker brings immediate benefits, not only in terms of application development and deployment speed and ease, 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.

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

- Containers provide an additional layer of protection by isolating between the applications and the host, and between the applications themselves without using incremental resources of the underlying infrastructure and by 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.
- The nature of containers facilitates fast and easy application of patches and updates to the OS, application and infrastructure layers, helping maintain overall security compliance.

www.docker.com

March 18, 2015