The attached DRAFT document (provided here for historical purposes), originally posted on April 10, 2017, has been superseded by the following publication:

Publication Number: **NIST Special Publication (SP) 800-190 (2nd Draft)**

Title: ***Application Container Security Guide***

Publication Date: **July 2017**

- Information about the attached Draft publication can be found at: https://csrc.nist.gov/publications/detail/sp/800-190/archive/2017-07-13
- Information on other NIST Computer Security Division publications and programs can be found at: https://csrc.nist.gov/publications

Draft NIST Special Publication 800-190

Application Container Security Guide

Murugiah Souppaya
John Morello
Karen Scarfone

C O M P U T E R   S E C U R I T Y

National Institute of
Standards and Technology
U.S. Department of Commerce

Draft NIST Special Publication 800-190

Application Container Security Guide

Murugiah Souppaya
Computer Security Division
Information Technology Laboratory

John Morello
Twistlock
Baton Rouge, Louisiana

Karen Scarfone
Scarfone Cybersecurity
Clifton, Virginia

April 2017

U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Kent Rochford, Acting Under Secretary of Commerce for Standards and Technology and Acting Director

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 *et seq.*, Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

National Institute of Standards and Technology Special Publication 800-190
Natl. Inst. Stand. Technol. Spec. Publ. 800-190, 56 pages (April 2017)
CODEN: NSPUE2

Public comment period: *April 10, 2017* through *May 18, 2017*

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: 800-190comments@nist.gov

All comments are subject to release under the Freedom of Information Act (FOIA).

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Abstract

Application container technologies, also known as containers, are a form of operating system virtualization combined with application software packaging. Containers provide a portable, reusable, and automatable way to package and run applications. This publication explains the potential security concerns associated with the use of containers and provides recommendations for addressing these concerns.

Keywords

application; application container; application software packaging; container; container security; isolation; operating system virtualization; virtualization

Acknowledgements

The authors wish to thank their colleagues, who have reviewed drafts of this document and contributed to its technical content during its development, in particular Raghuram Yeluri from Intel Corporation, Paul Cichonski from Cisco Systems, Inc., and Michael Bartock and Jeffrey Cichonski from NIST.

Audience

The intended audience for this document is system and security administrators, security program managers, information system security officers, and others who have responsibilities for or are otherwise interested in the security of application container technologies.

This document assumes that readers have some operating system, networking, and security expertise, as well as expertise with virtualization technologies (hypervisors and virtual machines). Because of the constantly changing nature of application container technologies, readers are encouraged to take advantage of other resources, including those listed in this document, for more current and detailed information.

Trademark Information

Executive Summary

Operating system (OS) virtualization provides a virtualized OS for each application to keep each application isolated from all others on the server. Each application can only see and affect itself. Recently, OS virtualization has become increasingly popular due to advances in its ease of use and an increased focus in developer agility as a key benefit. Today's OS virtualization technologies are primarily focused on providing a portable, reusable, and automatable way to package and run apps. The terms *application container* or simply *container* are frequently used to refer to these technologies.

The purpose of the document is to explain the security concerns associated with container technologies and make practical recommendations for addressing those concerns when planning for, implementing, and maintaining containers. Many of the recommendations are specific to a particular layer within the container technology stack, which is depicted in Figure 1.

Organizations should follow these recommendations to help ensure the security of their container stack implementations and usage:

Tailor the organization's processes to support the new way of developing, running, and supporting applications made possible by containerization.

The introduction of containerization technologies might disrupt the existing culture and software development methodologies within the organization. Traditional development practices, patching techniques, and system upgrade processes might not directly apply to a containerized environment, and it is important that the employees within the organization are willing to adapt to a new model. New processes can consider and address any potential culture shock that is introduced by the technology shift. Education and training can be offered to anyone involved in the software development lifecycle.

Use container-specific OSes instead of general-purpose ones to reduce attack surfaces.

A container-specific OS is a minimalist OS explicitly designed to only run containers, with all other services and functionality disabled, and with read-only file systems and other hardening practices employed. When using a container-specific OS, attack surfaces are typically much smaller than they would be with a general-purpose OS, so there are fewer opportunities to attack and compromise a container-specific OS. Accordingly, whenever possible, organizations should use container-specific OSes to reduce their risk. However, it is important to note that container-specific OSes will still have vulnerabilities over time that require remediation.

Automate compliance with container runtime configuration standards to minimize vulnerabilities.

Organizations should have a configuration standard for each type of container runtime they use that establishes the requirements for the container runtime's configuration settings. Deviations from the standard could create weaknesses that attackers can take advantage of to compromise the container runtime or the containers running on top of the runtime. Accordingly, organizations

should use tools or processes that continuously assess container runtime configuration settings and immediately act to correct any deviations from the approved standard.

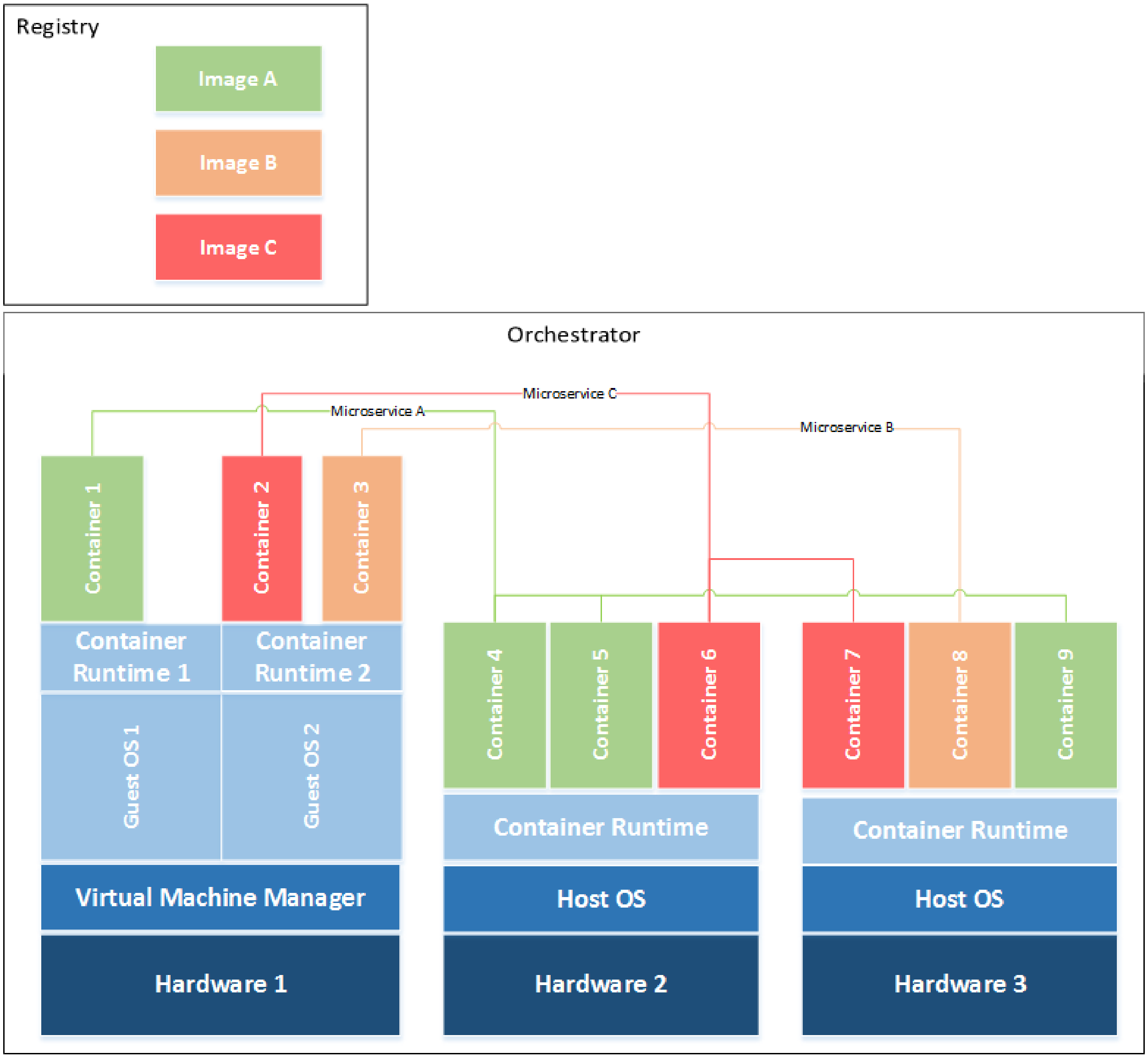

Figure 1: Container Technology Stack

Group containers by relative sensitivity and only run containers of a single sensitivity level on a single host OS kernel for additional defense in depth.

While most container runtime environments do an effective job of isolating containers from each other and from the host OS, in some cases it may be an unnecessary risk to run apps of different classification levels together on the same host OS. Grouping containers by purpose and sensitivity provides additional defense in depth. By grouping containers in this manner, it will be much more difficult for an attacker who compromises one of the groups to expand that compromise to other groups. This approach also ensures that any residual data, such as caches or local volumes mounted for temp files, stays within its security zone.

In larger-scale environments with hundreds of hosts and thousands of containers, this grouping must be automated to be practical to operationalize. Fortunately, common orchestration platforms typically include some notion of being able to group apps together, and container security tools can use attributes like container names and labels to enforce security policies across them.

Adopt container-specific vulnerability management tools and processes for images to prevent compromises.

Traditional vulnerability management tools make many assumptions about host durability, app update mechanisms, and update frequencies that are fundamentally misaligned with a containerized model. These tools are often unable to detect vulnerabilities within containerized stacks, leading to a false sense of safety. Organizations should use tools that take the pipeline-based build approach and immutable nature of containers and images into their design to provide more actionable and reliable results.

These tools and processes should take both image software vulnerabilities and configuration settings into account. Organizations should adopt tools and processes to validate and enforce compliance with secure configuration best practices for images. This should include having centralized reporting and monitoring of the current compliance state of each image, and preventing non-compliant images from being run.

Consider using hardware-based countermeasures to provide a basis for trusted computing.

Security should extend across all layers of the container stack. The current way of establishing trusted computing for all layers is to use a hardware root of trust. Within this trust is stored measurements of the host's firmware, software, and configuration data. Validating the current measurements against the stored measurements before booting the host provides assurance that the host can be trusted. The chain of trust rooted in hardware can be extended to the OS kernel and the OS components to enable cryptographic verification of boot mechanisms, system images, container runtimes, and container images. Trusted computing provides the most secure way to build, run, orchestrate, and manage containers.

Table of Contents

List of Appendices

List of Tables and Figures

1 Introduction

1.1 Purpose and Scope

The purpose of the document is to explain the security concerns associated with application container technologies, also known as containers, and make practical recommendations for addressing those concerns when planning for, implementing, and maintaining containers. The recommendations are intended to apply to most or all application container technologies.

All forms of virtualization other than application containers, such as virtual machines, are outside the scope of this document.

In addition to application container technologies, the term "container" is used to refer to concepts such as software that isolates enterprise data from personal data on mobile devices, and software that may be used to isolate applications from each other on desktop operating systems. While these may share some attributes with application container technologies, they are out of scope for this document.

This document assumes readers are already familiar with securing the technologies supporting and interacting with application container technologies. These include the following:

- The layers under application container technologies, including hardware, hypervisors, and operating systems;
- The client endpoint devices that use the applications within the containers; and
- The administrator endpoints used to manage the applications within the containers and the containers themselves.

Appendix A contains pointers to resources with information on securing these technologies. Sections 3 and 4 offer additional information on security considerations for container-specific operating systems. All further discussion of securing the technologies listed above is out of scope for this document.

1.2 Document Structure

The remainder of this document is organized into the following sections and appendices:

- Section 2 introduces containers, including their architectures, technical capabilities, attributes, and uses.
- Section 3 explains the major risks in the container technology stack.
- Section 4 discusses possible countermeasures for the risks identified in Section 3 and makes recommendations for selecting and using countermeasures.
- Section 5 defines threat scenario examples for containers.
- Section 6 presents actionable information for planning, implementing, operating, and maintaining a container technology stack.

- Section 7 provides a conclusion for the document.
- Appendix A lists NIST resources for securing systems and system components outside the container technology stack.
- Appendix B lists the NIST Special Publication 800-53 security controls and NIST Cybersecurity Framework subcategories that are most pertinent to application container technologies, explaining the relevancy of each.
- Appendix C provides an acronym and abbreviation list for the document.
- Appendix D presents a glossary of selected terms from the document.
- Appendix E contains a list of references for the document.

2 Introduction to Application Containers

NIST Special Publication (SP) 800-125 [1] defines *virtualization* as "the simulation of the software and/or hardware upon which other software runs." Virtualization has been in use for many years, but it is best known for enabling cloud computing. In cloud environments, *hardware virtualization* is used to run many instances of operating systems (OS) on a single physical server while keeping each instance separate. This allows more efficient use of hardware and supports multi-tenancy.

In hardware virtualization, each OS instance interacts with virtualized hardware. Another form of virtualization known as *operating system virtualization* has a similar concept; it provides a virtualized OS for each application to keep each application isolated from all others on the server. Each application can only see and affect itself.

Until recently, OS virtualization has not been widely used because hardware virtualization was considered easier to set up and run in order to achieve isolation. However, OS virtualization has become increasingly popular due to advances in its ease of use and an increased focus in developer agility as a key benefit. Today's OS virtualization technologies are primarily focused on providing a portable, reusable, and automatable way to package and run apps. The terms *application container* or simply *container* are frequently used to refer to these technologies. The term is meant as an analogy to shipping containers, which provide a standardized way of grouping disparate contents together while isolating them from each other.

Containers themselves are not new; various implementation of containers have existed since the early 2000s, starting with Solaris Zone and FreeBSD jails. Support initially became available in Linux in 2008 with the Linux Container (LXC) technology built into nearly all modern distributions. More recently, projects such as Docker and rkt have provided additional functionality designed to make OS component isolation features easier to use and scale. Container technologies are also available on the Windows platform beginning with Windows Server 2016. The fundamental architecture of all these implementations is consistent enough so that this document can discuss containers in detail while remaining implementation agnostic.

This section provides an introduction to containers for servers. First, it explains the architecture of containers, including all the major components typically found in a container implementation. Next, it describes the major technical capabilities and fundamental attributes of containers. Finally, the section briefly lists common uses for containers.

2.1 Container Architecture

Explaining the architecture of containers is made easier by comparing them with the architecture of virtual machines (VMs) from hardware virtualization technologies, which many readers are already familiar with. Figure 2 shows the VM architecture and two container architectures, one without VMs and one with.

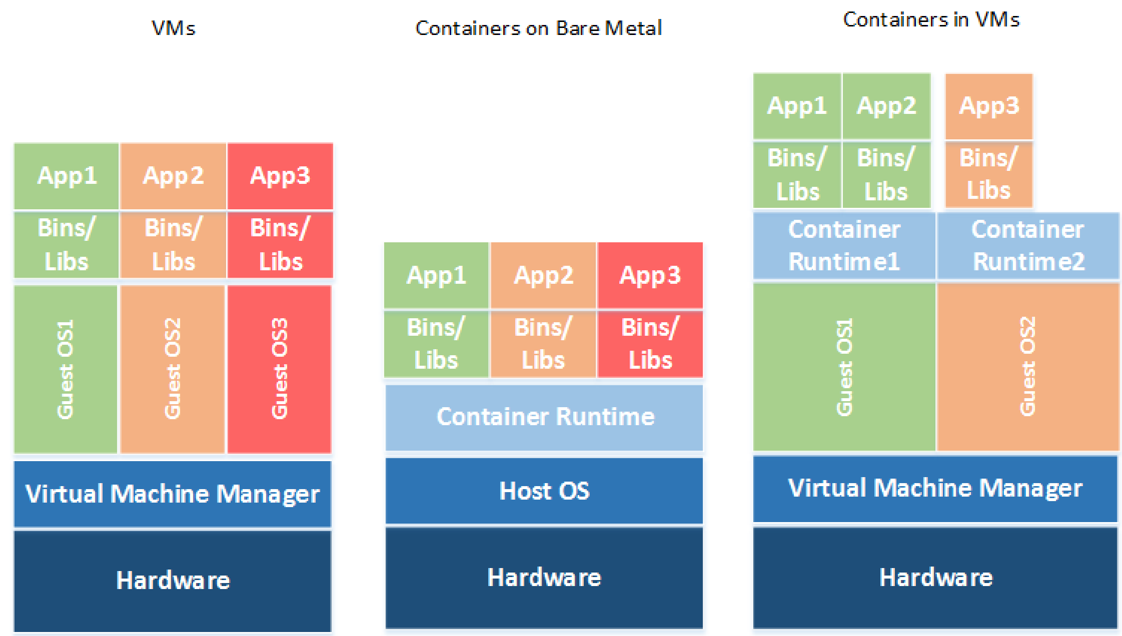

Figure 2: Virtual Machine and Container Architectures

Both VMs and containers allow multiple apps to share the same physical infrastructure, but they use different methods of separation. VMs use a hypervisor that provides hardware-level isolation of resources across VMs. Each VM sees its own virtual hardware and includes a complete guest OS in addition to the app and its data. VMs allow different OSes, such as Linux and Windows, to share the same physical hardware.

With containers, multiple apps share the same OS instance but are segregated from each other. Containers share the same OS kernel, so they cannot be run without a host OS present. In many cases, users will deploy containers inside of VMs, but this is not a requirement. Also, containers are OS-family specific; a Linux host can only run containers built for Linux, and a Windows host can only run Windows containers.

Containers can be run on an OS installed on “bare metal”, as shown in the middle of Figure 2, or an OS that runs within a VM, as shown on the right side of Figure 2. While containers are sometimes thought of as the next phase of virtualization, surpassing hardware virtualization, the reality for most organizations is less about revolution than evolution. Containers and hardware virtualization not only can, but very frequently do, coexist well and actually enhance each other’s capabilities. VMs provide many benefits, such as strong isolation, OS automation, and a wide and deep ecosystem of solutions. Organizations do not need to make a false choice between containers and VMs. Instead, organizations can continue to use VMs to deploy, partition, and manage their hardware, while using containers to package their apps and utilize each VM more efficiently.

The container technology stack, depicted in Figure 2, includes the following components:

- **Host operating system**: Containers share a common kernel that is part of the *host operating system*. It sits below the containers and provides OS capabilities to them. The host OSes used for running containers can generally be categorized into two types:
  - General-purpose OSes like Red Hat Enterprise Linux, Ubuntu, and Windows Server that can be used for running many kinds of apps and can have container-specific functionality added to them.
  - Container-specific OSes, like CoreOS [2], Project Atomic [3], and Google Container-Optimized OS [4], which are minimalistic OSes explicitly designed to only run containers. They typically do not come with package managers, and they actively discourage running applications outside containers. A container-specific OS includes the container runtime environment and a subset of core system administration tools. Often, these OSes use a read-only file system design to reduce the likelihood of an attacker being able to persist data within them, and they also utilize a simplified upgrade process since there is little concern around application compatibility.
- **Container runtime:** The layer above the host OS is the *container runtime*. It abstracts the underlying host OS from each container, such that each container sees its own dedicated view of the OS and is isolated from other containers running concurrently. The container runtime also provides management tools and application programming interfaces (APIs) to allow users to specify how to run containers on a given host. The runtime abstracts the complexity of manually creating all the necessary configurations and simplifies the process of starting, stopping, and operating containers. Examples of runtimes include Docker [5], LXC [6], rkt [7], and the Open Container Initiative Daemon [8].
- **Images:** *Images* are packages that contain all the files required to run a container. For example, an image to run Apache would include the httpd binary, along with associated libraries and configuration files. An image is executed within a container. Unlike a VM, an image does not contain an OS because that is provided by the host OS. Images are typically designed to be portable across machines and environments, so that an image created in a development lab can be easily moved to a test lab for evaluation, then copied into a production environment to run. Images often use techniques like layering and copy on write (in which shared master images are read only and changes are recorded to separate files) to minimize their size on disk and improve operational efficiency.
- **Registry**: Images are typically stored in central locations to make it easy to share, find, and reuse them across hosts. *Registries* are services that allow developers to easily store images as they are created, tag and catalog images to aid in discovery and reuse, and find and reuse images that others have created. When an image needs to be promoted from dev to test or production, the image can be pulled from this central registry. Registries are effectively special purpose file sharing apps and may be self-hosted or consumed as a service, such as with Amazon EC2 Container Registry [9] or Docker Hub [10].
- **Microservice:** Sets of containers that work together to compose an application are referred to as *microservices*. Unlike traditional architectures, which divide an application into a few tiers and have a component for each tier, in a container architecture a single app is often divided into many more components. With this modular approach, each container may have a single well-defined function. This allows more granular scaling of

the app because additional resources can be provided just to the containers with the function that needs them. It also makes iterative development easier because functionality is more self-contained.

- **Orchestrators:** Multiple container hosts can be grouped together and centrally managed by orchestration tools, also known as *orchestrators*. These are responsible for monitoring resource consumption, job execution, and machine health across multiple servers and/or VMs. This abstraction allows a developer to simply describe how many containers need to be running a given image and what resources, such as memory, processing, and disk need to be allocated to each. The orchestrator knows what is available within the cluster and dynamically assigns which containers will run on which hosts. Further, the orchestrator will monitor the health of hosts and containers and, depending on its configuration, may automatically restart containers on new hosts if the hosts they were initially running on failed. Many orchestrators can also enable cross-host container networking and service discovery. Examples of orchestrators include Kubernetes [11], Mesos [12], and Docker Swarm [13].

These components all play roles in running a containerized app. For example, in Figure 2, assume the user wants to run an app with three images. Rather than manually running containers for each image, the user tells the orchestrator the attributes of the app, including how many instances of each image is required and how many resources each container requires. The orchestrator knows the state of the machines in the cluster, including availability and resource consumption of each. The orchestrator then pulls the required images from the registry and runs them on containers across the cluster based on resource availability.

Note that all these components are not necessary to run containers. For example, a small, simple container implementation could omit a full-fledged orchestrator.

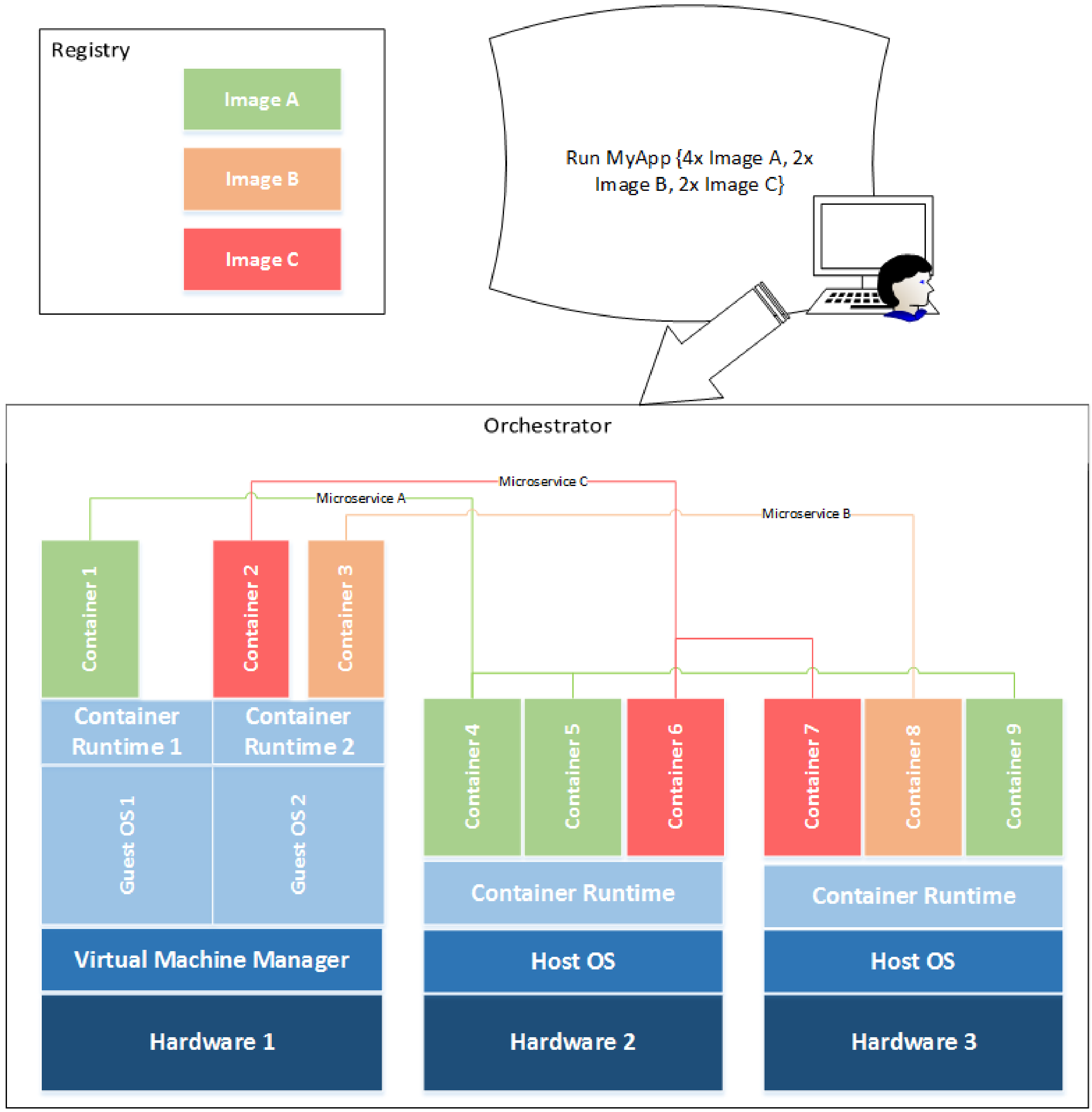

Figure 3: Interactions of Container Deployment Components

2.2 Container Technical Capabilities

The technical capabilities of containers vary by host OS. Containers are fundamentally a mechanism to give each app a unique view of a single OS, so the tools for achieving this separation are largely OS family-dependent. For example, the methods used to isolate processes from each other differ between Linux and Windows. However, while the underlying implementation may be different, container runtimes provide a common interface format that largely abstracts these differences from users.

All container platforms require the following technical capabilities provided by the host OS:

- **Namespace isolation**, which limits the resources a container may interact with. This includes file systems, network interfaces, interprocess communications, host names, user information, and processes. Namespace isolation ensures that applications and processes inside a container only see the physical and virtual resources allocated to that container. For example, if you run 'ps –A' inside a container running Apache on a server with many other containers running other apps, you would only see httpd listed in the results. Namespace isolation also allows individual containers to have their own IP addresses and interfaces. Containers on Linux use technologies like masked process identities to achieve namespace isolation, whereas on Windows, object namespaces are used.
- **Resource isolation**, which limits how much of a host's resources a given container can consume. For example, if your host OS has 10 gigabytes (GB) of total memory, you may wish to allocate 1 GB each to nine separate containers. No container should be able to interfere with the operations of another container, so resource isolation ensures that each container can only utilize the amount of resources assigned to it. On Linux, this is accomplished primarily with control groups (cgroups)[1], whereas on Windows job objects serve a similar purpose.
- **Filesystem virtualization**, which allows multiple containers to share the same physical storage without the ability to access or alter the storage of other containers. While arguably similar to namespace isolation, filesystem virtualization is called out separately because it also often involves optimizations to ensure that containers are efficiently using the host's storage through techniques like copy on write. For example, if multiple containers using the same image are running Apache on a single host, filesystem virtualization ensures that there is only one copy of the httpd binary stored on disk. If one of the containers modifies files within itself, only then will those copies be written out to storage as unique bits. On Linux, these capabilities are provided by technologies like the Advanced Multi-Layered Unification Filesystem (AUFS), whereas on Windows they are an extension of the NT File System (NTFS).

2.3 Container Attributes

Container technologies generally share several fundamental attributes:

- **Portable**. There are two main aspects to this:
  - Portability across the development lifecycle. The images used to create containers can be built directly by app developers and then moved into test and production without modification.
  - Portability across underlying platforms. The same container image should be able to run broadly across a family of host OSes and across any cloud provider that supports them.
- **Minimal**. A container only includes the specific software required to run the app within it. A container only includes the executables and libraries required by the app itself; all

[1] cgroups are collections of processes that can be managed independently, giving the kernel the software-based ability to meter subsystems such as memory, processor usage, and disk I/O. Administrators can control these subsystems either manually or programmatically.

other OS functionality is provided by the underlying host OS. Frequently, containers are single process entities and a given container only exists to run one app. Multiple containers then work together in a microservice to compose more complex apps.

- **Declarative**. Most container technologies have a declarative way of describing the components and requirements for the app. For example, an image for a web server would include not only the executables for the web server, but also some parseable data to describe how the web server should run, such as the ports it listens on or the configuration parameters it uses.
- **Immutable**. Most modern container technologies implement the concept of immutability. In other words, the containers themselves are stateless entities that are deployed but not changed. When a running container needs to be upgraded or have its contents changed, it is simply destroyed and recreated with a new image containing the updates. This provides the ability for developers and support engineers to make and push changes to applications at a much faster pace. Immutability is a fundamental operational difference between containers and hardware virtualization. Traditional VMs are typically run as stateful entities that are deployed, reconfigured, and upgraded throughout their life.

The immutable nature of containers also has implications for data persistence. Rather than intermingling the app with the data it uses, containers stress the concept of isolation. Data persistence should be achieved not through simple writes to the container file system, but instead by using external, persistent data stores such as databases or cluster-aware persistent volumes. Because containers are ephemeral, the data they use should be stored outside of the containers themselves so that when the next version of an app replaces the containers running the existing version, all data is still available to the new version.

Modern container technologies have largely emerged along with the adoption of DevOps (development and operations) practices that emphasize close coordination between development and operational teams. The portable and declarative nature of containers is particularly well suited to these practices because they allow an organization to have great consistency between development, test, and production environments. Organizations often utilize continuous integration processes to put their apps into containers directly in the build process itself, such that from the very beginning of the app's lifecycle, there is guaranteed consistency of its runtime environment.

Containers increase the effectiveness of build pipelines due to the immutable nature of container images. Containers shift the time and location of production code installation. In non-container systems, application installation happens in production (i.e., at server runtime), typically by running hand-crafted scripts that manage installation of application code (e.g., programming language runtime, dependent third-party libraries, init scripts, and OS tools) on servers. This means that any tests running in a pre-production build pipeline (and on developers' workstations) are not testing the actual production artifact, but a best-guess approximation contained in the build system. This approximation of production tends to drift from production over time, especially if the teams managing production and the build system are different. This scenario is the embodiment of the "it works on my machine" problem.

Using containers, the full application installation happens in the build system (i.e., at compile-time). The build system creates the full production artifact (i.e., the container image), which is an immutable snapshot of all userspace requirements of the application (i.e., programming language runtime, dependent third-party libraries, init scripts, and OS tools). In production the container image constructed by the build system is simply downloaded and run. This solves the "works on my machine" problem since the developer, build system, and production all run the same immutable artifact.

Modern container technologies often also emphasize reuse, such that a container image created by one developer can be easily shared and reused by other developers, either within his own organization or across the world. Registry services provide centralized image sharing and discovery services to make it easy for developers to find and reuse software created by others. This ease of use is also leading many popular software vendors and projects to use containers as a way to make it easy for customers to find and quickly run their software. For example, rather than directly installing an app like MongoDB on the host OS, a user can simply run a container image of MongoDB. Further, since the container runtime isolates containers from one another and the host OS, these apps can be run more safely and reliably, and users do not have to worry about them disturbing the underlying host OS.

2.4 Container Uses

Like any other technology, containers are not a panacea. They are a valuable tool for many scenarios, but are not necessarily the best choice for every scenario. For example, an organization with a large base of legacy off the shelf software is unlikely to be able to take advantage of containers for running most of that software since the vendors may not support it. However, most organizations will have multiple valuable uses for containers. Examples include:

- Agile development, where apps are frequently updated and deployed. The portability and declarative nature of containers makes these frequent updates more efficient and easier to test. This allows organizations to accelerate their innovation and deliver software more quickly. This also allows vulnerabilities in application code to be fixed and the updated software tested and deployed much faster.
- 'Scale out' scenarios, where an app may need to have many new instances deployed or decommissioned quickly depending on the load at a given point in time. The immutability of containers makes it easier to reliably scale out instances, knowing that each instance is exactly like all the others. Further, because containers are typically stateless, it is easier to decommission them when they are no longer needed.
- Net new apps, where developers can build for a microservices architecture from the beginning, ensuring more efficient iteration of the app and simplified deployment.

2.5 The Container Lifecycle

Containers do not exist in a vacuum; they are typically used as part of the overall lifecycle of an app and thus interact with other systems and user personas. Figure 4 shows the basic lifecycle phases. Because organizations are typically building and deploying many different apps at once, these lifecycle phases often occur concurrently within the same organization and should not be seen as progressive stages of maturity. Instead, think of them as cycles in an engine that is continuously running. In this metaphor, each app is a cylinder within the engine, and different apps may be at different phases of this lifecycle at the same time.

This section refers to tasks performed by development and operation personas during the lifecycle. Many organizations have merged their development and operations teams into combined DevOps teams that seek to increase the integration between building and running apps. Thus, the references in this section to these personas are focused on the types of job tasks being performed, not on strict titles or team organizational structures.

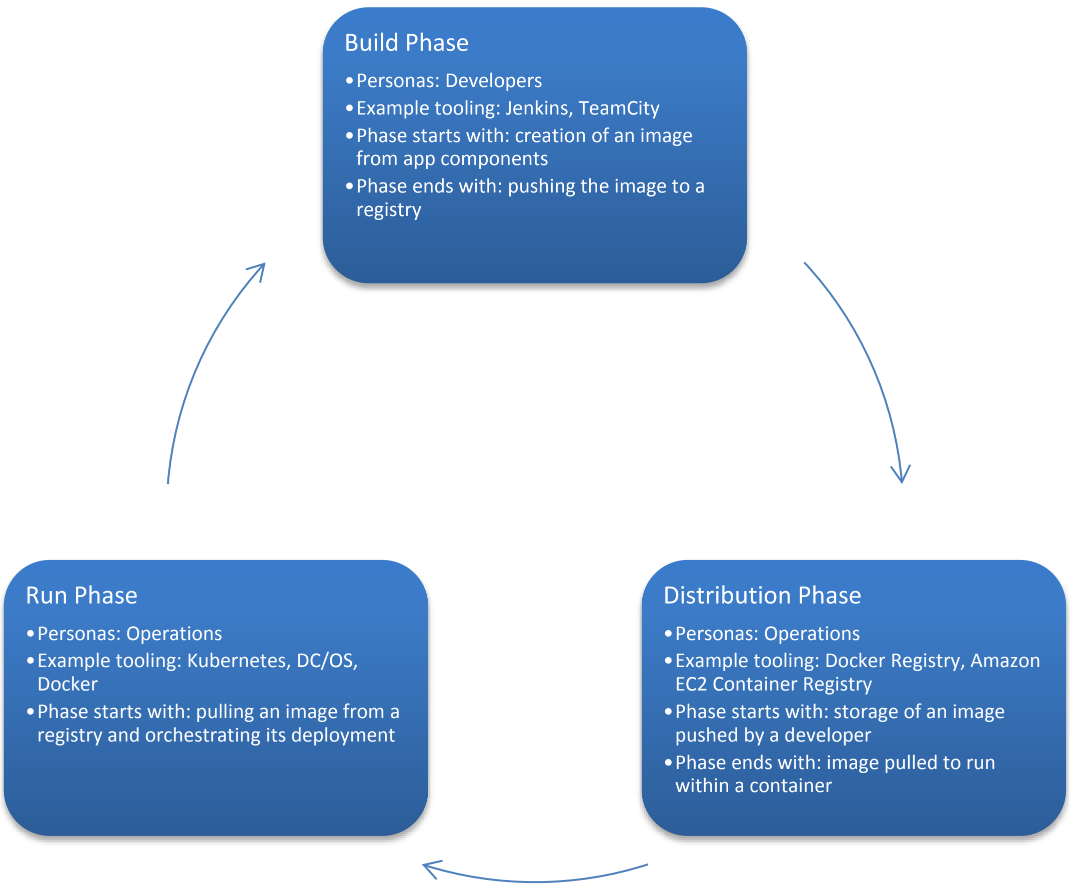

Figure 4: Container Lifecycle Phases

2.5.1 Build phase

The build phase is the portion of the lifecycle in which app components are compiled, collected, and placed into images. The build phase is mostly driven by developers who are working on creating or updating apps and packaging them in containers. The build phase typically uses build management and automation tools, such as Jenkins [14] and TeamCity [15], to assist with this "continuous integration" process. These tools take the various libraries, binaries, and other components of an application, perform testing on them, and then assemble images out of them. The build phase would normally begin with a developer creating a manifest for the app that describes how to build an image for it, and end with the build automation tool creating a ready-to-run image of the app.

2.5.2 Distribution phase

Once images are created by developers, they need to be stored in a predictable location they can be deployed from. These registries are essentially just file storage for images, wrapped in APIs that enable development and operations teams to automate common tasks like uploading new images, tagging images for identification, and downloading images for deployment. Registries, such as Docker Trusted Registry [16], Quay Container Registry [17], and Amazon EC2 Container Registry [9], are typically where developers output their images to at the end of the build phase. Once stored in the registry, they can be easily pulled and then run by operations personas across any environment in which they run containers. This is another example of the portability benefits of containers; the build phase may occur in a public cloud provider, which pushes an image to a registry hosted in a private cloud, which is then used to distribute images for running the app in a third location.

The distribution phase typically uses extensive automation to reduce the manual activities associated with uploading and deploying images. For example, organizations may have triggers in the build phase that automatically push images to a registry once tests pass. The registry may have further triggers that automate the deployment of new images once they have been added. This automation enables faster iteration on projects with more consistent results.

2.5.3 Run phase

Once an image is stored in a registry, it is ready to be pulled and run within a container. Operations personas, or the automation they create, typically perform the tasks associated with deploying an image from a registry into a set of containers. This deployment process is what actually results in a usable version of the app, running and ready to respond to requests. When an image is deployed into a container, the image itself is not changed, but instead a copy of it is placed within the container and transitioned from being a dormant set of app code to a running instance of the app. Images are typically deployed from registries via orchestration tools, such as Kubernetes [11] or DC/OS [18], that are configured to pull the most up-to-date version of an image from the registry so that the app is always up-to-date. This “continuous delivery” automation enables developers to simply build a new version of the image for their app, push it to the registry, and then rely on the run phase automation tooling to deploy it to the target environment.

3 Major Risks in the Container Technology Stack

This section identifies and analyzes the major risks in the container technology stack. It uses the data-centric system threat modeling approach described in NIST SP 800-154 [19] to examine a typical container stack as depicted in Figure 5. Because this analysis looks at the stack only, and not the technologies below the stack, it is applicable to most container deployments, whether using VMs or running on bare metal, at a public cloud provider or within an organization's onsite datacenter.

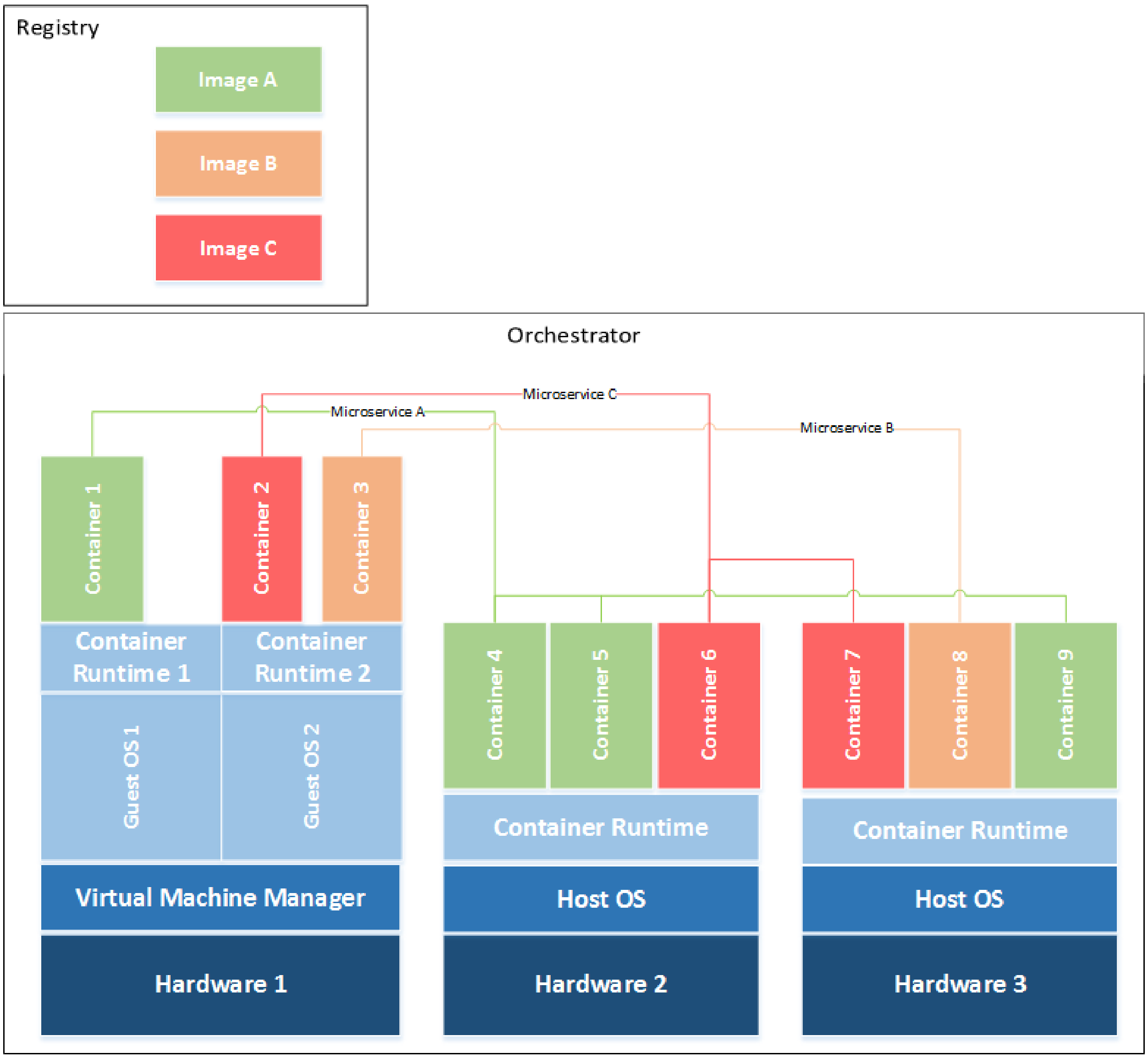

Figure 5: Container Technology Stack

This section begins by discussing the most important operational differences between VMs and containers, which all have security implications. The rest of the section walks through the container technology stack from lowest layer to highest layer, identifying and analyzing major risks relevant to each layer. Appendix A contains pointers to references for securing systems and system components outside the container technology stack.

3.1 Operational Differences Between Containers and VMs

While there are many technical differences between containers and VMs, there are also significant operational differences. These operational differences impact many aspects of container security.

- **Many more entities**. When an app is deployed via containers and microservices, there are many more discrete components for the app than if that app were run in a more monolithic, VM-centric model. For example, a simple two-tiered web app running in VMs may only have a cluster of web server VMs on the front end and a cluster of database VMs on the backend. This same app, decomposed into microservices, may have many different front end containers, each running a different part of the web portion of the app, as well as multiple database and cache instances on the backend. These microservices make iteration and scaling easier, but result in more objects to understand, manage, and secure. Security tools and operations must be adapted to deal with this larger number of objects.
- **Much greater rate of change.** One of the primary drivers for customers to adopt containers is the agility it gives them from a development standpoint, making it easier and faster to respond to business needs through rapid iteration of apps. Organizations may go from deploying a new version of their app every quarter, to deploying new components weekly or daily. Legacy security tools and processes often assume far less dynamic operations and may need to be adjusted to adapt to the rate of change in containerized environments.
- **Security is largely the responsibility of the developer.** Good security practices in development have always been a core part of an effective security strategy. However, in the past, organizations often had a clear differentiation between development and operations, and the operations team often had the responsibility of monitoring and maintaining the apps after deployment. Because containers are built directly from images created by developers, the responsibility for securing those images is much further ‘upstream’ with containers. For example, instead of the operations team patching a web server with a vulnerability, the developer is now responsible for performing the patching within the images and providing the new versions of the images to be run. This change in responsibilities often requires much greater coordination and cooperation between development and operations teams.
- **Security must be as portable as the containers.** One of the key factors driving adoption of containers is their portability. Developers find great value in being able to move containers and images across many different environments, such as their developer workstation, a public cloud test environment, and a private cloud production environment. Unlike VMs, in which environments were more static and predictable, developers may move containers around many different locations during the course of normal operations. Thus, the security tools and processes used to protect them must not

make assumptions about specific cloud providers, host OSes, network topologies, or other aspects of the runtime environment which may frequently change.

- **Networking is much more ephemeral.** VMs and bare metal servers are typically allocated static IP addresses by an administrator, and those addresses remain relatively consistent over time. For example, a given VM may be assigned an IP address when it is originally created and use that same IP address for the months or years it continues to run. Conversely, containers are typically allocated IP addresses via whatever orchestration tool is being used. The IP addresses assigned to a given container are not typically known in advance, and no administrator is normally involved in assigning them. Because containers are created and destroyed much more frequently than VMs, these IP addresses change frequently over time as well, without human involvement. This makes it difficult or impossible to protect containers using security techniques that rely on static IP addresses, such as firewall rulesets filtering traffic based on IP address.

3.2 Host OS Risks

3.2.1 Improper user access rights

Container-specific OSes are typically used in conjunction with orchestrators that provide for container placement and scaling. In these deployments, the OS is typically not optimized to support multiuser scenarios since interactive user logon should be rare. If organizations rely on manual configuration and management, users may have greater access to the containerized apps they host than necessary.

3.2.2 Host component vulnerabilities

Container-specific OSes have a much smaller attack surface than that of general-purpose OSes. For example, they do not contain libraries and package managers that enable a general-purpose OS to directly run database and web server apps. However, even on container-specific OSes, there are foundational system components provided by the host OS—for example, the cryptographic libraries used to authenticate remote connections and the kernel primitives used for general process invocation and management. Like any other software, these components can have vulnerabilities and, because they exist low in the stack, these vulnerabilities can impact all the containers and applications that run on these hosts.

3.3 Container Runtime Risks

3.3.1 Vulnerabilities within the runtime software

While relatively rare, these vulnerabilities can be particularly dangerous if they allow 'container escape' scenarios in which malicious software is able to use those vulnerabilities to attack resources outside of the container in which it originated, including other containers and the host OS itself. An attacker may also be able to exploit vulnerabilities to compromise the runtime software itself, and then alter that software so it allows the attacker to access containers, monitor container-to-container communications, etc.

3.3.2 Unbounded network access from containers

By default in most container runtimes, individual containers are able to access each other and the host over the network. If a container is compromised and acting maliciously, allowing this network traffic may expose other resources in the environment to risk. For example, a compromised container may be used to scan the network it is connected to in order to find other weaknesses for an attacker to exploit.

Egress network access is more complex to manage in a containerized environment because so much of the connection is virtualized between containers. Thus, traffic from one container to another may appear simply as encapsulated packets on the wire without an understanding of the ultimate source, destination, or payload. Tools and operational processes that are not container aware are not able to inspect this traffic or determine whether it represents a threat.

3.3.3 Insecure container runtime configurations

Container runtimes are complex software and typically expose many configurable options to administrators. Often, configuring them improperly can lower the relative security of the system. For example, on Linux container hosts, the set of allowed system calls is often limited by default to only those required for safe operation of containers. If this list is widened, it may expose the runtime and host to increased risk from a compromised container.

Another example of an insecure runtime configuration is allowing containers to mount sensitive directories on the host. Containers should rarely make changes to the host file system and should almost never make changes to locations like /boot or /etc that control the basic functionality of the host OS. If a container is allowed to make changes to these paths, a compromised container could potentially be used to elevate privileges and attack the host itself as well as other containers running on the host.

3.3.4 Shared kernel

While containers provide strong software-level isolation of resources, the use of a shared kernel invariably results in a larger inter-object attack surface than seen with hypervisors. In other words, the level of isolation provided by container runtimes is not as high as that provided by hypervisors.

3.4 Image Risks

3.4.1 Image vulnerabilities

Because images are effectively static archive files that include all the components used to run a given application, the components within this image may often be out of date and missing critical security updates. For example, if an image is created with fully up-to-date components, that image may continue to be free from vulnerabilities for days or weeks after its creation. However, at some point in the future the components included in that image will likely have vulnerabilities discovered in them, and thus the image overall will no longer be up-to-date.

Unlike traditional operational patterns in which deployed software is updated 'in the field' on the systems it runs on, with containers these updates must be made upstream in the images themselves, which are then redeployed. Thus, a common risk in containerized environments is deployed images having vulnerabilities because the version of the image being run does not include all the necessary updates.

3.4.2 Image configuration

In addition to software defects, images may also have configuration defects as well. For example, an image could be configured to run as root or include executables set to run with excessive privileges. Much like in a traditional server or VM, where a poor configuration can still expose a fully up-to-date system to attack, so too can a poorly configured image increase risk even if all the included components are up-to-date.

3.4.3 Embedded malware

Because images are just collections of files packaged together, malicious files could be included intentionally or inadvertently within them. Organizations often build images from base layers provided by third parties of which the full provenance is not known. Especially in these cases, an organization can be exposed to risk by malware being embedded within the image. This malware would have the same set of capabilities as any other component within the image and thus could be used to attack other containers or hosts within the environment.

3.4.4 Embedded secrets

Many applications require secrets to enable secure communication between various components. For example, a web application may need a username and password to connect to a backend database. When an app is packaged in a container, these secrets can be embedded directly into the image. However, this practice creates a security risk because anyone with access to the image file can easily parse it to learn these secrets. Potential sensitive data includes connection strings, SSH private keys, and x.509 private keys.

3.4.5 Image trust

One of the most common high-risk scenarios in any environment is the execution of untrusted software. The portability and ease of reuse of containers increase the temptation for teams to run images from external sources that may not be well validated or trustworthy. For example, when troubleshooting a problem with a web application, a user may find another version of that application available in an image provided by a third party. Using this externally provided image results in the same types of risks that external software traditionally has, such as introducing malware, leaking data, or including components with vulnerabilities.

3.5 Registry Risks

3.5.1 Insecure connections to registries

Images often contain sensitive components like an organization's line of business application. While, ideally, images should not include secrets or user data, the software itself is often proprietary to an organization and should be protected in transit. If connections to registries are performed over insecure channels, the contents of images are subject to the same confidentiality risks as any other data transmitted in the clear.

3.5.2 Stale images in registries

Because registries are typically the source location for all the images an organization deploys, over time the set of images they store can include many vulnerable, out-of-date versions. While these vulnerable images do not directly pose a threat simply by being stored in the registry, they increase the likelihood of user error resulting in the deployment of a known-bad version.

3.6 Orchestrator Risks

3.6.1 Unbounded administrative access

Historically, many orchestration tools assumed that all users that interacted with them were administrators and that those administrators should have environment-wide control. However, in many cases, a single orchestrator may run many different apps, each managed by different teams, and with different sensitivity levels. If the access provided to users and groups is not scoped to their specific needs, a malicious or careless user could affect or subvert the operation of other containers managed by the orchestrator.

3.6.2 Weak or unmanaged credentials

Orchestration tools often include their own authentication directory, which may be separate from the typical directories already in use within an organization. This can lead to weaker account management practices and 'orphaned' accounts in the orchestrator because these systems are less rigorously managed. Because many of these accounts are highly privileged within the orchestrator, compromise of them can lead to systemwide compromise.

3.6.3 Unmanaged inter-container network traffic

In most containerized environments, traffic between individual nodes is routed over a virtual overlay network. This overlay network is typically managed by the orchestration tool and is often opaque to existing network security and management tools. For example, instead of seeing database queries being sent from a web server container to a database container on another host, traditional network filters would only see encrypted packets flowing between two hosts, with no visibility into the actual container endpoints, nor the traffic being sent. This can create a security 'blindness' scenario in which organizations are unable to effectively monitor traffic within their own networks.

3.6.4 Mixing of workload sensitivity levels

Orchestrators are typically focused primarily on driving the scale and density of workloads. This means that, by default, they can place workloads of differing sensitivity levels on the same host. For example, in a default configuration, an orchestrator may place a container running a public-facing web server on the same host as one processing sensitive financial data, simply because that host happens to have the most available resources at the time of deployment. This can put the container processing sensitive financial data at significantly greater risk of compromise.

4 Countermeasures for Mitigating the Major Risks

This section discusses possible countermeasures for the major risks identified in Section 3 and makes recommendations for selecting and using countermeasures.

4.1 Hardware Countermeasures

Software-based security is regularly defeated, as acknowledged in NIST SP 800-164 [20]. NIST defines trusted computing requirements in NIST SPs 800-147 [21], 800-155 [22], and 800-164. To NIST, “trusted” means that the platform behaves as it is expected to: the software inventory is accurate, the configuration settings and security controls are in place and operating as they should, and so on. “Trusted” also means that it is known that no unauthorized person has tampered with the software or its configuration on the hosts.

The currently available way to provide trusted computing is to:

1. Measure firmware, software, and configuration data before it is executed using a Root of Trust for Measurement (RTM).
2. Store those measurements in a hardware root of trust, like a trusted platform module (TPM).
3. Validate that the current measurements match the expected measurements. If so, it can be attested that the platform can be trusted to behave as expected.

TPM-enabled devices can check the integrity of the machine during the boot process, enabling protection and detection mechanisms to function in hardware, at pre-boot, and in the secure boot process. This same trust and integrity assurance can be extended beyond the OS and the boot loader to the container runtimes and applications.

The increasing complexity of systems and the deeply embedded nature of today’s threats means that security should extend across all the layers of the container stack, starting with the hardware and firmware. This would form a distributed trusted computing model and provide the most trusted and secure way to build, run, orchestrate, and manage containers.

The trusted computing model should start with measured/secure boot, which provides a verified system platform, and build a chain of trust rooted in hardware and extended to the bootloaders, the OS kernel, and the OS components to enable cryptographic verification of boot mechanisms, system images, container runtimes, and container images. In the container stack, these techniques are currently applicable at the hardware, hypervisor, and host OS layers, with early work in progress to apply these to container-specific components.

4.2 Host OS Countermeasures

For customers using container-specific OSes, the threats are typically more minimal to start with since the OSes are specifically designed to host containers and have other services and functionality disabled. Further, because these optimized OSes are designed specifically for hosting containers, they typically feature read-only file systems and employ other hardening practices by default. Whenever possible, organizations should use these minimalistic OSes to reduce their attack surfaces and mitigate the typical risks and hardening activities associated with general-purpose OSes. This section is thus focused primarily on risks relevant to these container-optimized OSes.

4.2.1 Vulnerabilities in core system components

Organizations should implement management practices and tools to validate the versioning of components provided for base OS management and functionality. Even though container-specific OSes have a much more minimal set of components than general-purpose OSes, they still do have vulnerabilities and still require remediation. Organizations should use tools provided by the OS vendor or other trusted organizations to regularly check for and apply updates to all software components used within the OS.

Not as obvious, but equally critical to this approach, is ensuring that apps are built, tested, and operated with clear segmentation between the app and the host OS. Containerized apps should not rely on host-specific configurations or data storage because those dependencies often make it more difficult to utilize minimal host OSes. Furthermore, from an operational standpoint, apps should be built and operated to achieve resiliency through horizontal scaling across multiple nodes. This is important for host OS remediation because it enables simple updates to all the hosts in a deployment, removing one of the most common barriers to timely remediation of security vulnerabilities.

4.2.2 Improper user access rights

Though most container deployments rely on orchestrators to distribute jobs across hosts, organizations should still ensure that all authentication to the OS is audited, anomalies are monitored, and any escalation to performed privileged operations is logged. This makes it possible to identify anomalous access patterns such as an individual logging on to a host directly and running privileged commands.

Additionally, organizations should ensure that the orchestrator provides only the specific set of access required to the specific resources required for an administrator to perform their job. For example, a developer working on project foo should only able to manage resources associated with project foo and not be able to access resources for project bar. In cases where the orchestrator does not provide this capability natively, third-party solutions should be implemented to do so.

4.3 Container Runtime Countermeasures

4.3.1 Vulnerabilities within the runtime software

The container runtime must be carefully monitored for vulnerabilities and when problems are detected, they must be remediated quickly. A vulnerable runtime exposes all containers it supports, as well as the host itself, to potentially significant risk. Organizations should use tools to look for Common Vulnerabilities and Exposures (CVEs) vulnerabilities in the runtimes deployed, to upgrade any instances at risk, and to ensure that orchestrators only allow deployments to properly maintained runtimes.

4.3.2 Unbounded network access from containers

Organizations should control the egress network traffic sent by containers. At minimum, these controls should be in place at network borders, ensuring containers are not able to send traffic across networks of differing sensitivity levels, such as from an environment hosting secure data to the internet, similar to the patterns used for traditional architectures. However, the virtualized networking model of inter-container traffic poses an additional challenge.

Because containers deployed across multiple hosts typically communicate over a virtual, encrypted network, traditional network devices are often blind to this traffic. Additionally, containers are typically assigned dynamic IP addresses automatically when deployed by orchestrators, and these addresses change continuously as the app is scaled and load balanced. Thus, ideally, organizations use a combination of existing network level devices and more application-aware network filtering. App-aware tools should be able to not just see the inter-container traffic, but also to dynamically generate the rules used to filter this traffic based on the specific characteristics of the apps running in the containers. This dynamic rule management is critical due to the scale and rate of change of containerized apps, as well as their ephemeral networking topology.

Specifically, app-aware tools should provide the following capabilities:

- Automated determination of proper container networking surfaces, including both inbound ports and process-port bindings;
- Detection of traffic flows both between containers and other network entities, over both 'on the wire' traffic and encapsulated traffic; and
- Detection of network anomalies, such as unexpected east-west traffic flows, port scanning, or outbound access to potentially dangerous destinations.

4.3.3 Insecure container runtime configurations

Organizations should automate compliance with container runtime configuration standards. Documented technical implementation guidance, such as the Center for Internet Security Docker Benchmark, provides details on options and recommended settings, but operationalizing this guidance depends on automation. Organizations can use a variety of tools to 'scan' and assess their compliance at a point in time, but such approaches do not scale. Instead, organizations should use tools or processes that continuously assess configuration settings across the environment and actively enforce them.

Additionally, mandatory access control technologies like SELinux [23] and AppArmor [24] provide enhanced control and isolation for containers. For example, these technologies can be used to provide additional segmentation and assurance that containers should only be able to access specific file paths, processes, and network sockets, further constraining the ability of even a compromised container to impact the host or other containers.

4.3.4 Shared kernel

While most container runtime environments do an effective job of isolating containers from each other and from the host OS, in some cases it may be an unnecessary risk to run apps of different classification levels together on the same runtime. Segmenting containers by purpose and sensitivity provides additional defense in depth. For example, consider a scenario in which a host is running containers for both a financial database and a public-facing blog. While normally the container runtime will securely isolate these environments from each other, there is also a shared responsibility amongst the DevOps teams for each app to operate them properly. If the DevOps team for the blog were to run their app in a privileged mode and it was compromised, the attacker may be able to escalate privileges to attack the database.

Thus, a best practice is to group containers together by relative sensitivity and to ensure that a given host kernel only runs containers of a single sensitivity level. This segmentation may be provided by using multiple physical servers, but modern hypervisors also provide strong enough isolation to effectively mitigate these risks. From the previous example, this may mean that the organization has two sensitivity levels for their containers. One is for financial apps and the database is included in that group. The other is for web apps and the blog is included in that group. The organization would then have two pools of VMs that would each host containers of a single severity level. For example, the host called vm-financial may host the containers running the financial database as well as the tax reporting software, while a host called vm-web may host the blog and the public website.

By segmenting containers in this manner, it will be much more difficult for an attacker who compromises one of the segments to expand that compromise to other segments. This approach also ensures that any residual data, such as caches or local volumes mounted for temp files, stays within its security zone. From the previous example, this zoning would ensure that any financial data cached locally and residually after container termination would never be available on a host running an app at a lower sensitivity level.

In larger-scale environments with hundreds of hosts and thousands of containers, this segmentation must be automated to be practical to operationalize. Fortunately, common orchestration platforms typically include some notion of being able to group apps together, and container security tools can use attributes like container names and labels to enforce security policies across them. In these environments, additional layers of defense in depth beyond simple host isolation may also leverage this segmentation. For example, an organization may implement separate hosting 'zones' or networks to not only isolate these containers within hypervisors but also to isolate their network traffic more discretely.

4.3.5 Compromised containers

Existing host-based intrusion detection processes and tools are often unable to detect and prevent attacks within containers due to the differing technical architecture and operational practices previously discussed. Organizations should implement additional tools that are container aware and designed to operate at the scale and change rate typically seen with containers. These tools should be able to automatically profile containerized apps and build protection profiles for them to minimize human interaction. These profiles should then be able to detect anomalies at runtime, including events such as:

- Invalid or unexpected process execution,
- Invalid or unexpected system calls,

- Changes to protected configuration files and binaries,
- Writes to unexpected locations and file types,
- Creation of unexpected network listeners,
- Traffic sent to unexpected network destinations, and
- Malware storage or execution.

4.4 Image Countermeasures

4.4.1 Image vulnerabilities

There is a need for container-specific vulnerability management tools and processes. Traditional vulnerability management tools make many assumptions about host durability, app update mechanisms, and update frequencies that are fundamentally misaligned with a containerized model. These tools are often unable to detect vulnerabilities within containerized stacks, leading to a false sense of safety. Organizations should use tools that take the pipeline-based build approach and immutable nature of containers and images into their design to provide more actionable and reliable results. Key aspects of effective tools and processes include:

1. Integration with the entire lifecycle of images and containers, from the beginning of the build process, to whatever registries the organization is using, to runtime.
2. Visibility into vulnerabilities at all layers of the image, not just the base layer of the image but also application frameworks and custom software the organization is using.
3. Policy driven enforcement; organizations should be able to create 'quality gates' at each stage of the build and deployment process to ensure that only images that meet the vulnerable policy are allowed to progress. For example, organizations should be able to configure a rule in the build process to prevent the progression of images that include vulnerabilities with Common Vulnerability Scoring System (CVSS) ratings above a selected threshold.

4.4.2 Image configuration

In addition to software vulnerabilities, images may be configured in ways that increase security risks and violate organizational policies. For example, images should be configured to run as non-privileged users and should not allow remote access to themselves. Organizations should adopt tools and processes to validate and enforce compliance with these secure configuration best practices. Such tools and processes should include:

1. Validation of image configuration settings including both vendor recommendations and custom / 3rd party best practices.
2. Centralized reporting and monitoring of image compliance state to identify weaknesses and risks at the organizational level.
3. Enforcement of compliance requirements by preventing the running of non-compliant images.

4.4.3 Malware

Organizations should use tools and practices to monitor images for malware both at rest and when running in containers. These processes should include:

1. Identification of malware within images both in registries and on hosts,
2. The usage of comprehensive malware signature sets and detection heuristics based on actual 'in the wild' attacks,
3. The detection of malware introduced to a container at runtime; for example, if a container is subverted and the attacker downloads a rootkit into it.

4.4.4 Embedded secrets

Sensitive data should never be stored within image files. Instead, these secrets should be stored outside of the images and provided dynamically at runtime as needed. Most orchestration platforms, such as Docker Swarm and Kubernetes, include secret management natively. These platforms not only provide secure secret storage and 'just in time' injection to containers, but also make it much simpler to integrate secret management into the build and deployment processes. For example, an organization could use these tools to securely provision the database connection string into a web app container. The platform would ensure that only the web app container had access to this secret, that it is not persisted to disk, and that anytime the web app is deployed, the secret is provisioned into it.

Organizations may also integrate their container deployments with existing enterprise secret management systems that are already in use for storing secrets in non-container environments. These tools typically provide APIs to retrieve secrets securely as containers are deployed, which eliminates the need to persist them within images.

4.4.5 Image trust

Organizations should enforce a set of trusted images and registries and ensure that only images from this set are allowed to run in their environment, thus mitigating the risk of untrusted or malicious components being deployed.

To mitigate these risks, organizations should take a multilayered approach to ensure that only trusted, valid images are run within their environment. Such an approach should include:

- Capability to centrally control exactly what images and registries are trusted in their environment;
- Discrete identification of each image by cryptographic signature, using a NIST-validated implementation[2];
- Quality gates to ensure that only images that have been validated from a compliance and vulnerability state are allowed to be pushed to these locations;
- Enforcement to ensure that all hosts in the environment only run images from these approved lists; and
- Ongoing monitoring and maintenance of these repositories to ensure images within them are maintained and updated as vulnerabilities and configuration requirements change.

---

[2] For more information on NIST-validated cryptographic implementations, see the Cryptographic Module Validation Program (CMVP) page at http://csrc.nist.gov/groups/STM/cmvp/.

4.5 Registry Countermeasures

4.5.1 Insecure connections to registries

Organizations should configure their container runtimes to only connect to registries over encrypted channels. The specific steps vary between runtime and orchestrator, but the key goal is to ensure that all data pulled from a registry is encrypted in transit between the registry and the destination.

4.5.2 Stale images in registries

The risk of using stale images can be mitigated through two primary methods. First, organizations can prune registries of unsafe, vulnerable images that should no longer be used. This process can be automated based on time triggers and labels associated with images. Second, operational practices should emphasize accessing images using immutable names that specify discrete versions of images to be used. For example, rather than configuring a deployment job to use the image called my-app, configure it to deploy specific versions of the image, such as my-app:2.3 and my-app:2.4 to ensure that specific, known good instances of images are deployed as part of each job.

4.6 Orchestrator Countermeasures

4.6.1 Unbounded administrative access

Especially because of their wide-ranging span of control, orchestrators should use a least privileged access model in which users are only granted ability to perform the specific actions on the specific hosts, containers, and images their job role requires. For examples, members of the test team should only be given access to the images used in testing and the hosts used for running them, and should only be able to manipulate the containers they created. Test team members should have limited or no access to containers used in production.

4.6.2 Weak or unmanaged credentials

Access to cluster-wide administrative accounts should be tightly controlled as these accounts provide ability to affect all resources in the environment. Organizations should also implement single sign on to existing directory systems where applicable. Single sign on simplifies the orchestrator authentication experience, makes it easier for users to use strong authentication credentials, and centralizes auditing of access, making anomaly detection more effective.

4.6.3 Mixing of workload sensitivity levels

Orchestrators should be configured to isolate deployments to specific sets of hosts by sensitivity levels. The particular approach for implementing this varies depending on the orchestrator in use, but the general model is to define rules that prevent high sensitivity workloads from being placed on the same host as those running lower sensitivity workloads. This can be accomplished through the use of host 'pinning' within the orchestrator or even simply by having separate, individually managed clusters for each classification level.

5 Container Threat Scenario Examples

To illustrate the effectiveness of the recommended mitigations from Section 4, consider the following threat scenario examples for containers.

5.1 Exploit of a Vulnerability within an Image

One of the most common threats to a containerized environment is application-level vulnerabilities in the software within containers. For example, an organization may build an image based on a common web application. If that application has a vulnerability, it may be used to subvert the application within the container. Once compromised, the attacker may be able to map other systems in the environment, attempt to elevate privileges within the compromised container, or abuse the container for use in attacks on other systems (such as acting as a file dropper or command and control endpoint).

Organizations that adopt the recommendations would have multiple layers of defense in depth against such threats:

1. Detecting the vulnerable image early in the deployment process and having controls in place to prevent vulnerable images from being deployed would prevent the vulnerability from being introduced into production.
2. Container-aware network monitoring and filtering would detect anomalous connections to other containers during the attempt to map other systems.
3. Container-aware process monitoring and malware detection would detect the running of invalid or unexpected malicious processes and the data they introduce into the environment.

5.2 Exploit of the Container Runtime

While a rare occurrence, if a container runtime were compromised, an attacker could utilize this access to attack all the containers on the host and even the host itself.

Relevant mitigations for this threat scenario include:

1. The usage of mandatory access control capabilities can provide additional barriers to ensure that process and file system activity is still segmented within the defined boundaries.
2. Segmentation of workloads ensures that the scope of the compromise would be limited to applications of a common classification level that are sharing the host. For example, a compromised runtime on a host only running web applications would not impact runtimes on other hosts running containers for financial applications.
3. Security tools that can report on the vulnerability state of runtimes and prevent the deployment of images to vulnerable ones can prevent workloads from running there.

5.3 Running a Poisoned Image

Because images are easily sourced from public locations, often with unknown provenance, an attacker may embed malicious software within images known to be used by a target. For example, if an attacker determines that a target is active on a discussion board about a particular project and uses images provided by that project's web site, the attacker may seek to craft malicious versions of these images for use in an attack.

Relevant mitigations include:

1. Ensuring that only trusted images are allowed to run will prevent images from external, unvetted sources from being used.
2. Automated scanning of images for vulnerabilities and malware may detect malicious code such as rootkits embedded within an image.

6 Secure Container Technology Stack Planning and Implementation

It is critically important to carefully plan before installing, configuring, and deploying container technology stacks. This helps ensure that the container environment is as secure as possible and is in compliance with all relevant organizational policies, external regulations, and other requirements.

There is a great deal of similarity in the planning and implementation recommendations for container technology stacks and virtualization solutions. Section 5 of NIST SP 800-125 [1] already contains a full set of recommendations for virtualization solutions. Instead of repeating all those recommendations here, this section points readers to that document and states that, besides the exceptions listed below, organizations should apply all the NIST SP 800-125 Section 5 recommendations in a container technology stack context. For example, instead of creating a virtualization security policy, create a container technology stack security policy.

This section of the document lists exceptions and additions to the NIST SP 800-125 Section 5 recommendations, grouped by the corresponding phase in the planning and implementation life cycle.

6.1 Initiation Phase

Organizations should consider how other security policies may be affected by containers and adjust these policies as needed to take containers into consideration. For example, policies for incident response (especially forensics) and vulnerability management may need to be adjusted to take into account the special requirements of containers.

The introduction of containerization technologies might disrupt the existing culture and software development methodologies within the organization. To take full advantage of the benefits containers can provide, the organization's processes should be tailored to support this new way of developing, running, and supporting applications. Traditional development practices, patching techniques, and system upgrade processes might not directly apply to a containerized environment, and it is important that the employees within the organization are willing to adapt to a new model. New processes can consider and address any potential culture shock that is introduced by the technology shift. Education and training can be offered to anyone involved in the software development lifecycle to allow people to become comfortable and excited for the new way to build, ship, and run applications.

6.2 Planning and Design Phase

The primary container-specific consideration for the planning and design phase is forensics. Because containers mostly build on components already present in OSes, the tools and techniques for performing forensics in a containerized environment are mostly an evolution of existing practices. The immutable nature of containers and images can actually improve forensic capabilities because the demarcation between what an image should do and what actually occurred during an incident is clearer. For example, if a container launched to run a web server suddenly starts a mail relay, it is very clear that the new process was not part of the original image used to create the container. On traditional platforms, with less separation between the OS and apps, making this differentiation can be much more difficult.

Organizations that are familiar with process, memory, and disk incident response activities will find them largely similar when working with containers. However, there are some differences to keep in mind as well.

Containers typically use a layered file system that is virtualized from the host OS. Directly examining paths on the hosts typically only reveals the outer boundary of these layers, not the files and data within them. Thus, when responding to incidents in containerized environments, users should identify the specific storage provider in use and understand how to properly examine its contents offline.

Containers are typically connected to each other using virtualized overlay networks. These overlay networks frequently use encapsulation and encryption to allow the traffic to be routed over existing networks securely. However, this means that when investigating incidents on container networks, particularly when doing any live packet analysis, the tools used must be aware of these virtualized networks and understand how to extract the embedded IP frames from within them for parsing with existing tools.

Process and memory activity within containers is largely similar to that which would be observed within traditional apps, but with different parent processes. For example, container runtimes may spawn all processes within containers in a nested fashion in which the runtime is the top-level process with first-level descendants per container and second-level descendants for each process within the container. For example:

```
├─containerd─┬───┬───[container1─┬─bash]
│            │   │               └─8*[{thread}]]
│            │   ├─container2────┬─start.sh─┬─mongod───22*[{mongod}]
│            │   │               │          └─node─┬─4*[{V8 WorkerThread}]
│            │   │               │                 └─5*[{node}]
│            │   │               └─8*[{thread}]
│            │   ├─container3────┬─mysqld───28*[{mysqld}]
│            │   │               └─8*[{thread}]
```

6.3 Implementation Phase

After the container technology stack has been designed, the next step is to implement and test a prototype of the design before putting the solution into production. Be aware that container technology stacks do not offer the types of introspection capabilities that VM technologies do.

In addition to the NIST SP 800-125 items, it is important to also evaluate the container technology stack's isolation capabilities. Ensure that processes within the container can access all resources they are permitted to and cannot view or access any other resources.

Implementation may also require altering the configuration of other security controls and technologies, such as security event logging, network management, code repositories, and authentication servers.

When the prototype evaluation has been completed and the container technology stack is ready for production usage, the stack should initially be used for a small number of applications. Problems that occur are likely to affect multiple applications, so it is helpful to identify these problems early on so they can be addressed before further deployment. A phased deployment also provides time for developers and IT staff (e.g., system administrators, help desk) to be trained on its usage and support.

6.4 Operations and Maintenance Phase

Operational processes that are particularly important for maintaining the security of container technology stacks, and thus should be performed regularly, include updating all images and distributing those updated images to containers to take the place of older images.

6.5 Disposition Phase

The ability for containers to be deployed and destroyed automatically based on the needs of an application allows for highly efficient systems but can also introduce some challenges for records retention, forensic, and event data requirements. Organizations should make sure that appropriate mechanisms are in place to satisfy their data retention policies. Example of issues that should be addressed are how containers and images should be destroyed, what data should be extracted from a container before disposal and how that data extraction should be performed, how cryptographic keys used by a container should be revoked or deleted, etc.

Data stores and media that support the containerized environment should be included in any disposal plans developed by the organization.

7 Conclusion

While containers represent a transformational change in the way apps are built and run, they do not fundamentally upend decades of information security best practices. On the contrary, the most important aspects of container security are simply refinements of well-established techniques and principles. Containers provide new constructs for hosting apps, but they run on the same basic stack as the VMs most organizations are already using. Securing containers is as much a function of securing the underlying stack as it is using any container-specific techniques.

Earlier, this document discussed some of the differences between securing containers and securing the same apps in VMs. It is useful to summarize the guidance in this document around those points.

There are many more entities, so your security processes and tools must be able to scale accordingly. Scale does not just mean the total number of objects supported in a database, but also how effectively and autonomously policy can be managed. Many organizations struggle with the burden of managing security across hundreds of VMs. As container-centric architectures become the norm and these organizations are responsible for thousands or tens of thousands of instances, their security practices should emphasize automation and efficiency to keep up.

With containers there is a much higher rate of change, moving from updating an app a few times a year to a few times a week or even a day. What used to be acceptable to do manually no longer is. Automation is not just important to deal with the net number of entities, but also how frequently those entities change. Being able to centrally express policy and have software manage enforcement of it across the environment is vital. Organizations that adopt containers should be prepared to manage this frequency of change, which may require fundamentally new operational practices and organizational evolution.

Security is largely in the hands of the developer, so organizations should ensure that those developers have all the security data they need to make good decisions. That data should be integrated with the tooling they already use and should allow security teams to not just notify but also actively enforce quality throughout the development cycle. Organizations that are successful at this transition gain security benefit in being able to respond to vulnerabilities faster and with less operational burden than ever before.

Security must be as portable as the containers themselves, so organizations should adopt techniques and tools that are open and work across platforms and environments. Many organizations will see developers build in one environment, test in another, and deploy in a third, so having consistency in assessment and enforcement across these is key. Portability is also not just environmental but also temporal. Continuous integration and deployment practices erode the traditional walls between phases of the development and deployment cycle, so organizations need to ensure consistent, automated security practices across creation of the image, storage of the image in registries, and running of the images in containers.

Organizations that navigate these changes do not just reach a basic stasis of their existing security policies with containers, but instead can begin to leverage containers to actually improve their overall security. The immutability and declarative nature of containers enables organizations to begin realizing the vision of more automated, app-centric security that requires minimal manual involvement and that updates itself as the apps change. Containers are an enabling capability in organizations moving from reactive, manual, high-cost security models to those that enable better scale and efficiency, thus lowering risk.

Appendix A—NIST Resources for Security Outside the Container Stack

This appendix lists NIST resources for securing systems and system components outside the container stack. Many more resources are available from other organizations.

Table 1: NIST Resources for Security Outside the Container Stack

Resource Name and URI	Applicability
SP 800-40 Revision 3, *Guide to Enterprise Patch Management Technologies* https://doi.org/10.6028/NIST.SP.800-40r3	All IT products and systems
SP 800-46 Revision 2, *Guide to Enterprise Telework, Remote Access, and Bring Your Own Device (BYOD) Security* https://doi.org/10.6028/NIST.SP.800-46r2	Client operating systems, client applications
SP 800-53 Revision 4, *Security and Privacy Controls for Federal Information Systems and Organizations* https://doi.org/10.6028/NIST.SP.800-53r4	All IT products and systems
SP 800-70 Revision 3, *National Checklist Program for IT Products: Guidelines for Checklist Users and Developers* http://dx.doi.org/10.6028/NIST.SP.800-70r3	Server operating systems, client operating systems, server applications, client applications
SP 800-83 Revision 1, *Guide to Malware Incident Prevention and Handling for Desktops and Laptops* https://doi.org/10.6028/NIST.SP.800-83r1	Client operating systems, client applications
SP 800-123, *Guide to General Server Security* https://doi.org/10.6028/NIST.SP.800-123	Servers
SP 800-124 Revision 1, *Guidelines for Managing the Security of Mobile Devices in the Enterprise* https://doi.org/10.6028/NIST.SP.800-124r1	Mobile devices
SP 800-125, *Guide to Security for Full Virtualization Technologies* https://doi.org/10.6028/NIST.SP.800-125	Hypervisors and virtual machines
SP 800-125A, *Security Recommendations for Hypervisor Deployment* http://csrc.nist.gov/publications/drafts/800-125a/sp800-125a_draft.pdf	Hypervisors and virtual machines
SP 800-125B, *Secure Virtual Network Configuration for Virtual Machine (VM) Protection* https://doi.org/10.6028/NIST.SP.800-125B	Hypervisors and virtual machines
SP 800-147, *BIOS Protection Guidelines* https://doi.org/10.6028/NIST.SP.800-147	Client hardware
SP 800-155, *BIOS Integrity Measurement Guidelines* http://csrc.nist.gov/publications/drafts/800-155/draft-SP800-155_Dec2011.pdf	Client hardware
SP 800-164, *Guidelines on Hardware-Rooted Security in Mobile Devices* http://csrc.nist.gov/publications/drafts/800-164/sp800_164_draft.pdf	Mobile devices

Appendix B—NIST Cybersecurity Framework and NIST SP 800-53 Security Controls Related to Container Stack Security

The security controls from NIST SP 800-53 Revision 4 [25] that are most important for container stack security are listed in Table 2.

Table 2: Security Controls from NIST SP 800-53 for Container Stack Security

NIST SP 800-53 Control	Related Controls	References
AC-2, Account Management	AC-3, AC-4, AC-5, AC-6, AC-10, AC-17, AC-19, AC-20, AU-9, IA-2, IA-4, IA-5, IA-8, CM-5, CM-6, CM-11, MA-3, MA-4, MA-5, PL-4, SC-13	
AC-3, Access Enforcement	AC-2, AC-4, AC-5, AC-6, AC-16, AC-17, AC-18, AC-19, AC-20, AC-21, AC- 22, AU-9, CM-5, CM-6, CM-11, MA-3, MA-4, MA-5, PE-3	
AC-4, Information Flow Enforcement	AC-3, AC-17, AC-19, AC-21, CM-6, CM-7, SA-8, SC-2, SC-5, SC-7, SC-18	
AC-6, Least Privilege	AC-2, AC-3, AC-5, CM-6, CM-7, PL-2	
AC-17, Remote Access	AC-2, AC-3, AC-18, AC-19, AC-20, CA-3, CA-7, CM-8, IA-2, IA-3, IA-8, MA-4, PE-17, PL-4, SC-10, SI-4	NIST SPs 800-46, 800-77, 800-113, 800-114, 800-121
AT-3, Role-Based Security Training	AT-2, AT-4, PL-4, PS-7, SA-3, SA-12, SA-16	C.F.R. Part 5 Subpart C (5C.F.R.930.301); NIST SPs 800-16, 800- 50
AU-2, Audit Events	AC-6, AC-17, AU-3, AU-12, MA-4, MP-2, MP-4, SI-4	NIST SP 800-92; https://idmanagement.gov/
AU-5, Response to Audit Processing Failures	AU-4, SI-12	
AU-6, Audit Review, Analysis, and Reporting	AC-2, AC-3, AC-6, AC-17, AT-3, AU-7, AU-16, CA-7, CM-5, CM-10, CM-11, IA-3, IA-5, IR-5, IR-6, MA-4, MP-4, PE-3, PE-6, PE-14, PE-16, RA-5, SC-7, SC-18, SC-19, SI-3, SI-4, SI-7	
AU-8, Time Stamps	AU-3, AU-12	
AU-9, Protection of Audit Information	AC-3, AC-6, MP-2, MP-4, PE-2, PE-3, PE-6	
AU-12, Audit Generation	AC-3, AU-2, AU-3, AU-6, AU-7	
CA-9, Internal System Connections	AC-3, AC-4, AC-18, AC-19, AU-2, AU-12, CA- 7, CM-2, IA-3, SC-7, SI-4	
CM-2, Baseline Configuration	CM-3, CM-6, CM-8, CM-9, SA-10, PM-5, PM-7	NIST SP 800-128
CM-3, Configuration Change Control	CA-7, CM-2, CM-4, CM-5, CM-6, CM-9, SA-10, SI- 2, SI-12	NIST SP 800-128
CM-4, Security Impact Analysis	CA-2, CA-7, CM-3, CM-9, SA-4, SA-5, SA-10, SI-2	NIST SP 800-128
CM-5, Access Restrictions for Change	AC-3, AC-6, PE-3	
CM-6, Configuration Settings	AC-19, CM-2, CM-3, CM-7, SI-4	OMB Memoranda 07-11, 07-18, 08-22; NIST SPs 800-70, 800-128; https://nvd.nist.gov; https://checklists.nist.gov; https://www.nsa.gov

NIST SP 800-53 Control	Related Controls	References
CM-7, Least Functionality	AC-6, CM-2, RA-5, SA-5, SC-7	DoD Instruction 8551.01
CM-9, Configuration Management Plan	CM-2, CM-3, CM-4, CM-5, CM-8, SA-10	NIST SP 800-128
CP-2, Contingency Plan	AC-14, CP-6, CP-7, CP-8, CP-9, CP-10, IR-4, IR-8, MP-2, MP-4, MP-5, PM-8, PM-11	Federal Continuity Directive 1; NIST SP 800-34
CP-9, Information System Backup	CP-2, CP- 6, MP-4, MP-5, SC-13	NIST SP 800-34
CP-10, Information System Recovery and Reconstitution	CA-2, CA-6, CA-7, CP-2, CP-6, CP-7, CP-9, SC-24	Federal Continuity Directive 1; NIST SP 800-34
IA-2, Identification and Authentication (Organizational Users)	AC-2, AC-3, AC-14, AC-17, AC-18, IA-4, IA-5, IA-8	HSPD-12; OMB Memoranda 04-04, 06-16, 11-11; FIPS 201; NIST SPs 800-63, 800-73, 800-76, 800-78; FICAM Roadmap and Implementation Guidance; https://idmanagement.gov/
IA-4, Identifier Management	AC-2, IA-2, IA-3, IA-5, IA-8, SC-37	FIPS 201; NIST SPs 800-73, 800-76, 800-78
IA-5, Authenticator Management	AC-2, AC-3, AC-6, CM-6, IA-2, IA-4, IA-8, PL-4, PS-5, PS-6, SC-12, SC-13, SC-17, SC-28	OMB Memoranda 04-04, 11-11; FIPS 201; NIST SPs 800-63, 800-73, 800-76, 800-78; FICAM Roadmap and Implementation Guidance; https://idmanagement.gov/
IR-1, Incident Response Policy and Procedures	PM-9	NIST SPs 800-12, 800-61, 800-83, 800-100
IR-4, Incident Handling	AU-6, CM-6, CP-2, CP-4, IR-2, IR-3, IR-8, PE-6, SC-5, SC-7, SI-3, SI-4, SI-7	EO 13587; NIST SP 800-61
MA-2, Controlled Maintenance	CM-3, CM-4, MA-4, MP-6, PE-16, SA-12, SI-2	
MA-4, Nonlocal Maintenance	AC- 2, AC-3, AC-6, AC-17, AU-2, AU-3, IA-2, IA-4, IA-5, IA-8, MA-2, MA-5, MP-6, PL-2, SC-7, SC-10, SC-17	FIPS 140-2, 197, 201; NIST SPs 800-63, 800-88; CNSS Policy 15
PL-2, System Security Plan	AC-2, AC-6, AC-14, AC-17, AC-20, CA-2, CA-3, CA-7, CM-9, CP-2, IR-8, MA-4, MA-5, MP-2, MP-4, MP-5, PL-7, PM-1, PM-7, PM-8, PM-9, PM-11, SA-5, SA-17	NIST SP 800-18
PL-4, Rules of Behavior	AC-2, AC-6, AC-8, AC-9, AC-17, AC-18, AC-19, AC-20, AT-2, AT-3, CM-11, IA-2, IA-4, IA-5, MP-7, PS-6, PS-8, SA-5	NIST SP 800-18
RA-2, Security Categorization	CM-8, MP-4, RA-3, SC-7	FIPS 199; NIST SPs 800-30, 800-39, 800-60
RA-3, Risk Assessment	RA-2, PM-9	OMB Memorandum 04-04; NIST SPs 800-30, 800-39; https://idmanagement.gov/
SA-10, Developer Configuration Management	CM-3, CM-4, CM-9, SA-12, SI-2	NIST SP 800-128

NIST SP 800-53 Control	Related Controls	References
SA-11, Developer Security Testing and Evaluation	CA-2, CM-4, SA-3, SA-4, SA-5, SI-2	ISO/IEC 15408; NIST SP 800-53A; https://nvd.nist.gov; http://cwe.mitre.org; http://cve.mitre.org; http://capec.mitre.org
SA-15, Development Process, Standards, and Tools	SA-3, SA-8	
SA-19, Component Authenticity	PE-3, SA-12, SI-7	
SC-2, Application Partitioning	SA-4, SA-8, SC-3	
SC-4, Information in Shared Resources	AC-3, AC-4, MP-6	
SC-6, Resource Availability		
SC-8, Transmission Confidentiality and Integrity	AC-17, PE-4	FIPS 140-2, 197; NIST SPs 800-52, 800-77, 800-81, 800-113; CNSS Policy 15; NSTISSI No. 7003
SI-2, Flaw Remediation	CA-2, CA-7, CM-3, CM-5, CM-8, MA-2, IR-4, RA-5, SA-10, SA-11, SI-11	NIST SPs 800-40, 800-128
SI-4, Information System Monitoring	AC-3, AC-4, AC-8, AC-17, AU-2, AU-6, AU-7, AU-9, AU-12, CA-7, IR-4, PE-3, RA-5, SC-7, SC-26, SC-35, SI-3, SI-7	NIST SPs 800-61, 800-83, 800-92, 800-137
SI-7, Software, Firmware, and Information Integrity	SA-12, SC-8, SC-13, SI-3	NIST SPs 800-147, 800-155

The list below details the NIST Cybersecurity Framework [26] subcategories that are most important for container stack security.

- **Identify: Asset Management**
  - ID.AM-3: Organizational communication and data flows are mapped
  - ID.AM-5: Resources (e.g., hardware, devices, data, and software) are prioritized based on their classification, criticality, and business value
- **Identify: Risk Assessment**
  - ID.RA-1: Asset vulnerabilities are identified and documented
  - ID.RA-3: Threats, both internal and external, are identified and documented
  - ID.RA-4: Potential business impacts and likelihoods are identified
  - ID.RA-5: Threats, vulnerabilities, likelihoods, and impacts are used to determine risk
  - ID.RA-6: Risk responses are identified and prioritized
- **Protect: Access Control**
  - PR.AC-1: Identities and credentials are managed for authorized devices and users
  - PR.AC-2: Physical access to assets is managed and protected
  - PR.AC-3: Remote access is managed

  - PR.AC-4: Access permissions are managed, incorporating the principles of least privilege and separation of duties
- **Protect: Awareness and Training**
  - PR.AT-2: Privileged users understand roles & responsibilities
  - PR.AT-5: Physical and information security personnel understand roles & responsibilities
- **Protect: Data Security**
  - PR.DS-2: Data-in-transit is protected
  - PR.DS-4: Adequate capacity to ensure availability is maintained
  - PR.DS-5: Protections against data leaks are implemented
  - PR.DS-6: Integrity checking mechanisms are used to verify software, firmware, and information integrity
- **Protect: Information Protection Processes and Procedures**
  - PR.IP-1: A baseline configuration of information technology/industrial control systems is created and maintained
  - PR.IP-3: Configuration change control processes are in place
  - PR.IP-6: Data is destroyed according to policy
  - PR.IP-9: Response plans (Incident Response and Business Continuity) and recovery plans (Incident Recovery and Disaster Recovery) are in place and managed
  - PR.IP-12: A vulnerability management plan is developed and implemented
- **Protect: Maintenance**
  - PR.MA-1: Maintenance and repair of organizational assets is performed and logged in a timely manner, with approved and controlled tools
  - PR.MA-2: Remote maintenance of organizational assets is approved, logged, and performed in a manner that prevents unauthorized access
- **Protect: Protective Technology**
  - PR.PT-1: Audit/log records are determined, documented, implemented, and reviewed in accordance with policy
  - PR.PT-3: Access to systems and assets is controlled, incorporating the principle of least functionality
- **Detect: Anomalies and Events**
  - DE.AE-2: Detected events are analyzed to understand attack targets and methods
- **Detect: Security Continuous Monitoring**
  - DE.CM-1: The network is monitored to detect potential cybersecurity events
  - DE.CM-7: Monitoring for unauthorized personnel, connections, devices, and software is performed
- **Respond: Response Planning**
  - RS.RP-1: Response plan is executed during or after an event
- **Respond: Analysis**
  - RS.AN-1: Notifications from detection systems are investigated
  - RS.AN-3: Forensics are performed
- **Respond: Mitigation**
  - RS.MI-1: Incidents are contained
  - RS.MI-2: Incidents are mitigated
  - RS.MI-3: Newly identified vulnerabilities are mitigated or documented as accepted risks

- **Recover: Recovery Planning**
  - RC.RP-1: Recovery plan is executed during or after an event

Table 3 lists the security controls from NIST SP 800-53 Revision 4 [25] that can be accomplished partially or completely by using container stack technology. The rightmost column lists the sections of this document that map to each NIST SP 800-53 control.

Table 3: NIST SP 800-53 Controls Supported by Container Stacks

NIST SP 800-53 Control	Container Stack Relevancy	Related Sections of This Document
CM-3, Configuration Change Control	Images can be used to help manage change control for applications.	2.3, 2.4, 2.5, 3.1, 4.4
SC-2, Application Partitioning	Separating user functionality from administrator functionality can be accomplished in part by using containers or other virtualization technologies so that the functionality is performed in different containers.	2 (introduction), 2.1, 4.3.4
SC-3, Security Function Isolation	Separating security functions from non-security functions can be accomplished in part by using containers or other virtualization technologies so that the functions are performed in different containers.	2 (introduction), 2.1, 4.3.4
SC-4, Information in Shared Resources	Container stacks are designed to restrict each container's access to shared resources so that information cannot inadvertently be leaked from one container to another.	2 (introduction), 2.1, 2.2, 4.3
SC-6, Resource Availability	The maximum resources available for each container can be specified, thus protecting the availability of resources by not allowing any container to consume excessive resources.	2.1, 2.2
SC-7, Boundary Protection	Boundaries can be established and enforced between containers to restrict their communications with each other.	2 (introduction), 2.1, 2.2, 4.3
SC-39, Process Isolation	Multiple containers can run processes simultaneously on the same host, but those processes are isolated from each other.	2 (introduction), 2.1, 2.2, 2.3, 4.3
SI-7, Software, Firmware, and Information Integrity	Unauthorized changes to the contents of images can easily be detected and the altered image replaced with a known good copy.	2.1, 4.4, 4.5
SI-14, Non-Persistence	Images running within containers are replaced as needed with new image versions, so data, files, executables, and other information stored within running images is not persistent.	2.3, 4.4

Similar to Table 3, Table 4 lists the NIST Cybersecurity Framework [26] subcategories that can be accomplished partially or completely by using container stack technology. The rightmost column lists the sections of this document that map to each Cybersecurity Framework subcategory.

Table 4: NIST Cybersecurity Framework Subcategories Supported by Container Stacks

Cybersecurity Framework Subcategory	Container Stack Relevancy	Related Sections of This Document
PR.DS-4: Adequate capacity to ensure availability is maintained	The maximum resources available for each container can be specified, thus protecting the availability of resources by not allowing any container to consume excessive resources.	2.1, 2.2
PR.DS-5: Protections against data leaks are implemented	Container stacks are designed to restrict each container's access to shared resources so that information cannot inadvertently be leaked from one container to another.	2 (introduction), 2.1, 2.2, 4.3
PR.DS-6: Integrity checking mechanisms are used to verify software, firmware, and information integrity	Unauthorized changes to the contents of images can easily be detected and the altered image replaced with a known good copy.	2.1, 4.4, 4.5
PR.DS-7: The development and testing environment(s) are separate from the production environment	Using containers makes it easier to have separate development, testing, and production environments because the same image can be used in all environments without adjustments.	2.1, 2.3
PR.IP-3: Configuration change control processes are in place	Images can be used to help manage change control for applications.	2.3, 2.4, 2.5, 3.1, 4.4

Information on these controls and guidelines on possible implementations can be found in the following NIST publications:

- *FIPS 140-2, Security Requirements for Cryptographic Modules*
- *FIPS 197, Advanced Encryption Standard (AES)*
- *FIPS 199, Standards for Security Categorization of Federal Information and Information Systems*
- *FIPS 201-2, Personal Identity Verification (PIV) of Federal Employees and Contractors*
- *Draft SP 800-12 Rev. 1, An Introduction to Information Security*
- *Draft SP 800-16 Rev. 1, A Role-Based Model for Federal Information Technology/Cybersecurity Training*
- *SP 800-18 Rev. 1, Guide for Developing Security Plans for Federal Information Systems*
- *SP 800-30 Rev. 1, Guide for Conducting Risk Assessments*
- *SP 800-34 Rev. 1, Contingency Planning Guide for Federal Information Systems*
- *SP 800-39, Managing Information Security Risk: Organization, Mission, and Information System View*
- *SP 800-40 Rev. 3, Guide to Enterprise Patch Management Technologies*
- *SP 800-46 Rev. 2, Guide to Enterprise Telework, Remote Access, and Bring Your Own Device (BYOD) Security*
- *SP 800-50, Building an Information Technology Security Awareness and Training Program*

- *SP 800-52 Rev. 1, Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations*
- *SP 800-53 Rev. 4, Security and Privacy Controls for Federal Information Systems and Organizations*
- *SP 800-53A Rev. 4, Assessing Security and Privacy Controls in Federal Information Systems and Organizations: Building Effective Assessment Plans*
- *SP 800-60 Rev. 1 Vol. 1, Guide for Mapping Types of Information and Information Systems to Security Categories*
- *SP 800-61 Rev. 2, Computer Security Incident Handling Guide*
- *Draft SP 800-63 Rev. 3, Digital Identity Guidelines*
- *SP 800-70 Rev. 3, National Checklist Program for IT Products: Guidelines for Checklist Users and Developers*
- *SP 800-73-4, Interfaces for Personal Identity Verification*
- *SP 800-76-2, Biometric Specifications for Personal Identity Verification*
- *SP 800-77, Guide to IPsec VPNs*
- *SP 800-78-4, Cryptographic Algorithms and Key Sizes for Personal Identification Verification (PIV)*
- *SP 800-81-2, Secure Domain Name System (DNS) Deployment Guide*
- *SP 800-83 Rev. 1, Guide to Malware Incident Prevention and Handling for Desktops and Laptops*
- *SP 800-88 Rev. 1, Guidelines for Media Sanitization*
- *SP 800-92, Guide to Computer Security Log Management*
- *SP 800-100, Information Security Handbook: A Guide for Managers*
- *SP 800-113, Guide to SSL VPNs*
- *SP 800-114 Rev. 1, User's Guide to Telework and Bring Your Own Device (BYOD) Security*
- *Draft SP 800-121 Rev. 2, Guide to Bluetooth Security*
- *SP 800-128, Guide for Security-Focused Configuration Management of Information Systems*
- *SP 800-137, Information Security Continuous Monitoring (ISCM) for Federal Information Systems and Organizations*
- *SP 800-147, BIOS Protection Guidelines*
- *Draft SP 800-155, BIOS Integrity Measurement Guidelines*

Appendix C—Acronyms and Abbreviations

Selected acronyms and abbreviations used in this paper are defined below.

API	Application Programming Interface
AUFS	Advanced Multi-Layered Unification Filesystem
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DevOps	Development and Operations
FIPS	Federal Information Processing Standards
FISMA	Federal Information Security Modernization Act
FOIA	Freedom of Information Act
GB	Gigabyte
I/O	Input/Output
IP	Internet Protocol
IT	Information Technology
ITL	Information Technology Laboratory
LXC	Linux Container
NIST	National Institute of Standards and Technology
NTFS	NT File System
OMB	Office of Management and Budget
OS	Operating System
RTM	Root of Trust for Measurement
SP	Special Publication
SSH	Secure Shell
TPM	Trusted Platform Module
VM	Virtual Machine

Appendix D—Glossary

Container	A method for packaging and securely running an application on a shared virtual operating system. Also known as an application container or a server application container.
Container runtime	The layer above the host operating system that provides management tools and APIs to allow users to specify how to run containers on a given host.
Filesystem virtualization	A form of virtualization that allows multiple containers to share the same physical storage, while providing each container its own unique view of that storage and prohibiting that container from viewing or tampering with the storage of other containers.
Image	A package that contains all the files required to run a container.
Isolation	The ability to keep multiple instances of software separated so that each instance only sees and can affect itself.
Microservice	A set of containers that work together to compose an application.
Namespace isolation	A form of isolation that limits the resources a container may interact with.
Operating system virtualization	A virtual implementation of the operating system interface that can be used to run applications written for the same operating system. [from [1]]
Orchestrator	A tool for centrally managing groups of container hosts, including monitoring resource consumption, job execution, and machine health.
Registry	A service that allows developers to easily storage images as they are created, tag and catalog images to aid in discovery and reuse, and find and reuse images that others have created.
Resource isolation	A form of isolation that limits how much of a host's resources a given container can consume.
Virtual machine	A simulated environment created by virtualization. [from [1]]
Virtualization	The simulation of the software and/or hardware upon which other software runs. [from [1]]

Appendix E—References

[1] NIST Special Publication (SP) 800-125, Guide to Security for Full Virtualization Technologies, National Institute of Standards and Technology, Gaithersburg, Maryland, January 2011, 35pp. https://doi.org/10.6028/NIST.SP.800-125.

[2] CoreOS, https://coreos.com

[3] Project Atomic, http://www.projectatomic.io

[4] Google Container-Optimized OS, https://cloud.google.com/container-optimized-os/docs/

[5] Docker, https://www.docker.com/

[6] Linux Containers, https://linuxcontainers.org

[7] rkt, https://coreos.com/rkt/

[8] Open Container Initiative Daemon (OCID), https://github.com/kubernetes-incubator/cri-o

[9] Amazon EC2 Container Registry (ECR), https://aws.amazon.com/ecr/

[10] Docker Hub, https://hub.docker.com/

[11] Kubernetes, https://kubernetes.io/

[12] Apache Mesos, http://mesos.apache.org/

[13] Docker Swarm, https://github.com/docker/swarm

[14] Jenkins, https://jenkins.io

[15] TeamCity, https://www.jetbrains.com/teamcity/

[16] Docker Trusted Registry, https://hub.docker.com/r/docker/dtr/

[17] Quay Container Registry, https://quay.io

[18] DC/OS, https://dcos.io

[19] NIST Special Publication (SP) 800-154, *Guide to Data-Centric System Threat Modeling (Draft)*, National Institute of Standards and Technology, Gaithersburg, Maryland, March 2016, 25pp. http://csrc.nist.gov/publications/drafts/800-154/sp800_154_draft.pdf.

[20] NIST Special Publication (SP) 800-164, *Guidelines on Hardware-Rooted Security in Mobile Devices (Draft)*, National Institute of Standards and Technology, Gaithersburg, Maryland, October 2012, 33pp. http://csrc.nist.gov/publications/drafts/800-164/sp800_164_draft.pdf.

[21] NIST Special Publication (SP) 800-147, *BIOS Protection Guidelines*, National Institute of Standards and Technology, Gaithersburg, Maryland, April 2011, 26pp. https://doi.org/10.6028/NIST.SP.800-147.

[22] NIST Special Publication (SP) 800-155, *BIOS Integrity Measurement Guidelines (Draft)*, National Institute of Standards and Technology, Gaithersburg, Maryland, December 2011, 47pp. http://csrc.nist.gov/publications/drafts/800-155/draft-SP800-155_Dec2011.pdf.

[23] Security Enhanced Linux (SELinux), https://selinuxproject.org/page/Main_Page

[24] AppArmor, http://wiki.apparmor.net/index.php/Main_Page

[25] NIST Special Publication (SP) 800-53 Revision 4, *Security and Privacy Controls for Federal Information Systems and Organizations*, National Institute of Standards and Technology, Gaithersburg, Maryland, April 2013 (including updates as of January 15, 2014), 460pp. https://doi.org/10.6028/NIST.SP.800-53r4.

[26] *Framework for Improving Critical Infrastructure Cybersecurity Version 1.0*, National Institute of Standards and Technology, Gaithersburg, Maryland, February 12, 2014. https://www.nist.gov/document-3766.