1 Introduction

1.1 Purpose and Scope

The purpose of the document is to explain the security concerns associated with application container technologies and make practical recommendations for addressing those concerns when planning for, implementing, and maintaining containers. Some aspects of containers may vary among technologies, but the recommendations in this document 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 administrative tools 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 technical capabilities, technology architectures, and uses.
- Section 3 explains the major risks for the core components of application container technologies.
- Section 4 recommends countermeasures for the risks identified in Section 3.
- Section 5 defines threat scenario examples for containers.
- Section 6 presents actionable information for planning, implementing, operating, and maintaining container technologies.
- Section 7 provides the conclusion for the document.

- Appendix A lists NIST resources for securing non-core components of container technologies.
- 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

This section provides an introduction to containers for server applications (apps). First, it defines the basic concepts for application virtualization and containers needed to understand the rest of the document. Next, this section explains how containers interact with the operating system they run on top of. The next portion of the section illustrates the overall architecture of container technologies, defining all the major components typically found in a container implementation and explaining the workflows between components. The last part of the section describes common uses for containers.

2.1 Basic Concepts for Application Virtualization and 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 (OSs) 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 multiple virtualized OSs above a single actual OS kernel. This approach is often called an *OS container*, and various implementations of OS containers have existed since the early 2000s, starting with Solaris Zone and FreeBSD jails.[1] Support initially became available in Linux in 2008 with the Linux Container (LXC) technology built into nearly all modern distributions. OS containers are different from the application containers that are the topic of this guide because OS containers are designed to provide an environment that behaves much like a normal OS in which multiple apps and services may co-exist.

Recently, application virtualization has become increasingly popular due to advances in its ease of use and a greater focus on developer agility as a key benefit. In *application virtualization*, the same shared OS kernel is exposed virtually to multiple discrete apps. OS components keep each app instance isolated from all others on the server. In this case, each app sees only the OS and itself, and is isolated from other apps that may be running on this same OS kernel.

The key difference between OS virtualization and application virtualization is that with application virtualization, each virtual instance typically runs only a single app. Today's application 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.

[1] For more information on the concept of jails, see https://www.freebsd.org/doc/handbook/jails.html.

Unlike traditional app architectures, which often divide an app into a few tiers (e.g., web, app, and database) and have a server or VM for each tier, container architectures often have an app divided into many more components, each with a single well-defined function and typically running in its own container(s). Each app component runs in a separate container. In application container technologies, sets of containers that work together to compose an app are referred to as *microservices*. With this approach, app deployment is more flexible and scalable. Development is also simpler because functionality is more self-contained. However, there are many more objects to manage and secure, which may cause problems for app management and security tools and processes.

Most application container technologies implement the concept of immutability. In other words, the containers themselves should be operated as stateless entities that are deployed but not changed.[2] When a running container needs to be upgraded or have its contents changed, it is simply destroyed and replaced with a new container that has the updates. This enables developers and support engineers to make and push changes to apps at a much faster pace. Organizations may go from deploying a new version of their app every quarter, to deploying new components weekly or daily. 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. Legacy security tools and processes often assume largely static operations and may need to be adjusted to adapt to the rate of change in containerized environments.

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 root file system, but instead by using external, persistent data stores such as databases or cluster-aware persistent volumes. The data containers 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 development and operations (DevOps) practices that seek to increase the integration between building and running apps, emphasizing close coordination between development and operational teams.[3] 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. Container images—packages containing the files required to run containers—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 without needing to make any modifications. The downside of this is that the security tools and processes

[2] Note that while containers make immutability practical and realistic, they do not *require* it, so organizations need to adapt their operational practices to take advantage of it.

[3] This document refers to tasks performed by DevOps personas. The references to these personas are focused on the types of job tasks being performed, not on strict titles or team organizational structures.

used to protect containers should not make assumptions about specific cloud providers, host OSs, network topologies, or other aspects of the container runtime environment which may frequently change. Even more critically, security should be consistent across all these environments and throughout the app lifecycle from development to test to production.

Recently, projects such as Docker [2] and rkt [3] 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.

2.2 Containers and the Host Operating System

Explaining the deployment of apps within containers is made easier by comparing it with the deployment of apps within virtual machines (VMs) from hardware virtualization technologies, which many readers are already familiar with. Figure 2 shows the VM deployment on the left, a container deployment without VMs (installed on "bare metal") in the middle, and a container deployment that runs within a VM on the right.

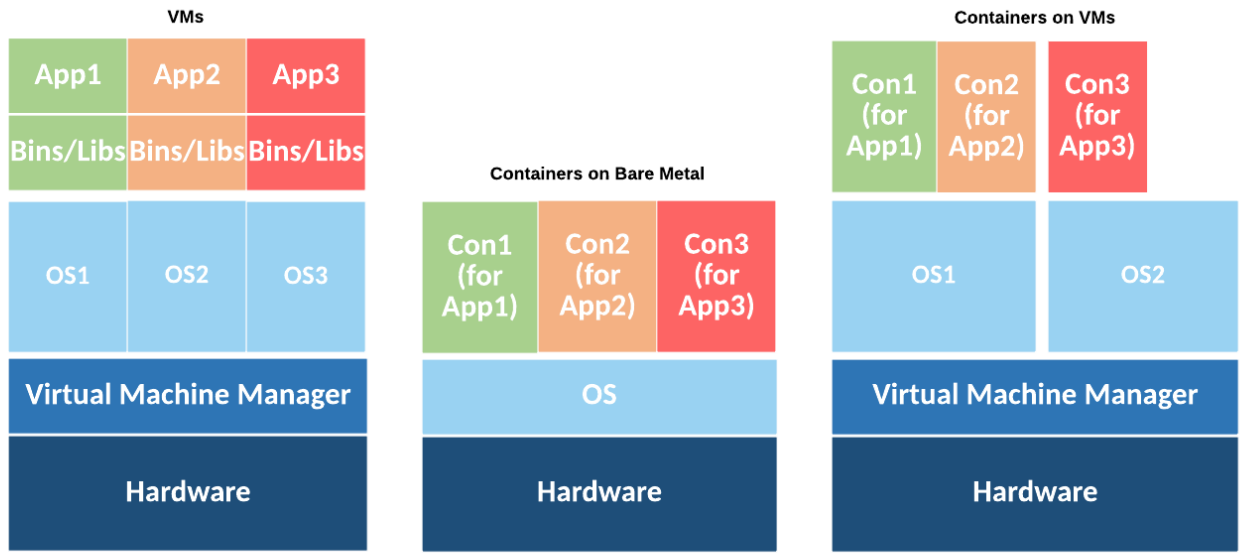

Figure 2: Virtual Machine and Container Deployments

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 OSs, such as Linux and Windows, to share the same physical hardware.

With containers, multiple apps share the same OS kernel instance but are segregated from each other. The OS kernel is part of what is called the *host operating system*. The host OS sits below the containers and provides OS capabilities to them. Containers are OS-family specific; a Linux host can only run containers built for Linux, and a Windows host can only run Windows containers. Also, a container built for one OS family should run on any recent OS from that family.

There are two general categories of host OSs used for running containers. *General-purpose OSs* like Red Hat Enterprise Linux, Ubuntu, and Windows Server can be used for running many kinds of apps and can have container-specific functionality added to them. *Container-specific OSs*, like CoreOS Container Linux [4], Project Atomic [5], and Google Container-Optimized OS [6] are minimalistic OSs explicitly designed to only run containers. They typically do not come with package managers, they have only a subset of the core administration tools, and they actively discourage running apps outside containers. Often, a container-specific OS uses a read-only file system design to reduce the likelihood of an attacker being able to persist data within it, and it also utilizes a simplified upgrade process since there is little concern around app compatibility.

Every host OS used for running containers has binaries that establish and maintain the environment for each container, also known as the *container runtime*. The container runtime coordinates multiple OS components that isolate resources and resource usage so that each container sees its own dedicated view of the OS and is isolated from other containers running concurrently. Effectively, the containers and the host OS interact through the container runtime. The container runtime also provides management tools and application programming interfaces (APIs) to allow DevOps personnel and others to specify how to run containers on a given host. The runtime eliminates the need to manually create all the necessary configurations and simplifies the process of starting, stopping, and operating containers. Examples of runtimes include Docker [2], rkt [3], and the Open Container Initiative Daemon [7].

Examples of technical capabilities the container runtime ensures the host OS provides include the following:

- **Namespace isolation** limits which resources a container may interact with. This includes file systems, network interfaces, interprocess communications, host names, user information, and processes. Namespace isolation ensures that apps 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 host with many other containers running other apps, you would only see httpd listed in the results. Namespace isolation provides each container with its own networking stack, including unique interfaces and IP addresses. Containers on Linux use technologies like masked process identities to achieve namespace isolation, whereas on Windows, object namespaces are used.
- **Resource allocation** 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 allocation ensures that each container can only utilize the amount of resources assigned to it. On Linux, this is

accomplished primarily with control groups (cgroups)[4], whereas on Windows job objects serve a similar purpose.

- **Filesystem virtualization** 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 the specifically changed bits will be written to disk, and those changes will only be visible to the container that executed them. 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).

The technical capabilities of containers vary by host OS family. 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, commonly used container runtimes provide a common interface format that largely abstracts these differences from users.

While containers provide a strong degree of isolation, they do not offer as clear and concrete of a security boundary as a VM. Because containers share the same kernel and can be run with varying capabilities and privileges on a host, the degree of segmentation between them is far less than that provided to VMs by a hypervisor. Thus, carelessly configured environments can result in containers having the ability to interact with each other and the host far more easily and directly than multiple VMs on the same host.

Although 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 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.

2.3 Container Technology Architecture

Figure 3 shows the five tiers of the container technology architecture:

[4] 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.

1. Developer systems (generate images and send them to testing and accreditation)
2. Testing and accreditation systems (validate and verify the contents of images, sign images, and send images to the registry)
3. Registries (store images and distribute images to the orchestrator upon request)
4. Orchestrators (convert images into containers and deploy containers to hosts)
5. Hosts (run and stop containers as directed by the orchestrator)

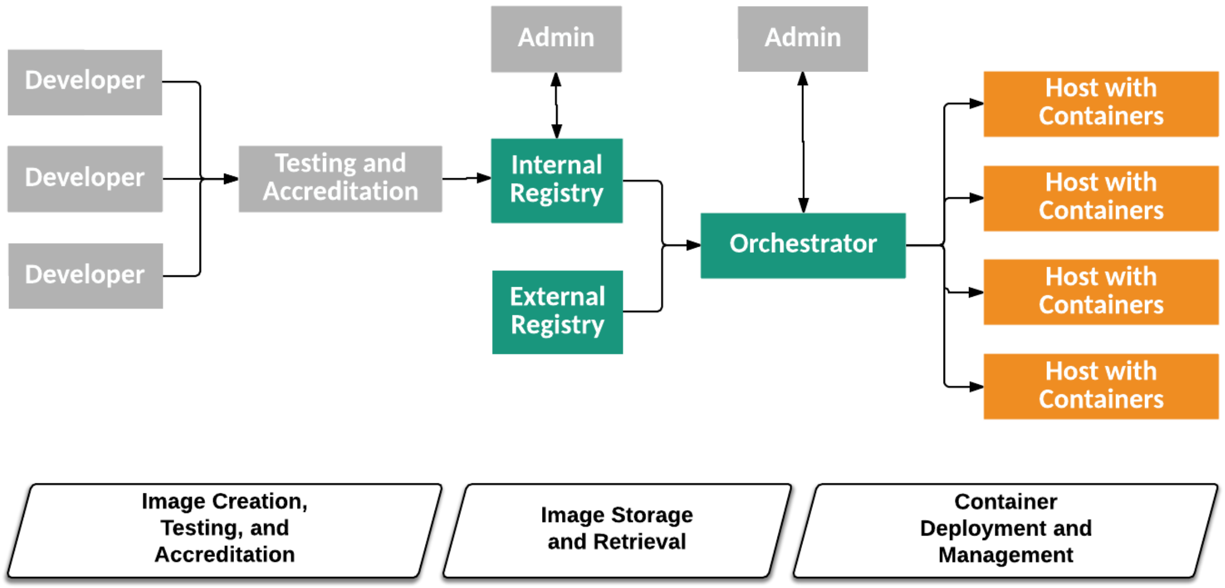

Figure 3: Container Technology Architecture Tiers, Components, and Lifecycle Phases

Although there are many administrator system personas involved in the overall process, the figure depicts only the administrator systems for the internal registry and the orchestrator.

The systems in gray (developer systems, testing and accreditation system, and administrator systems) are outside the scope of the container technology architecture, but they do have important interactions with it. In most organizations that use containers, the development and test environments also leverage containers, and this consistency is one of the key benefits of using containers. This document does not focus on systems in these environments because the recommendations for securing them are largely the same as those for the production environment. The systems in green (internal registry, external registry, and orchestrator) are core components of a container technology architecture. Finally, the systems in orange (hosts with containers) are where the containers are used.

Another way to understand the container technology architecture is to consider the container lifecycle phases, which are depicted at the bottom of Figure 3. The three phases are discussed in more detail below.

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.

2.3.1 Image Creation, Testing, and Accreditation

In the first phase of the container lifecycle, an app's components are built and placed into an image (or perhaps into multiple images). An *image* is a package that contains 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 should only include the executables and libraries required by the app itself; all other OS functionality is provided by the OS kernel within the underlying host OS. 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.

Because images are built in layers, the underlying layer upon which all other components are added is often called the *base layer*. Base layers are typically minimalistic distributions of common OSs like Ubuntu and Windows Nano Server with the OS kernel omitted. Users begin building their full images by starting with one of these base layers, then adding application frameworks and their own custom code to develop a fully deployable image of their unique app. Container runtimes support using images from within the same OS family, even if the specific host OS version is dissimilar. For example, a Red Hat host running Docker can run images created on any Linux base layer, such as Alpine or Ubuntu. However, it cannot run images created with a Windows base layer.

The image creation process is managed by developers responsible for packaging an app for handoff to testing. Image creation typically uses build management and automation tools, such as Jenkins [8] and TeamCity [9], to assist with what is called the "continuous integration" process. These tools take the various libraries, binaries, and other components of an app, perform testing on them, and then assemble images out of them based on the developer-created manifest that describes how to build an image for the app.

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 machine-parseable data to describe how the web server should run, such as the ports it listens on or the configuration parameters it uses.

After image creation, organizations typically perform testing and accreditation. For example, test automation tools and personnel would use the images built to validate the functionality of the final form application, and security teams would perform accreditation on these same images. The consistency of building, testing, and accrediting exactly the same artifacts for an app is one of the key operational and security benefits of containers.

2.3.2 Image Storage and Retrieval

Images are typically stored in central locations to make it easy to control, 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 for identification and version control to aid in discovery and reuse, and find and download images that others have created. Registries may be self-hosted or consumed as a service. Examples of registries include Amazon EC2 Container Registry [10], Docker Hub [11], Docker Trusted Registry [12], and Quay Container Registry [13].

Registries provide APIs that enable automating common image-related tasks. For example, organizations may have triggers in the image creation 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.

Once stored in a registry, images can be easily pulled and then run by DevOps personas across any environment in which they run containers. This is another example of the portability benefits of containers; image creation 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.

2.3.3 Container Deployment and Management

Tools known as *orchestrators* enable DevOps personas or automation working on their behalf to pull images from registries, deploy those images into containers, and manage the running 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. Examples of orchestrators are Kubernetes [14], Mesos [15], and Docker Swarm [16].

Note that a small, simple container implementation could omit a full-fledged orchestrator. Orchestrators may also be circumvented or unnecessary in other circumstances. For example, a host could directly contact a registry in order to pull an image from it for a container runtime. To simplify the discussions in this publication, the use of an orchestrator will be assumed.

The abstraction provided by an orchestrator allows a DevOps persona to simply specify 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 the state of each host within the cluster, including what resources are available for each host, and determines which containers will run on which hosts. The orchestrator then pulls the required images from the registry and runs them as containers with the designated resources.

Orchestration tools are also responsible for monitoring container resource consumption, job execution, and machine health across hosts. Depending on its configuration, an orchestrator may automatically restart containers on new hosts if the hosts they were initially running on failed. Many orchestrators enable cross-host container networking and service discovery. Most orchestrators also include a software-defined networking (SDN) component known as an *overlay network* that can be used to isolate communication between apps that share the same physical network.

When apps in containers need to be updated, the existing containers are not changed, but rather they are destroyed and new containers created from updated images. This is a key operational difference with containers: the baseline software from the initial deployment should not change over time, and updates are done by replacing the entire image at once. This approach has significant potential security benefits because it enables organizations to build, test, validate, and deploy exactly the same software in exactly the same configuration in each phase. As updates are made to apps, organizations can ensure that the most recent versions are used, typically by leveraging orchestrators. Orchestrators are usually 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, test the image, push it to the registry, and then rely on the automation tools to deploy it to the target environment.

This means that all vulnerability management, including patches and configuration settings, is typically taken care of by the developer when building a new image version. With containers, developers are largely responsible for the security of apps and images instead of the operations team. This change in responsibilities often requires much greater coordination and cooperation among personnel than was previously necessary. Organizations adopting containers should ensure that clear process flows and team responsibilities are established for each stakeholder group.

Container management includes security management and monitoring. However, security controls designed for non-container environments are often not well suited for use with containers. For example, consider security controls that take IP addresses into account. This works well for VMs and bare metal servers with static IP addresses that remain the same for months or years. Conversely, containers are typically allocated IP addresses by orchestrators, and because containers are created and destroyed much more frequently than VMs, these IP addresses change frequently over time as well. 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. Additionally, a container network can include communications between containers on the same node, across different nodes, and even across clouds.

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 app code to be fixed and the updated software tested and deployed much faster.
- Environmental consistency and compartmentalization, where developers can have identical yet separate environments for building, testing, and running the app. Containers give developers the ability to run the entirety of an exact copy of a production app locally on a development laptop system, limiting the need for coordination and sharing of testing environments as well as eliminating the hassle of stale testing environments.

- '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.
- Cloud-native apps, where developers can build for a microservices architecture from the beginning, ensuring more efficient iteration of the app and simplified deployment.

Containers provide additional benefits; for example, they can 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, app installation happens in production (i.e., at server runtime), typically by running hand-crafted scripts that manage installation of app 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.

With container technologies, the build system installs the app within the image it creates (i.e., at compile-time). The image is an immutable snapshot of all userspace requirements of the app (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 the same organization or among other organizations. 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.

3 Major Risks for Core Components of Container Technologies

This section identifies and analyzes major risks for the core components of container technologies—images, registries, orchestrators, containers, and host OSs. Because the analysis looks at core components only, it is applicable to most container deployments regardless of container technology, host OS platform, or location (public cloud, private cloud, etc.) Two types of risks are considered:

1. **Compromise of an image or container.** This risk was evaluated using the data-centric system threat modeling approach described in NIST SP 800-154 [17]. The primary "data" to protect is the images and containers, which may hold app files, data files, etc. The secondary data to protect is container data within shared host resources such as memory, storage, and network interfaces.
2. **Misuse of a container to attack other containers, the host OS, other hosts, etc.**

All other risks involving the core components, as well as risks involving non-core container technology components, including developer systems, testing and accreditation systems, administrator systems, and host hardware and virtual machine managers, are outside the scope of this document. Appendix A contains pointers to general references for securing non-core container technology components.

3.1 Image Risks

3.1.1 Image vulnerabilities

Because images are effectively static archive files that include all the components used to run a given app, components within an image may be missing critical security updates or are otherwise outdated. An image created with fully up-to-date components may be free of known vulnerabilities for days or weeks after its creation, but at some time vulnerabilities will be discovered in one or more image components, and thus the image will no longer be up-to-date.

Unlike traditional operational patterns in which deployed software is updated 'in the field' on the hosts 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 containers having vulnerabilities because the version of the image used to generate the containers has vulnerabilities.

3.1.2 Image configuration defects

In addition to software defects, images may also have configuration defects. For example, an image may not be configured with a specific user account to "run as" and thus run with greater privileges than needed. As another example, an image may include an SSH daemon, which exposes the container to unnecessary network risk. 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.

3.1.3 Embedded malware

Because images are just collections of files packaged together, malicious files could be included intentionally or inadvertently within them. Such malware would have the same capabilities as any other component within the image and thus could be used to attack other containers or hosts within the environment. A possible source of embedded malware is the use of base layers and other images provided by third parties of which the full provenance is not known.

3.1.4 Embedded clear text secrets

Many apps require secrets to enable secure communication between components. For example, a web app may need a username and password to connect to a backend database. Other examples of embedded secrets include connection strings, SSH private keys, and X.509 private keys. When an app is packaged into an image, these secrets can be embedded directly into the image file system. However, this practice creates a security risk because anyone with access to the image can easily parse it to learn these secrets.

3.1.5 Use of untrusted images

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 app, a user may find another version of that app 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.

3.2 Registry Risks

3.2.1 Insecure connections to registries

Images often contain sensitive components like an organization’s proprietary software and embedded secrets. 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. There is also an increased risk of man-in-the-middle attacks that could intercept network traffic intended for registries and steal developer or administrator credentials within that traffic, provide fraudulent or outdated images to orchestrators, etc.

3.2.2 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 accidental deployment of a known-vulnerable version.

3.2.3 Insufficient authentication and authorization restrictions

Because registries may contain images used to run sensitive or proprietary apps and to access sensitive data, insufficient authentication and authorization requirements can lead to intellectual property loss and expose significant technical details about an app to an attacker. Even more critically, because registries are typically trusted as a source of valid, approved software, compromise of a registry can potentially lead to compromise of downstream containers and hosts.

3.3 Orchestrator Risks

3.3.1 Unbounded administrative access

Historically, many orchestrators were designed with the assumption that all users interacting with them would be administrators and 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.

3.3.2 Unauthorized access

Orchestrators often include their own authentication directory service, 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.

Containers typically use data storage volumes that are managed by the orchestration tool and are not host specific. Because a container may run on any given node within a cluster, the data required by the app within the container must be available to the container regardless of which host it is running on. At the same time, many organizations manage data that must be encrypted at rest to prevent unauthorized access.

3.3.3 Poorly separated 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 orchestrator 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. Although an encrypted overlay network provides many operational and security benefits, it can also create a security ‘blindness’ scenario in which organizations are unable to effectively monitor traffic within their own networks.

Potentially even more critical is the risk of traffic from different apps sharing the same virtual networks. If apps of different sensitivity levels, such as a public-facing web site and an internal treasury management app, are using the same virtual network, sensitive internal apps may be exposed to greater risk from network attack. For example, if the public-facing web site is compromised, attackers may be able to use shared networks to attack the treasury app.

3.3.4 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. In the case of a critical vulnerability in the web server, this can put the container processing sensitive financial data at significantly greater risk of compromise.

3.3.5 Orchestrator node trust

Maintenance of trust between the nodes in the environment requires special care. The orchestrator is the most foundational node. Weak orchestrator configurations can expose the orchestrator and all other container technology components to increased risk. Examples of possible consequences include:

- Unauthorized hosts joining the cluster and running containers
- The compromise of a single cluster host implying compromise of the entire cluster—for example, if the same key pairs used for authentication are shared across all nodes
- Communications between the orchestrator and DevOps personnel, administrators, and hosts being unencrypted and unauthenticated

3.4 Container Risks

3.4.1 Vulnerabilities within the runtime software

While relatively uncommon, vulnerabilities within the runtime software are particularly dangerous if they allow ‘container escape’ scenarios in which malicious software can attack resources in 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 other containers, monitor container-to-container communications, etc.

3.4.2 Unbounded network access from containers

By default in most container runtimes, individual containers are able to access each other and the host OS 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. This risk is related to that from poorly separated virtual networks, as discussed in Section 3.3.3, but different because it is focused more on flows from containers to any outbound destination, not on app “cross talk” scenarios.

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 network without directly indicating 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.

3.4.3 Insecure container runtime configurations

Container runtimes typically expose many configurable options to administrators. Setting 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 containers and the host OS to increased risk from a compromised container. Similarly, if a container is run in privileged mode, it has access to all the devices on the host, thus allowing it to essentially act as part of the host OS and impact all other containers running on it.

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 OS file system and should almost never make changes to locations that control the basic functionality of the host OS (e.g., /boot or /etc for Linux containers, C:\Windows for Windows containers). If a compromised container is allowed to make changes to these paths, it could be used to elevate privileges and attack the host itself as well as other containers running on the host.

3.4.4 App vulnerabilities

Even when organizations are taking the precautions recommended in this guide, containers may still be compromised due to flaws in the apps they run. This is not a problem with containers themselves, but instead is just the manifestation of typical software flaws within a container environment. For example, a containerized web app may be vulnerable to cross-site scripting vulnerabilities, and a database front end container may be subject to Structured Query Language (SQL) injection. When a container is compromised, it can be misused in many ways, such as granting unauthorized access to sensitive information or enabling attacks against other containers or the host OS.

3.4.5 Rogue containers

Rogue containers are unplanned or unsanctioned containers in an environment. This can be a common occurrence, especially in development environments, where app developers may launch containers as a means of testing their code. If these containers are not put through the rigors of vulnerability scanning and proper configuration, they may be more susceptible to exploits. Rogue containers therefore pose additional risk to the organization, especially when they persist in the environment without the awareness of development teams and security administrators.

3.5 Host OS Risks

3.5.1 Large attack surface

Every host OS has an *attack surface*, which is the collection of all ways attackers can attempt to access and exploit the host OS's vulnerabilities. For example, any network-accessible service provides a potential entry point for attackers, adding to the attack surface. The larger the attack surface is, the better the odds are that an attacker can find and access a vulnerability, leading to a compromise of the host OS and the containers running on top of it.

3.5.2 Shared kernel

Container-specific OSs have a much smaller attack surface than that of general-purpose OSs. 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, although 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, even for container-specific OSs. In other words, the level of isolation provided by container runtimes is not as high as that provided by hypervisors.

3.5.3 Host OS component vulnerabilities

All host OSs, even container-specific ones, provide foundational system components—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 container technology architecture, they can impact all the containers and apps that run on these hosts.

3.5.4 Improper user access rights

Container-specific OSs are typically not optimized to support multiuser scenarios since interactive user logon should be rare. Organizations are exposed to risk when users log on directly to hosts to manage containers, rather than going through an orchestration layer. Direct management enables wide-ranging changes to the system and all containers on it, and can potentially enable a user that only needs to manage a specific app's containers to impact many others.

3.5.5 Host OS file system tampering

Insecure container configurations can expose host volumes to greater risk of file tampering. For example, if a container is allowed to mount sensitive directories on the host OS, that container can then change files in those directories. These changes could impact the stability and security of the host and all other containers running on it.

4 Countermeasures for Major Risks

This section recommends countermeasures for the major risks identified in Section 3.

4.1 Image Countermeasures

4.1.1 Image vulnerabilities

There is a need for container technology-specific vulnerability management tools and processes. Traditional vulnerability management tools make many assumptions about host durability and app update mechanisms and frequencies that are fundamentally misaligned with a containerized model. These tools are often unable to detect vulnerabilities within containers, 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, 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. Visibility should be centralized across the organization and provide flexible reporting and monitoring views aligned with organizations' business processes.
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 organization's vulnerability and configuration policies 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) [18] ratings above a selected threshold.

4.1.2 Image configuration defects

Organizations should adopt tools and processes to validate and enforce compliance with secure configuration best practices. For example, images should be configured to run as non-privileged users. Tools and processes that should be adopted include:

1. Validation of image configuration settings, including vendor recommendations and third-party best practices.
2. Ongoing, continuously updated, centralized reporting and monitoring of image compliance state to identify weaknesses and risks at the organizational level.
3. Enforcement of compliance requirements by optionally preventing the running of non-compliant images.
4. Use of base layers from trusted sources only, frequent updates of base layers, and selection of base layers from minimalistic technologies like Alpine Linux and Windows Nano Server to reduce attack surface areas.

A final recommendation for image configuration is that SSH and other remote administration tools designed to provide remote shells to hosts should never be enabled within containers. Containers should be run in an immutable manner to derive the greatest security benefit from their use. Enabling remote access to them via these tools implies a degree of change that violates this principle and exposes them to greater risk of network-based attack. Instead, all remote management of containers should be done through the container runtime APIs, which may be accessed via orchestration tools, or by creating remote shell sessions to the host on which the container is running.

4.1.3 Embedded malware

Organizations should continuously monitor all images for embedded malware. The monitoring processes should include the use of malware signature sets and behavioral detection heuristics based largely on actual "in the wild" attacks.

4.1.4 Embedded clear text secrets

Secrets should be stored outside of images and provided dynamically at runtime as needed. Most orchestrators, such as Docker Swarm and Kubernetes, include native management of secrets. These orchestrators not only provide secure storage of secrets 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 application container. The orchestrator can ensure that only the web application 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.

Regardless of the tool chosen, organizations should ensure that secrets are only provided to the specific containers that require them, based on a pre-defined and administrator-controlled setting, and that secrets are always encrypted at rest and in transit using Federal Information Processing Standard (FIPS) 140 approved cryptographic algorithms[5] contained in validated cryptographic modules.

4.1.5 Use of untrusted images

Organizations should maintain 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 that includes:

---

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

- 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[6];
- Enforcement to ensure that all hosts in the environment only run images from these approved lists;
- Validation of image signatures before image execution to ensure images are from trusted sources and have not been tampered with; and
- Ongoing monitoring and maintenance of these repositories to ensure images within them are maintained and updated as vulnerabilities and configuration requirements change.

4.2 Registry Countermeasures

4.2.1 Insecure connections to registries

Organizations should configure their development tools, orchestrators, and container runtimes to only connect to registries over encrypted channels. The specific steps vary between tools, but the key goal is to ensure that all data pushed to and pulled from a registry occurs between trusted endpoints and is encrypted in transit.

4.2.2 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.

Another option is using a “latest” tag for images and referencing this tag in deployment automation. However, because this tag is only a label attached to the image and not a guarantee of freshness, organizations should be cautious to not overly trust it. Regardless of whether an organization chooses to use discrete names or to use a “latest” tag, it is critical that processes be put in place to ensure that either the automation is using the most recent unique name or the images tagged “latest” actually do represent the most up-to-date versions.

4.2.3 Insufficient authentication and authorization restrictions

All access to registries that contain proprietary or sensitive images should require authentication. Any write access to a registry should require authentication to ensure that only images from trusted entities can be added to it. For example, only allow developers to push images to the specific repositories they are responsible for, rather than being able to update any repository.

[6] For more information on NIST-validated cryptographic implementations, see the Cryptographic Module Validation Program (CMVP) page at https://csrc.nist.gov/projects/cryptographic-module-validation-program.

Organizations should consider federating with existing accounts, such as their own or a cloud provider's directory service to take advantage of security controls already in place for those accounts. All write access to registries should be audited and any read actions for sensitive images should similarly be logged.

Registries also provide an opportunity to apply context-aware authorization controls to actions. For example, organizations can configure their continuous integration processes to allow images to be signed by the authorized personnel and pushed to a registry only after they have passed a vulnerability scan and compliance assessment. Organizations should integrate these automated scans into their processes to prevent the promotion and deployment of vulnerable or misconfigured images.

4.3 Orchestrator Countermeasures

4.3.1 Unbounded administrative access

Especially because of their wide-ranging span of control, orchestrators should use a least privilege access model in which users are only granted the ability to perform the specific actions on the specific hosts, containers, and images their job roles require. For example, 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.

4.3.2 Unauthorized access

Access to cluster-wide administrative accounts should be tightly controlled as these accounts provide ability to affect all resources in the environment. Organizations should use strong authentication methods, such as requiring multifactor authentication instead of just a password.

Organizations should 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.

Traditional approaches for data at rest encryption often involve the use of host-based capabilities that may be incompatible with containers. Thus, organizations should use tools for encrypting data used with containers that allow the data to be accessed properly from containers regardless of the node they are running on. Such encryption tools should provide the same barriers to unauthorized access and tampering, using the same cryptographic approaches as those defined in NIST SP 800-111 [19].

4.3.3 Poorly separated inter-container network traffic

Orchestrators should be configured to separate network traffic into discrete virtual networks by sensitivity level. While per-app segmentation is also possible, for most organizations and use cases, simply defining networks by sensitivity level provides sufficient mitigation of risk with a manageable degree of complexity. For example, public-facing apps can share a virtual network, internal apps can use another, and communication between the two should occur through a small number of well-defined interfaces.

4.3.4 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 sensitivity level.

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 sensitivity levels together on the same host OS. Segmenting containers by purpose, sensitivity, and threat posture provides additional defense in depth. Concepts such as application tiering and network and host segmentation should be taken into consideration when planning app deployments. For example, suppose a host is running containers for both a financial database and a public-facing blog. While normally the container runtime will effectively isolate these environments from each other, there is also a shared responsibility amongst the DevOps teams for each app to operate them securely and eliminate unnecessary risk. If the blog app were to be compromised by an attacker, there would be far fewer layers of defense to protect the database if the two apps are running on the same host.

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. An attacker who compromises a single server would have limited capabilities to perform reconnaissance and attacks on other containers of a similar sensitivity level and not have any additional access beyond it. This approach also ensures that any residual data, such as caches or local volumes mounted for temp files, stays within the data's 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 such that traffic for apps of one sensitivity level is separate from that of other sensitivity levels.

4.3.5 Orchestrator node trust

Orchestration platforms should be configured to provide features that create a secure environment for all the apps they run. Orchestrators should ensure that nodes are securely introduced to the cluster, have a persistent identity throughout their lifecycle, and can also provide an accurate inventory of nodes and their connectivity states. Organizations should ensure that orchestration platforms are designed specifically to be resilient to compromise of individual nodes without compromising the overall security of the cluster. A compromised node must be able to be isolated and removed from the cluster without disrupting or degrading overall cluster operations. Finally, organizations should choose orchestrators that provide mutually authenticated network connections between cluster members and end-to-end encryption of intra-cluster traffic. Because of the portability of containers, many deployments may occur across networks organizations do not directly control, so a secure-by-default posture is particularly important for this scenario.

4.4 Container Countermeasures

4.4.1 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.

4.4.2 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 should use a combination of existing network level devices and more app-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 traffic flows within the organization’s network, port scanning, or outbound access to potentially dangerous destinations.

4.4.3 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 [20], 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 (MAC) technologies like SELinux [21] and AppArmor [22] provide enhanced control and isolation for containers running Linux OSs. 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. MAC technologies provide protection at the host OS layer, ensuring that only specific files, paths, and processes are accessible to containerized apps. Organizations are encouraged to use the MAC technologies provided by their host OSs in all container deployments.

Secure computing (seccomp)[7] profiles are another mechanism that can be used to constrain the system-level capabilities containers are allocated at runtime. Common container runtimes like Docker include default seccomp profiles that drop system calls that are unsafe and typically unnecessary for container operation. Additionally, custom profiles can be created and passed to container runtimes to further limit their capabilities. At a minimum, organizations should ensure that containers are run with the default profiles provided by their runtime and should consider using additional profiles for high-risk apps.

4.4.4 App vulnerabilities

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

[7] For more information on seccomp, see https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

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 using behavioral learning and build security profiles for them to minimize human interaction. These profiles should then be able to prevent and 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.

Containers should also be run with their root filesystems in read-only mode. This approach isolates writes to specifically defined directories, which can then be more easily monitored by the aforementioned tools. Furthermore, using read-only filesystems makes the containers more resilient to compromise since any tampering is isolated to these specific locations and can be easily separated from the rest of the app.

4.4.5 Rogue containers

Organizations should institute separate environments for development, test, production, and other scenarios, each with specific controls to provide role-based access control for container deployment and management activities. All container creation should be associated with individual user identities and logged to provide a clear audit trail of activity. Further, organizations are encouraged to use security tools that can enforce baseline requirements for vulnerability management and compliance prior to allowing an image to be run.

4.5 Host OS Countermeasures

4.5.1 Large attack surface

For organizations using container-specific OSs, the threats are typically more minimal to start with since the OSs are specifically designed to host containers and have other services and functionality disabled. Further, because these optimized OSs 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 OSs to reduce their attack surfaces and mitigate the typical risks and hardening activities associated with general-purpose OSs.

Organizations that cannot use a container-specific OS should follow the guidance in NIST SP 800-123, *Guide to General Server Security* [23] to reduce the attack surface of their hosts as much as possible. For example, hosts that run containers should only run containers and not run other apps, like a web server or database, outside of containers. The host OS should not run unnecessary system services, such as a print spooler, that increase its attack and patching surface areas. Finally, hosts should be continuously scanned for vulnerabilities and updates applied quickly, not just to the container runtime but also to lower-level components such as the kernel that containers rely upon for secure, compartmentalized operation.

4.5.2 Shared kernel

In addition to grouping container workloads onto hosts by sensitivity level, organizations should not mix containerized and non-containerized workloads on the same host instance. For example, if a host is running a web server container, it should not also run a web server (or any other app) as a regularly installed component directly within the host OS. Keeping containerized workloads isolated to container-specific hosts makes it simpler and safer to apply countermeasures and defenses that are optimized for protecting containers.

4.5.3 Host OS component vulnerabilities

Organizations should implement management practices and tools to validate the versioning of components provided for base OS management and functionality. Even though container-specific OSs have a much more minimal set of components than general-purpose OSs, 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. The OS should be kept up to date not only with security updates, but also the latest component updates recommended by the vendor. This is particularly important for the kernel and container runtime components as newer releases of these components often add additional security protections and capabilities beyond simply correcting vulnerabilities. Some organizations may choose to simply redeploy new OS instances with the necessary updates, rather than updating existing systems. This approach is also valid, although it often requires more sophisticated operational practices.

Host OSs should be operated in an immutable manner with no data or state stored uniquely and persistently on the host and no application-level dependencies provided by the host. Instead, all app components and dependencies should be packaged and deployed in containers. This enables the host to be operated in a nearly stateless manner with a greatly reduced attack surface. Additionally, it provides a more trustworthy way to identify anomalies and configuration drift.

4.5.4 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, login anomalies are monitored, and any escalation to perform 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 to manipulate containers.

4.5.5 Host file system tampering

Ensure that containers are run with the minimal set of file system permissions required. Very rarely should containers mount local file systems on a host. Instead, any file changes that containers need to persist to disk should be made within storage volumes specifically allocated for this purpose. In no case should containers be able to mount sensitive directories on a host's file system, especially those containing configuration settings for the operating system.

Organizations should use tools that can monitor what directories are being mounted by containers and prevent the deployment of containers that violate these policies.

4.6 Hardware Countermeasures

Software-based security is regularly defeated, as acknowledged in NIST SP 800-164 [24]. NIST defines trusted computing requirements in NIST SPs 800-147 [25], 800-155 [26], 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. Hardware root of trust is not a concept unique to containers, but container management and security tools can leverage attestations for the rest of the container technology architecture to ensure containers are being run in secure environments.

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 apps. Note that while standards are being developed to enable verification of hardware trust by users of cloud services, not all clouds expose this functionality to their customers. In cases where technical verification is not provided, organizations should address hardware trust requirements as part of their service agreements with cloud providers.

The increasing complexity of systems and the deeply embedded nature of today’s threats means that security should extend across all container technology components, 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. For container technologies, 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.

As of this writing, NIST is collaborating with industry partners to build reference architectures based on commercial off-the-shelf products that demonstrate the trusted computing model for container environments.[8]

[8] For more information on previous NIST efforts in this area, see NIST IR 7904, *Trusted Geolocation in the Cloud: Proof of Concept Implementation* [27].

5 Container Threat Scenario Examples

To illustrate the effectiveness of the recommended countermeasures 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 app. If that app has a vulnerability, it may be used to subvert the app 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 recommended countermeasures 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 an uncommon 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 apps of a common sensitivity level that are sharing the host. For example, a compromised runtime on a host only running web apps would not impact runtimes on other hosts running containers for financial apps.
3. Security tools that can report on the vulnerability state of runtimes and prevent the deployment of images to vulnerable runtimes 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 vetted, tested, validated, and digitally signed images are allowed to be uploaded to an organization's registries.
2. Ensuring that only trusted images are allowed to run, which will prevent images from external, unvetted sources from being used.
3. Automatically scanning images for vulnerabilities and malware, which may detect malicious code such as rootkits embedded within an image.
4. Implementing runtime controls that limit the container's ability to abuse resources, escalate privileges, and run executables.
5. Using container-level network segmentation to limit the "blast radius" of what the poisoned image might do.
6. Validating a container runtime operates following least-privilege and least-access principles.
7. Building a threat profile of the container's runtime. This includes, but is not limited to, processes, network calls, and filesystem changes.
8. Validating the integrity of images before runtime by leveraging hashes and digital signatures.
9. Restrict images from being run based on rules establishing acceptable vulnerability severity levels. For example, prevent images with vulnerabilities that have a Moderate or higher CVSS rating from being run.

6 Container Technology Life Cycle Security Considerations

It is critically important to carefully plan before installing, configuring, and deploying container technologies. 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 technologies 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 context. For example, instead of creating a virtualization security policy, create a container technology 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 container 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 apps. 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 with the new way to build, ship, and run apps.

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 OSs, 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 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 technologies do not offer the types of introspection capabilities that VM technologies do.

NIST SP 800-125 [1] cites several aspects of virtualization technologies that should be evaluated before production deployment, including authentication, connectivity and networking, app functionality, management, performance, and the security of the technology itself. In addition to those, it is important to also evaluate the container technology'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 require new security tools that are specifically focused on containers and cloud-native apps and that have visibility into their operations that traditional tools lack. Finally, deployment may also include altering the configuration of existing security controls and technologies, such as security event logging, network management, code repositories, and authentication servers both to work with containers directly and to integrate with these new container security tools.

When the prototype evaluation has been completed and the container technology is ready for production usage, containers should initially be used for a small number of apps. Problems that occur are likely to affect multiple apps, 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 technologies, and thus should be performed regularly, include updating all images and distributing those updated images to containers to take the place of older images. Other security best practices, such as performing vulnerability management and updates for other supporting layers like hosts and orchestrators, are also key ongoing operational tasks. Container security and monitoring tools should similarly be integrated with existing security information and event management (SIEM) tools to ensure container-related events flow through the same tools and processes used to provide security throughout the rest of the environment.

If and when security incidents occur within a containerized environment, organizations should be prepared to respond with processes and tools that are optimized for the unique aspects of containers. The core guidance outlined in NIST SP 800-61, *Computer Security Incident Handling Guide* [28], is very much applicable for containerized environments as well. However, organizations adopting containers should ensure they enhance their responses for some of the unique aspects of container security.

- Because containerized apps may be run by a different team than the traditional operations team, organizations should ensure that whatever teams are responsible for container operations are brought into the incident response plan and understand their role in it.
- As discussed throughout this document, the ephemeral and automated nature of container management may not be aligned with the asset management policies and tools an organization has traditionally used. Incident response teams must be able to know the roles, owners, and sensitivity levels of containers, and be able to integrate this data into their process.
- Clear procedures should be defined to respond to container related incidents. For example, if a particular image is being exploited, but that image is in use across hundreds of containers, the response team may need to shut down all of these containers to stop the attack. While single vulnerabilities have long been able to cause problems across many systems, with containers, the response may require rebuilding and redeploying a new image widely, rather than installing a patch to existing systems. This change in response

may involve different teams and approvals and should be understood and practiced ahead of time.

- As discussed previously, logging and other forensic data may be stored differently in a containerized environment. Incident response teams should be familiar with the different tools and techniques required to gather data and have documented processes specifically for these environments.

6.5 Disposition Phase

The ability for containers to be deployed and destroyed automatically based on the needs of an app 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. Examples 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

Containers represent a transformational change in the way apps are built and run. They do not necessitate dramatically new security best practices; on the contrary, most important aspects of container security are refinements of well-established techniques and principles. This document has updated and expanded general security recommendations to take the risks particular to container technologies into account.

This document has already 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.

In container environments there are many more entities, so 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 containers, 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 with 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. This may require fundamentally new operational practices and organizational evolution.

The use of containers shifts much of the responsibility for security to developers, so organizations should ensure their developers have all the information, skills, and tools they need to make sound decisions. Also, security teams should be enabled to 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 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 Securing Non-Core Components

This appendix lists NIST resources for securing non-core container technology components, including developer systems, testing and accreditation systems, administrator systems, and host hardware and virtual machine managers. Many more resources are available from other organizations.

Table 1: NIST Resources for Securing Non-Core Components

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 apps
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* https://doi.org/10.6028/NIST.SP.800-70r3	Server operating systems, client operating systems, server apps, client apps
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 apps
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 (Second Draft)* https://csrc.nist.gov/publications/detail/sp/800-125A/draft	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* https://csrc.nist.gov/publications/detail/sp/800-155/draft	Client hardware
SP 800-164, *Guidelines on Hardware-Rooted Security in Mobile Devices* https://csrc.nist.gov/publications/detail/sp/800-164/draft	Mobile devices

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

The security controls from NIST SP 800-53 Revision 4 [29] that are most important for container technologies are listed in Table 2.

Table 2: Security Controls from NIST SP 800-53 for Container Technology 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 [30] subcategories that are most important for container technology 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 [29] that can be accomplished partially or completely by using container technologies. 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 Technologies

NIST SP 800-53 Control	Container Technology Relevancy	Related Sections of This Document
CM-3, Configuration Change Control	Images can be used to help manage change control for apps.	2.1, 2.2, 2.3, 2.4, 4.1
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), 2.3, 4.5.2
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), 2.3, 4.5.2
SC-4, Information in Shared Resources	Container technologies 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), 2.2, 2.3, 4.4
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.	2.2, 2.3
SC-7, Boundary Protection	Boundaries can be established and enforced between containers to restrict their communications with each other.	2 (introduction), 2.2, 2.3, 4.4
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, 4.4
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.	2.3, 4.1, 4.2
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.	2.1, 2.3, 4.1

Similar to Table 3, Table 4 lists the NIST Cybersecurity Framework [30] subcategories that can be accomplished partially or completely by using container technologies. 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 Technologies

Cybersecurity Framework Subcategory	Container Technology 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.	2.2, 2.3