1

2

Application Container Security Guide

4

5

Murugiah Souppaya
John Morello
Karen Scarfone

9

10

11

12

13

14

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

16

17

NIST

**National Institute of
Standards and Technology**

U.S. Department of Commerce

18   **Draft NIST Special Publication 800-190**

19   # Application Container Security Guide

20
21

22   Murugiah Souppaya
23   *Computer Security Division*
24   *Information Technology Laboratory*
25
26   John Morello
27   *Twistlock*
28   *Baton Rouge, Louisiana*
29
30   Karen Scarfone
31   *Scarfone Cybersecurity*
32   *Clifton, Virginia*
33
34
35

36   April 2017
37
38

47   **Authority**

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

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

85   **Reports on Computer Systems Technology**

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

96

97   **Abstract**

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

103

104   **Keywords**

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

107

108   **Acknowledgements**

113

114   **Audience**

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

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

123

124   **Trademark Information**

125   All registered trademarks or trademarks belong to their respective organizations.

126

127

Executive Summary

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

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

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

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

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

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

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

159 **Automate compliance with container runtime configuration standards to minimize**
160 **vulnerabilities.**

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

165   should use tools or processes that continuously assess container runtime configuration settings
166   and immediately act to correct any deviations from the approved standard.
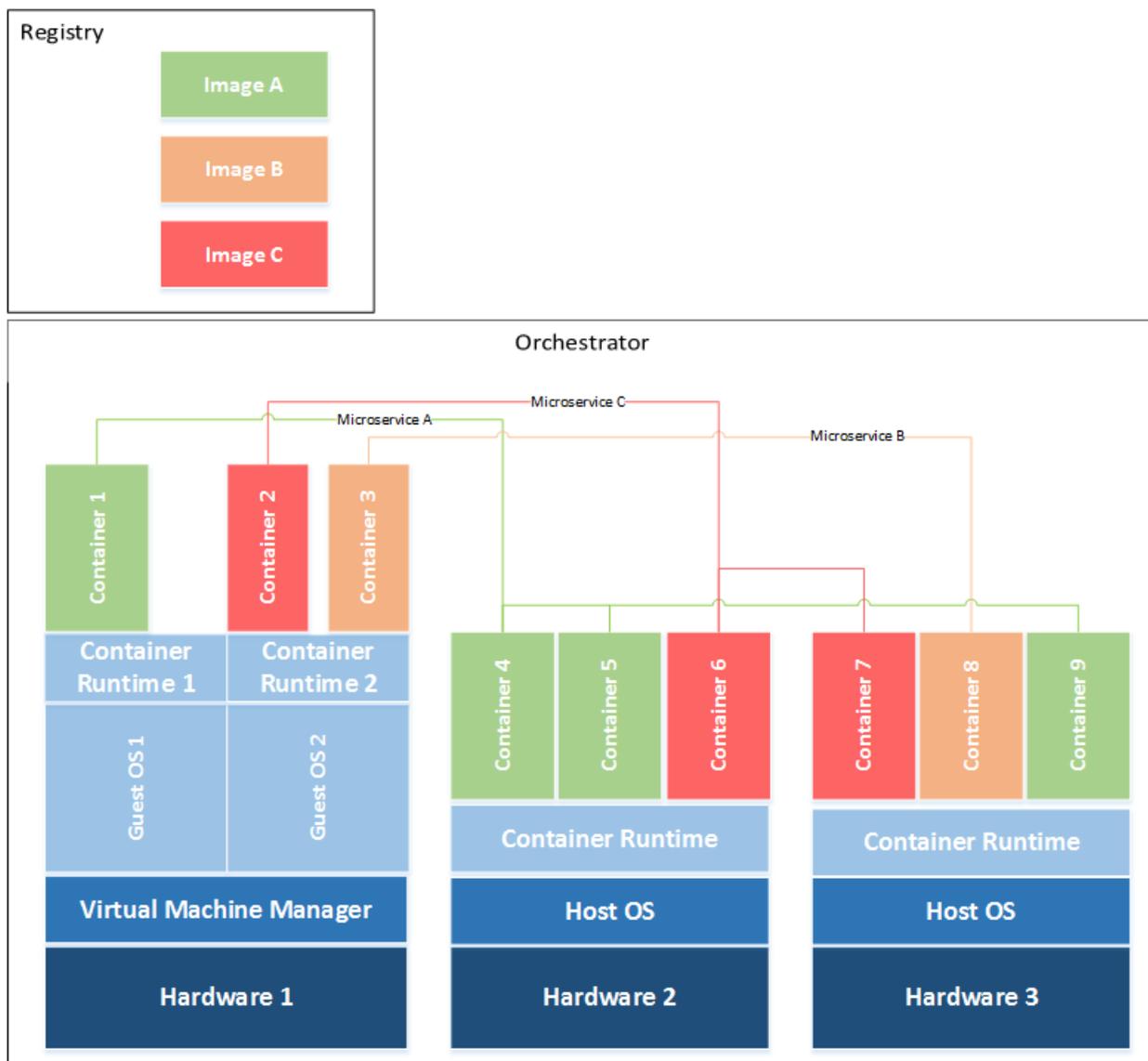
167

168   **Figure 1: Container Technology Stack**

169

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

172   While most container runtime environments do an effective job of isolating containers from each
173   other and from the host OS, in some cases it may be an unnecessary risk to run apps of different
174   classification levels together on the same host OS. Grouping containers by purpose and
175   sensitivity provides additional defense in depth. By grouping containers in this manner, it will be

176   much more difficult for an attacker who compromises one of the groups to expand that
177   compromise to other groups. This approach also ensures that any residual data, such as caches or
178   local volumes mounted for temp files, stays within its security zone.

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

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

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

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

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

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

206

207

208
209   **Table of Contents**

282
283   **List of Appendices**

290
291   **List of Tables and Figures**

302

1   Introduction

1.1   Purpose and Scope

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

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

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

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

- The layers under application container technologies, including hardware, hypervisors, and operating systems;

- The client endpoint devices that use the applications within the containers; and

- The administrator endpoints used to manage the applications within the containers and the containers themselves.

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

1.2   Document Structure

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

- Section 2 introduces containers, including their architectures, technical capabilities, attributes, and uses.

- Section 3 explains the major risks in the container technology stack.

- Section 4 discusses possible countermeasures for the risks identified in Section 3 and makes recommendations for selecting and using countermeasures.

- Section 5 defines threat scenario examples for containers.

- Section 6 presents actionable information for planning, implementing, operating, and maintaining a container technology stack.

337   ■ Section 7 provides a conclusion for the document.

338   ■ Appendix A lists NIST resources for securing systems and system components outside the
339   container technology stack.

340   ■ Appendix B lists the NIST Special Publication 800-53 security controls and NIST
341   Cybersecurity Framework subcategories that are most pertinent to application container
342   technologies, explaining the relevancy of each.

343   ■ Appendix C provides an acronym and abbreviation list for the document.

344   ■ Appendix D presents a glossary of selected terms from the document.

345   ■ Appendix E contains a list of references for the document.

346

347 ## 2   Introduction to Application Containers

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

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

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

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

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

378 ### 2.1   Container Architecture

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

383
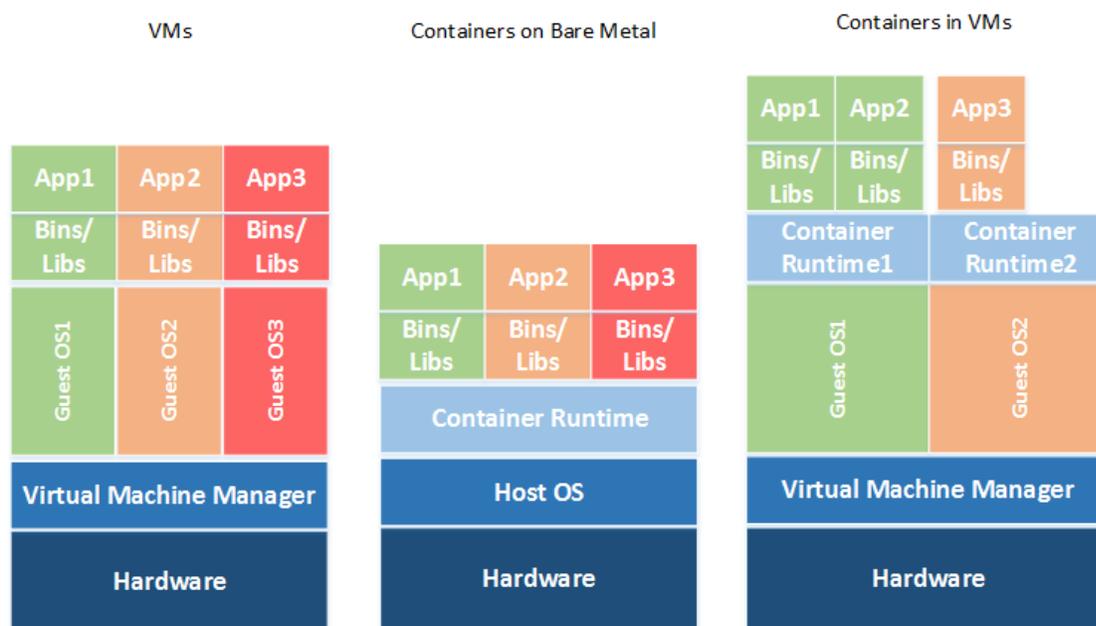
384

385
Figure 2: Virtual Machine and Container Architectures

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

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

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

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

407   • **Host operating system**: Containers share a common kernel that is part of the *host*
408   *operating system*. It sits below the containers and provides OS capabilities to them. The
409   host OSes used for running containers can generally be categorized into two types:
410   o General-purpose OSes like Red Hat Enterprise Linux, Ubuntu, and Windows Server
411   that can be used for running many kinds of apps and can have container-specific
412   functionality added to them.
413   o Container-specific OSes, like CoreOS [2], Project Atomic [3], and Google Container-
414   Optimized OS [4], which are minimalistic OSes explicitly designed to only run
415   containers. They typically do not come with package managers, and they actively
416   discourage running applications outside containers. A container-specific OS includes
417   the container runtime environment and a subset of core system administration tools.
418   Often, these OSes use a read-only file system design to reduce the likelihood of an
419   attacker being able to persist data within them, and they also utilize a simplified
420   upgrade process since there is little concern around application compatibility.
421   • **Container runtime:** The layer above the host OS is the *container runtime*. It abstracts
422   the underlying host OS from each container, such that each container sees its own
423   dedicated view of the OS and is isolated from other containers running concurrently. The
424   container runtime also provides management tools and application programming
425   interfaces (APIs) to allow users to specify how to run containers on a given host. The
426   runtime abstracts the complexity of manually creating all the necessary configurations
427   and simplifies the process of starting, stopping, and operating containers. Examples of
428   runtimes include Docker [5], LXC [6], rkt [7], and the Open Container Initiative Daemon
429   [8].
430   • **Images:** *Images* are packages that contain all the files required to run a container. For
431   example, an image to run Apache would include the httpd binary, along with associated
432   libraries and configuration files. An image is executed within a container. Unlike a VM,
433   an image does not contain an OS because that is provided by the host OS. Images are
434   typically designed to be portable across machines and environments, so that an image
435   created in a development lab can be easily moved to a test lab for evaluation, then copied
436   into a production environment to run. Images often use techniques like layering and copy
437   on write (in which shared master images are read only and changes are recorded to
438   separate files) to minimize their size on disk and improve operational efficiency.
439   • **Registry**: Images are typically stored in central locations to make it easy to share, find,
440   and reuse them across hosts. *Registries* are services that allow developers to easily store
441   images as they are created, tag and catalog images to aid in discovery and reuse, and find
442   and reuse images that others have created. When an image needs to be promoted from
443   dev to test or production, the image can be pulled from this central registry. Registries are
444   effectively special purpose file sharing apps and may be self-hosted or consumed as a
445   service, such as with Amazon EC2 Container Registry [9] or Docker Hub [10].
446   • **Microservice:** Sets of containers that work together to compose an application are
447   referred to as *microservices*. Unlike traditional architectures, which divide an application
448   into a few tiers and have a component for each tier, in a container architecture a single
449   app is often divided into many more components. With this modular approach, each
450   container may have a single well-defined function. This allows more granular scaling of

451   the app because additional resources can be provided just to the containers with the
452   function that needs them. It also makes iterative development easier because functionality
453   is more self-contained.
454   • **Orchestrators:** Multiple container hosts can be grouped together and centrally managed
455   by orchestration tools, also known as *orchestrators*. These are responsible for monitoring
456   resource consumption, job execution, and machine health across multiple servers and/or
457   VMs. This abstraction allows a developer to simply describe how many containers need
458   to be running a given image and what resources, such as memory, processing, and disk
459   need to be allocated to each. The orchestrator knows what is available within the cluster
460   and dynamically assigns which containers will run on which hosts. Further, the
461   orchestrator will monitor the health of hosts and containers and, depending on its
462   configuration, may automatically restart containers on new hosts if the hosts they were
463   initially running on failed. Many orchestrators can also enable cross-host container
464   networking and service discovery. Examples of orchestrators include Kubernetes [11],
465   Mesos [12], and Docker Swarm [13].

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

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

475
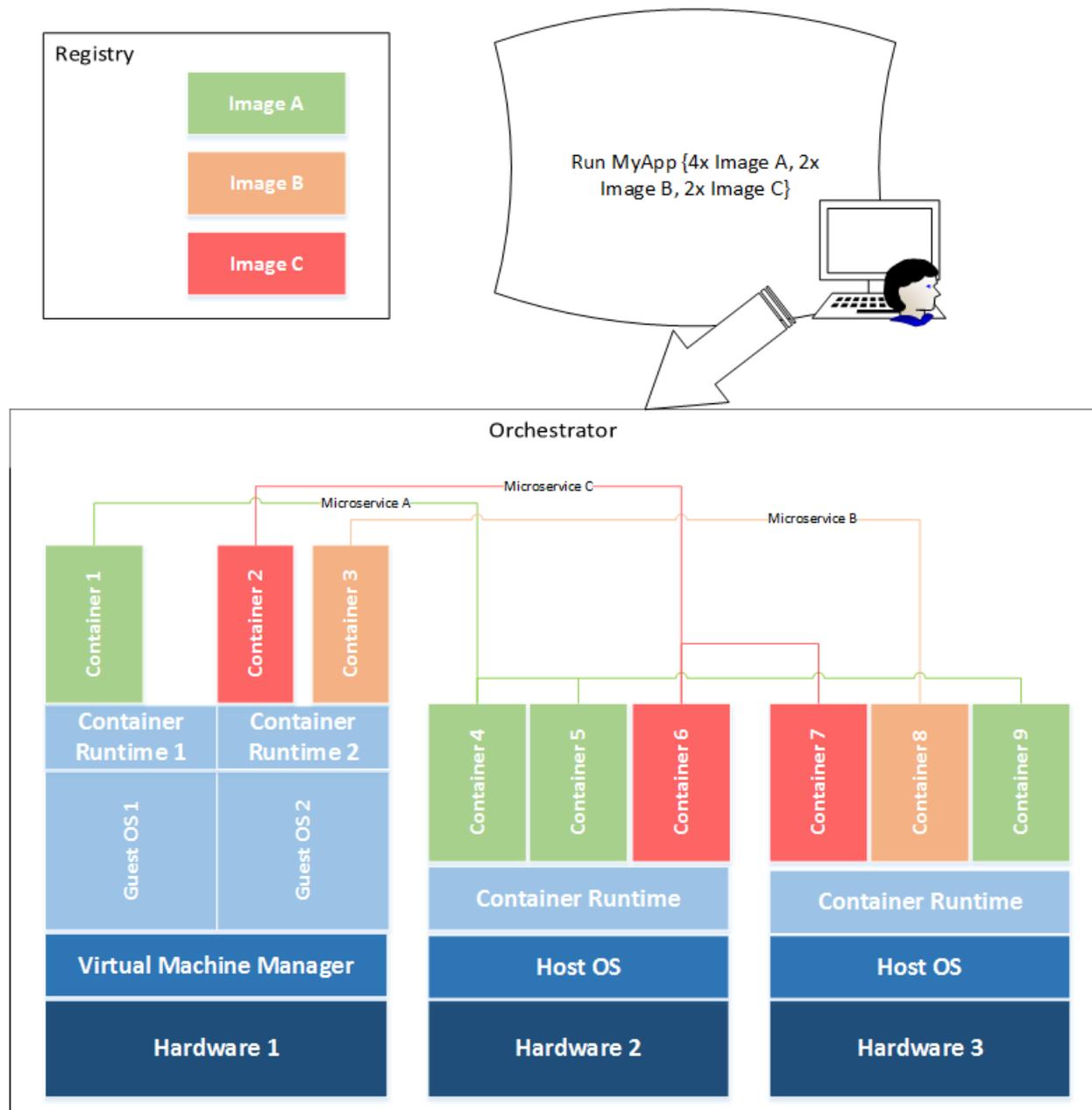
476

477   **Figure 3: Interactions of Container Deployment Components**

478   **2.2   Container Technical Capabilities**

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

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

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

513   **2.3   Container Attributes**

514   Container technologies generally share several fundamental attributes:

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

---

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

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

527   • **Declarative**. Most container technologies have a declarative way of describing the
528   components and requirements for the app. For example, an image for a web server would
529   include not only the executables for the web server, but also some parseable data to
530   describe how the web server should run, such as the ports it listens on or the
531   configuration parameters it uses.

532   • **Immutable**. Most modern container technologies implement the concept of immutability.
533   In other words, the containers themselves are stateless entities that are deployed but not
534   changed. When a running container needs to be upgraded or have its contents changed, it
535   is simply destroyed and recreated with a new image containing the updates. This provides
536   the ability for developers and support engineers to make and push changes to applications
537   at a much faster pace. Immutability is a fundamental operational difference between
538   containers and hardware virtualization. Traditional VMs are typically run as stateful
539   entities that are deployed, reconfigured, and upgraded throughout their life.

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

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

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

565   Using containers, the full application installation happens in the build system (i.e., at compile-
566   time). The build system creates the full production artifact (i.e., the container image), which is an

567   immutable snapshot of all userspace requirements of the application (i.e., programming language
568   runtime, dependent third-party libraries, init scripts, and OS tools). In production the container
569   image constructed by the build system is simply downloaded and run. This solves the "works on
570   my machine" problem since the developer, build system, and production all run the same
571   immutable artifact.

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

582   **2.4   Container Uses**

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

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

600   **2.5   The Container Lifecycle**

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

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

613

Build Phase
- Personas: Developers
- Example tooling: Jenkins, TeamCity
- Phase starts with: creation of an image from app components
- Phase ends with: pushing the image to a registry

Run Phase
- Personas: Operations
- Example tooling: Kubernetes, DC/OS, Docker
- Phase starts with: pulling an image from a registry and orchestrating its deployment

Distribution Phase
- Personas: Operations
- Example tooling: Docker Registry, Amazon EC2 Container Registry
- Phase starts with: storage of an image pushed by a developer
- Phase ends with: image pulled to run within a container

614

615   **Figure 4: Container Lifecycle Phases**

2.5.1   Build phase

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

626   **2.5.2   Distribution phase**

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

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

643   **2.5.3   Run phase**

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

656

657   **3   Major Risks in the Container Technology Stack**

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

664
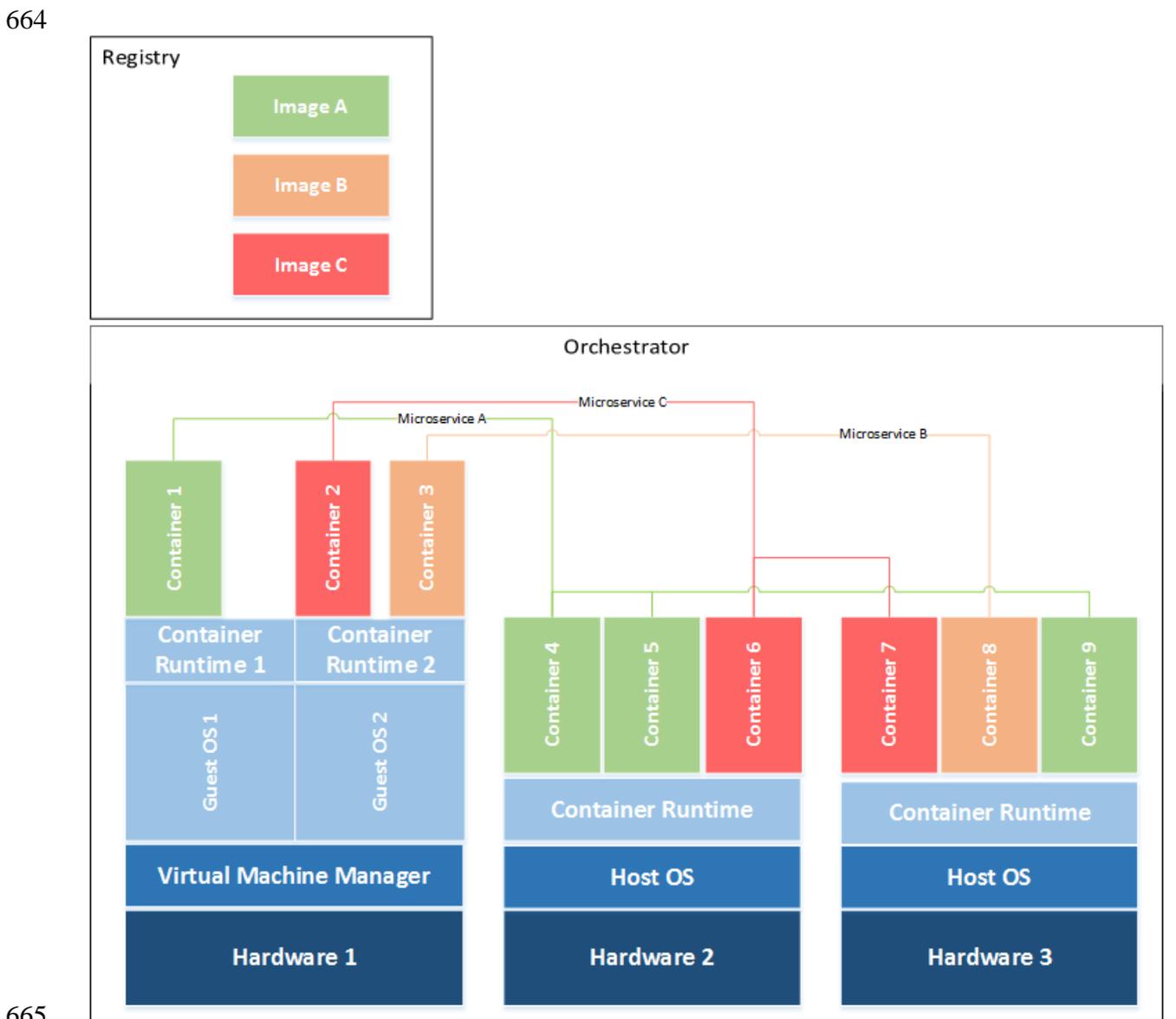
666   **Figure 5: Container Technology Stack**

667   This section begins by discussing the most important operational differences between VMs and
668   containers, which all have security implications. The rest of the section walks through the

669   container technology stack from lowest layer to highest layer, identifying and analyzing major
670   risks relevant to each layer. Appendix A contains pointers to references for securing systems and
671   system components outside the container technology stack.

672   **3.1   Operational Differences Between Containers and VMs**

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

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

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

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

724 ## 3.2   Host OS Risks

725 ### 3.2.1   Improper user access rights

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

731 ### 3.2.2   Host component vulnerabilities

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

740 ## 3.3   Container Runtime Risks

741 ### 3.3.1   Vulnerabilities within the runtime software

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

3.3.2   Unbounded network access from containers

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

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

3.3.3   Insecure container runtime configurations

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

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

3.3.4   Shared kernel

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

3.4   Image Risks

3.4.1   Image vulnerabilities

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

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

789   **3.4.2   Image configuration**

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

795   **3.4.3   Embedded malware**

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

802   **3.4.4   Embedded secrets**

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

809   **3.4.5   Image trust**

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

817   **3.5   Registry Risks**

818   **3.5.1   Insecure connections to registries**

819   Images often contain sensitive components like an organization's line of business application.
820   While, ideally, images should not include secrets or user data, the software itself is often

821   proprietary to an organization and should be protected in transit. If connections to registries are
822   performed over insecure channels, the contents of images are subject to the same confidentiality
823   risks as any other data transmitted in the clear.

824   **3.5.2   Stale images in registries**

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

829   **3.6   Orchestrator Risks**

830   **3.6.1   Unbounded administrative access**

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

837   **3.6.2   Weak or unmanaged credentials**

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

843   **3.6.3   Unmanaged inter-container network traffic**

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

852   **3.6.4   Mixing of workload sensitivity levels**

853   Orchestrators are typically focused primarily on driving the scale and density of workloads. This
854   means that, by default, they can place workloads of differing sensitivity levels on the same host.
855   For example, in a default configuration, an orchestrator may place a container running a public-
856   facing web server on the same host as one processing sensitive financial data, simply because

857   that host happens to have the most available resources at the time of deployment. This can put
858   the container processing sensitive financial data at significantly greater risk of compromise.

859 ## 4   Countermeasures for Mitigating the Major Risks

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

862 ### 4.1   Hardware Countermeasures

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

869 The currently available way to provide trusted computing is to:

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

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

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

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

890 ### 4.2   Host OS Countermeasures

891 For customers using container-specific OSes, the threats are typically more minimal to start with
892 since the OSes are specifically designed to host containers and have other services and
893 functionality disabled. Further, because these optimized OSes are designed specifically for
894 hosting containers, they typically feature read-only file systems and employ other hardening
895 practices by default. Whenever possible, organizations should use these minimalistic OSes to
896 reduce their attack surfaces and mitigate the typical risks and hardening activities associated with

897   general-purpose OSes. This section is thus focused primarily on risks relevant to these container-
898   optimized OSes.

4.2.1   Vulnerabilities in core system components

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

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

4.2.2   Improper user access rights

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

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

4.3   Container Runtime Countermeasures

4.3.1   Vulnerabilities within the runtime software

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

4.3.2 Unbounded network access from containers

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

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

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

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

4.3.3 Insecure container runtime configurations

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

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

4.3.4 Shared kernel

While most container runtime environments do an effective job of isolating containers from each other and from the host OS, in some cases it may be an unnecessary risk to run apps of different

973   classification levels together on the same runtime. Segmenting containers by purpose and
974   sensitivity provides additional defense in depth. For example, consider a scenario in which a host
975   is running containers for both a financial database and a public-facing blog. While normally the
976   container runtime will securely isolate these environments from each other, there is also a shared
977   responsibility amongst the DevOps teams for each app to operate them properly. If the DevOps
978   team for the blog were to run their app in a privileged mode and it was compromised, the
979   attacker may be able to escalate privileges to attack the database.

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

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

996   In larger-scale environments with hundreds of hosts and thousands of containers, this
997   segmentation must be automated to be practical to operationalize. Fortunately, common
998   orchestration platforms typically include some notion of being able to group apps together, and
999   container security tools can use attributes like container names and labels to enforce security
1000   policies across them. In these environments, additional layers of defense in depth beyond simple
1001   host isolation may also leverage this segmentation. For example, an organization may implement
1002   separate hosting 'zones' or networks to not only isolate these containers within hypervisors but
1003   also to isolate their network traffic more discretely.

1004   **4.3.5   Compromised containers**

1005   Existing host-based intrusion detection processes and tools are often unable to detect and prevent
1006   attacks within containers due to the differing technical architecture and operational practices
1007   previously discussed. Organizations should implement additional tools that are container aware
1008   and designed to operate at the scale and change rate typically seen with containers. These tools
1009   should be able to automatically profile containerized apps and build protection profiles for them
1010   to minimize human interaction. These profiles should then be able to detect anomalies at
1011   runtime, including events such as:

1012   • Invalid or unexpected process execution,
1013   • Invalid or unexpected system calls,

1014   • Changes to protected configuration files and binaries,
1015   • Writes to unexpected locations and file types,
1016   • Creation of unexpected network listeners,
1017   • Traffic sent to unexpected network destinations, and
1018   • Malware storage or execution.

4.4   Image Countermeasures

4.4.1   Image vulnerabilities

1021   There is a need for container-specific vulnerability management tools and processes. Traditional
1022   vulnerability management tools make many assumptions about host durability, app update
1023   mechanisms, and update frequencies that are fundamentally misaligned with a containerized
1024   model. These tools are often unable to detect vulnerabilities within containerized stacks, leading
1025   to a false sense of safety. Organizations should use tools that take the pipeline-based build
1026   approach and immutable nature of containers and images into their design to provide more
1027   actionable and reliable results. Key aspects of effective tools and processes include:

1028   1. Integration with the entire lifecycle of images and containers, from the beginning of the
1029   build process, to whatever registries the organization is using, to runtime.
1030   2. Visibility into vulnerabilities at all layers of the image, not just the base layer of the
1031   image but also application frameworks and custom software the organization is using.
1032   3. Policy driven enforcement; organizations should be able to create 'quality gates' at each
1033   stage of the build and deployment process to ensure that only images that meet the
1034   vulnerable policy are allowed to progress. For example, organizations should be able to
1035   configure a rule in the build process to prevent the progression of images that include
1036   vulnerabilities with Common Vulnerability Scoring System (CVSS) ratings above a
1037   selected threshold.

4.4.2   Image configuration

1039   In addition to software vulnerabilities, images may be configured in ways that increase security
1040   risks and violate organizational policies. For example, images should be configured to run as
1041   non-privileged users and should not allow remote access to themselves. Organizations should
1042   adopt tools and processes to validate and enforce compliance with these secure configuration
1043   best practices. Such tools and processes should include:

1044   1. Validation of image configuration settings including both vendor recommendations and
1045   custom / 3rd party best practices.
1046   2. Centralized reporting and monitoring of image compliance state to identify weaknesses
1047   and risks at the organizational level.
1048   3. Enforcement of compliance requirements by preventing the running of non-compliant
1049   images.

4.4.3   Malware

1051   Organizations should use tools and practices to monitor images for malware both at rest and
1052   when running in containers. These processes should include:

24

1053   1. Identification of malware within images both in registries and on hosts,
1054   2. The usage of comprehensive malware signature sets and detection heuristics based on
1055   actual 'in the wild' attacks,
1056   3. The detection of malware introduced to a container at runtime; for example, if a container
1057   is subverted and the attacker downloads a rootkit into it.

4.4.4 Embedded secrets

1059 Sensitive data should never be stored within image files. Instead, these secrets should be stored
1060 outside of the images and provided dynamically at runtime as needed. Most orchestration
1061 platforms, such as Docker Swarm and Kubernetes, include secret management natively. These
1062 platforms not only provide secure secret storage and 'just in time' injection to containers, but
1063 also make it much simpler to integrate secret management into the build and deployment
1064 processes. For example, an organization could use these tools to securely provision the database
1065 connection string into a web app container. The platform would ensure that only the web app
1066 container had access to this secret, that it is not persisted to disk, and that anytime the web app is
1067 deployed, the secret is provisioned into it.

1068 Organizations may also integrate their container deployments with existing enterprise secret
1069 management systems that are already in use for storing secrets in non-container environments.
1070 These tools typically provide APIs to retrieve secrets securely as containers are deployed, which
1071 eliminates the need to persist them within images.

4.4.5 Image trust

1073 Organizations should enforce a set of trusted images and registries and ensure that only images
1074 from this set are allowed to run in their environment, thus mitigating the risk of untrusted or
1075 malicious components being deployed.

1076 To mitigate these risks, organizations should take a multilayered approach to ensure that only
1077 trusted, valid images are run within their environment. Such an approach should include:

1078   • Capability to centrally control exactly what images and registries are trusted in their
1079   environment;
1080   • Discrete identification of each image by cryptographic signature, using a NIST-validated
1081   implementation[2];
1082   • Quality gates to ensure that only images that have been validated from a compliance and
1083   vulnerability state are allowed to be pushed to these locations;
1084   • Enforcement to ensure that all hosts in the environment only run images from these
1085   approved lists; and
1086   • Ongoing monitoring and maintenance of these repositories to ensure images within them
1087   are maintained and updated as vulnerabilities and configuration requirements change.

---

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

1088   **4.5   Registry Countermeasures**

1089   **4.5.1   Insecure connections to registries**

1090   Organizations should configure their container runtimes to only connect to registries over
1091   encrypted channels. The specific steps vary between runtime and orchestrator, but the key goal is
1092   to ensure that all data pulled from a registry is encrypted in transit between the registry and the
1093   destination.

1094   **4.5.2   Stale images in registries**

1095   The risk of using stale images can be mitigated through two primary methods. First,
1096   organizations can prune registries of unsafe, vulnerable images that should no longer be used.
1097   This process can be automated based on time triggers and labels associated with images.
1098   Second, operational practices should emphasize accessing images using immutable names that
1099   specify discrete versions of images to be used. For example, rather than configuring a
1100   deployment job to use the image called my-app, configure it to deploy specific versions of the
1101   image, such as my-app:2.3 and my-app:2.4 to ensure that specific, known good instances of
1102   images are deployed as part of each job.

1103   **4.6   Orchestrator Countermeasures**

1104   **4.6.1   Unbounded administrative access**

1105   Especially because of their wide-ranging span of control, orchestrators should use a least
1106   privileged access model in which users are only granted ability to perform the specific actions on
1107   the specific hosts, containers, and images their job role requires. For examples, members of the
1108   test team should only be given access to the images used in testing and the hosts used for running
1109   them, and should only be able to manipulate the containers they created. Test team members
1110   should have limited or no access to containers used in production.

1111   **4.6.2   Weak or unmanaged credentials**

1112   Access to cluster-wide administrative accounts should be tightly controlled as these accounts
1113   provide ability to affect all resources in the environment. Organizations should also implement
1114   single sign on to existing directory systems where applicable. Single sign on simplifies the
1115   orchestrator authentication experience, makes it easier for users to use strong authentication
1116   credentials, and centralizes auditing of access, making anomaly detection more effective.

1117   **4.6.3   Mixing of workload sensitivity levels**

1118   Orchestrators should be configured to isolate deployments to specific sets of hosts by sensitivity
1119   levels. The particular approach for implementing this varies depending on the orchestrator in use,
1120   but the general model is to define rules that prevent high sensitivity workloads from being placed
1121   on the same host as those running lower sensitivity workloads. This can be accomplished
1122   through the use of host 'pinning' within the orchestrator or even simply by having separate,
1123   individually managed clusters for each classification level.

1124

1125   **5   Container Threat Scenario Examples**

1126   To illustrate the effectiveness of the recommended mitigations from Section 4, consider the
1127   following threat scenario examples for containers.

1128   **5.1   Exploit of a Vulnerability within an Image**

1129   One of the most common threats to a containerized environment is application-level
1130   vulnerabilities in the software within containers. For example, an organization may build an
1131   image based on a common web application. If that application has a vulnerability, it may be used
1132   to subvert the application within the container. Once compromised, the attacker may be able to
1133   map other systems in the environment, attempt to elevate privileges within the compromised
1134   container, or abuse the container for use in attacks on other systems (such as acting as a file
1135   dropper or command and control endpoint).

1136   Organizations that adopt the recommendations would have multiple layers of defense in depth
1137   against such threats:

1138   1. Detecting the vulnerable image early in the deployment process and having controls in
1139   place to prevent vulnerable images from being deployed would prevent the vulnerability
1140   from being introduced into production.
1141   2. Container-aware network monitoring and filtering would detect anomalous connections
1142   to other containers during the attempt to map other systems.
1143   3. Container-aware process monitoring and malware detection would detect the running of
1144   invalid or unexpected malicious processes and the data they introduce into the
1145   environment.

1146   **5.2   Exploit of the Container Runtime**

1147   While a rare occurrence, if a container runtime were compromised, an attacker could utilize this
1148   access to attack all the containers on the host and even the host itself.

1149   Relevant mitigations for this threat scenario include:

1150   1. The usage of mandatory access control capabilities can provide additional barriers to
1151   ensure that process and file system activity is still segmented within the defined
1152   boundaries.
1153   2. Segmentation of workloads ensures that the scope of the compromise would be limited to
1154   applications of a common classification level that are sharing the host. For example, a
1155   compromised runtime on a host only running web applications would not impact
1156   runtimes on other hosts running containers for financial applications.
1157   3. Security tools that can report on the vulnerability state of runtimes and prevent the
1158   deployment of images to vulnerable ones can prevent workloads from running there.

1159   **5.3   Running a Poisoned Image**

1160   Because images are easily sourced from public locations, often with unknown provenance, an
1161   attacker may embed malicious software within images known to be used by a target. For

1162   example, if an attacker determines that a target is active on a discussion board about a particular
1163   project and uses images provided by that project's web site, the attacker may seek to craft
1164   malicious versions of these images for use in an attack.

1165   Relevant mitigations include:

1166   1. Ensuring that only trusted images are allowed to run will prevent images from external,
1167   unvetted sources from being used.
1168   2. Automated scanning of images for vulnerabilities and malware may detect malicious
1169   code such as rootkits embedded within an image.

1170

1171

6   Secure Container Technology Stack Planning and Implementation

1172

1173   It is critically important to carefully plan before installing, configuring, and deploying container
1174   technology stacks. This helps ensure that the container environment is as secure as possible and
1175   is in compliance with all relevant organizational policies, external regulations, and other
1176   requirements.

1177   There is a great deal of similarity in the planning and implementation recommendations for
1178   container technology stacks and virtualization solutions. Section 5 of NIST SP 800-125 [1]
1179   already contains a full set of recommendations for virtualization solutions. Instead of repeating
1180   all those recommendations here, this section points readers to that document and states that,
1181   besides the exceptions listed below, organizations should apply all the NIST SP 800-125 Section
1182   5 recommendations in a container technology stack context. For example, instead of creating a
1183   virtualization security policy, create a container technology stack security policy.

1184   This section of the document lists exceptions and additions to the NIST SP 800-125 Section 5
1185   recommendations, grouped by the corresponding phase in the planning and implementation life
1186   cycle.

6.1   Initiation Phase

1187

1188   Organizations should consider how other security policies may be affected by containers and
1189   adjust these policies as needed to take containers into consideration. For example, policies for
1190   incident response (especially forensics) and vulnerability management may need to be adjusted
1191   to take into account the special requirements of containers.

1192   The introduction of containerization technologies might disrupt the existing culture and software
1193   development methodologies within the organization. To take full advantage of the benefits
1194   containers can provide, the organization's processes should be tailored to support this new way
1195   of developing, running, and supporting applications. Traditional development practices, patching
1196   techniques, and system upgrade processes might not directly apply to a containerized
1197   environment, and it is important that the employees within the organization are willing to adapt
1198   to a new model. New processes can consider and address any potential culture shock that is
1199   introduced by the technology shift. Education and training can be offered to anyone involved in
1200   the software development lifecycle to allow people to become comfortable and excited for the
1201   new way to build, ship, and run applications.

6.2   Planning and Design Phase

1202

1203   The primary container-specific consideration for the planning and design phase is forensics.
1204   Because containers mostly build on components already present in OSes, the tools and
1205   techniques for performing forensics in a containerized environment are mostly an evolution of
1206   existing practices. The immutable nature of containers and images can actually improve forensic
1207   capabilities because the demarcation between what an image should do and what actually
1208   occurred during an incident is clearer. For example, if a container launched to run a web server
1209   suddenly starts a mail relay, it is very clear that the new process was not part of the original

1210   image used to create the container. On traditional platforms, with less separation between the OS
1211   and apps, making this differentiation can be much more difficult.

1212   Organizations that are familiar with process, memory, and disk incident response activities will
1213   find them largely similar when working with containers. However, there are some differences to
1214   keep in mind as well.

1215   Containers typically use a layered file system that is virtualized from the host OS. Directly
1216   examining paths on the hosts typically only reveals the outer boundary of these layers, not the
1217   files and data within them. Thus, when responding to incidents in containerized environments,
1218   users should identify the specific storage provider in use and understand how to properly
1219   examine its contents offline.

1220   Containers are typically connected to each other using virtualized overlay networks. These
1221   overlay networks frequently use encapsulation and encryption to allow the traffic to be routed
1222   over existing networks securely. However, this means that when investigating incidents on
1223   container networks, particularly when doing any live packet analysis, the tools used must be
1224   aware of these virtualized networks and understand how to extract the embedded IP frames from
1225   within them for parsing with existing tools.

1226   Process and memory activity within containers is largely similar to that which would be observed
1227   within traditional apps, but with different parent processes. For example, container runtimes may
1228   spawn all processes within containers in a nested fashion in which the runtime is the top-level
1229   process with first-level descendants per container and second-level descendants for each process
1230   within the container. For example:

```
1231    ├─containerd─────────  ──[container1──┬─bash]
1232                                          └─8*[{thread}]]
1233                          ├─container2─────────┬─start.sh─┬─mongod───22*[{mongod}]
1234                                                          └─node──┬─4*[{V8 WorkerThread}]
1235                                                                  └─5*[{node}]
1236                                                └─8*[{thread}]]
1237                          ├─container3─────────┬─mysqld──28*[{mysqld}]
1238                                                └─8*[{thread}]]
```
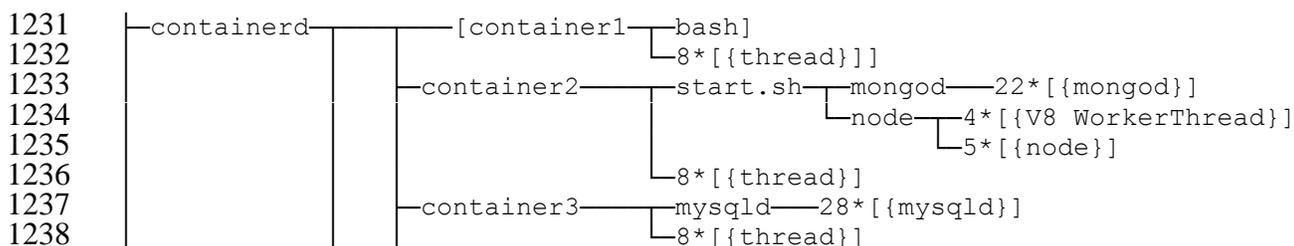
6.3   Implementation Phase

1240   After the container technology stack has been designed, the next step is to implement and test a
1241   prototype of the design before putting the solution into production. Be aware that container
1242   technology stacks do not offer the types of introspection capabilities that VM technologies do.

1243   In addition to the NIST SP 800-125 items, it is important to also evaluate the container
1244   technology stack's isolation capabilities. Ensure that processes within the container can access
1245   all resources they are permitted to and cannot view or access any other resources.

1246   Implementation may also require altering the configuration of other security controls and
1247   technologies, such as security event logging, network management, code repositories, and
1248   authentication servers.

1249   When the prototype evaluation has been completed and the container technology stack is ready
1250   for production usage, the stack should initially be used for a small number of applications.
1251   Problems that occur are likely to affect multiple applications, so it is helpful to identify these
1252   problems early on so they can be addressed before further deployment. A phased deployment
1253   also provides time for developers and IT staff (e.g., system administrators, help desk) to be
1254   trained on its usage and support.

1255   **6.4   Operations and Maintenance Phase**

1256   Operational processes that are particularly important for maintaining the security of container
1257   technology stacks, and thus should be performed regularly, include updating all images and
1258   distributing those updated images to containers to take the place of older images.

1259   **6.5   Disposition Phase**

1260   The ability for containers to be deployed and destroyed automatically based on the needs of an
1261   application allows for highly efficient systems but can also introduce some challenges for
1262   records retention, forensic, and event data requirements. Organizations should make sure that
1263   appropriate mechanisms are in place to satisfy their data retention policies. Example of issues
1264   that should be addressed are how containers and images should be destroyed, what data should
1265   be extracted from a container before disposal and how that data extraction should be performed,
1266   how cryptographic keys used by a container should be revoked or deleted, etc.

1267   Data stores and media that support the containerized environment should be included in any
1268   disposal plans developed by the organization.

1269

7   Conclusion

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

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

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

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

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

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

Organizations that navigate these changes do not just reach a basic stasis of their existing security policies with containers, but instead can begin to leverage containers to actually improve their overall security. The immutability and declarative nature of containers enables

1310   organizations to begin realizing the vision of more automated, app-centric security that requires
1311   minimal manual involvement and that updates itself as the apps change. Containers are an
1312   enabling capability in organizations moving from reactive, manual, high-cost security models to
1313   those that enable better scale and efficiency, thus lowering risk.

1314   **Appendix A—NIST Resources for Security Outside the Container Stack**

1315   This appendix lists NIST resources for securing systems and system components outside the
1316   container stack. Many more resources are available from other organizations.

1317   **Table 1: NIST Resources for Security Outside the Container Stack**

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

1318

1319

1320

1321
1322

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

1323 The security controls from NIST SP 800-53 Revision 4 [25] that are most important for container
1324 stack security are listed in Table 2.

1325 **Table 2: Security Controls from NIST SP 800-53 for Container Stack Security**

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

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

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

1326

1327   The list below details the NIST Cybersecurity Framework [26] subcategories that are most
1328   important for container stack security.

1329   • **Identify: Asset Management**
1330   o ID.AM-3: Organizational communication and data flows are mapped
1331   o ID.AM-5: Resources (e.g., hardware, devices, data, and software) are prioritized
1332   based on their classification, criticality, and business value
1333   • **Identify: Risk Assessment**
1334   o ID.RA-1: Asset vulnerabilities are identified and documented
1335   o ID.RA-3: Threats, both internal and external, are identified and documented
1336   o ID.RA-4: Potential business impacts and likelihoods are identified
1337   o ID.RA-5: Threats, vulnerabilities, likelihoods, and impacts are used to determine risk
1338   o ID.RA-6: Risk responses are identified and prioritized
1339   • **Protect: Access Control**
1340   o PR.AC-1: Identities and credentials are managed for authorized devices and users
1341   o PR.AC-2: Physical access to assets is managed and protected
1342   o PR.AC-3: Remote access is managed

1343   o PR.AC-4: Access permissions are managed, incorporating the principles of least
1344   privilege and separation of duties
1345   • **Protect: Awareness and Training**
1346   o PR.AT-2: Privileged users understand roles & responsibilities
1347   o PR.AT-5: Physical and information security personnel understand roles &
1348   responsibilities
1349   • **Protect: Data Security**
1350   o PR.DS-2: Data-in-transit is protected
1351   o PR.DS-4: Adequate capacity to ensure availability is maintained
1352   o PR.DS-5: Protections against data leaks are implemented
1353   o PR.DS-6: Integrity checking mechanisms are used to verify software, firmware, and
1354   information integrity
1355   • **Protect: Information Protection Processes and Procedures**
1356   o PR.IP-1: A baseline configuration of information technology/industrial control
1357   systems is created and maintained
1358   o PR.IP-3: Configuration change control processes are in place
1359   o PR.IP-6: Data is destroyed according to policy
1360   o PR.IP-9: Response plans (Incident Response and Business Continuity) and recovery
1361   plans (Incident Recovery and Disaster Recovery) are in place and managed
1362   o PR.IP-12: A vulnerability management plan is developed and implemented
1363   • **Protect: Maintenance**
1364   o PR.MA-1: Maintenance and repair of organizational assets is performed and logged
1365   in a timely manner, with approved and controlled tools
1366   o PR.MA-2: Remote maintenance of organizational assets is approved, logged, and
1367   performed in a manner that prevents unauthorized access
1368   • **Protect: Protective Technology**
1369   o PR.PT-1: Audit/log records are determined, documented, implemented, and reviewed
1370   in accordance with policy
1371   o PR.PT-3: Access to systems and assets is controlled, incorporating the principle of
1372   least functionality
1373   • **Detect: Anomalies and Events**
1374   o DE.AE-2: Detected events are analyzed to understand attack targets and methods
1375   • **Detect: Security Continuous Monitoring**
1376   o DE.CM-1: The network is monitored to detect potential cybersecurity events
1377   o DE.CM-7: Monitoring for unauthorized personnel, connections, devices, and software
1378   is performed
1379   • **Respond: Response Planning**
1380   o RS.RP-1: Response plan is executed during or after an event
1381   • **Respond: Analysis**
1382   o RS.AN-1: Notifications from detection systems are investigated
1383   o RS.AN-3: Forensics are performed
1384   • **Respond: Mitigation**
1385   o RS.MI-1: Incidents are contained
1386   o RS.MI-2: Incidents are mitigated
1387   o RS.MI-3: Newly identified vulnerabilities are mitigated or documented as accepted
1388   risks

1389   • **Recover: Recovery Planning**
1390   o RC.RP-1: Recovery plan is executed during or after an event
1391

1392   Table 3 lists the security controls from NIST SP 800-53 Revision 4 [25] that can be
1393   accomplished partially or completely by using container stack technology. The rightmost column
1394   lists the sections of this document that map to each NIST SP 800-53 control.

1395   **Table 3: NIST SP 800-53 Controls Supported by Container Stacks**

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

1396

1397   Similar to Table 3, Table 4 lists the NIST Cybersecurity Framework [26] subcategories that can
1398   be accomplished partially or completely by using container stack technology. The rightmost
1399   column lists the sections of this document that map to each Cybersecurity Framework
1400   subcategory.

1401   **Table 4: NIST Cybersecurity Framework Subcategories Supported by Container Stacks**

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

1402

1403   Information on these controls and guidelines on possible implementations can be found in the
1404   following NIST publications:

1405   • *FIPS 140-2, Security Requirements for Cryptographic Modules*
1406   • *FIPS 197, Advanced Encryption Standard (AES)*
1407   • *FIPS 199, Standards for Security Categorization of Federal Information and Information*
1408   *Systems*
1409   • *FIPS 201-2, Personal Identity Verification (PIV) of Federal Employees and Contractors*
1410   • *Draft SP 800-12 Rev. 1, An Introduction to Information Security*
1411   • *Draft SP 800-16 Rev. 1, A Role-Based Model for Federal Information*
1412   *Technology/Cybersecurity Training*
1413   • *SP 800-18 Rev. 1, Guide for Developing Security Plans for Federal Information Systems*
1414   • *SP 800-30 Rev. 1, Guide for Conducting Risk Assessments*
1415   • *SP 800-34 Rev. 1, Contingency Planning Guide for Federal Information Systems*
1416   • *SP 800-39, Managing Information Security Risk: Organization, Mission, and Information*
1417   *System View*
1418   • *SP 800-40 Rev. 3, Guide to Enterprise Patch Management Technologies*
1419   • *SP 800-46 Rev. 2, Guide to Enterprise Telework, Remote Access, and Bring Your Own*
1420   *Device (BYOD) Security*
1421   • *SP 800-50, Building an Information Technology Security Awareness and Training*
1422   *Program*

1423   • *SP 800-52 Rev. 1, Guidelines for the Selection, Configuration, and Use of Transport*
1424   *Layer Security (TLS) Implementations*
1425   • *SP 800-53 Rev. 4, Security and Privacy Controls for Federal Information Systems and*
1426   *Organizations*
1427   • *SP 800-53A Rev. 4, Assessing Security and Privacy Controls in Federal Information*
1428   *Systems and Organizations: Building Effective Assessment Plans*
1429   • *SP 800-60 Rev. 1 Vol. 1, Guide for Mapping Types of Information and Information*
1430   *Systems to Security Categories*
1431   • *SP 800-61 Rev. 2, Computer Security Incident Handling Guide*
1432   • *Draft SP 800-63 Rev. 3, Digital Identity Guidelines*
1433   • *SP 800-70 Rev. 3, National Checklist Program for IT Products: Guidelines for Checklist*
1434   *Users and Developers*
1435   • *SP 800-73-4, Interfaces for Personal Identity Verification*
1436   • *SP 800-76-2, Biometric Specifications for Personal Identity Verification*
1437   • *SP 800-77, Guide to IPsec VPNs*
1438   • *SP 800-78-4, Cryptographic Algorithms and Key Sizes for Personal Identification*
1439   *Verification (PIV)*
1440   • *SP 800-81-2, Secure Domain Name System (DNS) Deployment Guide*
1441   • *SP 800-83 Rev. 1, Guide to Malware Incident Prevention and Handling for Desktops and*
1442   *Laptops*
1443   • *SP 800-88 Rev. 1, Guidelines for Media Sanitization*
1444   • *SP 800-92, Guide to Computer Security Log Management*
1445   • *SP 800-100, Information Security Handbook: A Guide for Managers*
1446   • *SP 800-113, Guide to SSL VPNs*
1447   • *SP 800-114 Rev. 1, User's Guide to Telework and Bring Your Own Device (BYOD)*
1448   *Security*
1449   • *Draft SP 800-121 Rev. 2, Guide to Bluetooth Security*
1450   • *SP 800-128, Guide for Security-Focused Configuration Management of Information*
1451   *Systems*
1452   • *SP 800-137, Information Security Continuous Monitoring (ISCM) for Federal*
1453   *Information Systems and Organizations*
1454   • *SP 800-147, BIOS Protection Guidelines*
1455   • *Draft SP 800-155, BIOS Integrity Measurement Guidelines*
1456

1457

1458   **Appendix C—Acronyms and Abbreviations**

1459   Selected acronyms and abbreviations used in this paper are defined below.

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

1460

1461   **Appendix D—Glossary**

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

1462

1463   **Appendix E—References**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1464