

The attached DRAFT document (provided here for historical purposes), originally posted on July 13, 2017, has been superseded by the following publication:

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

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

Publication Date:         **September 2017**

- Final Publication: <https://doi.org/10.6028/NIST.SP.800-190> (direct link: <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf>).
- Information about the attached Draft publication can be found at: <https://csrc.nist.gov/publications/detail/sp/800-190/archive/2017-07-13>
- Information on other NIST Computer Security Division publications and programs can be found at: <https://csrc.nist.gov/publications>

2

---

---

3 **Application Container Security Guide**

---

4

5

6

Murugiah Souppaya

7

John Morello

8

Karen Scarfone

9

10

11

12

13

14

---

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

---

16

17

18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38

**Draft (2<sup>nd</sup>) NIST Special Publication 800-190**

# **Application Container Security Guide**

Murugiah Souppaya  
*Computer Security Division  
Information Technology Laboratory*

John Morello  
*Twistlock  
Baton Rouge, Louisiana*

Karen Scarfone  
*Scarfone Cybersecurity  
Clifton, Virginia*

July 2017



39  
40  
41  
42  
43  
44  
45  
46

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

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

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.

61 National Institute of Standards and Technology Special Publication 800-190  
62 Natl. Inst. Stand. Technol. Spec. Publ. 800-190, 62 pages (July 2017)  
63 CODEN: NSPUE2

64 Certain commercial entities, equipment, or materials may be identified in this document in order to describe an  
65 experimental procedure or concept adequately. Such identification is not intended to imply recommendation or  
66 endorsement by NIST, nor is it intended to imply that the entities, materials, or equipment are necessarily the best  
67 available for the purpose.

68 There may be references in this publication to other publications currently under development by NIST in  
69 accordance with its assigned statutory responsibilities. The information in this publication, including concepts and  
70 methodologies, may be used by federal agencies even before the completion of such companion publications. Thus,  
71 until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain  
72 operative. For planning and transition purposes, federal agencies may wish to closely follow the development of  
73 these new publications by NIST.

74 Organizations are encouraged to review all draft publications during public comment periods and provide feedback  
75 to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at  
76 <http://csrc.nist.gov/publications>.

77

**Public comment period: July 13, 2017 through August 11, 2017**

78

National Institute of Standards and Technology

79

Attn: Computer Security Division, Information Technology Laboratory

80

100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930

81

Email: [800-190comments@nist.gov](mailto:800-190comments@nist.gov)

82

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

84

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

109 The authors wish to thank their colleagues who have reviewed drafts of this document and  
110 contributed to its technical content during its development, in particular Raghuram Yeluri from  
111 Intel Corporation, Paul Cichonski from Cisco Systems, Inc., and Michael Bartock and Jeffrey  
112 Cichonski from NIST. The authors also acknowledge the organizations that provided feedback  
113 during the public comment period, including Docker, Motorola Solutions, United States  
114 Citizenship and Immigration Services (USCIS), and the US Army.

115

116

## Audience

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

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

126

127

## Trademark Information

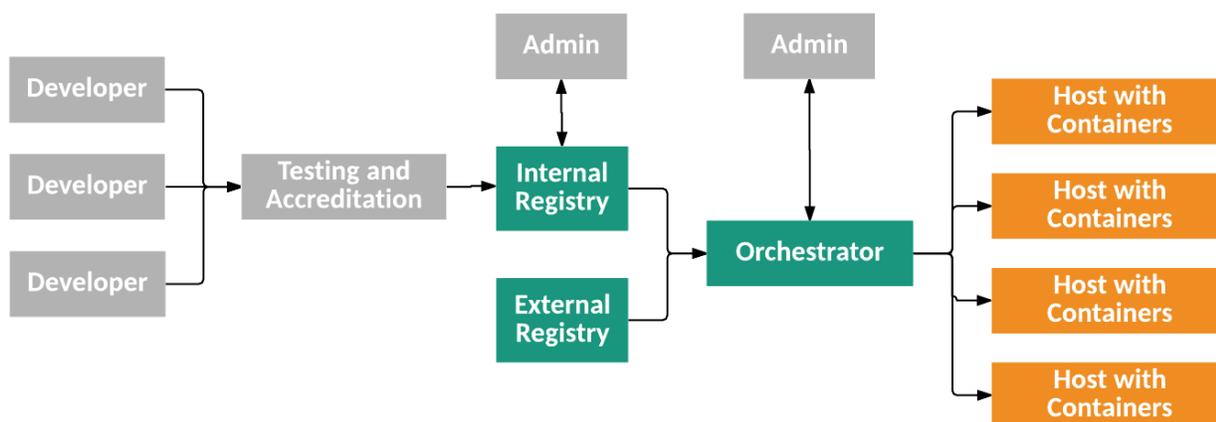
128 All registered trademarks or trademarks belong to their respective organizations.

129

## 130 Executive Summary

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

138 The purpose of the document is to explain the security concerns associated with container  
 139 technologies and make practical recommendations for addressing those concerns when planning  
 140 for, implementing, and maintaining containers. Many of the recommendations are specific to a  
 141 particular component or tier within the container technology architecture, which is depicted in  
 142 Figure 1.



143 **Figure 1: Container Technology Architecture Tiers and Components**

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

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

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

155 **Use container-specific host OSs instead of general-purpose ones to reduce attack surfaces.**

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

164 **Only run containers with the same purpose, sensitivity, and threat posture on a single host**  
165 **OS kernel for additional defense in depth.**

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

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

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

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

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

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

191 Security should extend across all tiers of the container technology. The current way of  
192 establishing trusted computing for all tiers is to use a hardware root of trust. Within this trust is

193 stored measurements of the host's firmware, software, and configuration data. Validating the  
194 current measurements against the stored measurements before booting the host provides  
195 assurance that the host can be trusted. The chain of trust rooted in hardware can be extended to  
196 the OS kernel and the OS components to enable cryptographic verification of boot mechanisms,  
197 system images, container runtimes, and container images. Trusted computing provides the most  
198 secure way to build, run, orchestrate, and manage containers.

199

200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231

**Table of Contents**

**Executive Summary ..... iv**

**1 Introduction ..... 1**

    1.1 Purpose and Scope ..... 1

    1.2 Document Structure ..... 1

**2 Introduction to Application Containers..... 3**

    2.1 Basic Concepts for Application Virtualization and Containers..... 3

    2.2 Containers and the Host Operating System..... 5

    2.3 Container Technology Architecture..... 7

        2.3.1 Image Creation, Testing, and Accreditation..... 8

        2.3.2 Image Storage and Retrieval..... 9

        2.3.3 Container Deployment and Management..... 10

    2.4 Container Uses ..... 11

**3 Major Risks for Core Components of Container Technologies ..... 13**

    3.1 Image Risks ..... 13

        3.1.1 Image vulnerabilities..... 13

        3.1.2 Image configuration ..... 13

        3.1.3 Embedded malware..... 14

        3.1.4 Embedded secrets..... 14

        3.1.5 Image trust ..... 14

    3.2 Registry Risks..... 14

        3.2.1 Insecure connections to registries ..... 14

        3.2.2 Stale images in registries ..... 14

        3.2.3 Insufficient authentication and authorization restrictions ..... 14

    3.3 Orchestrator Risks ..... 15

        3.3.1 Unbounded administrative access..... 15

        3.3.2 Unauthorized access ..... 15

        3.3.3 Poorly separated inter-container network traffic ..... 15

        3.3.4 Mixing of workload sensitivity levels ..... 16

        3.3.5 Orchestrator node trust..... 16

    3.4 Container Risks ..... 16

232 3.4.1 Vulnerabilities within the runtime software..... 16

233 3.4.2 Unbounded network access from containers..... 16

234 3.4.3 Insecure container runtime configurations..... 17

235 3.4.4 Application vulnerabilities ..... 17

236 3.5 Host OS Risks ..... 17

237 3.5.1 Large attack surface ..... 17

238 3.5.2 Shared kernel ..... 17

239 3.5.3 Host OS component vulnerabilities ..... 18

240 3.5.4 Improper user access rights ..... 18

241 3.5.5 Host OS file system tampering ..... 18

242 **4 Countermeasures for Major Risks..... 19**

243 4.1 Image Countermeasures ..... 19

244 4.1.1 Image vulnerabilities..... 19

245 4.1.2 Image configuration ..... 19

246 4.1.3 Embedded malware..... 20

247 4.1.4 Embedded secrets..... 20

248 4.1.5 Image trust ..... 20

249 4.2 Registry Countermeasures ..... 21

250 4.2.1 Insecure connections to registries..... 21

251 4.2.2 Stale images in registries ..... 21

252 4.2.3 Insufficient authentication and authorization restrictions ..... 21

253 4.3 Orchestrator Countermeasures ..... 22

254 4.3.1 Unbounded administrative access..... 22

255 4.3.2 Unauthorized access ..... 22

256 4.3.3 Poorly separated inter-container network traffic ..... 22

257 4.3.4 Mixing of workload sensitivity levels ..... 22

258 4.3.5 Orchestrator node trust..... 23

259 4.4 Container Countermeasures ..... 24

260 4.4.1 Vulnerabilities within the runtime software..... 24

261 4.4.2 Unbounded network access from containers..... 24

262 4.4.3 Insecure container runtime configurations..... 24

263 4.4.4 Application vulnerabilities ..... 25

264 4.5 Host OS Countermeasures ..... 26

265 4.5.1 Large attack surface ..... 26

266 4.5.2 Shared kernel ..... 26

267 4.5.3 Host OS component vulnerabilities ..... 26

268 4.5.4 Improper user access rights ..... 26

269 4.5.5 Host file system tampering ..... 27

270 4.6 Hardware Countermeasures ..... 27

271 **5 Container Threat Scenario Examples..... 29**

272 5.1 Exploit of a Vulnerability within an Image..... 29

273 5.2 Exploit of the Container Runtime ..... 29

274 5.3 Running a Poisoned Image..... 29

275 **6 Container Technology Life Cycle Security Considerations ..... 31**

276 6.1 Initiation Phase ..... 31

277 6.2 Planning and Design Phase..... 31

278 6.3 Implementation Phase ..... 32

279 6.4 Operations and Maintenance Phase..... 33

280 6.5 Disposition Phase ..... 34

281 **7 Conclusion ..... 35**

282 **List of Appendices**

283

284 **Appendix A— NIST Resources for Securing Non-Core Components ..... 37**

285 **Appendix B— NIST SP 800-53 and NIST Cybersecurity Framework Security**

286 **Controls Related to Container Technologies ..... 38**

287 **Appendix C— Acronyms and Abbreviations ..... 45**

288 **Appendix D— Glossary ..... 47**

289 **Appendix E— References..... 49**

290

291 **List of Tables and Figures**

292 Figure 1: Container Technology Architecture Tiers and Components .....iv

293 Figure 2: Virtual Machine and Container Deployments ..... 5

294 Figure 3: Container Technology Architecture Tiers, Components, and Lifecycle Phases8

295 Table 1: NIST Resources for Securing Non-Core Components ..... 37

296 Table 2: Security Controls from NIST SP 800-53 for Container Technology Security... 38  
297 Table 3: NIST SP 800-53 Controls Supported by Container Technologies ..... 42  
298 Table 4: NIST Cybersecurity Framework Subcategories Supported by Container  
299 Technologies..... 42  
300

## 301 **1 Introduction**

### 302 **1.1 Purpose and Scope**

303 The purpose of the document is to explain the security concerns associated with application  
304 container technologies and make practical recommendations for addressing those concerns when  
305 planning for, implementing, and maintaining containers. Some aspects of containers may vary  
306 among technologies, but the recommendations in this document are intended to apply to most or  
307 all application container technologies.

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

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

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

- 317 • The layers under application container technologies, including hardware, hypervisors,  
318 and operating systems;
- 319 • The administrative tools that use the applications within the containers; and
- 320 • The administrator endpoints used to manage the applications within the containers and  
321 the containers themselves.

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

### 326 **1.2 Document Structure**

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

- 328 • Section 2 introduces containers, including their technical capabilities, technology  
329 architectures, and uses.
- 330 • Section 3 explains the major risks for the core components of application container  
331 technologies.
- 332 • Section 4 recommends countermeasures for the risks identified in Section 3.
- 333 • Section 5 defines threat scenario examples for containers.
- 334 • Section 6 presents actionable information for planning, implementing, operating, and  
335 maintaining container technologies.
- 336 • Section 7 provides the conclusion for the document.

- 337
- 338
- 339
- 340
- 341
- 342
- 343
- 344
- Appendix A lists NIST resources for securing non-core components of container technologies.
  - Appendix B lists the NIST Special Publication 800-53 security controls and NIST Cybersecurity Framework subcategories that are most pertinent to application container technologies, explaining the relevancy of each.
  - Appendix C provides an acronym and abbreviation list for the document.
  - Appendix D presents a glossary of selected terms from the document.
  - Appendix E contains a list of references for the document.

345

## 2 Introduction to Application Containers

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

### 2.1 Basic Concepts for Application Virtualization and Containers

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

In hardware virtualization, each OS instance interacts with virtualized hardware. Another form of virtualization known as *operating system virtualization* has a similar concept; it provides multiple virtualized OSs above a single actual OS kernel. This approach is often called an *OS container*, and various implementation of OS containers have existed since the early 2000s, starting with Solaris Zone and FreeBSD jails.<sup>1</sup> Support initially became available in Linux in 2008 with the Linux Container (LXC) technology built into nearly all modern distributions. OS containers are different from the application containers that are the topic of this guide because OS containers are designed to provide an environment that behaves much like a normal OS in which multiple apps and services may co-exist.

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

The key difference between OS virtualization and application virtualization is that with app virtualization, each virtual instance typically runs only a single application. Today’s application virtualization technologies are primarily focused on providing a portable, reusable, and automatable way to package and run apps. The terms *application container* or simply *container* are frequently used to refer to these technologies. The term is meant as an analogy to shipping containers, which provide a standardized way of grouping disparate contents together while isolating them from each other.

---

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

382 Unlike traditional application architectures, which divide an application into a few tiers and have  
383 a component for each tier, container architectures often have an app divided into many more  
384 components, each with a single well-defined function. Each app component runs in a separate  
385 container. In application container technologies, sets of containers that work together to compose  
386 an application are referred to as *microservices*. With this approach, app deployment is more  
387 flexible and scalable. Development is also simpler because functionality is more self-contained.  
388 However, there are many more objects to manage and secure, which may cause problems for app  
389 management and security tools and processes.

390 Most application container technologies implement the concept of immutability. In other words,  
391 the containers themselves should be operated as stateless entities that are deployed but not  
392 changed.<sup>2</sup> When a running container needs to be upgraded or have its contents changed, it is  
393 simply destroyed and replaced with a new container that has the updates. This enables  
394 developers and support engineers to make and push changes to applications at a much faster  
395 pace. Organizations may go from deploying a new version of their app every quarter, to  
396 deploying new components weekly or daily. Immutability is a fundamental operational  
397 difference between containers and hardware virtualization. Traditional VMs are typically run as  
398 stateful entities that are deployed, reconfigured, and upgraded throughout their life. Legacy  
399 security tools and processes often assume largely static operations and may need to be adjusted  
400 to adapt to the rate of change in containerized environments.

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

408 Modern container technologies have largely emerged along with the adoption of development  
409 and operations (DevOps) practices that seek to increase the integration between building and  
410 running apps, emphasizing close coordination between development and operational teams.<sup>3</sup> The  
411 portable and declarative nature of containers is particularly well suited to these practices because  
412 they allow an organization to have great consistency between development, test, and production  
413 environments. Organizations often utilize continuous integration processes to put their apps into  
414 containers directly in the build process itself, such that from the very beginning of the app's  
415 lifecycle, there is guaranteed consistency of its runtime environment. Container images are  
416 typically designed to be portable across machines and environments, so that an image created in  
417 a development lab can be easily moved to a test lab for evaluation, then copied into a production  
418 environment to run without needing to make any modifications. The downside of this is that the  
419 security tools and processes used to protect containers should not make assumptions about  
420 specific cloud providers, host OSs, network topologies, or other aspects of the container runtime

---

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

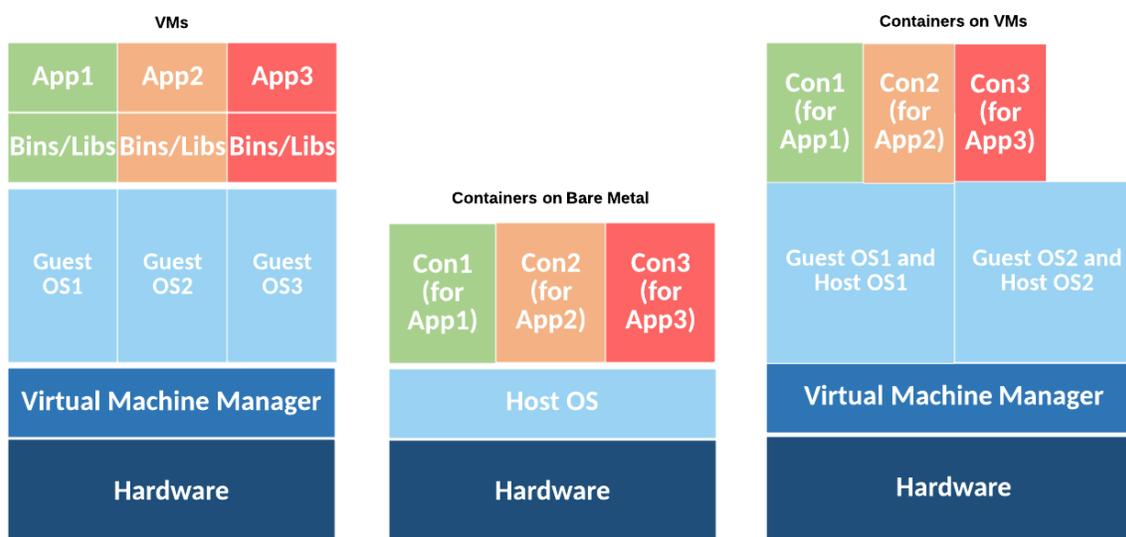
<sup>3</sup> This document refers to tasks performed by DevOps personas. The references to these personas are focused on the types of job tasks being performed, not on strict titles or team organizational structures.

421 environment which may frequently change. Even more critically, security should be consistent  
 422 across all these environments and throughout the application lifecycle from development to test  
 423 to production.

424 Recently, projects such as Docker [2] and rkt [3] have provided additional functionality designed  
 425 to make OS component isolation features easier to use and scale. Container technologies are also  
 426 available on the Windows platform beginning with Windows Server 2016. The fundamental  
 427 architecture of all these implementations is consistent enough so that this document can discuss  
 428 containers in detail while remaining implementation agnostic.

## 429 2.2 Containers and the Host Operating System

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



435 **Figure 2: Virtual Machine and Container Deployments**

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

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

447 There are two general categories of host OSs used for running containers. *General-purpose OSs*  
448 like Red Hat Enterprise Linux, Ubuntu, and Windows Server can be used for running many  
449 kinds of apps and can have container-specific functionality added to them. *Container-specific*  
450 *OSs*, like CoreOS Container Linux [4], Project Atomic [5], and Google Container-Optimized OS  
451 [6] are minimalistic OSs explicitly designed to only run containers. They typically do not come  
452 with package managers, they have only a subset of the core administration tools, and they  
453 actively discourage running applications outside containers. Often, a container-specific OS uses  
454 a read-only file system design to reduce the likelihood of an attacker being able to persist data  
455 within it, and it also utilizes a simplified upgrade process since there is little concern around  
456 application compatibility.

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

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

- 469 • **Namespace isolation**, which limits which resources a container may interact with. This  
470 includes file systems, network interfaces, interprocess communications, host names, user  
471 information, and processes. Namespace isolation ensures that applications and processes  
472 inside a container only see the physical and virtual resources allocated to that container.  
473 For example, if you run ‘ps -A’ inside a container running Apache on a host with many  
474 other containers running other apps, you would only see httpd listed in the results.  
475 Namespace isolation provides each container with its own networking stack, including  
476 unique interfaces and IP addresses. Containers on Linux use technologies like masked  
477 process identities to achieve namespace isolation, whereas on Windows, object  
478 namespaces are used.
- 479 • **Resource allocation**, which limits how much of a host’s resources a given container can  
480 consume. For example, if your host OS has 10 gigabytes (GB) of total memory, you may  
481 wish to allocate 1 GB each to nine separate containers. No container should be able to  
482 interfere with the operations of another container, so resource allocation ensures that each  
483 container can only utilize the amount of resources assigned to it. On Linux, this is  
484 accomplished primarily with control groups (cgroups)<sup>4</sup>, whereas on Windows job objects  
485 serve a similar purpose.

---

<sup>4</sup> 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.

486       • **Filesystem virtualization**, which allows multiple containers to share the same physical  
487 storage without the ability to access or alter the storage of other containers. While  
488 arguably similar to namespace isolation, filesystem virtualization is called out separately  
489 because it also often involves optimizations to ensure that containers are efficiently using  
490 the host's storage through techniques like copy on write. For example, if multiple  
491 containers using the same image are running Apache on a single host, filesystem  
492 virtualization ensures that there is only one copy of the httpd binary stored on disk. If one  
493 of the containers modifies files within itself, only the specifically changed bits will be  
494 written to disk, and those changes will only be visible to the container that executed  
495 them. On Linux, these capabilities are provided by technologies like the Advanced Multi-  
496 Layered Unification Filesystem (AUFS), whereas on Windows they are an extension of  
497 the NT File System (NTFS).

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

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

510 Although containers are sometimes thought of as the next phase of virtualization, surpassing  
511 hardware virtualization, the reality for most organizations is less about revolution than evolution.  
512 Containers and hardware virtualization not only can, but very frequently do, coexist well and  
513 actually enhance each other's capabilities. VMs provide many benefits, such as strong isolation,  
514 OS automation, and a wide and deep ecosystem of solutions. Organizations do not need to make  
515 a false choice between containers and VMs. Instead, organizations can continue to use VMs to  
516 deploy, partition, and manage their hardware, while using containers to package their apps and  
517 utilize each VM more efficiently.

## 518 **2.3 Container Technology Architecture**

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

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

526 It also depicts administrator systems for the internal registry and the orchestrator.

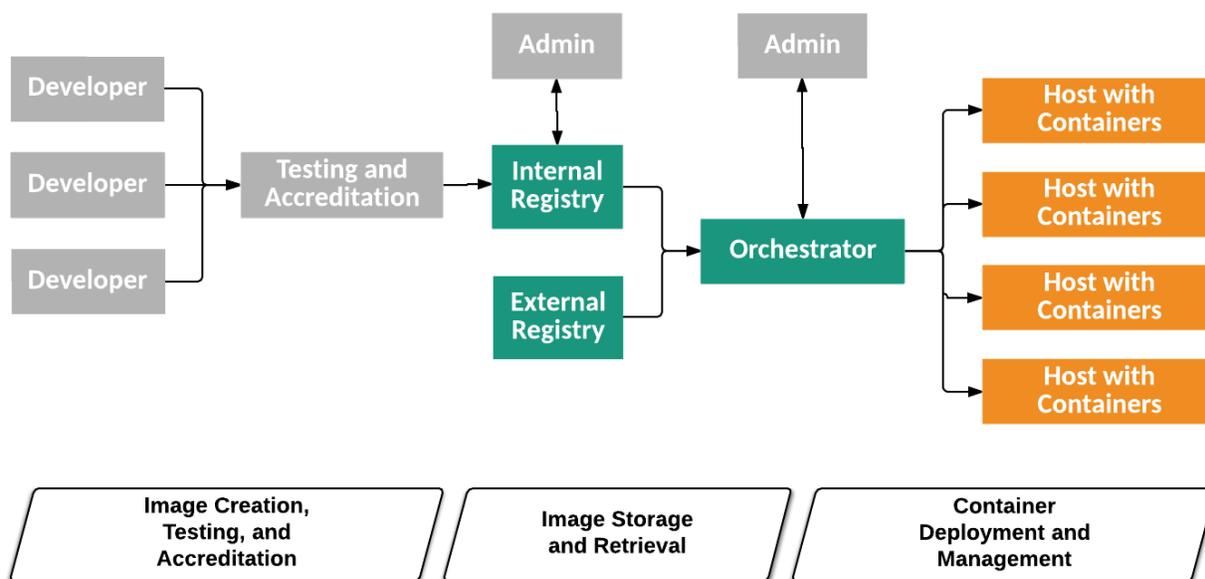


Figure 3: Container Technology Architecture Tiers, Components, and Lifecycle Phases

527 The systems in gray (developer systems, testing and accreditation system, and administrator  
 528 systems) are outside the scope of the container technology architecture, but they do have  
 529 important interactions with it. The systems in green (internal registry, external registry, and  
 530 orchestrator) are core components of a container technology architecture. Finally, the systems in  
 531 orange (hosts with containers) are where the containers are used.

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

535 Because organizations are typically building and deploying many different apps at once, these  
 536 lifecycle phases often occur concurrently within the same organization and should not be seen as  
 537 progressive stages of maturity. Instead, think of them as cycles in an engine that is continuously  
 538 running. In this metaphor, each app is a cylinder within the engine, and different apps may be at  
 539 different phases of this lifecycle at the same time.

### 540 2.3.1 Image Creation, Testing, and Accreditation

541 In the first phase of the container lifecycle, an app's components are built and placed into an  
 542 image. An *image* is a package that contains all the files required to run a container. For example,

543 an image to run Apache would include the httpd binary, along with associated libraries and  
544 configuration files. An image should only include the executables and libraries required by the  
545 app itself; all other OS functionality is provided by the OS kernel within the underlying host OS.  
546 Images often use techniques like layering and copy on write (in which shared master images are  
547 read only and changes are recorded to separate files) to minimize their size on disk and improve  
548 operational efficiency.

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

558 Image creation is mostly driven by developers who are working on creating or updating apps and  
559 packaging them. Image creation typically uses build management and automation tools, such as  
560 Jenkins [8] and TeamCity [9], to assist with what is called the “continuous integration” process.  
561 These tools take the various libraries, binaries, and other components of an application, perform  
562 testing on them, and then assemble images out of them based on the developer-created manifest  
563 that describes how to build an image for the app.

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

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

### 573 **2.3.2 Image Storage and Retrieval**

574 Images are typically stored in central locations to make it easy to share, find, and reuse them  
575 across hosts. *Registries* are services that allow developers to easily store images as they are  
576 created, tag and catalog images for identification and version control to aid in discovery and  
577 reuse, and find and download images that others have created. Registries may be self-hosted or  
578 consumed as a service. Examples of registries include Amazon EC2 Container Registry [10],  
579 Docker Hub [11], Docker Trusted Registry [12], and Quay Container Registry [13].

580 Registries provide APIs that enable automating common image-related tasks. For example,  
581 organizations may have triggers in the image creation phase that automatically push images to a  
582 registry once tests pass. The registry may have further triggers that automate the deployment of

583 new images once they have been added. This automation enables faster iteration on projects with  
584 more consistent results.

585 Once stored in a registry, images can be easily pulled and then run by DevOps personas across  
586 any environment in which they run containers. This is another example of the portability benefits  
587 of containers; image creation may occur in a public cloud provider, which pushes an image to a  
588 registry hosted in a private cloud, which is then used to distribute images for running the app in a  
589 third location.

### 590 **2.3.3 Container Deployment and Management**

591 Tools known as *orchestrators* enable DevOps personas or automation working on their behalf to  
592 pull images from registries, deploy those images into containers, and manage the running  
593 containers. This deployment process is what actually results in a usable version of the app,  
594 running and ready to respond to requests. When an image is deployed into a container, the image  
595 itself is not changed, but instead a copy of it is placed within the container and transitioned from  
596 being a dormant set of app code to a running instance of the app. Examples of orchestrators are  
597 Kubernetes [14], Mesos [15], and Docker Swarm [16].

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

602 The abstraction provided by an orchestrator allows a DevOps persona to simply specify how  
603 many containers need to be running a given image and what resources, such as memory,  
604 processing, and disk need to be allocated to each. The orchestrator knows the state of each host  
605 within the cluster, including what resources are available for each host, and determines which  
606 containers will run on which hosts. The orchestrator then pulls the required images from the  
607 registry and runs them as containers with the designated resources.

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

615 When apps in containers need to be updated, the existing containers are not changed, but rather  
616 they are destroyed and new containers created from updated images. This is a key operational  
617 difference with containers in that the baseline software from the initial deployment should not  
618 change over time, and the atomicity of updates is the entire image at once. This approach has  
619 significant potential security benefits because it enables organizations to build, test, validate, and  
620 deploy exactly the same software in exactly the same configuration in each phase. As updates are  
621 made to apps, organizations can ensure that the most recent versions are used, typically by  
622 leveraging orchestrators. Orchestrators are usually configured to pull the most up-to-date version

623 of an image from the registry so that the app is always up-to-date. This “continuous delivery”  
624 automation enables developers to simply build a new version of the image for their app, test the  
625 image, push it to the registry, and then rely on the automation tools to deploy it to the target  
626 environment.

627 This means that all vulnerability management, including patches and configuration settings, must  
628 be taken care of by the developer before building a new image version. With containers,  
629 developers are largely responsible for the security of apps and images instead of the operations  
630 team. This change in responsibilities often requires much greater coordination and cooperation  
631 among personnel than was previously necessary.

632 Container management includes security management and monitoring. Unfortunately, security  
633 controls designed for non-container environments are often not well suited for use with  
634 containers. For example, consider security controls that take IP addresses into account. This  
635 works well for VMs and bare metal servers with static IP addresses that remain the same for  
636 months or years. Conversely, containers are typically allocated IP addresses by orchestrators, and  
637 because containers are created and destroyed much more frequently than VMs, these IP  
638 addresses change frequently over time as well. This makes it difficult or impossible to protect  
639 containers using security techniques that rely on static IP addresses, such as firewall rulesets  
640 filtering traffic based on IP address. Additionally, a container network can include  
641 communications between containers on the same node, across different nodes, and even across  
642 clouds.

## 643 **2.4 Container Uses**

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

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

661 Containers provide additional benefits; for example, they can increase the effectiveness of build  
662 pipelines due to the immutable nature of container images. Containers shift the time and location  
663 of production code installation. In non-container systems, application installation happens in

664 production (i.e., at server runtime), typically by running hand-crafted scripts that manage  
665 installation of application code (e.g., programming language runtime, dependent third-party  
666 libraries, init scripts, and OS tools) on servers. This means that any tests running in a pre-  
667 production build pipeline (and on developers' workstations) are not testing the actual production  
668 artifact, but a best-guess approximation contained in the build system. This approximation of  
669 production tends to drift from production over time, especially if the teams managing production  
670 and the build system are different. This scenario is the embodiment of the "it works on my  
671 machine" problem.

672 With container technologies, the build system installs the application within the image it creates  
673 (i.e., at compile-time). The image is an immutable snapshot of all userspace requirements of the  
674 application (i.e., programming language runtime, dependent third-party libraries, init scripts, and  
675 OS tools). In production the container image constructed by the build system is simply  
676 downloaded and run. This solves the "works on my machine" problem since the developer, build  
677 system, and production all run the same immutable artifact.

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

688

## 689 **3 Major Risks for Core Components of Container Technologies**

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

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

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

### 706 **3.1 Image Risks**

#### 707 **3.1.1 Image vulnerabilities**

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

713 Unlike traditional operational patterns in which deployed software is updated ‘in the field’ on the  
714 hosts it runs on, with containers these updates must be made upstream in the images themselves,  
715 which are then redeployed. Thus, a common risk in containerized environments is deployed  
716 containers having vulnerabilities because the version of the image used to generate the containers  
717 has vulnerabilities.

#### 718 **3.1.2 Image configuration**

719 In addition to software defects, images may also have configuration defects. For example, an  
720 image may not have a user defined and thus not take advantage of the defense in depth provided  
721 by user namespaces. As another example, an image may include an SSH daemon, which exposes  
722 the container to unnecessary network risk. Much like in a traditional server or VM, where a poor  
723 configuration can still expose a fully up-to-date system to attack, so too can a poorly configured  
724 image increase risk even if all the included components are up-to-date.

### 725 **3.1.3 Embedded malware**

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

### 731 **3.1.4 Embedded secrets**

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

### 738 **3.1.5 Image trust**

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

## 746 **3.2 Registry Risks**

### 747 **3.2.1 Insecure connections to registries**

748 Images often contain sensitive components like an organization's proprietary software and  
749 embedded secrets. If connections to registries are performed over insecure channels, the contents  
750 of images are subject to the same confidentiality risks as any other data transmitted in the clear.  
751 There is also an increased risk of man-in-the-middle attacks that could intercept network traffic  
752 intended for registries and steal developer or administrator credentials, provide fraudulent or  
753 outdated images to orchestrators, etc.

### 754 **3.2.2 Stale images in registries**

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

### 759 **3.2.3 Insufficient authentication and authorization restrictions**

760 Because registries may contain images used to run sensitive or proprietary applications and to  
761 access sensitive data, insufficient authentication and authorization requirements can lead to

762 intellectual property loss and expose significant technical details about an application to an  
763 attacker. Even more critically, because registries are typically trusted as a source of valid,  
764 approved software, compromise of a registry can potentially lead to compromise of downstream  
765 containers and hosts.

### 766 **3.3 Orchestrator Risks**

#### 767 **3.3.1 Unbounded administrative access**

768 Historically, many orchestrators were designed with the assumption that all users interacting  
769 with them would be administrators and those administrators should have environment-wide  
770 control. However, in many cases, a single orchestrator may run many different apps, each  
771 managed by different teams, and with different sensitivity levels. If the access provided to users  
772 and groups is not scoped to their specific needs, a malicious or careless user could affect or  
773 subvert the operation of other containers managed by the orchestrator.

#### 774 **3.3.2 Unauthorized access**

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

780 Another concern regarding unauthorized access is the misuse of credentials, such as an attacker  
781 getting access to a password through social engineering or other means, then reusing that  
782 password to access the orchestrator.

#### 783 **3.3.3 Poorly separated inter-container network traffic**

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

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

### 797 **3.3.4 Mixing of workload sensitivity levels**

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

### 805 **3.3.5 Orchestrator node trust**

806 Many orchestrators exist and each supports a wide variety of configurations. Weak orchestrator  
807 configurations can expose the orchestrator and all other container technology components to  
808 increased risk. Examples of possible consequences include:

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

## 814 **3.4 Container Risks**

### 815 **3.4.1 Vulnerabilities within the runtime software**

816 While relatively uncommon, vulnerabilities within the runtime software are particularly  
817 dangerous if they allow ‘container escape’ scenarios in which malicious software can attack  
818 resources in other containers and the host OS itself. An attacker may also be able to exploit  
819 vulnerabilities to compromise the runtime software itself, and then alter that software so it allows  
820 the attacker to access other containers, monitor container-to-container communications, etc.

### 821 **3.4.2 Unbounded network access from containers**

822 By default in most container runtimes, individual containers are able to access each other and the  
823 host OS over the network. If a container is compromised and acting maliciously, allowing this  
824 network traffic may expose other resources in the environment to risk. For example, a  
825 compromised container may be used to scan the network it is connected to in order to find other  
826 weaknesses for an attacker to exploit. This risk is related to that from poorly separated virtual  
827 networks, as discussed in Section 3.3.3, but different because it is focused more on flows from  
828 containers to any outbound destination, not on app “cross talk” scenarios.

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

### 835 3.4.3 Insecure container runtime configurations

836 Container runtimes typically expose many configurable options to administrators. Setting them  
837 improperly can lower the relative security of the system. For example, on Linux container hosts,  
838 the set of allowed system calls is often limited by default to only those required for safe  
839 operation of containers. If this list is widened, it may expose containers and the host OS to  
840 increased risk from a compromised container. Similarly, if a container is run in privileged mode,  
841 it has access to all the devices on the host, thus allowing it to essentially act as part of the host  
842 OS and impact all other containers running on it.

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

### 849 3.4.4 Application vulnerabilities

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

## 858 3.5 Host OS Risks

### 859 3.5.1 Large attack surface

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

### 865 3.5.2 Shared kernel

866 Container-specific OSs have a much smaller attack surface than that of general-purpose OSs. For  
867 example, they do not contain libraries and package managers that enable a general-purpose OS to  
868 directly run database and web server apps. However, although containers provide strong  
869 software-level isolation of resources, the use of a shared kernel invariably results in a larger  
870 inter-object attack surface than seen with hypervisors, even for container-specific OSs. In other  
871 words, the level of isolation provided by container runtimes is not as high as that provided by  
872 hypervisors.

**873 3.5.3 Host OS component vulnerabilities**

874 All host OSs, even container-specific ones, provide foundational system components—for  
875 example, the cryptographic libraries used to authenticate remote connections and the kernel  
876 primitives used for general process invocation and management. Like any other software, these  
877 components can have vulnerabilities and, because they exist low in the container technology  
878 architecture, they can impact all the containers and applications that run on these hosts.

**879 3.5.4 Improper user access rights**

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

**886 3.5.5 Host OS file system tampering**

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

891

## 892 **4 Countermeasures for Major Risks**

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

### 894 **4.1 Image Countermeasures**

#### 895 **4.1.1 Image vulnerabilities**

896 There is a need for container technology-specific vulnerability management tools and processes.  
897 Traditional vulnerability management tools make many assumptions about host durability and  
898 app update mechanisms and frequencies that are fundamentally misaligned with a containerized  
899 model. These tools are often unable to detect vulnerabilities within containers, leading to a false  
900 sense of safety.

901 Organizations should use tools that take the pipeline-based build approach and immutable nature  
902 of containers and images into their design to provide more actionable and reliable results. Key  
903 aspects of effective tools and processes include:

- 904 1. Integration with the entire lifecycle of images, from the beginning of the build process, to  
905 whatever registries the organization is using, to runtime.
- 906 2. Visibility into vulnerabilities at all layers of the image, not just the base layer of the  
907 image but also application frameworks and custom software the organization is using.  
908 Visibility should be centralized across the organization and provide flexible reporting and  
909 monitoring views aligned with organizations' business processes.
- 910 3. Policy-driven enforcement; organizations should be able to create "quality gates" at each  
911 stage of the build and deployment process to ensure that only images that meet the  
912 organization's vulnerability and configuration policies are allowed to progress. For  
913 example, organizations should be able to configure a rule in the build process to prevent  
914 the progression of images that include vulnerabilities with Common Vulnerability  
915 Scoring System (CVSS) [18] ratings above a selected threshold.

#### 916 **4.1.2 Image configuration**

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

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

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

#### 937 **4.1.3 Embedded malware**

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

#### 941 **4.1.4 Embedded secrets**

942 Secrets should be stored outside of images and provided dynamically at runtime as needed. Most  
943 orchestrators, such as Docker Swarm and Kubernetes, include secret management natively.  
944 These orchestrators not only provide secure secret storage and ‘just in time’ injection to  
945 containers, but also make it much simpler to integrate secret management into the build and  
946 deployment processes. For example, an organization could use these tools to securely provision  
947 the database connection string into a web app container. The orchestrator would ensure that only  
948 the web app container had access to this secret, that it is not persisted to disk, and that anytime  
949 the web app is deployed, the secret is provisioned into it.

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

954 Regardless of the tool chosen, organizations should ensure that secrets are only provided to the  
955 specific containers that require them, based on a pre-defined and administrator-controlled setting,  
956 and that secrets are always encrypted at rest.

#### 957 **4.1.5 Image trust**

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

961 To mitigate these risks, organizations should take a multilayered approach that includes:

- 962 • Capability to centrally control exactly what images and registries are trusted in their  
963 environment;

- 964 • Discrete identification of each image by cryptographic signature, using a NIST-validated  
965 implementation<sup>5</sup>;
- 966 • Enforcement to ensure that all hosts in the environment only run images from these  
967 approved lists;
- 968 • Validation of image signatures before image execution to ensure images are from trusted  
969 sources and have not been tampered with; and
- 970 • Ongoing monitoring and maintenance of these repositories to ensure images within them  
971 are maintained and updated as vulnerabilities and configuration requirements change.

## 972 **4.2 Registry Countermeasures**

### 973 **4.2.1 Insecure connections to registries**

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

### 978 **4.2.2 Stale images in registries**

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

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

### 993 **4.2.3 Insufficient authentication and authorization restrictions**

994 All access to registries that contain proprietary or sensitive images should require authentication.  
995 Any write access to a registry should require authentication to ensure that only images from  
996 trusted entities can be added to it. In both cases, organizations should consider federating with  
997 existing accounts, such as their own or a cloud provider’s directory service to take advantage of

---

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

998 security controls already in place for those accounts. All write access to registries should be  
999 audited and any read actions for sensitive images should similarly be logged.

### 1000 **4.3 Orchestrator Countermeasures**

#### 1001 **4.3.1 Unbounded administrative access**

1002 Especially because of their wide-ranging span of control, orchestrators should use a least  
1003 privileged access model in which users are only granted ability to perform the specific actions on  
1004 the specific hosts, containers, and images their job role requires. For examples, members of the  
1005 test team should only be given access to the images used in testing and the hosts used for running  
1006 them, and should only be able to manipulate the containers they created. Test team members  
1007 should have limited or no access to containers used in production.

#### 1008 **4.3.2 Unauthorized access**

1009 Access to cluster-wide administrative accounts should be tightly controlled as these accounts  
1010 provide ability to affect all resources in the environment. Organizations should use strong  
1011 authentication methods, such as requiring multifactor authentication instead of just a password.

1012 Organizations should implement single sign-on to existing directory systems where applicable.  
1013 Single sign-on simplifies the orchestrator authentication experience, makes it easier for users to  
1014 use strong authentication credentials, and centralizes auditing of access, making anomaly  
1015 detection more effective.

#### 1016 **4.3.3 Poorly separated inter-container network traffic**

1017 Orchestrators should be configured to separate network traffic into discrete virtual networks by  
1018 sensitivity level. While per-app segmentation is also possible, for most organizations and use  
1019 cases, simply defining networks by sensitivity level provides sufficient mitigation of risk with a  
1020 manageable degree of complexity. For example, public-facing apps can share a virtual network,  
1021 internal apps can use another, and communication between the two should occur through a small  
1022 number of well-defined interfaces.

#### 1023 **4.3.4 Mixing of workload sensitivity levels**

1024 Orchestrators should be configured to isolate deployments to specific sets of hosts by sensitivity  
1025 levels. The particular approach for implementing this varies depending on the orchestrator in use,  
1026 but the general model is to define rules that prevent high sensitivity workloads from being placed  
1027 on the same host as those running lower sensitivity workloads. This can be accomplished  
1028 through the use of host ‘pinning’ within the orchestrator or even simply by having separate,  
1029 individually managed clusters for each sensitivity level.

1030 While most container runtime environments do an effective job of isolating containers from each  
1031 other and from the host OS, in some cases it may be an unnecessary risk to run apps of different  
1032 sensitivity levels together on the same host OS. Segmenting containers by purpose, sensitivity,  
1033 and threat posture provides additional defense in depth. Concepts such as application tiering and  
1034 network and host segmentation should be taken into consideration when planning app

1035 deployments. For example, suppose a host is running containers for both a financial database and  
1036 a public-facing blog. While normally the container runtime will effectively isolate these  
1037 environments from each other, there is also a shared responsibility amongst the DevOps teams  
1038 for each app to operate them securely and eliminate unnecessary risk. If the blog app were to be  
1039 compromised by an attacker, there would be far fewer layers of defense to protect the database if  
1040 the two apps are running on the same host.

1041 Thus, a best practice is to group containers together by relative sensitivity and to ensure that a  
1042 given host kernel only runs containers of a single sensitivity level. This segmentation may be  
1043 provided by using multiple physical servers, but modern hypervisors also provide strong enough  
1044 isolation to effectively mitigate these risks. From the previous example, this may mean that the  
1045 organization has two sensitivity levels for their containers. One is for financial apps and the  
1046 database is included in that group. The other is for web apps and the blog is included in that  
1047 group. The organization would then have two pools of VMs that would each host containers of a  
1048 single severity level. For example, the host called vm-financial may host the containers running  
1049 the financial database as well as the tax reporting software, while a host called vm-web may host  
1050 the blog and the public website.

1051 By segmenting containers in this manner, it will be much more difficult for an attacker who  
1052 compromises one of the segments to expand that compromise to other segments. This approach  
1053 also ensures that any residual data, such as caches or local volumes mounted for temp files, stays  
1054 within its security zone. From the previous example, this zoning would ensure that any financial  
1055 data cached locally and residually after container termination would never be available on a host  
1056 running an app at a lower sensitivity level.

1057 In larger-scale environments with hundreds of hosts and thousands of containers, this  
1058 segmentation must be automated to be practical to operationalize. Fortunately, common  
1059 orchestration platforms typically include some notion of being able to group apps together, and  
1060 container security tools can use attributes like container names and labels to enforce security  
1061 policies across them. In these environments, additional layers of defense in depth beyond simple  
1062 host isolation may also leverage this segmentation. For example, an organization may implement  
1063 separate hosting zones or networks to not only isolate these containers within hypervisors but  
1064 also to isolate their network traffic more discretely such that traffic for apps of one sensitivity  
1065 level is separate from that of other sensitivity levels.

#### 1066 **4.3.5 Orchestrator node trust**

1067 Orchestration platforms should be configured to provide features that create a secure  
1068 environment for all the apps they run. Orchestrators should ensure that nodes are securely  
1069 introduced to the cluster, have a persistent identity throughout their lifecycle, and can also  
1070 provide an accurate inventory of nodes and their connectivity states. Organizations should ensure  
1071 that orchestration platforms are designed specifically to be resilient to compromise of individual  
1072 nodes without compromising the overall security of the cluster. A compromised node must be  
1073 able to be isolated and removed from the cluster without disrupting or degrading overall cluster  
1074 operations. Finally, organizations should choose orchestrators that provide mutually  
1075 authenticated network connections between cluster members and end-to-end encryption of intra-  
1076 cluster traffic. Because of the portability of containers, many deployments may occur across

1077 networks organizations do not directly control, so a secure-by-default posture is particularly  
1078 important for this scenario.

#### 1079 **4.4 Container Countermeasures**

##### 1080 **4.4.1 Vulnerabilities within the runtime software**

1081 The container runtime must be carefully monitored for vulnerabilities and when problems are  
1082 detected, they must be remediated quickly. A vulnerable runtime exposes all containers it  
1083 supports, as well as the host itself, to potentially significant risk. Organizations should use tools  
1084 to look for Common Vulnerabilities and Exposures (CVEs) vulnerabilities in the runtimes  
1085 deployed, to upgrade any instances at risk, and to ensure that orchestrators only allow  
1086 deployments to properly maintained runtimes.

##### 1087 **4.4.2 Unbounded network access from containers**

1088 Organizations should control the egress network traffic sent by containers. At minimum, these  
1089 controls should be in place at network borders, ensuring containers are not able to send traffic  
1090 across networks of differing sensitivity levels, such as from an environment hosting secure data  
1091 to the internet, similar to the patterns used for traditional architectures. However, the virtualized  
1092 networking model of inter-container traffic poses an additional challenge.

1093 Because containers deployed across multiple hosts typically communicate over a virtual,  
1094 encrypted network, traditional network devices are often blind to this traffic. Additionally,  
1095 containers are typically assigned dynamic IP addresses automatically when deployed by  
1096 orchestrators, and these addresses change continuously as the app is scaled and load balanced.  
1097 Thus, ideally, organizations should use a combination of existing network level devices and  
1098 more application-aware network filtering. App-aware tools should be able to not just see the  
1099 inter-container traffic, but also to dynamically generate the rules used to filter this traffic based  
1100 on the specific characteristics of the apps running in the containers. This dynamic rule  
1101 management is critical due to the scale and rate of change of containerized apps, as well as their  
1102 ephemeral networking topology.

1103 Specifically, app-aware tools should provide the following capabilities:

- 1104 • Automated determination of proper container networking surfaces, including both  
1105 inbound ports and process-port bindings;
- 1106 • Detection of traffic flows both between containers and other network entities, over both  
1107 'on the wire' traffic and encapsulated traffic; and
- 1108 • Detection of network anomalies, such as unexpected traffic flows within the  
1109 organization's network, port scanning, or outbound access to potentially dangerous  
1110 destinations.

##### 1111 **4.4.3 Insecure container runtime configurations**

1112 Organizations should automate compliance with container runtime configuration standards.  
1113 Documented technical implementation guidance, such as the Center for Internet Security Docker  
1114 Benchmark [19], provides details on options and recommended settings, but operationalizing this

1115 guidance depends on automation. Organizations can use a variety of tools to “scan” and assess  
1116 their compliance at a point in time, but such approaches do not scale. Instead, organizations  
1117 should use tools or processes that continuously assess configuration settings across the  
1118 environment and actively enforce them.

1119 Additionally, mandatory access control (MAC) technologies like SELinux [20] and AppArmor  
1120 [21] provide enhanced control and isolation for containers running Linux OSs. For example,  
1121 these technologies can be used to provide additional segmentation and assurance that containers  
1122 should only be able to access specific file paths, processes, and network sockets, further  
1123 constraining the ability of even a compromised container to impact the host or other containers.  
1124 MAC technologies provide protection at the host OS layer, ensuring that only specific files,  
1125 paths, and processes are accessible to containerized apps. Organizations are encouraged to use  
1126 the MAC technologies provided by their host OSs in all container deployments.

1127 Secure computing (seccomp)<sup>6</sup> profiles are another mechanism that can be used to constrain the  
1128 system-level capabilities containers are allocated at runtime. Common container runtimes like  
1129 Docker include default seccomp profiles that drop system calls that are unsafe and typically  
1130 unnecessary for container operation. Additionally, custom profiles can be created and passed to  
1131 container runtimes to further limit their capabilities. At a minimum, organizations should ensure  
1132 that containers are run with the default profiles provided by their runtime and should consider  
1133 using additional profiles for high-risk apps.

#### 1134 **4.4.4 Application vulnerabilities**

1135 Existing host-based intrusion detection processes and tools are often unable to detect and prevent  
1136 attacks within containers due to the differing technical architecture and operational practices  
1137 previously discussed. Organizations should implement additional tools that are container aware  
1138 and designed to operate at the scale and change rate typically seen with containers. These tools  
1139 should be able to automatically profile containerized apps using behavioral learning and build  
1140 security profiles for them to minimize human interaction. These profiles should then be able to  
1141 detect anomalies at runtime, including events such as:

- 1142 • Invalid or unexpected process execution,
- 1143 • Invalid or unexpected system calls,
- 1144 • Changes to protected configuration files and binaries,
- 1145 • Writes to unexpected locations and file types,
- 1146 • Creation of unexpected network listeners,
- 1147 • Traffic sent to unexpected network destinations, and
- 1148 • Malware storage or execution.

1149 Containers should also be run in a read-only mode, in which changes are not allowed to their root  
1150 filesystems. This approach isolates writes to specifically defined directories, which can then be  
1151 more easily monitored by the aforementioned tools. Furthermore, using read-only filesystems

---

<sup>6</sup> For more information on seccomp, see [https://www.kernel.org/doc/Documentation/prctl/seccomp\\_filter.txt](https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt).

1152 makes the containers more resilient to compromise since any tampering is isolated to these  
1153 specific locations and can be easily separated from the rest of the app.

## 1154 **4.5 Host OS Countermeasures**

### 1155 **4.5.1 Large attack surface**

1156 For organizations using container-specific OSs, the threats are typically more minimal to start  
1157 with since the OSs are specifically designed to host containers and have other services and  
1158 functionality disabled. Further, because these optimized OSs are designed specifically for  
1159 hosting containers, they typically feature read-only file systems and employ other hardening  
1160 practices by default. Whenever possible, organizations should use these minimalistic OSs to  
1161 reduce their attack surfaces and mitigate the typical risks and hardening activities associated with  
1162 general-purpose OSs.

### 1163 **4.5.2 Shared kernel**

1164 In addition to grouping container workloads onto hosts by sensitivity level, organizations should  
1165 not mix containerized and non-containerized workloads on the same host instance. For example,  
1166 if a host is running a web server container, it should not also run a web server (or any other app)  
1167 as a regularly installed component directly within the host OS. Keeping containerized workloads  
1168 isolated to container-specific hosts makes it simpler and safer to apply countermeasures and  
1169 defenses that are optimized for protecting containers.

### 1170 **4.5.3 Host OS component vulnerabilities**

1171 Organizations should implement management practices and tools to validate the versioning of  
1172 components provided for base OS management and functionality. Even though container-  
1173 specific OSs have a much more minimal set of components than general-purpose OSs, they still  
1174 do have vulnerabilities and still require remediation. Organizations should use tools provided by  
1175 the OS vendor or other trusted organizations to regularly check for and apply updates to all  
1176 software components used within the OS. The OS should be kept up to date not only with  
1177 security updates, but also the latest component updates recommended by the vendor. This is  
1178 particularly important for the kernel and container runtime components as newer releases of  
1179 these components often add additional security protections and capabilities beyond simply  
1180 correcting vulnerabilities.

1181 Host OSs should be operated in an immutable manner with no data or state stored uniquely and  
1182 persistently on the host and no application-level dependencies provided by the host. Instead, all  
1183 application components and dependencies should be packaged and deployed in containers. This  
1184 enables the host to be operated in a nearly stateless manner with a greatly reduced attack surface  
1185 and a more trustworthy way to identify anomalies and configuration drift.

### 1186 **4.5.4 Improper user access rights**

1187 Though most container deployments rely on orchestrators to distribute jobs across hosts,  
1188 organizations should still ensure that all authentication to the OS is audited, login anomalies are  
1189 monitored, and any escalation to performed privileged operations is logged. This makes it

1190 possible to identify anomalous access patterns such as an individual logging on to a host directly  
1191 and running privileged commands to manipulate containers.

#### 1192 **4.5.5 Host file system tampering**

1193 Ensure that containers are run with the minimal set of file system permissions required. Very  
1194 rarely should containers mount local file systems on a host. Instead, any file changes that  
1195 containers need to persist to disk should be made within storage volumes specifically allocated  
1196 for this purpose. In no case should containers be able to mount sensitive directories on a host's  
1197 file system, especially those containing configuration settings for the operating system.  
1198 Organizations should use tools that can monitor what directories are being mounted by  
1199 containers and prevent the deployment of containers that violate these policies.

#### 1200 **4.6 Hardware Countermeasures**

1201 Software-based security is regularly defeated, as acknowledged in NIST SP 800-164 [22]. NIST  
1202 defines trusted computing requirements in NIST SPs 800-147 [23], 800-155 [24], and 800-164.  
1203 To NIST, "trusted" means that the platform behaves as it is expected to: the software inventory is  
1204 accurate, the configuration settings and security controls are in place and operating as they  
1205 should, and so on. "Trusted" also means that it is known that no unauthorized person has  
1206 tampered with the software or its configuration on the hosts. Hardware root of trust is not a  
1207 concept unique to containers, but container management and security tools can leverage  
1208 attestations for the rest of the container technology architecture to ensure containers are being  
1209 run in secure environments.

1210 The currently available way to provide trusted computing is to:

- 1211 1. Measure firmware, software, and configuration data before it is executed using a Root of  
1212 Trust for Measurement (RTM).
- 1213 2. Store those measurements in a hardware root of trust, like a trusted platform module  
1214 (TPM).
- 1215 3. Validate that the current measurements match the expected measurements. If so, it can be  
1216 attested that the platform can be trusted to behave as expected.

1217 TPM-enabled devices can check the integrity of the machine during the boot process, enabling  
1218 protection and detection mechanisms to function in hardware, at pre-boot, and in the secure boot  
1219 process. This same trust and integrity assurance can be extended beyond the OS and the boot  
1220 loader to the container runtimes and applications. Note that while standards are being developed  
1221 to enable verification of hardware trust by users of cloud services, not all clouds expose this  
1222 functionality to their customers. In cases where technical verification is not provided,  
1223 organizations should address hardware trust requirements as part of their service agreements with  
1224 cloud providers.

1225 The increasing complexity of systems and the deeply embedded nature of today's threats means  
1226 that security should extend across all container technology components, starting with the  
1227 hardware and firmware. This would form a distributed trusted computing model and provide the  
1228 most trusted and secure way to build, run, orchestrate, and manage containers.

1229 The trusted computing model should start with measured/secure boot, which provides a verified  
1230 system platform, and build a chain of trust rooted in hardware and extended to the bootloaders,  
1231 the OS kernel, and the OS components to enable cryptographic verification of boot mechanisms,  
1232 system images, container runtimes, and container images. For container technologies, these  
1233 techniques are currently applicable at the hardware, hypervisor, and host OS layers, with early  
1234 work in progress to apply these to container-specific components.

1235 As of this writing, NIST is collaborating with industry partners to build reference architectures  
1236 based on commercial off-the-shelf products that demonstrate the trusted computing model for  
1237 container environments.<sup>7</sup>

1238

---

<sup>7</sup> For more information on previous NIST efforts in this area, see NIST IR 7904, *Trusted Geolocation in the Cloud: Proof of Concept Implementation* [28].

## 1239 **5 Container Threat Scenario Examples**

1240 To illustrate the effectiveness of the recommended mitigations from Section 4, consider the  
1241 following threat scenario examples for containers.

### 1242 **5.1 Exploit of a Vulnerability within an Image**

1243 One of the most common threats to a containerized environment is application-level  
1244 vulnerabilities in the software within containers. For example, an organization may build an  
1245 image based on a common web application. If that application has a vulnerability, it may be used  
1246 to subvert the application within the container. Once compromised, the attacker may be able to  
1247 map other systems in the environment, attempt to elevate privileges within the compromised  
1248 container, or abuse the container for use in attacks on other systems (such as acting as a file  
1249 dropper or command and control endpoint).

1250 Organizations that adopt the recommendations would have multiple layers of defense in depth  
1251 against such threats:

- 1252 1. Detecting the vulnerable image early in the deployment process and having controls in  
1253 place to prevent vulnerable images from being deployed would prevent the vulnerability  
1254 from being introduced into production.
- 1255 2. Container-aware network monitoring and filtering would detect anomalous connections  
1256 to other containers during the attempt to map other systems.
- 1257 3. Container-aware process monitoring and malware detection would detect the running of  
1258 invalid or unexpected malicious processes and the data they introduce into the  
1259 environment.

### 1260 **5.2 Exploit of the Container Runtime**

1261 While an uncommon occurrence, if a container runtime were compromised, an attacker could  
1262 utilize this access to attack all the containers on the host and even the host itself.

1263 Relevant mitigations for this threat scenario include:

- 1264 1. The usage of mandatory access control capabilities can provide additional barriers to  
1265 ensure that process and file system activity is still segmented within the defined  
1266 boundaries.
- 1267 2. Segmentation of workloads ensures that the scope of the compromise would be limited to  
1268 applications of a common sensitivity level that are sharing the host. For example, a  
1269 compromised runtime on a host only running web applications would not impact  
1270 runtimes on other hosts running containers for financial applications.
- 1271 3. Security tools that can report on the vulnerability state of runtimes and prevent the  
1272 deployment of images to vulnerable ones can prevent workloads from running there.

### 1273 **5.3 Running a Poisoned Image**

1274 Because images are easily sourced from public locations, often with unknown provenance, an  
1275 attacker may embed malicious software within images known to be used by a target. For

1276 example, if an attacker determines that a target is active on a discussion board about a particular  
1277 project and uses images provided by that project's web site, the attacker may seek to craft  
1278 malicious versions of these images for use in an attack.

1279 Relevant mitigations include:

- 1280 1. Ensuring that only trusted images are allowed to run, which will prevent images from  
1281 external, unvetted sources from being used.
- 1282 2. Automatically scanning images for vulnerabilities and malware, which may detect  
1283 malicious code such as rootkits embedded within an image.
- 1284 3. Implementing runtime controls that limit the container's ability to abuse resources,  
1285 escalate privileges, and run executables.
- 1286 4. Using container-level network segmentation to limit the "blast radius" of what the  
1287 poisoned image might do.
- 1288 5. Validating a container image's runtime operates following least-privilege and least-access  
1289 principles.
- 1290 6. Building a threat profile of the container's runtime. This includes, but is not limited to,  
1291 processes, network calls, and filesystem changes.
- 1292 7. Leveraging the use of digitally hashed or signed images to validate images before  
1293 runtime as integrity and tampering checks.

1294

1295

## 1296 **6 Container Technology Life Cycle Security Considerations**

1297 It is critically important to carefully plan before installing, configuring, and deploying container  
1298 technologies. This helps ensure that the container environment is as secure as possible and is in  
1299 compliance with all relevant organizational policies, external regulations, and other  
1300 requirements.

1301 There is a great deal of similarity in the planning and implementation recommendations for  
1302 container technologies and virtualization solutions. Section 5 of NIST SP 800-125 [1] already  
1303 contains a full set of recommendations for virtualization solutions. Instead of repeating all those  
1304 recommendations here, this section points readers to that document and states that, besides the  
1305 exceptions listed below, organizations should apply all the NIST SP 800-125 Section 5  
1306 recommendations in a container technology context. For example, instead of creating a  
1307 virtualization security policy, create a container technology security policy.

1308 This section of the document lists exceptions and additions to the NIST SP 800-125 Section 5  
1309 recommendations, grouped by the corresponding phase in the planning and implementation life  
1310 cycle.

### 1311 **6.1 Initiation Phase**

1312 Organizations should consider how other security policies may be affected by containers and  
1313 adjust these policies as needed to take containers into consideration. For example, policies for  
1314 incident response (especially forensics) and vulnerability management may need to be adjusted  
1315 to take into account the special requirements of containers.

1316 The introduction of container technologies might disrupt the existing culture and software  
1317 development methodologies within the organization. To take full advantage of the benefits  
1318 containers can provide, the organization's processes should be tailored to support this new way  
1319 of developing, running, and supporting applications. Traditional development practices, patching  
1320 techniques, and system upgrade processes might not directly apply to a containerized  
1321 environment, and it is important that the employees within the organization are willing to adapt  
1322 to a new model. New processes can consider and address any potential culture shock that is  
1323 introduced by the technology shift. Education and training can be offered to anyone involved in  
1324 the software development lifecycle to allow people to become comfortable with the new way to  
1325 build, ship, and run applications.

### 1326 **6.2 Planning and Design Phase**

1327 The primary container-specific consideration for the planning and design phase is forensics.  
1328 Because containers mostly build on components already present in OSs, the tools and techniques  
1329 for performing forensics in a containerized environment are mostly an evolution of existing  
1330 practices. The immutable nature of containers and images can actually improve forensic  
1331 capabilities because the demarcation between what an image should do and what actually  
1332 occurred during an incident is clearer. For example, if a container launched to run a web server  
1333 suddenly starts a mail relay, it is very clear that the new process was not part of the original

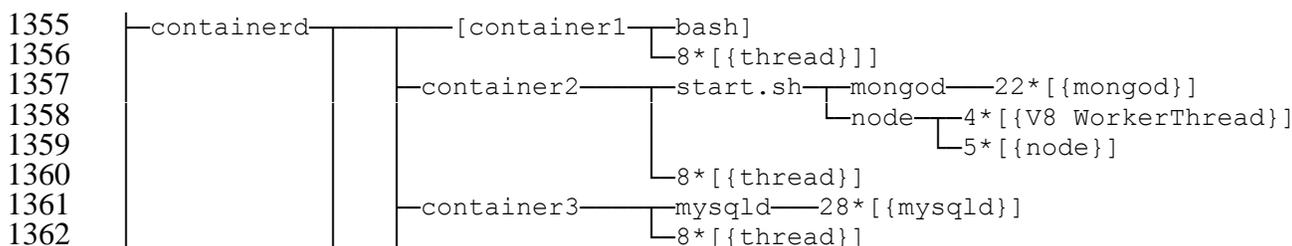
1334 image used to create the container. On traditional platforms, with less separation between the OS  
1335 and apps, making this differentiation can be much more difficult.

1336 Organizations that are familiar with process, memory, and disk incident response activities will  
1337 find them largely similar when working with containers. However, there are some differences to  
1338 keep in mind as well.

1339 Containers typically use a layered file system that is virtualized from the host OS. Directly  
1340 examining paths on the hosts typically only reveals the outer boundary of these layers, not the  
1341 files and data within them. Thus, when responding to incidents in containerized environments,  
1342 users should identify the specific storage provider in use and understand how to properly  
1343 examine its contents offline.

1344 Containers are typically connected to each other using virtualized overlay networks. These  
1345 overlay networks frequently use encapsulation and encryption to allow the traffic to be routed  
1346 over existing networks securely. However, this means that when investigating incidents on  
1347 container networks, particularly when doing any live packet analysis, the tools used must be  
1348 aware of these virtualized networks and understand how to extract the embedded IP frames from  
1349 within them for parsing with existing tools.

1350 Process and memory activity within containers is largely similar to that which would be observed  
1351 within traditional apps, but with different parent processes. For example, container runtimes may  
1352 spawn all processes within containers in a nested fashion in which the runtime is the top-level  
1353 process with first-level descendants per container and second-level descendants for each process  
1354 within the container. For example:



### 1363 6.3 Implementation Phase

1364 After the container technology has been designed, the next step is to implement and test a  
1365 prototype of the design before putting the solution into production. Be aware that container  
1366 technologies do not offer the types of introspection capabilities that VM technologies do.

1367 NIST SP 800-125 [1] cites several aspects of virtualization technologies that should be evaluated  
1368 before production deployment, including authentication, connectivity and networking,  
1369 application functionality, management, performance, and the security of the technology itself. In  
1370 addition to those, it is important to also evaluate the container technology's isolation capabilities.  
1371 Ensure that processes within the container can access all resources they are permitted to and  
1372 cannot view or access any other resources.

1373 Implementation may also require altering the configuration of other security controls and  
1374 technologies, such as security event logging, network management, code repositories, and  
1375 authentication servers.

1376 When the prototype evaluation has been completed and the container technology is ready for  
1377 production usage, containers should initially be used for a small number of applications.  
1378 Problems that occur are likely to affect multiple applications, so it is helpful to identify these  
1379 problems early on so they can be addressed before further deployment. A phased deployment  
1380 also provides time for developers and IT staff (e.g., system administrators, help desk) to be  
1381 trained on its usage and support.

#### 1382 **6.4 Operations and Maintenance Phase**

1383 Operational processes that are particularly important for maintaining the security of container  
1384 technologies, and thus should be performed regularly, include updating all images and  
1385 distributing those updated images to containers to take the place of older images. Other security  
1386 best practices, such as performing vulnerability management and updates for other supporting  
1387 layers like hosts and orchestrators, are also key ongoing operational tasks. Container security and  
1388 monitoring tools should similarly be integrated with existing security information and event  
1389 management (SIEM) tools to ensure container-related events flow through the same tools and  
1390 processes used to provide security throughout the rest of the environment.

1391 If and when security incidents occur within a containerized environment, organizations should be  
1392 prepared to respond with processes and tools that are optimized for the unique aspects of  
1393 containers. The core guidance outlined in NIST SP 800-61, *Computer Security Incident*  
1394 *Handling Guide* [25], is very much applicable for containerized environments as well. However,  
1395 organizations adopting containers should ensure they enhance their responses for some of the  
1396 unique aspects of container security.

- 1397 • Because containerized apps may be run by a different team than the traditional operations  
1398 team, organizations should ensure that whatever teams are responsible for container  
1399 operations are brought into the incident response plan and understand their role in it.
- 1400 • As discussed throughout this document, the ephemeral and automated nature of container  
1401 management may not be aligned with the asset management policies and tools an  
1402 organization has traditionally used. Incident response team must be able to know the  
1403 roles, owners, and sensitivity levels of containers, and be able to integrate this data into  
1404 their process.
- 1405 • Clear procedures should be defined to response to container related incidents. For  
1406 example, if a particular image is being exploited, but that image is in use across hundreds  
1407 of containers, the response team may need to shut down all of these containers to stop the  
1408 attack. While single vulnerabilities have long been able to cause problems across many  
1409 systems, with containers, the response may require rebuilding and redeploying a new  
1410 image widely, rather than installing a patch to existing systems. This change in response  
1411 may involve different teams and approvals and should be understood and practiced ahead  
1412 of time.
- 1413 • As discussed previously, logging and other forensic data may be stored differently in a  
1414 containerized environment. Incident response teams should be familiar with the different

1415 tools and techniques required to gather data and have documented processes specifically  
1416 for these environments.

## 1417 **6.5 Disposition Phase**

1418 The ability for containers to be deployed and destroyed automatically based on the needs of an  
1419 application allows for highly efficient systems but can also introduce some challenges for  
1420 records retention, forensic, and event data requirements. Organizations should make sure that  
1421 appropriate mechanisms are in place to satisfy their data retention policies. Example of issues  
1422 that should be addressed are how containers and images should be destroyed, what data should  
1423 be extracted from a container before disposal and how that data extraction should be performed,  
1424 how cryptographic keys used by a container should be revoked or deleted, etc.

1425 Data stores and media that support the containerized environment should be included in any  
1426 disposal plans developed by the organization.

1427

**7 Conclusion**

1429 Containers represent a transformational change in the way apps are built and run. They do not  
1430 necessitate dramatically new security best practices; on the contrary, most important aspects of  
1431 container security are refinements of well-established techniques and principles. This document  
1432 has updated and expanded general security recommendations to take the risks particular to  
1433 container technologies into account.

1434 This document has already discussed some of the differences between securing containers and  
1435 securing the same apps in VMs. It is useful to summarize the guidance in this document around  
1436 those points.

1437 In container environments there are many more entities, so security processes and tools must be  
1438 able to scale accordingly. Scale does not just mean the total number of objects supported in a  
1439 database, but also how effectively and autonomously policy can be managed. Many  
1440 organizations struggle with the burden of managing security across hundreds of VMs. As  
1441 container-centric architectures become the norm and these organizations are responsible for  
1442 thousands or tens of thousands of containers, their security practices should emphasize  
1443 automation and efficiency to keep up.

1444 With containers there is a much higher rate of change, moving from updating an app a few times  
1445 a year to a few times a week or even a day. What used to be acceptable to do manually no longer  
1446 is. Automation is not just important to deal with the net number of entities, but also how  
1447 frequently those entities change. Being able to centrally express policy and have software  
1448 manage enforcement of it across the environment is vital. Organizations that adopt containers  
1449 should be prepared to manage this frequency of change. This may require fundamentally new  
1450 operational practices and organizational evolution.

1451 The use of containers shifts much of the responsibility for security to developers, so  
1452 organizations should ensure their developers have all the information, skills, and tools they need  
1453 to make sound decisions. Also, security teams should be enabled to actively enforce quality  
1454 throughout the development cycle. Organizations that are successful at this transition gain  
1455 security benefit in being able to respond to vulnerabilities faster and with less operational burden  
1456 than ever before.

1457 Security must be as portable as the containers themselves, so organizations should adopt  
1458 techniques and tools that are open and work across platforms and environments. Many  
1459 organizations will see developers build in one environment, test in another, and deploy in a third,  
1460 so having consistency in assessment and enforcement across these is key. Portability is also not  
1461 just environmental but also temporal. Continuous integration and deployment practices erode the  
1462 traditional walls between phases of the development and deployment cycle, so organizations  
1463 need to ensure consistent, automated security practices across creation of the image, storage of  
1464 the image in registries, and running of the images in containers.

1465 Organizations that navigate these changes can begin to leverage containers to actually improve  
1466 their overall security. The immutability and declarative nature of containers enables  
1467 organizations to begin realizing the vision of more automated, app-centric security that requires

1468 minimal manual involvement and that updates itself as the apps change. Containers are an  
1469 enabling capability in organizations moving from reactive, manual, high-cost security models to  
1470 those that enable better scale and efficiency, thus lowering risk.

1471 **Appendix A—NIST Resources for Securing Non-Core Components**

1472 This appendix lists NIST resources for securing non-core container technology components,  
 1473 including developer systems, testing and accreditation systems, administrator systems, and host  
 1474 hardware and virtual machine managers. Many more resources are available from other  
 1475 organizations.

1476 **Table 1: NIST Resources for Securing Non-Core Components**

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

1477

1478

1479  
1480

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

1481  
1482

The security controls from NIST SP 800-53 Revision 4 [26] that are most important for container technologies are listed in Table 2.

1483

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

NIST SP 800-53 Control	Related Controls	References
AC-2, Account Management	AC-3, AC-4, AC-5, AC-6, AC-10, AC-17, AC-19, AC-20, AU-9, IA-2, IA-4, IA-5, IA-8, CM-5, CM-6, CM-11, MA-3, MA-4, MA-5, PL-4, SC-13	
AC-3, Access Enforcement	AC-2, AC-4, AC-5, AC-6, AC-16, AC-17, AC-18, AC-19, AC-20, AC-21, AC- 22, AU-9, CM-5, CM-6, CM-11, MA-3, MA-4, MA-5, PE-3	
AC-4, Information Flow Enforcement	AC-3, AC-17, AC-19, AC-21, CM-6, CM-7, SA-8, SC-2, SC-5, SC-7, SC-18	
AC-6, Least Privilege	AC-2, AC-3, AC-5, CM-6, CM-7, PL-2	
AC-17, Remote Access	AC-2, AC-3, AC-18, AC-19, AC-20, CA-3, CA-7, CM-8, IA-2, IA-3, IA-8, MA-4, PE-17, PL-4, SC-10, SI-4	NIST SPs 800-46, 800-77, 800-113, 800-114, 800-121
AT-3, Role-Based Security Training	AT-2, AT-4, PL-4, PS-7, SA-3, SA-12, SA-16	C.F.R. Part 5 Subpart C (5C.F.R.930.301); NIST SPs 800-16, 800- 50
AU-2, Audit Events	AC-6, AC-17, AU-3, AU-12, MA-4, MP-2, MP-4, SI-4	NIST SP 800-92; <a href="https://idmanagement.gov/">https://idmanagement.gov/</a>
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; <a href="https://nvd.nist.gov/">https://nvd.nist.gov/</a> ; <a href="https://checklists.nist.gov/">https://checklists.nist.gov/</a> ; <a href="https://www.nsa.gov">https://www.nsa.gov</a>

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; <a href="https://idmanagement.gov/">https://idmanagement.gov/</a>
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; <a href="https://idmanagement.gov/">https://idmanagement.gov/</a>
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; <a href="https://idmanagement.gov/">https://idmanagement.gov/</a>
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; <a href="https://nvd.nist.gov">https://nvd.nist.gov</a> ; <a href="http://cwe.mitre.org">http://cwe.mitre.org</a> ; <a href="http://cve.mitre.org">http://cve.mitre.org</a> ; <a href="http://capec.mitre.org">http://capec.mitre.org</a>
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

1484

1485 The list below details the NIST Cybersecurity Framework [27] subcategories that are most  
 1486 important for container technology security.

- 1487 • **Identify: Asset Management**
- 1488 ○ ID.AM-3: Organizational communication and data flows are mapped
- 1489 ○ ID.AM-5: Resources (e.g., hardware, devices, data, and software) are prioritized
- 1490 based on their classification, criticality, and business value
- 1491 • **Identify: Risk Assessment**
- 1492 ○ ID.RA-1: Asset vulnerabilities are identified and documented
- 1493 ○ ID.RA-3: Threats, both internal and external, are identified and documented
- 1494 ○ ID.RA-4: Potential business impacts and likelihoods are identified
- 1495 ○ ID.RA-5: Threats, vulnerabilities, likelihoods, and impacts are used to determine risk
- 1496 ○ ID.RA-6: Risk responses are identified and prioritized
- 1497 • **Protect: Access Control**
- 1498 ○ PR.AC-1: Identities and credentials are managed for authorized devices and users
- 1499 ○ PR.AC-2: Physical access to assets is managed and protected
- 1500 ○ PR.AC-3: Remote access is managed

- 1501 ○ PR.AC-4: Access permissions are managed, incorporating the principles of least
- 1502 privilege and separation of duties
- 1503 ● **Protect: Awareness and Training**
- 1504 ○ PR.AT-2: Privileged users understand roles & responsibilities
- 1505 ○ PR.AT-5: Physical and information security personnel understand roles &
- 1506 responsibilities
- 1507 ● **Protect: Data Security**
- 1508 ○ PR.DS-2: Data-in-transit is protected
- 1509 ○ PR.DS-4: Adequate capacity to ensure availability is maintained
- 1510 ○ PR.DS-5: Protections against data leaks are implemented
- 1511 ○ PR.DS-6: Integrity checking mechanisms are used to verify software, firmware, and
- 1512 information integrity
- 1513 ● **Protect: Information Protection Processes and Procedures**
- 1514 ○ PR.IP-1: A baseline configuration of information technology/industrial control
- 1515 systems is created and maintained
- 1516 ○ PR.IP-3: Configuration change control processes are in place
- 1517 ○ PR.IP-6: Data is destroyed according to policy
- 1518 ○ PR.IP-9: Response plans (Incident Response and Business Continuity) and recovery
- 1519 plans (Incident Recovery and Disaster Recovery) are in place and managed
- 1520 ○ PR.IP-12: A vulnerability management plan is developed and implemented
- 1521 ● **Protect: Maintenance**
- 1522 ○ PR.MA-1: Maintenance and repair of organizational assets is performed and logged
- 1523 in a timely manner, with approved and controlled tools
- 1524 ○ PR.MA-2: Remote maintenance of organizational assets is approved, logged, and
- 1525 performed in a manner that prevents unauthorized access
- 1526 ● **Protect: Protective Technology**
- 1527 ○ PR.PT-1: Audit/log records are determined, documented, implemented, and reviewed
- 1528 in accordance with policy
- 1529 ○ PR.PT-3: Access to systems and assets is controlled, incorporating the principle of
- 1530 least functionality
- 1531 ● **Detect: Anomalies and Events**
- 1532 ○ DE.AE-2: Detected events are analyzed to understand attack targets and methods
- 1533 ● **Detect: Security Continuous Monitoring**
- 1534 ○ DE.CM-1: The network is monitored to detect potential cybersecurity events
- 1535 ○ DE.CM-7: Monitoring for unauthorized personnel, connections, devices, and software
- 1536 is performed
- 1537 ● **Respond: Response Planning**
- 1538 ○ RS.RP-1: Response plan is executed during or after an event
- 1539 ● **Respond: Analysis**
- 1540 ○ RS.AN-1: Notifications from detection systems are investigated
- 1541 ○ RS.AN-3: Forensics are performed
- 1542 ● **Respond: Mitigation**
- 1543 ○ RS.MI-1: Incidents are contained
- 1544 ○ RS.MI-2: Incidents are mitigated
- 1545 ○ RS.MI-3: Newly identified vulnerabilities are mitigated or documented as accepted
- 1546 risks

- 1547 • **Recover: Recovery Planning**
- 1548 ○ RC.RP-1: Recovery plan is executed during or after an event
- 1549

1550 Table 3 lists the security controls from NIST SP 800-53 Revision 4 [26] that can be  
 1551 accomplished partially or completely by using container technologies. The rightmost column  
 1552 lists the sections of this document that map to each NIST SP 800-53 control.

1553 **Table 3: NIST SP 800-53 Controls Supported by Container Technologies**

NIST SP 800-53 Control	Container Technology Relevancy	Related Sections of This Document
CM-3, Configuration Change Control	Images can be used to help manage change control for applications.	2.1, 2.2, 2.3, 2.4, 4.1
SC-2, Application Partitioning	Separating user functionality from administrator functionality can be accomplished in part by using containers or other virtualization technologies so that the functionality is performed in different containers.	2 (introduction), 2.3, 4.5.2
SC-3, Security Function Isolation	Separating security functions from non-security functions can be accomplished in part by using containers or other virtualization technologies so that the functions are performed in different containers.	2 (introduction), 2.3, 4.5.2
SC-4, Information in Shared Resources	Container technologies are designed to restrict each container's access to shared resources so that information cannot inadvertently be leaked from one container to another.	2 (introduction), 2.2, 2.3, 4.4
SC-6, Resource Availability	The maximum resources available for each container can be specified, thus protecting the availability of resources by not allowing any container to consume excessive resources.	2.2, 2.3
SC-7, Boundary Protection	Boundaries can be established and enforced between containers to restrict their communications with each other.	2 (introduction), 2.2, 2.3, 4.4
SC-39, Process Isolation	Multiple containers can run processes simultaneously on the same host, but those processes are isolated from each other.	2 (introduction), 2.1, 2.2, 2.3, 4.4
SI-7, Software, Firmware, and Information Integrity	Unauthorized changes to the contents of images can easily be detected and the altered image replaced with a known good copy.	2.3, 4.1, 4.2
SI-14, Non-Persistence	Images running within containers are replaced as needed with new image versions, so data, files, executables, and other information stored within running images is not persistent.	2.1, 2.3, 4.1

1554

1555 Similar to Table 3, Table 4 lists the NIST Cybersecurity Framework [27] subcategories that can  
 1556 be accomplished partially or completely by using container technologies. The rightmost column  
 1557 lists the sections of this document that map to each Cybersecurity Framework subcategory.

1558 **Table 4: NIST Cybersecurity Framework Subcategories Supported by Container Technologies**

Cybersecurity Framework Subcategory	Container Technology Relevancy	Related Sections of This Document
PR.DS-4: Adequate capacity to ensure availability is maintained	The maximum resources available for each container can be specified, thus protecting the availability of resources by not allowing any container to consume excessive resources.	2.2, 2.3

Cybersecurity Framework Subcategory	Container Technology Relevancy	Related Sections of This Document
PR.DS-5: Protections against data leaks are implemented	Container technologies are designed to restrict each container's access to shared resources so that information cannot inadvertently be leaked from one container to another.	2 (introduction), 2.2, 2.3, 4.4
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.	2.3, 4.1, 4.2
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.	2.1, 2.2, 2.3, 2.4, 4.1

1559

1560 Information on these controls and guidelines on possible implementations can be found in the  
1561 following NIST publications:

- 1562 • [\*FIPS 140-2, Security Requirements for Cryptographic Modules\*](#)
- 1563 • [\*FIPS 197, Advanced Encryption Standard \(AES\)\*](#)
- 1564 • [\*FIPS 199, Standards for Security Categorization of Federal Information and Information\*](#)
- 1565 [\*Systems\*](#)
- 1566 • [\*FIPS 201-2, Personal Identity Verification \(PIV\) of Federal Employees and Contractors\*](#)
- 1567 • [\*SP 800-12 Rev. 1, An Introduction to Information Security\*](#)
- 1568 • [\*Draft SP 800-16 Rev. 1, A Role-Based Model for Federal Information\*](#)
- 1569 [\*Technology/Cybersecurity Training\*](#)
- 1570 • [\*SP 800-18 Rev. 1, Guide for Developing Security Plans for Federal Information Systems\*](#)
- 1571 • [\*SP 800-30 Rev. 1, Guide for Conducting Risk Assessments\*](#)
- 1572 • [\*SP 800-34 Rev. 1, Contingency Planning Guide for Federal Information Systems\*](#)
- 1573 • [\*SP 800-39, Managing Information Security Risk: Organization, Mission, and Information\*](#)
- 1574 [\*System View\*](#)
- 1575 • [\*SP 800-40 Rev. 3, Guide to Enterprise Patch Management Technologies\*](#)
- 1576 • [\*SP 800-46 Rev. 2, Guide to Enterprise Telework, Remote Access, and Bring Your Own\*](#)
- 1577 [\*Device \(BYOD\) Security\*](#)
- 1578 • [\*SP 800-50, Building an Information Technology Security Awareness and Training\*](#)
- 1579 [\*Program\*](#)
- 1580 • [\*SP 800-52 Rev. 1, Guidelines for the Selection, Configuration, and Use of Transport\*](#)
- 1581 [\*Layer Security \(TLS\) Implementations\*](#)
- 1582 • [\*SP 800-53 Rev. 4, Security and Privacy Controls for Federal Information Systems and\*](#)

- 1583        *Organizations*
- 1584        • *SP 800-53A Rev. 4, Assessing Security and Privacy Controls in Federal Information*
- 1585        *Systems and Organizations: Building Effective Assessment Plans*
- 1586        • *SP 800-60 Rev. 1 Vol. 1, Guide for Mapping Types of Information and Information*
- 1587        *Systems to Security Categories*
- 1588        • *SP 800-61 Rev. 2, Computer Security Incident Handling Guide*
- 1589        • *SP 800-63 Rev. 3, Digital Identity Guidelines*
- 1590        • *SP 800-70 Rev. 3, National Checklist Program for IT Products: Guidelines for Checklist*
- 1591        *Users and Developers*
- 1592        • *SP 800-73-4, Interfaces for Personal Identity Verification*
- 1593        • *SP 800-76-2, Biometric Specifications for Personal Identity Verification*
- 1594        • *SP 800-77, Guide to IPsec VPNs*
- 1595        • *SP 800-78-4, Cryptographic Algorithms and Key Sizes for Personal Identification*
- 1596        *Verification (PIV)*
- 1597        • *SP 800-81-2, Secure Domain Name System (DNS) Deployment Guide*
- 1598        • *SP 800-83 Rev. 1, Guide to Malware Incident Prevention and Handling for Desktops and*
- 1599        *Laptops*
- 1600        • *SP 800-88 Rev. 1, Guidelines for Media Sanitization*
- 1601        • *SP 800-92, Guide to Computer Security Log Management*
- 1602        • *SP 800-100, Information Security Handbook: A Guide for Managers*
- 1603        • *SP 800-113, Guide to SSL VPNs*
- 1604        • *SP 800-114 Rev. 1, User's Guide to Telework and Bring Your Own Device (BYOD)*
- 1605        *Security*
- 1606        • *SP 800-121 Rev. 2, Guide to Bluetooth Security*
- 1607        • *SP 800-128, Guide for Security-Focused Configuration Management of Information*
- 1608        *Systems*
- 1609        • *SP 800-137, Information Security Continuous Monitoring (ISCM) for Federal*
- 1610        *Information Systems and Organizations*
- 1611        • *SP 800-147, BIOS Protection Guidelines*
- 1612        • *Draft SP 800-155, BIOS Integrity Measurement Guidelines*
- 1613
- 1614

**1615 Appendix C—Acronyms and Abbreviations**

1616 Selected acronyms and abbreviations used in this paper are defined below.

AES	Advanced Encryption Standard
API	Application Programming Interface
AUFS	Advanced Multi-Layered Unification Filesystem
BIOS	Basic Input/Output System
BYOD	Bring Your Own Device
cgroup	Control Group
CIS	Center for Internet Security
CMVP	Cryptographic Module Validation Program
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DevOps	Development and Operations
DNS	Domain Name System
FIPS	Federal Information Processing Standards
FIRST	Forum for Incident Response and Security Teams
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
MAC	Mandatory Access Control
NIST	National Institute of Standards and Technology
NTFS	NT File System
OMB	Office of Management and Budget
OS	Operating System
PIV	Personal Identity Verification
RTM	Root of Trust for Measurement

SDN	Software-Defined Networking
seccomp	Secure Computing
SIEM	Security Information and Event Management
SP	Special Publication
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Sockets Layer
TLS	Transport Layer Security
TPM	Trusted Platform Module
URI	Uniform Resource Identifier
US	United States
USCIS	United States Citizenship and Immigration Services
VM	Virtual Machine
VPN	Virtual Private Network

1617

1618

**Appendix D—Glossary**

Application virtualization	A form of virtualization that exposes a single shared operating system kernel to multiple discrete application instances, each of which is kept isolated from all others on the host.
Base layer	The underlying layer of an image upon which all other components are added.
Container	A method for packaging and securely running an application within an application virtualization environment. Also known as an application container or a server application container.
Container runtime	The environment for each container; comprised of binaries coordinating multiple operating system components that isolate resources and resource usage for running containers.
Container-specific operating system	A minimalistic host operating system explicitly designed to only run containers.
Filesystem virtualization	A form of virtualization that allows multiple containers to share the same physical storage without the ability to access or alter the storage of other containers.
General-purpose operating system	A host operating system that can be used to run many kinds of applications, not just applications in containers.
Host operating system	The operating system kernel shared by multiple applications within an application virtualization architecture.
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 that enables DevOps personas or automation working on their behalf to pull images from registries, deploy those images into containers, and manage the running containers. Orchestrators are also responsible for monitoring container resource consumption, job execution, and machine health across hosts.
Overlay network	A software-defined networking component included in most orchestrators that can be used to isolate communication between applications that share the same physical network.
Registry	A service that allows developers to easily storage images as they are created, tag and catalog images for identification and version control to aid in discovery and reuse, and find and download images that others have created.
Resource allocation	A mechanism for limiting 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]]

1619

1620

**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] Docker, <https://www.docker.com/>
- [3] rkt, <https://coreos.com/rkt/>
- [4] CoreOS Container Linux, <https://coreos.com/os/docs/latest>
- [5] Project Atomic, <http://www.projectatomic.io>
- [6] Google Container-Optimized OS, <https://cloud.google.com/container-optimized-os/docs/>
- [7] Open Container Initiative Daemon (OCID), <https://github.com/kubernetes-incubator/cri-o>
- [8] Jenkins, <https://jenkins.io>
- [9] TeamCity, <https://www.jetbrains.com/teamcity/>
- [10] Amazon EC2 Container Registry (ECR), <https://aws.amazon.com/ecr/>
- [11] Docker Hub, <https://hub.docker.com/>
- [12] Docker Trusted Registry, <https://hub.docker.com/r/docker/dtr/>
- [13] Quay Container Registry, <https://quay.io>
- [14] Kubernetes, <https://kubernetes.io/>
- [15] Apache Mesos, <http://mesos.apache.org/>
- [16] Docker Swarm, <https://github.com/docker/swarm>
- [17] 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](http://csrc.nist.gov/publications/drafts/800-154/sp800_154_draft.pdf).
- [18] *Common Vulnerability Scoring System v3.0: Specification Document*, Forum for Incident Response and Security Teams (FIRST). <https://www.first.org/cvss/specification-document>.

- [19] CIS Docker Benchmark, Center for Internet Security (CIS).  
<https://www.cisecurity.org/benchmark/docker/>.
- [20] Security Enhanced Linux (SELinux), [https://selinuxproject.org/page/Main\\_Page](https://selinuxproject.org/page/Main_Page)
- [21] AppArmor, [http://wiki.apparmor.net/index.php/Main\\_Page](http://wiki.apparmor.net/index.php/Main_Page)
- [22] 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](http://csrc.nist.gov/publications/drafts/800-164/sp800_164_draft.pdf).
- [23] 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>.
- [24] 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](http://csrc.nist.gov/publications/drafts/800-155/draft-SP800-155_Dec2011.pdf).
- [25] NIST Internal Report (IR) 7904, *Trusted Geolocation in the Cloud: Proof of Concept Implementation*, National Institute of Standards and Technology, Gaithersburg, Maryland, December 2015, 59 pp. <https://dx.doi.org/10.6028/NIST.IR.7904>
- [26] NIST Special Publication (SP) 800-61 Revision 2, *Computer Security Incident Handling Guide*, National Institute of Standards and Technology, Gaithersburg, Maryland, August 2012, 79 pp. <https://dx.doi.org/10.6028/NIST.SP.800-61r2>.
- [27] 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>.
- [28] *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>.