



# **OpenShift Container Platform 3.5**

## **Container Security Guide**

OpenShift Container Platform 3.5 Container Security Guide



# OpenShift Container Platform 3.5 Container Security Guide

---

OpenShift Container Platform 3.5 Container Security Guide

## Legal Notice

Copyright © 2018 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## Abstract

Secure your cluster using these recommendations

## Table of Contents

<b>CHAPTER 1. INTRODUCTION</b>	<b>4</b>
1.1. ABOUT THIS GUIDE	4
1.2. WHAT ARE CONTAINERS?	4
Further Reading	4
1.3. CONTAINER SECURITY IN OPENSIFT CONTAINER PLATFORM	4
Further Reading	5
<b>CHAPTER 2. CONTAINER HOSTS AND MULTI-TENANCY</b>	<b>6</b>
2.1. HOW CONTAINERS ARE SECURED ON RHEL	6
Further Reading	6
2.2. MULTI-TENANCY: VIRTUALIZATION VERSUS CONTAINERS	6
Further Reading	7
<b>CHAPTER 3. CONTAINER CONTENT</b>	<b>8</b>
3.1. SECURITY INSIDE THE CONTAINER	8
Further Reading	8
3.2. CONTAINER CONTENT SCANNING	8
3.3. INTEGRATING EXTERNAL SCANNING TOOLS WITH OPENSIFT	8
3.3.1. Image Metadata	8
3.3.1.1. Example Annotation Keys	9
3.3.1.2. Example Annotation Values	10
3.3.2. Annotating Image Objects	11
3.3.2.1. Example Annotate CLI Command	11
3.3.3. Controlling Pod Execution	11
3.3.3.1. Example Annotation	11
3.3.4. Integration Reference	11
3.3.4.1. Example REST API Call	11
<b>CHAPTER 4. REGISTRIES</b>	<b>13</b>
4.1. WHERE DO YOUR CONTAINERS COME FROM?	13
4.2. IMMUTABLE AND CERTIFIED CONTAINERS	13
Further Reading	13
4.3. RED HAT REGISTRY AND RED HAT CONTAINER CATALOG	13
Further Reading	14
4.4. OPENSIFT CONTAINER REGISTRY	14
Further Reading	14
<b>CHAPTER 5. BUILD PROCESS</b>	<b>15</b>
5.1. BUILD ONCE, DEPLOY EVERYWHERE	15
5.2. BUILD MANAGEMENT AND SECURITY	15
Further Reading	16
5.3. SECURING INPUTS DURING BUILDS	16
Further Reading	16
5.4. DESIGNING YOUR BUILD PROCESS	17
Further Reading	17
<b>CHAPTER 6. DEPLOYMENT</b>	<b>18</b>
6.1. CONTROLLING WHAT CAN BE DEPLOYED IN A CONTAINER	18
Further Reading	19
6.2. SECRETS AND CONFIGMAPS	19
Further Reading	19
6.3. SECURITY CONTEXT CONSTRAINTS (SCCS)	19
Further Reading	20

6.4. CONTINUOUS DEPLOYMENT TOOLING	20
<b>CHAPTER 7. SECURING THE CONTAINER PLATFORM</b> .....	<b>21</b>
7.1. CONTAINER ORCHESTRATION	21
Further Reading	21
7.2. AUTHENTICATION AND AUTHORIZATION	21
7.2.1. Controlling Access Using OAuth	21
Further Reading	22
7.2.2. API Access Control and Management	22
7.2.3. Red Hat SSO	22
7.2.4. Secure Self-service Web Console	22
Further Reading	23
7.3. MANAGING CERTIFICATES FOR THE PLATFORM	23
7.3.1. Configuring Custom Certificates	23
Further Reading	23
<b>CHAPTER 8. NETWORK SECURITY</b> .....	<b>25</b>
8.1. NETWORK NAMESPACES	25
Further Reading	25
8.2. ISOLATING APPLICATIONS	25
<b>CHAPTER 9. ATTACHED STORAGE</b> .....	<b>26</b>
9.1. PERSISTENT VOLUME PLUG-INS	26
Further Reading	26
9.2. SHARED STORAGE	26
Further Reading	26
9.3. BLOCK STORAGE	26
Further Reading	26
<b>CHAPTER 10. REVISION HISTORY: CONTAINER SECURITY GUIDE</b> .....	<b>28</b>
10.1. MON MAY 22 2017	28



# CHAPTER 1. INTRODUCTION

## 1.1. ABOUT THIS GUIDE

This guide provides a high-level walkthrough of the container security measures available in OpenShift Container Platform, including solutions for the host layer, the container and orchestration layer, and the build and application layer. This guide contains the following information:

- Why container security is important and how it compares with existing security standards.
- Which container security measures are provided by the host (RHEL) layer and which are provided by OpenShift Container Platform.
- How to evaluate your container content and sources for vulnerabilities.
- How to design your build and deployment process to proactively check container content.
- How to control access to containers via authentication and authorization.
- How networking and attached storage are secured in OpenShift Container Platform.
- Containerized solutions for API management and SSO.

## 1.2. WHAT ARE CONTAINERS?

Containers package an application and all its dependencies into a single image that can be promoted from development, to test, to production, without change.

Containers provide consistency across environments and multiple deployment targets: physical servers, virtual machines (VMs), and private or public cloud.

Some of the benefits of using containers include:

INFRASTRUCTURE	APPLICATIONS
Sandboxed application processes on a shared Linux OS kernel	Package my application and all of its dependencies
Simpler, lighter, and denser than virtual machines	Deploy to any environment in seconds and enable CI/CD
Portable across different environments	Easily access and share containerized components

### Further Reading

- *OpenShift Container Platform Architecture: [Core Concepts](#) → [Containers and Images](#)*
- *[Red Hat Enterprise Linux Atomic Host Container Security Guide](#)*

## 1.3. CONTAINER SECURITY IN OPENSIFT CONTAINER PLATFORM



This guide describes the key elements of security for each layer of the container solution stack, while also showing how OpenShift Container Platform can be used to create, deploy, and manage containers at scale, with security in mind at every stage and every layer.

**Further Reading**

- [\*Red Hat Enterprise Linux Atomic Host Overview of Containers in Red Hat Systems\*](#)
- [\*Red Hat Enterprise Linux Atomic Host Container Security Guide\*](#)

## CHAPTER 2. CONTAINER HOSTS AND MULTI-TENANCY

### 2.1. HOW CONTAINERS ARE SECURED ON RHEL

Containers enable you to simplify multi-tenancy deployments by deploying multiple applications on a single host, using the kernel and the docker runtime to spin up each container.

You must have an operating system (OS) that can secure the host kernel and secure containers from each other. In Linux, containers are just a special type of process, so securing containers is the same as securing any running process. Containers should run as a non-root user. Dropping the privilege level or creating containers with the least amount of privileges possible is recommended.

Because OpenShift Container Platform runs on Red Hat Enterprise Linux (RHEL) and RHEL Atomic Host, the following concepts apply by default to any deployed OpenShift Container Platform cluster and are at the core of what make containers secure on the platform.

- *Linux namespaces* enable creating an abstraction of a particular global system resource to make it appear as a separate instance to processes within a namespace. Consequently, several containers can use the same resource simultaneously without creating a conflict. See [Overview of Containers in Red Hat Systems](#) for details on the types of namespaces (e.g., mount, PID, and network).
- *SELinux* provides an additional layer of security to keep containers isolated from each other and from the host. SELinux allows administrators to enforce mandatory access controls (MAC) for every user, application, process, and file.
- *CGroups* (control groups) limit, account for, and isolate the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. CGroups are used to ensure that containers on the same host are not impacted by each other.
- *Secure computing mode (seccomp)* profiles can be associated with a container to restrict available system calls.
- Deploying containers using *RHEL Atomic Host* reduces the attack surface by minimizing the host environment and tuning it for containers.

#### Further Reading

- Linux man page: [namespaces\(7\)](#)
- *Red Hat Enterprise Linux Atomic Host Overview of Containers in Red Hat Systems*: [Secure Containers with SELinux](#)
- *Red Hat Enterprise Linux Resource Management Guide*: [Introduction to Control Groups \(CGroups\)](#)
- *Red Hat Enterprise Linux Atomic Host Container Security Guide*: [Linux Capabilities and seccomp](#)
- Kernel documentation: [seccomp](#)

### 2.2. MULTI-TENANCY: VIRTUALIZATION VERSUS CONTAINERS

Traditional virtualization also enables multi-tenancy, but in a very different way from containers. Virtualization relies on a hypervisor spinning up guest virtual machines (VMs), each of which has its own operating system (OS), as well as the running application and its dependencies.

With VMs, the hypervisor isolates the guests from each other and from the host kernel. Fewer individuals and processes have access to the hypervisor, reducing the attack surface on the physical server. That said, security must still be monitored: one guest VM may be able to use hypervisor bugs to gain access to another VM or the host kernel. And, when the OS needs patching, it must be patched on all guest VMs using that OS.

Containers can be run inside guest VMs, and there may be use cases where this is desirable. For example, you may be deploying a traditional application in a container, perhaps in order to lift-and-shift an application to the cloud. However, container multi-tenancy on a single host provides a more lightweight, flexible, and easier-to-scale deployment solution. This deployment model is particularly appropriate for cloud-native applications.

### Further Reading

- *Red Hat Enterprise Linux Atomic Host Overview of Containers in Red Hat Systems:* [Linux Containers Compared to KVM Virtualization](#)

## CHAPTER 3. CONTAINER CONTENT

### 3.1. SECURITY INSIDE THE CONTAINER

Applications and infrastructures are composed of readily available components, many of which are open source packages such as, the Linux operating system, JBoss Web Server, PostgreSQL, and Node.js.

Containerized versions of these packages are also available. However, you need to know where the packages originally came from, who built them, and whether there is any malicious code inside them.

Some questions to answer include:

- Will what is inside the containers compromise your infrastructure?
- Are there known vulnerabilities in the application layer?
- Are the runtime and OS layers current?

#### Further Reading

- [OpenShift Container Platform Using Images](#)
  - Reference documentation on framework, database, and service container images provided by Red Hat for use on OpenShift Container Platform

### 3.2. CONTAINER CONTENT SCANNING

Container scanning tools can leverage continuously updated vulnerability databases to ensure that you always have the latest information on known vulnerabilities for your container content. The list of known vulnerabilities constantly evolves; you must check the contents of your container images when you first download them and continue to track vulnerability status over time for all of your approved and deployed images.

RHEL provides a pluggable API to support multiple scanners. You can also use Red Hat CloudForms with OpenSCAP to scan container images for security issues. See the [Red Hat Enterprise Linux Security Guide](#) for general information on OpenSCAP in RHEL, and the [Red Hat CloudForms Policies and Profiles Guide](#) for specifics on OpenSCAP integration.

OpenShift Container Platform enables you to leverage such scanners with your CI/CD process. For example, you can integrate static code analysis tools that test for security flaws in your source code and software composition analysis tools that identify open source libraries in order to provide metadata on those libraries such as known vulnerabilities. This is covered in more detail in [Build Process](#).

### 3.3. INTEGRATING EXTERNAL SCANNING TOOLS WITH OPENSIFT

OpenShift Container Platform makes use of [object annotations](#) to extend functionality. External tools, such as vulnerability scanners, may annotate image objects with metadata to summarize results and control pod execution. This section describes the recognized format of this annotation so it may be reliably used in consoles to display useful data to users.

#### 3.3.1. Image Metadata

There are different types of image quality data, including package vulnerabilities and open source software (OSS) license compliance. Additionally, there may be more than one provider of this metadata. To that end, the following annotation format has been reserved:

```
quality.images.openshift.io/<qualityType>.<providerId>: {}
```

**Table 3.1. Annotation Key Format**

Component	Description	Acceptable Values
<b>qualityType</b>	Metadata type	<b>vulnerability</b> <b>license</b> <b>operations</b> <b>policy</b>
<b>providerId</b>	Provider ID string	<b>openscap</b> <b>redhatcatalog</b> <b>redhatinsights</b> <b>blackduck</b> <b>jfrog</b>

### 3.3.1.1. Example Annotation Keys

```
quality.images.openshift.io/vulnerability.blackduck: {}
quality.images.openshift.io/vulnerability.jfrog: {}
quality.images.openshift.io/license.blackduck: {}
quality.images.openshift.io/vulnerability.openscap: {}
```

The value of the image quality annotation is structured data that must adhere to the following format:

**Table 3.2. Annotation Value Format**

Field	Required?	Description	Type
<b>name</b>	Yes	Provider display name	String
<b>timestamp</b>	Yes	Scan timestamp	String
<b>description</b>	No	Short description	String
<b>reference</b>	Yes	URL of information source and/or more details. Required so user may validate the data.	String
<b>scannerVersion</b>	No	Scanner version	String
<b>compliant</b>	No	Compliance pass/fail	Boolean
<b>summary</b>	No	Summary of issues found	List (see table below)

The **summary** field must adhere to the following format:

**Table 3.3. Summary Field Value Format**

Field	Description	Type
<b>label</b>	Display label for component (e.g., "critical", "important", "moderate", "low" or "health")	String
<b>data</b>	Data for this component (e.g., count of vulnerabilities found or score)	String
<b>severityIndex</b>	Component index allowing for ordering and assigning graphical representation. The value is range <b>0</b> . . <b>3</b> where <b>0</b> = low.	Integer
<b>reference</b>	URL of information source and/or more details. Optional.	String

### 3.3.1.2. Example Annotation Values

This example shows an OpenSCAP annotation for an image with vulnerability summary data and a compliance boolean:

#### OpenSCAP Annotation

```
{
  "name": "OpenSCAP",
  "description": "OpenSCAP vulnerability score",
  "timestamp": "2016-09-08T05:04:46Z",
  "reference": "https://www.open-scap.org/930492",
  "compliant": true,
  "scannerVersion": "1.2",
  "summary": [
    { "label": "critical", "data": "4", "severityIndex": 3, "reference": null },
    { "label": "important", "data": "12", "severityIndex": 2, "reference": null },
    { "label": "moderate", "data": "8", "severityIndex": 1, "reference": null },
    { "label": "low", "data": "26", "severityIndex": 0, "reference": null }
  ]
}
```

This example shows a [Red Hat Container Catalog](#) annotation for an image with health index data with an external URL for additional details:

#### Red Hat Container Catalog Annotation

```
{
  "name": "Red Hat Container Catalog",
```

```

    "description": "Container health index",
    "timestamp": "2016-09-08T05:04:46Z",
    "reference": "https://access.redhat.com/errata/RHBA-2016:1566",
    "compliant": null,
    "scannerVersion": "1.2",
    "summary": [
      { "label": "Health index", "data": "B", "severityIndex": 1,
"reference": null }
    ]
  }
}

```

### 3.3.2. Annotating Image Objects

While [image stream objects](#) are what an end-user of OpenShift Container Platform operates against, [image objects](#) are annotated with security metadata. Image objects are cluster-scoped, pointing to a single image that may be referenced by many image streams and tags.

#### 3.3.2.1. Example Annotate CLI Command

Replace `<image>` with an image digest, for example

**sha256: fec8a395afe3e804b3db5cb277869142d2b5c561ebb517585566e160ff321988:**

```

$ oc annotate image <image> \
  quality.images.openshift.io/vulnerability.redhatcatalog='{ \
    "name": "Red Hat Container Catalog", \
    "description": "Container health index", \
    "timestamp": "2016-09-08T05:04:46Z", \
    "compliant": null, \
    "scannerVersion": "1.2", \
    "reference": "https://access.redhat.com/errata/RHBA-2016:1566", \
    "summary": "[ \
      { "label": "Health index", "data": "B", "severityIndex": 1,
"reference": null } ]" }'

```

### 3.3.3. Controlling Pod Execution

To programmatically control if an image may be run, the `images.openshift.io/deny-execution` image policy may be used. See [Image Policy](#) for more information.

#### 3.3.3.1. Example Annotation

```

annotations:
  images.openshift.io/deny-execution: true

```

### 3.3.4. Integration Reference

In most cases, external tools such as vulnerability scanners will develop a script or plug-in that watches for image updates, performs scanning and annotate the associated image object with the results. Typically this automation calls the OpenShift Container Platform REST API to write the annotation. See [REST API Reference](#) for general information on the REST API and **PATCH** call to update images.

#### 3.3.4.1. Example REST API Call

This example call using **curl** will overwrite the value of the annotation. Be sure to replace the values for **<token>**, **<openshift\_server>**, **<image\_id>**, and **<image\_annotation>**.

### Patch API Call

```
$ curl -X PATCH \
-H "Authorization: Bearer <token>" \
-H "Content-Type: application/merge-patch+json" \
https://<openshift_server>:8443/oapi/v1/images/<image_id> \
--data '{ <image_annotation> }'
```

Below is example **PATCH** payload data.

### Patch Call Data

```
{
  "metadata": {
    "annotations": {
      "quality.images.openshift.io/vulnerability.redhatcatalog":
        "{ 'name': 'Red Hat Container Catalog', 'description': 'Container
health index', 'timestamp': '2016-09-08T05:04:46Z', 'compliant': null,
'reference': 'https://access.redhat.com/errata/RHBA-2016:1566', 'summary':
[{'label': 'Health index', 'data': '4', 'severityIndex': 1, 'reference':
null}] }"
    }
  }
}
```



## CHAPTER 4. REGISTRIES

### 4.1. WHERE DO YOUR CONTAINERS COME FROM?

There are tools you can use to scan and track the contents of your downloaded and deployed container images. However, there are many public sources of container images. When using public container registries, you can add a layer of protection by using trusted sources.

### 4.2. IMMUTABLE AND CERTIFIED CONTAINERS

Consuming security updates is particularly important when managing *immutable containers*. Immutable containers are containers that will never be changed while running. When you deploy immutable containers, you do not step into the running container to replace one or more binaries; you rebuild and redeploy an updated container image.

Red Hat certified images are:

- Free of known vulnerabilities in the platform components or layers.
- Compatible across the RHEL platforms, from bare metal to cloud.
- Supported by Red Hat.

The list of known vulnerabilities is constantly evolving, so you must track the contents of your deployed container images, as well as newly downloaded images, over time. You can use [Red Hat Security Advisories \(RHSA\)](#)s to alert you to any newly discovered issues in Red Hat certified container images, and direct you to the updated image.

#### Further Reading

- More on immutable containers in OpenShift Container Platform:
  - *OpenShift Container Platform Architecture*: [Image Streams](#)
  - *OpenShift Container Platform Developer Guide*: [Referencing Images in Image Streams](#)

### 4.3. RED HAT REGISTRY AND RED HAT CONTAINER CATALOG

Red Hat provides certified containers for Red Hat products and partner offerings via the *Red Hat Registry*, which is a public container registry hosted by Red Hat at [registry.access.redhat.com](https://registry.access.redhat.com). The *Red Hat Container Catalog* enables you to identify bug fix or security advisories associated with container images provided in the Red Hat Registry.

Container content is monitored for vulnerabilities by Red Hat and updated regularly. When Red Hat releases security updates, such as fixes to **glibc**, Drown, or Dirty Cow, any affected container images are also rebuilt and pushed to the Red Hat Registry.

Red Hat uses a "health index" for security risk with containers provided through the Red Hat Container Catalog. These containers consume software provided by Red Hat and the errata process, so old, stale containers are insecure whereas new, fresh containers are more secure.

To illustrate the age of containers, the Red Hat Container Catalog uses a grading system. A freshness grade is a measure of the oldest and most severe security errata available for an image. "A" is more up-to-date than "F". See [Container Health Index grades as used inside the Red Hat Container Catalog](#) for more details on this grading system.

### Further Reading

- [Red Hat Container Catalog FAQ](#)
- [Red Hat Product Security Center](#)
- [Red Hat Security Advisories](#)

## 4.4. OPENSIFT CONTAINER REGISTRY

OpenShift Container Platform includes the *OpenShift Container Registry*, a private registry that runs integrated with the platform that you can use to manage your container images. The OpenShift Container Registry provides role-based access controls that allow you to manage who can pull and push which container images.

OpenShift Container Platform also supports integration with other private registries you may already be using.

### Further Reading

- *OpenShift Container Platform Architecture*: [Infrastructure Components](#) → [Image Registry](#)

## CHAPTER 5. BUILD PROCESS

### 5.1. BUILD ONCE, DEPLOY EVERYWHERE

In a container environment, the software build process is the stage in the life cycle where application code is integrated with the required runtime libraries. Managing this build process is key to securing the software stack.

Using OpenShift Container Platform as the standard platform for container builds enables you to guarantee the security of the build environment. Adhering to a "build once, deploy everywhere" philosophy ensures that the product of the build process is exactly what is deployed in production.

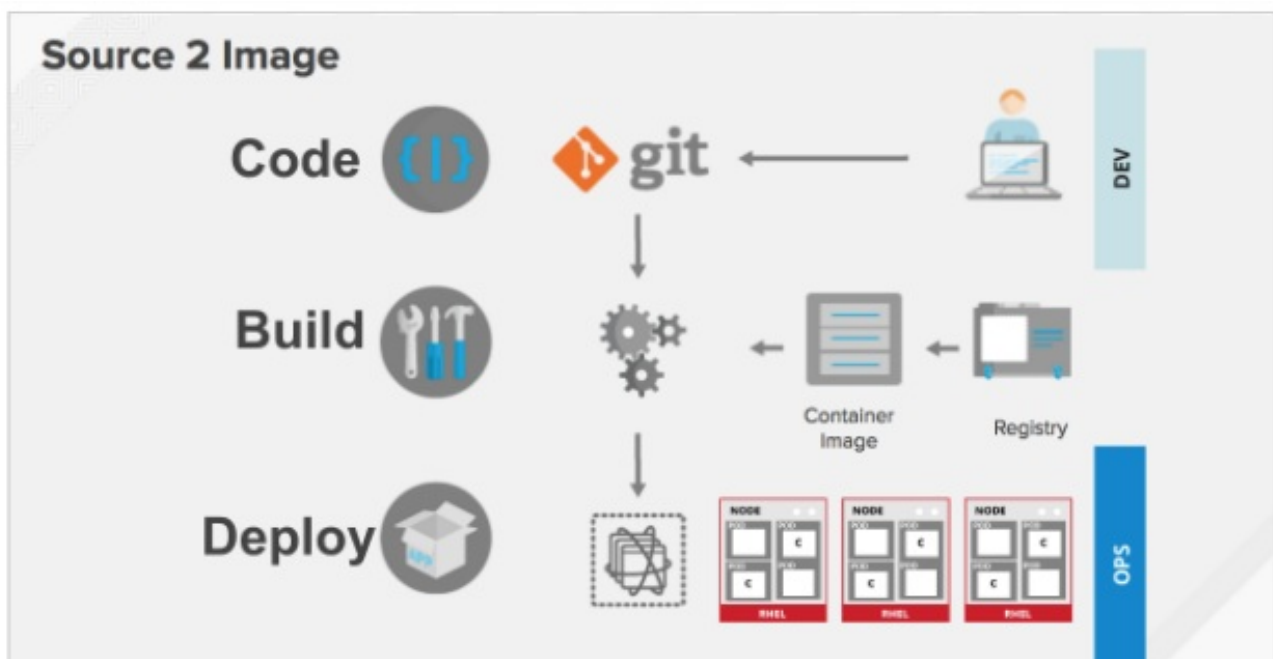
It is also important to maintain the immutability of your containers. You should not patch running containers, but rebuild and redeploy them.

### 5.2. BUILD MANAGEMENT AND SECURITY

You can use source-to-Image (S2I) to combine source code and base images. *Builder images* make use of S2I to enable your development and operations teams to collaborate on a reproducible build environment.

When developers commit code with Git for an application using build images, OpenShift Container Platform can perform the following functions:

- Trigger, either via webhooks on the code repository or other automated continuous integration (CI) process, to automatically assemble a new image from available artifacts, the S2I builder image, and the newly committed code.
- Automatically deploy the newly-built image for testing.
- Promote the tested image to production where it can be automatically deployed using CI process.



You can use OpenShift Container Registry to manage access to final images. Both S2I and native build images are automatically pushed to the OpenShift Container Registry.

In addition to the included Jenkins for CI, you can also integrate your own build / CI environment with OpenShift Container Platform using RESTful APIs, as well as use any API-compliant image registry.

### Further Reading

- *OpenShift Container Platform Developer Guide*
  - [How Builds Work](#)
  - [Triggering Builds](#)
- *OpenShift Container Platform Architecture*: [Source-to-Image \(S2I\) Build](#)
- *OpenShift Container Platform Using Images*: [Other Images](#) → [Jenkins](#)

## 5.3. SECURING INPUTS DURING BUILDS

In some scenarios, build operations require credentials to access dependent resources, but it is undesirable for those credentials to be available in the final application image produced by the build. You can define input secrets for this purpose.

For example, when building a Node.js application, you can set up your private mirror for Node.js modules. In order to download modules from that private mirror, you have to supply a custom `.npmrc` file for the build that contains a URL, user name, and password. For security reasons, you do not want to expose your credentials in the application image.

Using this example scenario, you can add an input secret to a new **BuildConfig**:

1. Create the secret, if it does not exist:

```
$ oc secrets new secret-npmrc .npmrc=~/.npmrc
```

This creates a new secret named **secret-npmrc**, which contains the base64 encoded content of the `~/.npmrc` file.

2. Add the secret to the **source** section in the existing **BuildConfig**:

```
source:
  git:
    uri: https://github.com/sclorg/nodejs-ex.git
  secrets:
    - secret:
        name: secret-npmrc
```

3. To include the secret in a new **BuildConfig**, run the following command:

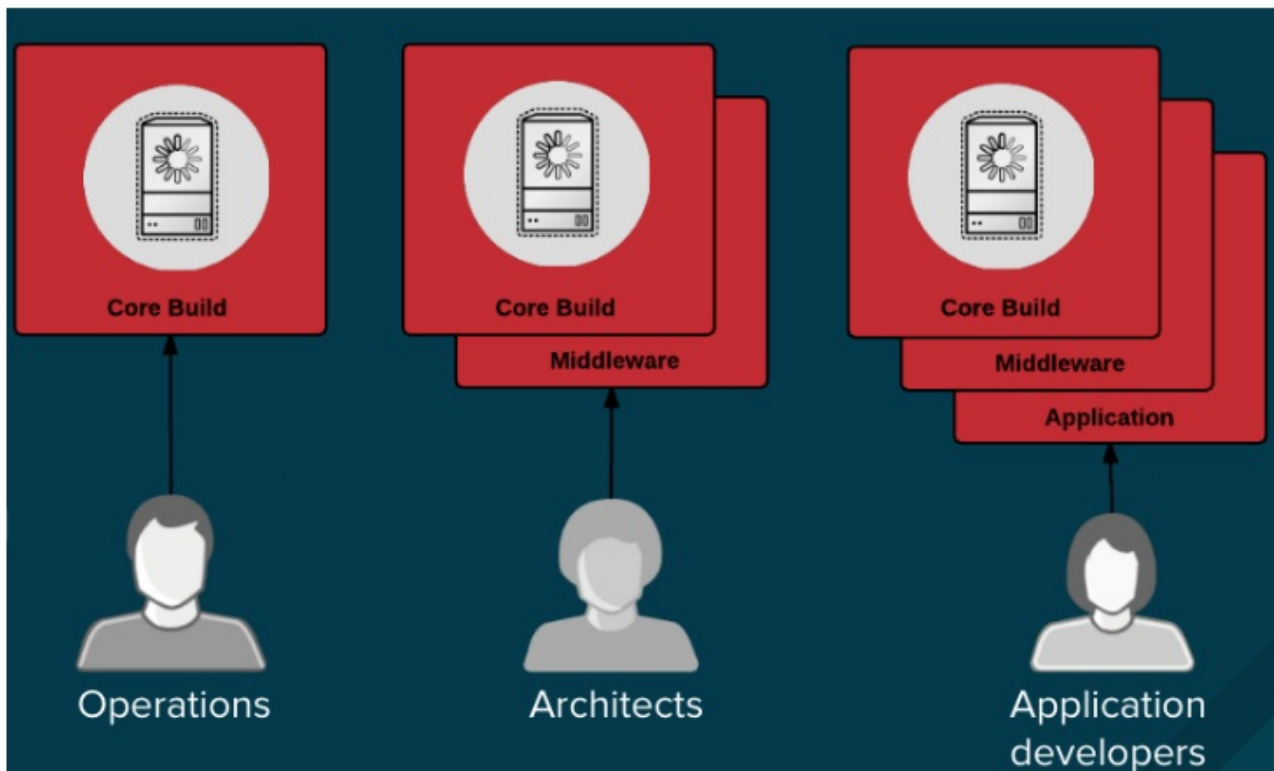
```
$ oc new-build \
  openshift/nodejs-010-centos7~https://github.com/sclorg/nodejs-
  ex.git \
  --build-secret secret-npmrc
```

### Further Reading

- *OpenShift Container Platform Developer Guide*: [Input Secrets](#)

## 5.4. DESIGNING YOUR BUILD PROCESS

You can design your container image management and build process to use container layers so that you can separate control.



For example, an operations team manages base images, while architects manage middleware, runtimes, databases, and other solutions. Developers can then focus on application layers and just write code.

Because new vulnerabilities are identified daily, you need to proactively check container content over time. To do this, you should integrate automated security testing into your build or CI process. For example:

- SAST / DAST – Static and Dynamic security testing tools.
- Scanners for real-time checking against known vulnerabilities. Tools like these catalog the open source packages in your container, notify you of any known vulnerabilities, and update you when new vulnerabilities are discovered in previously scanned packages.

Your CI process should include policies that flag builds with issues discovered by security scans so that your team can take appropriate action to address those issues. You should sign your custom built containers to ensure that nothing is tampered with between build and deployment.

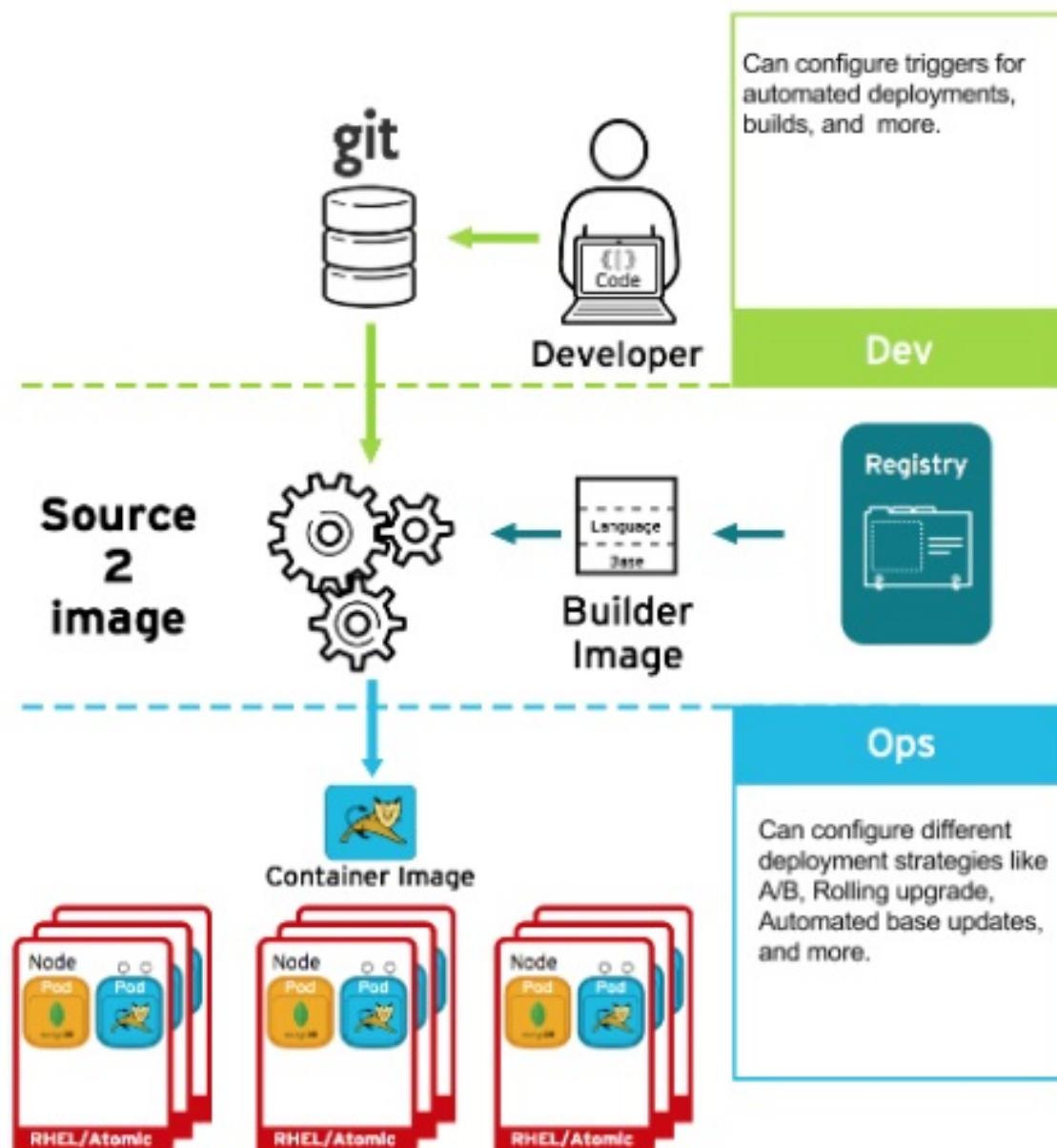
### Further Reading

- *Red Hat Enterprise Linux Atomic Host Managing Containers:* [Signing Container Images](#)

## CHAPTER 6. DEPLOYMENT

### 6.1. CONTROLLING WHAT CAN BE DEPLOYED IN A CONTAINER

If something happens during the build process, or if a vulnerability is discovered after an image has been deployed, you can use tooling for automated, policy-based deployment. You can use triggers to rebuild and replace images instead of patching running containers, which is not recommended.



For example, you build an application using three container image layers: core, middleware, and applications. An issue is discovered in the core image and that image is rebuilt. After the build is complete, the image is pushed to the OpenShift Container Registry. OpenShift Container Platform detects that the image has changed and automatically rebuilds and deploys the application image, based on the defined triggers. This change incorporates the fixed libraries and ensures that the production code is identical to the most current image.

The `oc set triggers` command can be used to set a deployment trigger for a deployment configuration. For example, to set an **ImageChangeTrigger** in a deployment configuration called **frontend**:

```
$ oc set triggers dc/frontend \
  --from-image=myproject/origin-ruby-sample:latest \
  -c helloworld
```

### Further Reading

- *OpenShift Container Platform Developer Guide*
  - [How Deployments Work](#)
  - [Setting Deployment Triggers](#)
  - [Application Life Cycle Management → Promoting Applications Across Environments](#)

## 6.2. SECRETS AND CONFIGMAPS

The **Secret** object type provides a mechanism to hold sensitive information such as passwords, OpenShift Container Platform client configuration files, ***dockercfg*** files, and private source repository credentials. Secrets decouple sensitive content from pods. You can mount secrets into containers using a volume plug-in or the system can use secrets to perform actions on behalf of a pod.

For example, to add a secret to your deployment configuration using the web console so that it can access a private image repository:

1. Create a new project.
2. Navigate to **Resources** → **Secrets** and create a new secret. Set **Secret Type** to **Image Secret** and **Authentication Type** to **Image Registry Credentials** to enter credentials for accessing a private image repository.
3. When creating a deployment configuration (for example, from the **Add to Project** → **Deploy Image** page), set the **Pull Secret** to your new secret.

**ConfigMaps** are similar to secrets, but are designed to support working with strings that do not contain sensitive information. The **ConfigMap** object holds key-value pairs of configuration data that can be consumed in pods or used to store configuration data for system components such as controllers.

### Further Reading

- *OpenShift Container Platform Developer Guide*
  - [Secrets](#)
  - [ConfigMaps](#)

## 6.3. SECURITY CONTEXT CONSTRAINTS (SCCS)

You can use *security context constraints* (SCCs) to define a set of conditions that a pod (a collection of containers) must run with in order to be accepted into the system.

Some aspects that can be managed by SCCs include:

- Running of privileged containers.
- Capabilities a container can request to be added.

- Use of host directories as volumes.
- SELinux context of the container.
- Container user ID.

If you have the required permissions, you can adjust the default SCC policies to be more permissive.

#### Further Reading

- *OpenShift Container Platform Architecture*: [Security Context Constraints](#)
- *OpenShift Container Platform Installation and Configuration*: [Security Warning](#)
  - Discusses privileged containers

## 6.4. CONTINUOUS DEPLOYMENT TOOLING

You can integrate your own continuous deployment (CD) tooling with OpenShift Container Platform.

By leveraging CI/CD and OpenShift Container Platform, you can automate the process of rebuilding the application to incorporate the latest fixes, testing, and ensuring that it is deployed everywhere within the environment.



## CHAPTER 7. SECURING THE CONTAINER PLATFORM

### 7.1. CONTAINER ORCHESTRATION

APIs are key to automating container management at scale. APIs are used to:

- Validate and configure the data for pods, services, and replication controllers.
- Perform project validation on incoming requests and invoke triggers on other major system components.

#### Further Reading

- OpenShift Container Platform *Architecture*: [How Is OpenShift Container Platform Secured?](#)

### 7.2. AUTHENTICATION AND AUTHORIZATION

#### 7.2.1. Controlling Access Using OAuth

You can use API access control via authentication and authorization for securing your container platform. The OpenShift Container Platform master includes a built-in OAuth server. Users can obtain OAuth access tokens to authenticate themselves to the API.

As an administrator, you can configure OAuth to authenticate using an *identity provider*, such as LDAP, GitHub, or Google. The Deny All identity provider is used by default for new OpenShift Container Platform deployments, but you can configure this at initial installation time or post-installation. See [Configuring authentication and user agent](#) for a full list of identity providers.

For example, to configure the GitHub identity provider post-installation:

1. Edit the master configuration file at `/etc/origin/master-config.yaml`.
2. Modify the `oauthConfig` stanza per the following:

```
oauthConfig:
  ...
  identityProviders:
  - name: github
    challenge: false
    login: true
    mappingMethod: claim
    provider:
      apiVersion: v1
      kind: GitHubIdentityProvider
      clientID: ...
      clientSecret: ...
      organizations:
      - myorganization1
      - myorganization2
      teams:
      - myorganization1/team-a
      - myorganization2/team-b
```

**NOTE**

See the [GitHub](#) section in Configuring Authentication for more detailed information and usage.

3. After saving your changes, restart the master service for the changes to take effect:

```
# systemctl restart atomic-openshift-master
```

**Further Reading**

- *OpenShift Container Platform Architecture*
  - [Additional Concepts → Authentication](#)
  - [Additional Concepts → Authorization](#)
- *OpenShift Container Platform CLI Reference*
- *OpenShift Container Platform Developer Guide: [CLI Authentication](#)*

**7.2.2. API Access Control and Management**

Applications can have multiple, independent API services which have different endpoints that require management. OpenShift Container Platform includes a containerized version of the 3scale API gateway so that you can manage your APIs and control access.

3scale gives you a variety of standard options for API authentication and security, which can be used alone or in combination to issue credentials and control access: Standard API keys, Application ID and key pair, and OAuth 2.0.

You can restrict access to specific end points, methods, and services and apply access policy for groups of users. Application plans allow you to set rate limits for API usage and control traffic flow for groups of developers.

For a tutorial on using APIcast v2, the containerized 3scale API Gateway, see [Running APIcast on Red Hat OpenShift](#).

**7.2.3. Red Hat SSO**

The Red Hat Single Sign-On (RH-SSO) Server enables you to secure your applications by providing Web SSO capabilities based on standards, including SAML 2.0, OpenID Connect, and OAuth 2.0. The Server can act as a SAML or OpenID Connect–based identity provider (IdP), mediating with your enterprise user directory or third-party identity provider for identity information and your applications using standards-based tokens. You can integrate Red Hat SSO with LDAP-based directory services including Microsoft Active Directory and Red Hat Enterprise Linux Identity Management.

See [Red Hat JBoss SSO for OpenShift](#) documentation for usage tutorials.

**7.2.4. Secure Self-service Web Console**

OpenShift Container Platform provides a self-service web console to ensure that teams do not access other environments without authorization. OpenShift Container Platform ensures a secure multi-tenant master by providing the following:

- Access to the master uses Transport Layer Security (TLS)
- Access to the API Server uses X.509 certificates or OAuth access tokens
- Project quota limits the damage that a rogue token could do
- Etcd is not exposed directly to the cluster

### Further Reading

- *OpenShift Container Platform Architecture*: [Infrastructure Components](#) → [Web Console](#)
- *OpenShift Container Platform Developer Guide*: [Web Console Authentication](#)

## 7.3. MANAGING CERTIFICATES FOR THE PLATFORM

OpenShift Container Platform has multiple components within its framework that use REST-based HTTPS communication leveraging encryption via TLS certificates. OpenShift Container Platform's Ansible-based installer configures these certificates during installation. There are some primary components that generate this traffic:

- masters (API server and controllers)
- etcd
- nodes
- registry
- router

### 7.3.1. Configuring Custom Certificates

You can configure custom serving certificates for the public host names of the API server and web console during initial installation or when redeploying certificates. You can also use a custom CA.

During initial advanced installations using Ansible playbooks, custom certificates can be configured using the **openshift\_master\_overwrite\_named\_certificates** Ansible variable, which is configurable in the inventory file. For example:

```
openshift_master_named_certificates=[{"certfile":
"/path/on/host/to/custom1.crt", "keyfile": "/path/on/host/to/custom1.key",
"cafile": "/path/on/host/to/custom-ca1.crt"}]
```

See the Advanced Installation's [Configuring Custom Certificates](#) for more options and instructions on how to run the installation playbook.

The installer provides Ansible playbooks for checking on the expiration dates of all cluster certificates. Additional playbooks can automatically redeploy all certificates at once using the current CA, redeploy specific certificates only, or redeploy a newly generated or custom CA on its own. See [Redeploying Certificates](#) for more on these playbooks.

### Further Reading

- *OpenShift Container Platform Installation and Configuration*

- [Configuring Custom Certificates](#)
- [Checking Certificate Expirations](#)
- [Redeploying Certificates](#)

## CHAPTER 8. NETWORK SECURITY

### 8.1. NETWORK NAMESPACES

OpenShift Container Platform uses software-defined networking (SDN) to provide a unified cluster network that enables communication between containers across the cluster.

Using network namespaces, you can isolate pod networks. Each pod gets its own IP and port range to bind to, thereby isolating pod networks from each other on the node. Pods from different projects cannot send packets to or receive packets from pods and services of a different project. You can use this to isolate developer, test and production environments within a cluster.

OpenShift Container Platform also provides the ability to control egress traffic using either a router or firewall method. For example, you can use IP whitelisting to control database access.

#### Further Reading

- *OpenShift Container Platform Architecture: [Networking](#)*
- *OpenShift Container Platform Cluster Administration: [Managing Networking](#)*
- *Red Hat Enterprise Linux Atomic Host Managing Containers: [Running Super-Privileged Containers](#)*

### 8.2. ISOLATING APPLICATIONS

OpenShift Container Platform enables you to segment network traffic on a single cluster to make multi-tenant clusters that isolate users, teams, applications, and environments.

For example, to isolate a project network in the cluster and vice versa, run:

```
$ oc adm pod-network isolate-projects <project1> <project2>
```

In the above example, all of the pods and services in **<project1>** and **<project2>** can not access any pods and services from other non-global projects in the cluster and vice versa.

## CHAPTER 9. ATTACHED STORAGE

### 9.1. PERSISTENT VOLUME PLUG-INS

Containers are useful for both stateless and stateful applications. Protecting attached storage is a key element of securing stateful services.

OpenShift Container Platform provides plug-ins for multiple types of storage, including NFS, AWS Elastic Block Stores (EBS), GCE Persistent Disks, GlusterFS, iSCSI, RADOS (Ceph) and Cinder. Data in transit is encrypted via HTTPS for all OpenShift Container Platform components communicating with each other.

You can mount **PersistentVolume** (PV) on a host in any way supported by your storage type. Different types of storage have different capabilities and each PV's access modes are set to the specific modes supported by that particular volume.

For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only. Each PV has its own set of access modes describing that specific PV's capabilities, such as **ReadWriteOnce**, **ReadOnlyMany**, and **ReadWriteMany**.

#### Further Reading

- *OpenShift Container Platform Architecture*: [Additional Concepts → Storage](#)
- *OpenShift Container Platform Installation and Configuration*: [Configuring Persistent Storage → Volume Security](#)

### 9.2. SHARED STORAGE

For shared storage providers like NFS, Ceph, and Gluster, the PV registers its group ID (GID) as an annotation on the PV resource. Then, when the PV is claimed by the pod, the annotated GID is added to the supplemental groups of the pod, giving that pod access to the contents of the shared storage.

#### Further Reading

- *OpenShift Container Platform Installation and Configuration*
  - [Persistent Storage Using NFS](#)
  - [Persistent Storage Using Ceph RBD](#)
  - [Persistent Storage Using GlusterFS](#)

### 9.3. BLOCK STORAGE

For block storage providers like AWS Elastic Block Store (EBS), GCE Persistent Disks, and iSCSI, OpenShift Container Platform uses SELinux capabilities to secure the root of the mounted volume for non-privileged pods, making the mounted volume owned by and only visible to the container with which it is associated.

#### Further Reading

- *OpenShift Container Platform Installation and Configuration*
  - [Persistent Storage Using AWS Elastic Block Storage](#)

- [Persistent Storage Using GCE Persistent Disk](#)
- [Persistent Storage Using iSCSI](#)

## CHAPTER 10. REVISION HISTORY: CONTAINER SECURITY GUIDE

### 10.1. MON MAY 22 2017

Affected Topic	Description of Change
All topics	New guide providing a high-level walkthrough of the container security measures available in OpenShift Container Platform.