



OpenShift Container Platform 4.6

Security and compliance

Learning about and managing security for OpenShift Container Platform

OpenShift Container Platform 4.6 Security and compliance

Learning about and managing security for OpenShift Container Platform

Legal Notice

Copyright © 2023 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, the Red Hat 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 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

This document discusses container security, configuring certificates, and enabling encryption to help secure the cluster.

Table of Contents

CHAPTER 1. OPENSIFT CONTAINER PLATFORM SECURITY AND COMPLIANCE	9
1.1. SECURITY OVERVIEW	9
Container security	9
Auditing	9
Certificates	9
Encrypting data	10
Vulnerability scanning	10
1.2. COMPLIANCE OVERVIEW	10
Compliance checking	10
File integrity checking	10
1.3. ADDITIONAL RESOURCES	10
CHAPTER 2. CONTAINER SECURITY	11
2.1. UNDERSTANDING CONTAINER SECURITY	11
2.1.1. What are containers?	12
2.1.2. What is OpenShift Container Platform?	12
2.2. UNDERSTANDING HOST AND VM SECURITY	13
2.2.1. Securing containers on Red Hat Enterprise Linux CoreOS (RHCOS)	13
2.2.2. Comparing virtualization and containers	14
2.2.3. Securing OpenShift Container Platform	15
2.3. HARDENING RHCOS	15
2.3.1. Choosing what to harden in RHCOS	16
2.3.2. Choosing how to harden RHCOS	16
2.3.2.1. Hardening before installation	16
2.3.2.2. Hardening during installation	16
2.3.2.3. Hardening after the cluster is running	16
2.4. CONTAINER IMAGE SIGNATURES	17
2.4.1. Enabling signature verification for Red Hat Container Registries	17
2.4.2. Verifying the signature verification configuration	20
2.4.3. Additional resources	25
2.5. UNDERSTANDING COMPLIANCE	25
2.5.1. Understanding compliance and risk management	25
2.6. SECURING CONTAINER CONTENT	25
2.6.1. Securing inside the container	25
2.6.2. Creating redistributable images with UBI	26
2.6.3. Security scanning in RHEL	26
2.6.3.1. Scanning OpenShift images	26
2.6.4. Integrating external scanning	27
2.6.4.1. Image metadata	27
2.6.4.1.1. Example annotation keys	27
2.6.4.1.2. Example annotation values	28
2.6.4.2. Annotating image objects	29
2.6.4.2.1. Example annotate CLI command	29
2.6.4.3. Controlling pod execution	30
2.6.4.3.1. Example annotation	30
2.6.4.4. Integration reference	30
2.6.4.4.1. Example REST API call	30
2.7. USING CONTAINER REGISTRIES SECURELY	30
2.7.1. Knowing where containers come from?	31
2.7.2. Immutable and certified containers	31
2.7.3. Getting containers from Red Hat Registry and Ecosystem Catalog	31

2.7.4. OpenShift Container Registry	32
2.7.5. Storing containers using Red Hat Quay	32
2.8. SECURING THE BUILD PROCESS	33
2.8.1. Building once, deploying everywhere	33
2.8.2. Managing builds	34
2.8.3. Securing inputs during builds	34
2.8.4. Designing your build process	35
2.8.5. Building Knative serverless applications	36
2.8.6. Additional resources	36
2.9. DEPLOYING CONTAINERS	36
2.9.1. Controlling container deployments with triggers	36
2.9.2. Controlling what image sources can be deployed	37
2.9.3. Using signature transports	39
2.9.4. Creating secrets and config maps	39
2.9.5. Automating continuous deployment	40
2.10. SECURING THE CONTAINER PLATFORM	40
2.10.1. Isolating containers with multitenancy	40
2.10.2. Protecting control plane with admission plug-ins	41
2.10.2.1. Security context constraints (SCCs)	41
2.10.2.2. Granting roles to service accounts	42
2.10.3. Authentication and authorization	42
2.10.3.1. Controlling access using OAuth	42
2.10.3.2. API access control and management	42
2.10.3.3. Red Hat Single Sign-On	42
2.10.3.4. Secure self-service web console	42
2.10.4. Managing certificates for the platform	43
2.10.4.1. Configuring custom certificates	43
2.11. SECURING NETWORKS	44
2.11.1. Using network namespaces	44
2.11.2. Isolating pods with network policies	44
2.11.3. Using multiple pod networks	44
2.11.4. Isolating applications	44
2.11.5. Securing ingress traffic	44
2.11.6. Securing egress traffic	45
2.12. SECURING ATTACHED STORAGE	45
2.12.1. Persistent volume plug-ins	45
2.12.2. Shared storage	46
2.12.3. Block storage	46
2.13. MONITORING CLUSTER EVENTS AND LOGS	46
2.13.1. Watching cluster events	47
2.13.2. Logging	48
2.13.3. Audit logs	48
CHAPTER 3. CONFIGURING CERTIFICATES	49
3.1. REPLACING THE DEFAULT INGRESS CERTIFICATE	49
3.1.1. Understanding the default ingress certificate	49
3.1.2. Replacing the default ingress certificate	49
Additional resources	50
3.2. ADDING API SERVER CERTIFICATES	50
3.2.1. Add an API server named certificate	50
3.3. SECURING SERVICE TRAFFIC USING SERVICE SERVING CERTIFICATE SECRETS	52
3.3.1. Understanding service serving certificates	52
3.3.2. Add a service certificate	53

3.3.3. Add the service CA bundle to a config map	54
3.3.4. Add the service CA bundle to an API service	55
3.3.5. Add the service CA bundle to a custom resource definition	55
3.3.6. Add the service CA bundle to a mutating webhook configuration	56
3.3.7. Add the service CA bundle to a validating webhook configuration	57
3.3.8. Manually rotate the generated service certificate	58
3.3.9. Manually rotate the service CA certificate	59
3.4. UPDATING THE CA BUNDLE	60
3.4.1. Understanding the CA Bundle certificate	60
3.4.2. Replacing the CA Bundle certificate	60
Additional resources	61
CHAPTER 4. CERTIFICATE TYPES AND DESCRIPTIONS	62
4.1. USER-PROVIDED CERTIFICATES FOR THE API SERVER	62
4.1.1. Purpose	62
4.1.2. Location	62
4.1.3. Management	62
4.1.4. Expiration	62
4.1.5. Customization	62
Additional resources	62
4.2. PROXY CERTIFICATES	62
4.2.1. Purpose	62
Additional resources	63
4.2.2. Managing proxy certificates during installation	63
4.2.3. Location	63
4.2.4. Expiration	64
4.2.5. Services	64
4.2.6. Management	64
4.2.7. Customization	64
4.2.8. Renewal	65
4.3. SERVICE CA CERTIFICATES	65
4.3.1. Purpose	65
4.3.2. Expiration	65
4.3.3. Management	66
4.3.4. Services	66
Additional resources	67
4.4. NODE CERTIFICATES	67
4.4.1. Purpose	67
4.4.2. Management	67
Additional resources	67
4.5. BOOTSTRAP CERTIFICATES	67
4.5.1. Purpose	67
4.5.2. Management	67
4.5.3. Expiration	67
4.5.4. Customization	68
4.6. ETCDCERTIFICATES	68
4.6.1. Purpose	68
4.6.2. Expiration	68
4.6.3. Management	68
4.6.4. Services	68
Additional resources	68
4.7. OLM CERTIFICATES	68
4.7.1. Management	68

4.8. USER-PROVIDED CERTIFICATES FOR DEFAULT INGRESS	69
4.8.1. Purpose	69
4.8.2. Location	69
4.8.3. Management	69
4.8.4. Expiration	69
4.8.5. Services	69
4.8.6. Customization	69
Additional resources	69
4.9. INGRESS CERTIFICATES	70
4.9.1. Purpose	70
4.9.2. Location	70
4.9.3. Workflow	70
4.9.4. Expiration	72
4.9.5. Services	72
4.9.6. Management	72
4.9.7. Renewal	72
4.10. MONITORING AND CLUSTER LOGGING OPERATOR COMPONENT CERTIFICATES	73
4.10.1. Expiration	73
4.10.2. Management	73
4.11. CONTROL PLANE CERTIFICATES	73
4.11.1. Location	73
4.11.2. Management	73
CHAPTER 5. COMPLIANCE OPERATOR	74
5.1. COMPLIANCE OPERATOR RELEASE NOTES	74
5.1.1. OpenShift Compliance Operator 0.1.53	74
5.1.1.1. Bug fixes	74
5.1.1.2. Known issue	75
5.1.2. OpenShift Compliance Operator 0.1.52	75
5.1.2.1. New features and enhancements	75
5.1.2.2. Bug fixes	75
5.1.2.3. Known issue	76
5.1.3. OpenShift Compliance Operator 0.1.49	76
5.1.3.1. Bug fixes	76
5.1.4. OpenShift Compliance Operator 0.1.48	77
5.1.4.1. Bug fixes	77
5.1.5. OpenShift Compliance Operator 0.1.47	77
5.1.5.1. New features and enhancements	77
5.1.5.2. Bug fixes	77
5.1.6. OpenShift Compliance Operator 0.1.44	78
5.1.6.1. New features and enhancements	78
5.1.6.2. Templating and variable use	79
5.1.6.3. Bug fixes	79
5.1.7. Release Notes for Compliance Operator 0.1.39	79
5.1.7.1. New features and enhancements	80
5.1.8. Additional resources	80
5.2. SUPPORTED COMPLIANCE PROFILES	80
5.2.1. Compliance profiles	80
5.2.2. Additional resources	82
5.3. INSTALLING THE COMPLIANCE OPERATOR	82
5.3.1. Installing the Compliance Operator through the web console	82
5.3.2. Installing the Compliance Operator using the CLI	83
5.3.3. Additional resources	84

5.4. COMPLIANCE OPERATOR SCANS	84
5.4.1. Running compliance scans	85
5.4.2. Scheduling the result server pod on a worker node	87
5.5. UNDERSTANDING THE COMPLIANCE OPERATOR	88
5.5.1. Compliance Operator profiles	89
5.6. MANAGING THE COMPLIANCE OPERATOR	92
5.6.1. Updating security content	92
5.6.2. Using image streams	92
5.6.3. ProfileBundle CR example	93
5.6.4. Additional resources	94
5.7. TAILORING THE COMPLIANCE OPERATOR	94
5.7.1. Using tailored profiles	94
5.8. RETRIEVING COMPLIANCE OPERATOR RAW RESULTS	96
5.8.1. Obtaining Compliance Operator raw results from a persistent volume	97
5.9. MANAGING COMPLIANCE OPERATOR RESULT AND REMEDIATION	98
5.9.1. Filters for compliance check results	98
5.9.2. Reviewing a remediation	99
5.9.3. Applying remediation when using customized machine config pools	100
5.9.4. Applying a remediation	101
5.9.5. Remediating a platform check manually	102
5.9.6. Updating remediations	103
5.9.7. Unapplying a remediation	104
5.9.8. Removing a KubeletConfig remediation	104
5.9.9. Inconsistent ComplianceScan	106
5.9.10. Additional resources	107
5.10. PERFORMING ADVANCED COMPLIANCE OPERATOR TASKS	107
5.10.1. Using the ComplianceSuite and ComplianceScan objects directly	107
5.10.2. Using raw tailored profiles	107
5.10.3. Performing a rescan	108
5.10.4. Setting custom storage size for results	109
5.10.4.1. Using custom result storage values	109
5.10.5. Applying remediations generated by suite scans	110
5.10.6. Automatically update remediations	110
5.11. TROUBLESHOOTING THE COMPLIANCE OPERATOR	110
5.11.1. Anatomy of a scan	111
5.11.1.1. Compliance sources	111
5.11.1.2. The ScanSetting and ScanSettingBinding objects lifecycle and debugging	111
5.11.1.3. ComplianceSuite custom resource lifecycle and debugging	112
5.11.1.4. ComplianceScan custom resource lifecycle and debugging	113
5.11.1.4.1. Pending phase	113
5.11.1.4.2. Launching phase	113
5.11.1.4.3. Running phase	113
5.11.1.4.4. Aggregating phase	114
5.11.1.4.5. Done phase	115
5.11.1.5. ComplianceRemediation controller lifecycle and debugging	115
5.11.1.6. Useful labels	117
5.11.2. Getting support	117
5.12. UNINSTALLING THE COMPLIANCE OPERATOR	117
5.12.1. Uninstalling the OpenShift Compliance Operator from OpenShift Container Platform	117
5.13. UNDERSTANDING THE CUSTOM RESOURCE DEFINITIONS	119
5.13.1. CRDs workflow	119
5.13.2. Defining the compliance scan requirements	119
5.13.2.1. ProfileBundle object	119

5.13.2.2. Profile object	120
5.13.2.3. Rule object	121
5.13.2.4. TailoredProfile object	122
5.13.3. Configuring the compliance scan settings	123
5.13.3.1. ScanSetting object	123
5.13.4. Processing the compliance scan requirements with compliance scans settings	124
5.13.4.1. ScanSettingBinding object	124
5.13.5. Tracking the compliance scans	125
5.13.5.1. ComplianceSuite object	125
5.13.5.2. Advanced ComplianceScan Object	126
5.13.6. Viewing the compliance results	127
5.13.6.1. ComplianceCheckResult object	127
5.13.6.2. ComplianceRemediation object	128
CHAPTER 6. FILE INTEGRITY OPERATOR	130
6.1. FILE INTEGRITY OPERATOR RELEASE NOTES	130
6.1.1. OpenShift File Integrity Operator 0.1.30	130
6.1.1.1. Bug fixes	130
6.1.2. OpenShift File Integrity Operator 0.1.24	130
6.1.2.1. New features and enhancements	130
6.1.2.2. Bug fixes	130
6.1.3. OpenShift File Integrity Operator 0.1.22	130
6.1.3.1. Bug fixes	131
6.1.4. OpenShift File Integrity Operator 0.1.21	131
6.1.4.1. New features and enhancements	131
6.1.4.2. Bug fixes	131
6.1.5. Additional resources	131
6.2. INSTALLING THE FILE INTEGRITY OPERATOR	131
6.2.1. Installing the File Integrity Operator using the web console	131
6.2.2. Installing the File Integrity Operator using the CLI	132
6.2.3. Additional resources	133
6.3. UNDERSTANDING THE FILE INTEGRITY OPERATOR	133
6.3.1. Creating the FileIntegrity custom resource	134
6.3.2. Checking the FileIntegrity custom resource status	134
6.3.3. FileIntegrity custom resource phases	135
6.3.4. Understanding the FileIntegrityNodeStatuses object	135
6.3.5. FileIntegrityNodeStatus CR status types	136
6.3.5.1. FileIntegrityNodeStatus CR success example	136
6.3.5.2. FileIntegrityNodeStatus CR failure status example	137
6.3.6. Understanding events	138
6.4. CONFIGURING THE CUSTOM FILE INTEGRITY OPERATOR	140
6.4.1. Viewing FileIntegrity object attributes	140
6.4.2. Important attributes	140
6.4.3. Examine the default configuration	141
6.4.4. Understanding the default File Integrity Operator configuration	141
6.4.5. Supplying a custom AIDE configuration	142
6.4.6. Defining a custom File Integrity Operator configuration	142
6.4.7. Changing the custom File Integrity configuration	143
6.5. PERFORMING ADVANCED CUSTOM FILE INTEGRITY OPERATOR TASKS	143
6.5.1. Reinitializing the database	143
6.5.2. Machine config integration	144
6.5.3. Exploring the daemon sets	144
6.6. TROUBLESHOOTING THE FILE INTEGRITY OPERATOR	145

6.6.1. General troubleshooting	145
6.6.2. Checking the AIDE configuration	145
6.6.3. Determining the FileIntegrity object's phase	145
6.6.4. Determining that the daemon set's pods are running on the expected nodes	146
CHAPTER 7. VIEWING AUDIT LOGS	147
7.1. ABOUT THE API AUDIT LOG	147
7.2. VIEWING THE AUDIT LOGS	148
7.3. FILTERING AUDIT LOGS	151
7.4. GATHERING AUDIT LOGS	152
7.5. ADDITIONAL RESOURCES	152
CHAPTER 8. CONFIGURING THE AUDIT LOG POLICY	154
8.1. ABOUT AUDIT LOG POLICY PROFILES	154
8.2. CONFIGURING THE AUDIT LOG POLICY	154
CHAPTER 9. CONFIGURING TLS SECURITY PROFILES	156
9.1. UNDERSTANDING TLS SECURITY PROFILES	156
9.2. VIEWING TLS SECURITY PROFILE DETAILS	157
9.3. CONFIGURING THE TLS SECURITY PROFILE FOR THE INGRESS CONTROLLER	159
9.4. CONFIGURING THE TLS SECURITY PROFILE FOR THE CONTROL PLANE	161
CHAPTER 10. CONFIGURING SECCOMP PROFILES	165
10.1. ENABLING THE DEFAULT SECCOMP PROFILE FOR ALL PODS	165
10.2. CONFIGURING A CUSTOM SECCOMP PROFILE	166
10.2.1. Setting up the custom seccomp profile	166
10.2.2. Applying the custom seccomp profile to the workload	167
10.3. ADDITIONAL RESOURCES	167
CHAPTER 11. ALLOWING JAVASCRIPT-BASED ACCESS TO THE API SERVER FROM ADDITIONAL HOSTS	169
11.1. ALLOWING JAVASCRIPT-BASED ACCESS TO THE API SERVER FROM ADDITIONAL HOSTS	169
CHAPTER 12. ENCRYPTING ETCD DATA	171
12.1. ABOUT ETCD ENCRYPTION	171
12.2. ENABLING ETCD ENCRYPTION	171
12.3. DISABLING ETCD ENCRYPTION	172
CHAPTER 13. SCANNING PODS FOR VULNERABILITIES	174
13.1. RUNNING THE RED HAT QUAY CONTAINER SECURITY OPERATOR	174
13.2. QUERYING IMAGE VULNERABILITIES FROM THE CLI	176

CHAPTER 1. OPENSIFT CONTAINER PLATFORM SECURITY AND COMPLIANCE

1.1. SECURITY OVERVIEW

It is important to understand how to properly secure various aspects of your OpenShift Container Platform cluster.

Container security

A good starting point to understanding OpenShift Container Platform security is to review the concepts in [Understanding container security](#). This and subsequent sections provide 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. These sections also include information on the following topics:

- Why container security is important and how it compares with existing security standards.
- Which container security measures are provided by the host (RHCOS and 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 through authentication and authorization.
- How networking and attached storage are secured in OpenShift Container Platform.
- Containerized solutions for API management and SSO.

Auditing

OpenShift Container Platform auditing provides a security-relevant chronological set of records documenting the sequence of activities that have affected the system by individual users, administrators, or other components of the system. Administrators can [configure the audit log policy](#) and [view audit logs](#).

Certificates

Certificates are used by various components to validate access to the cluster. Administrators can [replace the default ingress certificate](#), [add API server certificates](#), or [add a service certificate](#).

You can also review more details about the types of certificates used by the cluster:

- [User-provided certificates for the API server](#)
- [Proxy certificates](#)
- [Service CA certificates](#)
- [Node certificates](#)
- [Bootstrap certificates](#)
- [etcd certificates](#)
- [OLM certificates](#)

- [User-provided certificates for default ingress](#)
- [Ingress certificates](#)
- [Monitoring and cluster logging Operator component certificates](#)
- [Control plane certificates](#)

Encrypting data

You can [enable etcd encryption](#) for your cluster to provide an additional layer of data security. For example, it can help protect the loss of sensitive data if an etcd backup is exposed to the incorrect parties.

Vulnerability scanning

Administrators can use the Red Hat Quay Container Security Operator to run [vulnerability scans](#) and review information about detected vulnerabilities.

1.2. COMPLIANCE OVERVIEW

For many OpenShift Container Platform customers, regulatory readiness, or compliance, on some level is required before any systems can be put into production. That regulatory readiness can be imposed by national standards, industry standards, or the organization's corporate governance framework.

Compliance checking

Administrators can use the [Compliance Operator](#) to run compliance scans and recommend remediations for any issues found.

File integrity checking

Administrators can use the [File Integrity Operator](#) to continually run file integrity checks on cluster nodes and provide a log of files that have been modified.

1.3. ADDITIONAL RESOURCES

- [Understanding authentication](#)
- [Configuring the internal OAuth server](#)
- [Understanding identity provider configuration](#)
- [Using RBAC to define and apply permissions](#)
- [Managing security context constraints](#)

CHAPTER 2. CONTAINER SECURITY

2.1. UNDERSTANDING CONTAINER SECURITY

Securing a containerized application relies on multiple levels of security:

- Container security begins with a trusted base container image and continues through the container build process as it moves through your CI/CD pipeline.



IMPORTANT

Image streams by default do not automatically update. This default behavior might create a security issue because security updates to images referenced by an image stream do not automatically occur. For information about how to override this default behavior, see [Configuring periodic importing of imagestreamtags](#).

- When a container is deployed, its security depends on it running on secure operating systems and networks, and establishing firm boundaries between the container itself and the users and hosts that interact with it.
- Continued security relies on being able to scan container images for vulnerabilities and having an efficient way to correct and replace vulnerable images.

Beyond what a platform such as OpenShift Container Platform offers out of the box, your organization will likely have its own security demands. Some level of compliance verification might be needed before you can even bring OpenShift Container Platform into your data center.

Likewise, you may need to add your own agents, specialized hardware drivers, or encryption features to OpenShift Container Platform, before it can meet your organization's security standards.

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. It then points you to specific OpenShift Container Platform documentation to help you achieve those security measures.

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 (RHCOS and 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 through authentication and authorization.
- How networking and attached storage are secured in OpenShift Container Platform.
- Containerized solutions for API management and SSO.

The goal of this guide is to understand the incredible security benefits of using OpenShift Container Platform for your containerized workloads and how the entire Red Hat ecosystem plays a part in making

and keeping containers secure. It will also help you understand how you can engage with the OpenShift Container Platform to achieve your organization's security goals.

2.1.1. 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. A container might be part of a larger application that works closely with other containers.

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 operating system 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

See [Understanding Linux containers](#) from the Red Hat Customer Portal to find out more about Linux containers. To learn about RHEL container tools, see [Building, running, and managing containers](#) in the RHEL product documentation.

2.1.2. What is OpenShift Container Platform?

Automating how containerized applications are deployed, run, and managed is the job of a platform such as OpenShift Container Platform. At its core, OpenShift Container Platform relies on the Kubernetes project to provide the engine for orchestrating containers across many nodes in scalable data centers.

Kubernetes is a project, which can run using different operating systems and add-on components that offer no guarantees of supportability from the project. As a result, the security of different Kubernetes platforms can vary.

OpenShift Container Platform is designed to lock down Kubernetes security and integrate the platform with a variety of extended components. To do this, OpenShift Container Platform draws on the extensive Red Hat ecosystem of open source technologies that include the operating systems, authentication, storage, networking, development tools, base container images, and many other components.

OpenShift Container Platform can leverage Red Hat's experience in uncovering and rapidly deploying fixes for vulnerabilities in the platform itself as well as the containerized applications running on the platform. Red Hat's experience also extends to efficiently integrating new components with OpenShift Container Platform as they become available and adapting technologies to individual customer needs.

Additional resources

- [OpenShift Container Platform architecture](#)

- [OpenShift Security Guide](#)

2.2. UNDERSTANDING HOST AND VM SECURITY

Both containers and virtual machines provide ways of separating applications running on a host from the operating system itself. Understanding RHCOS, which is the operating system used by OpenShift Container Platform, will help you see how the host systems protect containers and hosts from each other.

2.2.1. Securing containers on Red Hat Enterprise Linux CoreOS (RHCOS)

Containers simplify the act of deploying many applications to run on the same host, using the same kernel and container runtime to spin up each container. The applications can be owned by many users and, because they are kept separate, can run different, and even incompatible, versions of those applications at the same time without issue.

In Linux, containers are just a special type of process, so securing containers is similar in many ways to securing any other running process. An environment for running containers starts with an operating system that can secure the host kernel from containers and other processes running on the host, as well as secure containers from each other.

Because OpenShift Container Platform 4.6 runs on RHCOS hosts, with the option of using Red Hat Enterprise Linux (RHEL) as worker nodes, the following concepts apply by default to any deployed OpenShift Container Platform cluster. These RHEL security features are at the core of what makes running containers in OpenShift more secure:

- *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 computing resource simultaneously without creating a conflict. Container namespaces that are separate from the host by default include mount table, process table, network interface, user, control group, UTS, and IPC namespaces. Those containers that need direct access to host namespaces need to have elevated permissions to request that access. See [Overview of Containers in Red Hat Systems](#) from the RHEL 7 container documentation for details on the types of namespaces.
- *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.



WARNING

Disabling SELinux on RHCOS nodes is not supported.

- *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. See page 94 of the [OpenShift Security Guide](#) for details about seccomp.

- Deploying containers using *RHCOS* reduces the attack surface by minimizing the host environment and tuning it for containers. The [CRI-O container engine](#) further reduces that attack surface by implementing only those features required by Kubernetes and OpenShift to run and manage containers, as opposed to other container engines that implement desktop-oriented standalone features.

RHCOS is a version of Red Hat Enterprise Linux (RHEL) that is specially configured to work as control plane (master) and worker nodes on OpenShift Container Platform clusters. So RHCOS is tuned to efficiently run container workloads, along with Kubernetes and OpenShift services.

To further protect RHCOS systems in OpenShift Container Platform clusters, most containers, except those managing or monitoring the host system itself, should run as a non-root user. Dropping the privilege level or creating containers with the least amount of privileges possible is recommended best practice for protecting your own OpenShift Container Platform clusters.

Additional resources

- [How nodes enforce resource constraints](#)
- [Managing security context constraints](#)
- [Available platforms](#)
- [Machine requirements for a cluster with user-provisioned infrastructure](#)
- [Choosing how to configure RHCOS](#)
- [Ignition](#)
- [Kernel arguments](#)
- [Kernel modules](#)
- [FIPS cryptography](#)
- [Disk encryption](#)
- [Chrony time service](#)
- [OpenShift Container Platform cluster updates](#)

2.2.2. Comparing virtualization and containers

Traditional virtualization provides another way to keep application environments separate on the same physical host. However, virtual machines work in a different way than containers. Virtualization relies on a hypervisor spinning up guest virtual machines (VMs), each of which has its own operating system (OS), represented by a running kernel, 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 might be able to use hypervisor bugs to gain access to another VM or the host kernel. And, when the OS needs to be patched, it must be patched on all guest VMs using that OS.

Containers can be run inside guest VMs, and there might be use cases where this is desirable. For example, you might be deploying a traditional application in a container, perhaps in order to lift-and-shift an application to the cloud.

Container separation on a single host, however, provides a more lightweight, flexible, and easier-to-scale deployment solution. This deployment model is particularly appropriate for cloud-native applications. Containers are generally much smaller than VMs and consume less memory and CPU.

See [Linux Containers Compared to KVM Virtualization](#) in the RHEL 7 container documentation to learn about the differences between container and VMs.

2.2.3. Securing OpenShift Container Platform

When you deploy OpenShift Container Platform, you have the choice of an installer-provisioned infrastructure (there are several available platforms) or your own user-provisioned infrastructure. Some low-level security-related configuration, such as enabling FIPS compliance or adding kernel modules required at first boot, might benefit from a user-provisioned infrastructure. Likewise, user-provisioned infrastructure is appropriate for disconnected OpenShift Container Platform deployments.

Keep in mind that, when it comes to making security enhancements and other configuration changes to OpenShift Container Platform, the goals should include:

- Keeping the underlying nodes as generic as possible. You want to be able to easily throw away and spin up similar nodes quickly and in prescriptive ways.
- Managing modifications to nodes through OpenShift Container Platform as much as possible, rather than making direct, one-off changes to the nodes.

In pursuit of those goals, most node changes should be done during installation through Ignition or later using MachineConfigs that are applied to sets of nodes by the Machine Config Operator. Examples of security-related configuration changes you can do in this way include:

- Adding kernel arguments
- Adding kernel modules
- Enabling support for FIPS cryptography
- Configuring disk encryption
- Configuring the chrony time service

Besides the Machine Config Operator, there are several other Operators available to configure OpenShift Container Platform infrastructure that are managed by the Cluster Version Operator (CVO). The CVO is able to automate many aspects of OpenShift Container Platform cluster updates.

2.3. HARDENING RHCOS

RHCOS was created and tuned to be deployed in OpenShift Container Platform with few if any changes needed to RHCOS nodes. Every organization adopting OpenShift Container Platform has its own requirements for system hardening. As a RHEL system with OpenShift-specific modifications and features added (such as Ignition, ostree, and a read-only `/usr` to provide limited immutability), RHCOS can be hardened just as you would any RHEL system. Differences lie in the ways you manage the hardening.

A key feature of OpenShift Container Platform and its Kubernetes engine is to be able to quickly scale applications and infrastructure up and down as needed. Unless it is unavoidable, you do not want to make direct changes to RHCOS by logging into a host and adding software or changing settings. You want to have the OpenShift Container Platform installer and control plane manage changes to RHCOS so new nodes can be spun up without manual intervention.

So, if you are setting out to harden RHCOS nodes in OpenShift Container Platform to meet your security needs, you should consider both what to harden and how to go about doing that hardening.

2.3.1. Choosing what to harden in RHCOS

The [RHEL 8 Security Hardening](#) guide describes how you should approach security for any RHEL system.

Use this guide to learn how to approach cryptography, evaluate vulnerabilities, and assess threats to various services. Likewise, you can learn how to scan for compliance standards, check file integrity, perform auditing, and encrypt storage devices.

With the knowledge of what features you want to harden, you can then decide how to harden them in RHCOS.

2.3.2. Choosing how to harden RHCOS

Direct modification of RHCOS systems in OpenShift Container Platform is discouraged. Instead, you should think of modifying systems in pools of nodes, such as worker nodes and control plane nodes (also known as the master nodes). When a new node is needed, in non-bare metal installs, you can request a new node of the type you want and it will be created from an RHCOS image plus the modifications you created earlier.

There are opportunities for modifying RHCOS before installation, during installation, and after the cluster is up and running.

2.3.2.1. Hardening before installation

For bare metal installations, you can add hardening features to RHCOS before beginning the OpenShift Container Platform installation. For example, you can add kernel options when you boot the RHCOS installer to turn security features on or off, such as SELinux booleans or various low-level settings, such as symmetric multithreading.

Although bare metal RHCOS installations are more difficult, they offer the opportunity of getting operating system changes in place before starting the OpenShift Container Platform installation. This can be important when you need to ensure that certain features, such as disk encryption or special networking settings, be set up at the earliest possible moment.



WARNING

Disabling SELinux on RHCOS nodes is not supported.

2.3.2.2. Hardening during installation

You can interrupt the OpenShift installation process and change Ignition configs. Through Ignition configs, you can add your own files and systemd services to the RHCOS nodes. You can also make some basic security-related changes to the **install-config.yaml** file used for installation. Contents added in this way are available at each node's first boot.

2.3.2.3. Hardening after the cluster is running

After the OpenShift Container Platform cluster is up and running, there are several ways to apply hardening features to RHCOS:

- **Daemon set:** If you need a service to run on every node, you can add that service with a Kubernetes [DaemonSet](#) object.
- **Machine config:** **MachineConfig** objects contain a subset of Ignition configs in the same format. By applying machine configs to all worker or control plane nodes, you can ensure that the next node of the same type that is added to the cluster has the same changes applied.

All of the features noted here are described in the OpenShift Container Platform product documentation.

Additional resources

- [OpenShift Security Guide](#)
- [Choosing how to configure RHCOS](#)
- [Modifying Nodes](#)
- [Manually creating the installation configuration file](#)
- [Creating the Kubernetes manifest and Ignition config files](#)
- [Creating Red Hat Enterprise Linux CoreOS \(RHCOS\) machines using an ISO image](#)
- [Customizing nodes](#)
- [Adding kernel arguments to Nodes](#)
- [Installation configuration parameters](#) - see **fips**
- [Support for FIPS cryptography](#)
- [RHEL core crypto components](#)

2.4. CONTAINER IMAGE SIGNATURES

Red Hat delivers signatures for the images in the Red Hat Container Registries. Those signatures can be automatically verified when being pulled to OpenShift Container Platform 4 clusters by using the Machine Config Operator (MCO).

[Quay.io](#) serves most of the images that make up OpenShift Container Platform, and only the release image is signed. Release images refer to the approved OpenShift Container Platform images, offering a degree of protection against supply chain attacks. However, some extensions to OpenShift Container Platform, such as logging, monitoring, and service mesh, are shipped as Operators from the Operator Lifecycle Manager (OLM). Those images ship from the [Red Hat Ecosystem Catalog Container images](#) registry.

To verify the integrity of those images between Red Hat registries and your infrastructure, enable signature verification.

2.4.1. Enabling signature verification for Red Hat Container Registries

Enabling container signature validation requires files that link the registry URLs to the sigstore and then specifies the keys which verify the images.

Procedure

1. Create the files that link the registry URLs to the sigstore and that specifies the key to verify the image.

- Create the **policy.json** file:

```
$ cat > policy.json <<EOF
{
  "default": [
    {
      "type": "insecureAcceptAnything"
    }
  ],
  "transports": {
    "docker": {
      "registry.access.redhat.com": [
        {
          "type": "signedBy",
          "keyType": "GPGKeys",
          "keyPath": "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release"
        }
      ],
      "registry.redhat.io": [
        {
          "type": "signedBy",
          "keyType": "GPGKeys",
          "keyPath": "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release"
        }
      ]
    },
    "docker-daemon": {
      "": [
        {
          "type": "insecureAcceptAnything"
        }
      ]
    }
  }
}
EOF
```

- Create the **registry.access.redhat.com.yaml** file:

```
$ cat <<EOF > registry.access.redhat.com.yaml
docker:
  registry.access.redhat.com:
    sigstore: https://access.redhat.com/webassets/docker/content/sigstore
EOF
```

- Create the **registry.redhat.io.yaml** file:

```
$ cat <<EOF > registry.redhat.io.yaml
docker:
  registry.redhat.io:
    sigstore: https://registry.redhat.io/containers/sigstore
EOF
```

2. Set the files with a **base64** encode format that will be used for the machine config template:

```
$ export ARC_REG=$( cat registry.access.redhat.com.yaml | base64 -w0 )
$ export RIO_REG=$( cat registry.redhat.io.yaml | base64 -w0 )
$ export POLICY_CONFIG=$( cat policy.json | base64 -w0 )
```

3. Create a machine config that writes the exported files to disk on the worker nodes:

```
$ cat > 51-worker-rh-registry-trust.yaml <<EOF
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 51-worker-rh-registry-trust
spec:
  config:
    ignition:
      config: {}
      security:
        tls: {}
      timeouts: {}
      version: 2.2.0
    networkd: {}
    passwd: {}
    storage:
      files:
        - contents:
            source: data:text/plain;charset=utf-8;base64,${ARC_REG}
            verification: {}
          filesystem: root
          mode: 420
          path: /etc/containers/registries.d/registry.access.redhat.com.yaml
        - contents:
            source: data:text/plain;charset=utf-8;base64,${RIO_REG}
            verification: {}
          filesystem: root
          mode: 420
          path: /etc/containers/registries.d/registry.redhat.io.yaml
        - contents:
            source: data:text/plain;charset=utf-8;base64,${POLICY_CONFIG}
            verification: {}
          filesystem: root
          mode: 420
          path: /etc/containers/policy.json
    osImageURL: ""
EOF
```

4. Apply the created machine config:

```
$ oc apply -f 51-worker-rh-registry-trust.yaml
```

5. Create a machine config, which writes the exported files to disk on the master nodes:

```
$ cat > 51-master-rh-registry-trust.yaml <<EOF
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: master
  name: 51-master-rh-registry-trust
spec:
  config:
    ignition:
      config: {}
      security:
        tls: {}
      timeouts: {}
      version: 2.2.0
    networkd: {}
    passwd: {}
    storage:
      files:
        - contents:
            source: data:text/plain;charset=utf-8;base64,${ARC_REG}
            verification: {}
            filesystem: root
            mode: 420
            path: /etc/containers/registries.d/registry.access.redhat.com.yaml
        - contents:
            source: data:text/plain;charset=utf-8;base64,${RIO_REG}
            verification: {}
            filesystem: root
            mode: 420
            path: /etc/containers/registries.d/registry.redhat.io.yaml
        - contents:
            source: data:text/plain;charset=utf-8;base64,${POLICY_CONFIG}
            verification: {}
            filesystem: root
            mode: 420
            path: /etc/containers/policy.json
      osImageURL: ""
EOF
```

6. Apply the master machine config changes to the cluster:

```
$ oc apply -f 51-master-rh-registry-trust.yaml
```

2.4.2. Verifying the signature verification configuration

After you apply the machine configs to the cluster, the Machine Config Controller detects the new **MachineConfig** object and generates a new **rendered-worker-`<hash>`** version.

Prerequisites

- You enabled signature verification by using a machine config file.

Procedure

1. On the command line, run the following command to display information about a desired worker:

```
$ oc describe machineconfigpool/worker
```

Example output of initial worker monitoring

```
Name:      worker
Namespace:
Labels:    machineconfiguration.openshift.io/mco-built-in=
Annotations: <none>
API Version: machineconfiguration.openshift.io/v1
Kind:      MachineConfigPool
Metadata:
  Creation Timestamp: 2019-12-19T02:02:12Z
  Generation:        3
  Resource Version:  16229
  Self Link:         /apis/machineconfiguration.openshift.io/v1/machineconfigpools/worker
  UID:               92697796-2203-11ea-b48c-fa163e3940e5
Spec:
  Configuration:
    Name: rendered-worker-f6819366eb455a401c42f8d96ab25c02
    Source:
      API Version: machineconfiguration.openshift.io/v1
      Kind:        MachineConfig
      Name:        00-worker
      API Version: machineconfiguration.openshift.io/v1
      Kind:        MachineConfig
      Name:        01-worker-container-runtime
      API Version: machineconfiguration.openshift.io/v1
      Kind:        MachineConfig
      Name:        01-worker-kubelet
      API Version: machineconfiguration.openshift.io/v1
      Kind:        MachineConfig
      Name:        51-worker-rh-registry-trust
      API Version: machineconfiguration.openshift.io/v1
      Kind:        MachineConfig
      Name:        99-worker-92697796-2203-11ea-b48c-fa163e3940e5-registries
      API Version: machineconfiguration.openshift.io/v1
      Kind:        MachineConfig
      Name:        99-worker-ssh
    Machine Config Selector:
      Match Labels:
        machineconfiguration.openshift.io/role: worker
    Node Selector:
      Match Labels:
        node-role.kubernetes.io/worker:
  Paused:      false
Status:
  Conditions:
    Last Transition Time: 2019-12-19T02:03:27Z
  Message:
```

```

Reason:
Status:      False
Type:       RenderDegraded
Last Transition Time: 2019-12-19T02:03:43Z
Message:
Reason:
Status:      False
Type:       NodeDegraded
Last Transition Time: 2019-12-19T02:03:43Z
Message:
Reason:
Status:      False
Type:       Degraded
Last Transition Time: 2019-12-19T02:28:23Z
Message:
Reason:
Status:      False
Type:       Updated
Last Transition Time: 2019-12-19T02:28:23Z
Message:      All nodes are updating to rendered-worker-
f6819366eb455a401c42f8d96ab25c02
Reason:
Status:      True
Type:       Updating
Configuration:
Name: rendered-worker-d9b3f4ffcf65c30dcf591a0e8cf9b2e
Source:
  API Version:  machineconfiguration.openshift.io/v1
  Kind:        MachineConfig
  Name:        00-worker
  API Version:  machineconfiguration.openshift.io/v1
  Kind:        MachineConfig
  Name:        01-worker-container-runtime
  API Version:  machineconfiguration.openshift.io/v1
  Kind:        MachineConfig
  Name:        01-worker-kubelet
  API Version:  machineconfiguration.openshift.io/v1
  Kind:        MachineConfig
  Name:        99-worker-92697796-2203-11ea-b48c-fa163e3940e5-registries
  API Version:  machineconfiguration.openshift.io/v1
  Kind:        MachineConfig
  Name:        99-worker-ssh
Degraded Machine Count:  0
Machine Count:          1
Observed Generation:    3
Ready Machine Count:    0
Unavailable Machine Count: 1
Updated Machine Count:  0
Events:                 <none>

```

2. Run the **oc describe** command again:

```
$ oc describe machineconfigpool/worker
```

Example output after the worker is updated

```

...
  Last Transition Time: 2019-12-19T04:53:09Z
  Message:           All nodes are updated with rendered-worker-
f6819366eb455a401c42f8d96ab25c02
  Reason:
  Status:           True
  Type:            Updated
  Last Transition Time: 2019-12-19T04:53:09Z
  Message:
  Reason:
  Status:           False
  Type:            Updating
Configuration:
  Name: rendered-worker-f6819366eb455a401c42f8d96ab25c02
  Source:
    API Version:    machineconfiguration.openshift.io/v1
    Kind:           MachineConfig
    Name:           00-worker
    API Version:    machineconfiguration.openshift.io/v1
    Kind:           MachineConfig
    Name:           01-worker-container-runtime
    API Version:    machineconfiguration.openshift.io/v1
    Kind:           MachineConfig
    Name:           01-worker-kubelet
    API Version:    machineconfiguration.openshift.io/v1
    Kind:           MachineConfig
    Name:           51-worker-rh-registry-trust
    API Version:    machineconfiguration.openshift.io/v1
    Kind:           MachineConfig
    Name:           99-worker-92697796-2203-11ea-b48c-fa163e3940e5-registries
    API Version:    machineconfiguration.openshift.io/v1
    Kind:           MachineConfig
    Name:           99-worker-ssh
  Degraded Machine Count:  0
  Machine Count:           3
  Observed Generation:    4
  Ready Machine Count:    3
  Unavailable Machine Count: 0
  Updated Machine Count:  3
...

```



NOTE

The **Observed Generation** parameter shows an increased count based on the generation of the controller-produced configuration. This controller updates this value even if it fails to process the specification and generate a revision. The **Configuration Source** value points to the **51-worker-rh-registry-trust** configuration.

3. Confirm that the **policy.json** file exists with the following command:

```
$ oc debug node/<node> -- chroot /host cat /etc/containers/policy.json
```

Example output

```

Starting pod/<node>-debug ...
To use host binaries, run `chroot /host`
{
  "default": [
    {
      "type": "insecureAcceptAnything"
    }
  ],
  "transports": {
    "docker": {
      "registry.access.redhat.com": [
        {
          "type": "signedBy",
          "keyType": "GPGKeys",
          "keyPath": "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release"
        }
      ],
      "registry.redhat.io": [
        {
          "type": "signedBy",
          "keyType": "GPGKeys",
          "keyPath": "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release"
        }
      ]
    },
    "docker-daemon": {
      "": [
        {
          "type": "insecureAcceptAnything"
        }
      ]
    }
  }
}

```

4. Confirm that the **registry.redhat.io.yaml** file exists with the following command:

```

$ oc debug node/<node> -- chroot /host cat
/etc/containers/registries.d/registry.redhat.io.yaml

```

Example output

```

Starting pod/<node>-debug ...
To use host binaries, run `chroot /host`
docker:
  registry.redhat.io:
    sigstore: https://registry.redhat.io/containers/sigstore

```

5. Confirm that the **registry.access.redhat.com.yaml** file exists with the following command:

```

$ oc debug node/<node> -- chroot /host cat
/etc/containers/registries.d/registry.access.redhat.com.yaml

```

Example output

```
Starting pod/<node>-debug ...
To use host binaries, run `chroot /host`
docker:
  registry.access.redhat.com:
    sigstore: https://access.redhat.com/webassets/docker/content/sigstore
```

2.4.3. Additional resources

- [Machine Config Overview](#)

2.5. UNDERSTANDING COMPLIANCE

For many OpenShift Container Platform customers, regulatory readiness, or compliance, on some level is required before any systems can be put into production. That regulatory readiness can be imposed by national standards, industry standards or the organization's corporate governance framework.

2.5.1. Understanding compliance and risk management

FIPS compliance is one of the most critical components required in highly secure environments, to ensure that only supported cryptographic technologies are allowed on nodes.



IMPORTANT

The use of FIPS Validated / Modules in Process cryptographic libraries is only supported on OpenShift Container Platform deployments on the **x86_64** architecture.

To understand Red Hat's view of OpenShift Container Platform compliance frameworks, refer to the Risk Management and Regulatory Readiness chapter of the [OpenShift Security Guide Book](#).

Additional resources

- [Installing a cluster in FIPS mode](#)

2.6. SECURING CONTAINER CONTENT

To ensure the security of the content inside your containers you need to start with trusted base images, such as Red Hat Universal Base Images, and add trusted software. To check the ongoing security of your container images, there are both Red Hat and third-party tools for scanning images.

2.6.1. Securing 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, what versions are used, 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 operating system layers current?

By building your containers from Red Hat [Universal Base Images](#) (UBI) you are assured of a foundation for your container images that consists of the same RPM-packaged software that is included in Red Hat Enterprise Linux. No subscriptions are required to either use or redistribute UBI images.

To assure ongoing security of the containers themselves, security scanning features, used directly from RHEL or added to OpenShift Container Platform, can alert you when an image you are using has vulnerabilities. OpenSCAP image scanning is available in RHEL and the [Red Hat Quay Container Security Operator](#) can be added to check container images used in OpenShift Container Platform.

2.6.2. Creating redistributable images with UBI

To create containerized applications, you typically start with a trusted base image that offers the components that are usually provided by the operating system. These include the libraries, utilities, and other features the application expects to see in the operating system's file system.

Red Hat Universal Base Images (UBI) were created to encourage anyone building their own containers to start with one that is made entirely from Red Hat Enterprise Linux rpm packages and other content. These UBI images are updated regularly to keep up with security patches and free to use and redistribute with container images built to include your own software.

Search the [Red Hat Ecosystem Catalog](#) to both find and check the health of different UBI images. As someone creating secure container images, you might be interested in these two general types of UBI images:

- **UBI:** There are standard UBI images for RHEL 7 and 8 (**ubi7/ubi** and **ubi8/ubi**), as well as minimal images based on those systems (**ubi7/ubi-minimal** and **ubi8/ubi-minimal**). All of these images are preconfigured to point to free repositories of RHEL software that you can add to the container images you build, using standard **yum** and **dnf** commands. Red Hat encourages people to use these images on other distributions, such as Fedora and Ubuntu.
- **Red Hat Software Collections** Search the Red Hat Ecosystem Catalog for **rhscl/** to find images created to use as base images for specific types of applications. For example, there are Apache httpd (**rhscl/httpd-***), Python (**rhscl/python-***), Ruby (**rhscl/ruby-***), Node.js (**rhscl/nodejs-***) and Perl (**rhscl/perl-***) rhscl images.

Keep in mind that while UBI images are freely available and redistributable, Red Hat support for these images is only available through Red Hat product subscriptions.

See [Using Red Hat Universal Base Images](#) in the Red Hat Enterprise Linux documentation for information on how to use and build on standard, minimal and init UBI images.

2.6.3. Security scanning in RHEL

For Red Hat Enterprise Linux (RHEL) systems, OpenSCAP scanning is available from the **openscap-utils** package. In RHEL, you can use the **openscap-podman** command to scan images for vulnerabilities. See [Scanning containers and container images for vulnerabilities](#) in the Red Hat Enterprise Linux documentation.

OpenShift Container Platform enables you to leverage RHEL 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.

2.6.3.1. Scanning OpenShift images

For the container images that are running in OpenShift Container Platform and are pulled from Red Hat Quay registries, you can use an Operator to list the vulnerabilities of those images. The [Red Hat Quay Container Security Operator](#) can be added to OpenShift Container Platform to provide vulnerability reporting for images added to selected namespaces.

Container image scanning for Red Hat Quay is performed by the [Clair security scanner](#). In Red Hat Quay, Clair can search for and report vulnerabilities in images built from RHEL, CentOS, Oracle, Alpine, Debian, and Ubuntu operating system software.

2.6.4. Integrating external scanning

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

2.6.4.1. 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 2.1. Annotation key format

Component	Description	Acceptable values
qualityType	Metadata type	vulnerability license operations policy
providerId	Provider ID string	openscap redhatcatalog redhatinsights blackduck jfrog

2.6.4.1.1. 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 2.2. Annotation value format

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

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

Table 2.3. Summary field value format

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

2.6.4.1.2. 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 the [Container images section of the Red Hat Ecosystem Catalog](#) annotation for an image with health index data with an external URL for additional details:

Red Hat Ecosystem Catalog annotation

```
{
  "name": "Red Hat Ecosystem 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 }
  ]
}
```

2.6.4.2. 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.

2.6.4.2.1. Example annotate CLI command

Replace **<image>** with an image digest, for example **sha256:401e359e0f45bfdcf004e258b72e253fd07fba8cc5c6f2ed4f4608fb119ecc2:**

```
$ oc annotate image <image> \
  quality.images.openshift.io/vulnerability.redhatcatalog='{ \
  "name": "Red Hat Ecosystem Catalog", \
  "description": "Container health index", \
  "timestamp": "2020-06-01T05:04:46Z", \
  "compliant": null, \
  "scannerVersion": "1.2", \
  "reference": "https://access.redhat.com/errata/RHBA-2020:2347", \
  "summary": "[ \
    { "label": "Health index", "data": "B", "severityIndex": 1, "reference": null } ]' }
```

2.6.4.3. Controlling pod execution

Use the **images.openshift.io/deny-execution** image policy to programmatically control if an image can be run.

2.6.4.3.1. Example annotation

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

2.6.4.4. Integration reference

In most cases, external tools such as vulnerability scanners develop a script or plug-in that watches for image updates, performs scanning, and annotates the associated image object with the results. Typically this automation calls the OpenShift Container Platform 4.6 REST APIs to write the annotation. See OpenShift Container Platform REST APIs for general information on the REST APIs.

2.6.4.4.1. Example REST API call

The following example call using **curl** overrides 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>:6443/apis/image.openshift.io/v1/images/<image_id> \
  --data '{ <image_annotation> }'
```

The following is an example of **PATCH** payload data:

Patch call data

```
{
  "metadata": {
    "annotations": {
      "quality.images.openshift.io/vulnerability.redhatcatalog":
        "{ 'name': 'Red Hat Ecosystem Catalog', 'description': 'Container health index', 'timestamp': '2020-06-01T05:04:46Z', 'compliant': null, 'reference': 'https://access.redhat.com/errata/RHBA-2020:2347', 'summary': [{'label': 'Health index', 'data': '4', 'severityIndex': 1, 'reference': null}] }"
    }
  }
}
```

Additional resources

- [Image stream objects](#)

2.7. USING CONTAINER REGISTRIES SECURELY

Container registries store container images to:

- Make images accessible to others
- Organize images into repositories that can include multiple versions of an image
- Optionally limit access to images, based on different authentication methods, or make them publicly available

There are public container registries, such as Quay.io and Docker Hub where many people and organizations share their images. The Red Hat Registry offers supported Red Hat and partner images, while the Red Hat Ecosystem Catalog offers detailed descriptions and health checks for those images. To manage your own registry, you could purchase a container registry such as [Red Hat Quay](#).

From a security standpoint, some registries provide special features to check and improve the health of your containers. For example, Red Hat Quay offers container vulnerability scanning with Clair security scanner, build triggers to automatically rebuild images when source code changes in GitHub and other locations, and the ability to use role-based access control (RBAC) to secure access to images.

2.7.1. Knowing where 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.

2.7.2. 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. From an operational standpoint, you rebuild and redeploy an updated container image to replace a container instead of changing it.

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 \(RHSAs\)](#) to alert you to any newly discovered issues in Red Hat certified container images, and direct you to the updated image. Alternatively, you can go to the Red Hat Ecosystem Catalog to look up that and other security-related issues for each Red Hat image.

2.7.3. Getting containers from Red Hat Registry and Ecosystem Catalog

Red Hat lists certified container images for Red Hat products and partner offerings from the [Container Images](#) section of the Red Hat Ecosystem Catalog. From that catalog, you can see details of each image, including CVE, software packages listings, and health scores.

Red Hat images are actually stored in what is referred to as the *Red Hat Registry*, which is represented by a public container registry (registry.access.redhat.com) and an authenticated registry (registry.redhat.io). Both include basically the same set of container images, with registry.redhat.io including some additional images that require authentication with Red Hat subscription credentials.

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** to reflect the security risk for each container provided through the Red Hat Ecosystem Catalog. Because containers consume software provided by Red Hat and the errata process, old, stale containers are insecure whereas new, fresh containers are more secure.

To illustrate the age of containers, the Red Hat Ecosystem 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 Ecosystem Catalog](#) for more details on this grading system.

Refer to the [Red Hat Product Security Center](#) for details on security updates and vulnerabilities related to Red Hat software. Check out [Red Hat Security Advisories](#) to search for specific advisories and CVEs.

2.7.4. OpenShift Container Registry

OpenShift Container Platform includes the *OpenShift Container Registry*, a private registry running as an integrated component of 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 that you might already be using, such as Red Hat Quay.

Additional resources

- [Integrated OpenShift Container Platform registry](#)

2.7.5. Storing containers using Red Hat Quay

[Red Hat Quay](#) is an enterprise-quality container registry product from Red Hat. Development for Red Hat Quay is done through the upstream [Project Quay](#). Red Hat Quay is available to deploy on-premise or through the hosted version of Red Hat Quay at [Quay.io](#).

Security-related features of Red Hat Quay include:

- **Time machine:** Allows images with older tags to expire after a set period of time or based on a user-selected expiration time.
- **Repository mirroring:** Lets you mirror other registries for security reasons, such as hosting a public repository on Red Hat Quay behind a company firewall, or for performance reasons, to keep registries closer to where they are used.
- **Action log storage:** Save Red Hat Quay logging output to [Elasticsearch storage](#) to allow for later search and analysis.
- **Clair security scanning:** Scan images against a variety of Linux vulnerability databases, based on the origins of each container image.
- **Internal authentication:** Use the default local database to handle RBAC authentication to Red Hat Quay or choose from LDAP, Keystone (OpenStack), JWT Custom Authentication, or External Application Token authentication.

- **External authorization (OAuth):** Allow authorization to Red Hat Quay from GitHub, GitHub Enterprise, or Google Authentication.
- **Access settings:** Generate tokens to allow access to Red Hat Quay from docker, rkt, anonymous access, user-created accounts, encrypted client passwords, or prefix username autocompletion.

Ongoing integration of Red Hat Quay with OpenShift Container Platform continues, with several OpenShift Container Platform Operators of particular interest. The [Quay Bridge Operator](#) lets you replace the internal OpenShift Container Platform registry with Red Hat Quay. The [Quay Red Hat Quay Container Security Operator](#) lets you check vulnerabilities of images running in OpenShift Container Platform that were pulled from Red Hat Quay registries.

2.8. SECURING THE BUILD PROCESS

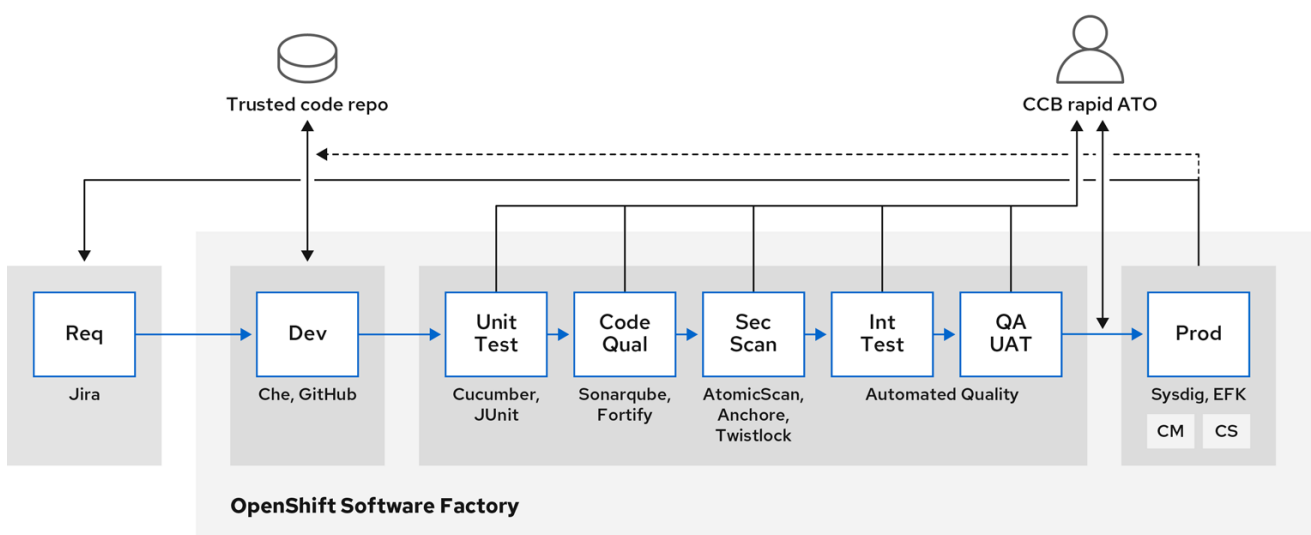
In a container environment, the software build process is the stage in the life cycle where application code is integrated with the required runtime libraries. Managing this build process is key to securing the software stack.

2.8.1. Building once, deploying everywhere

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.

As your software moves through the stages of building, testing, and production, it is important that the tools making up your software supply chain be trusted. The following figure illustrates the process and tools that could be incorporated into a trusted software supply chain for containerized software:



107_OpenShift_0720

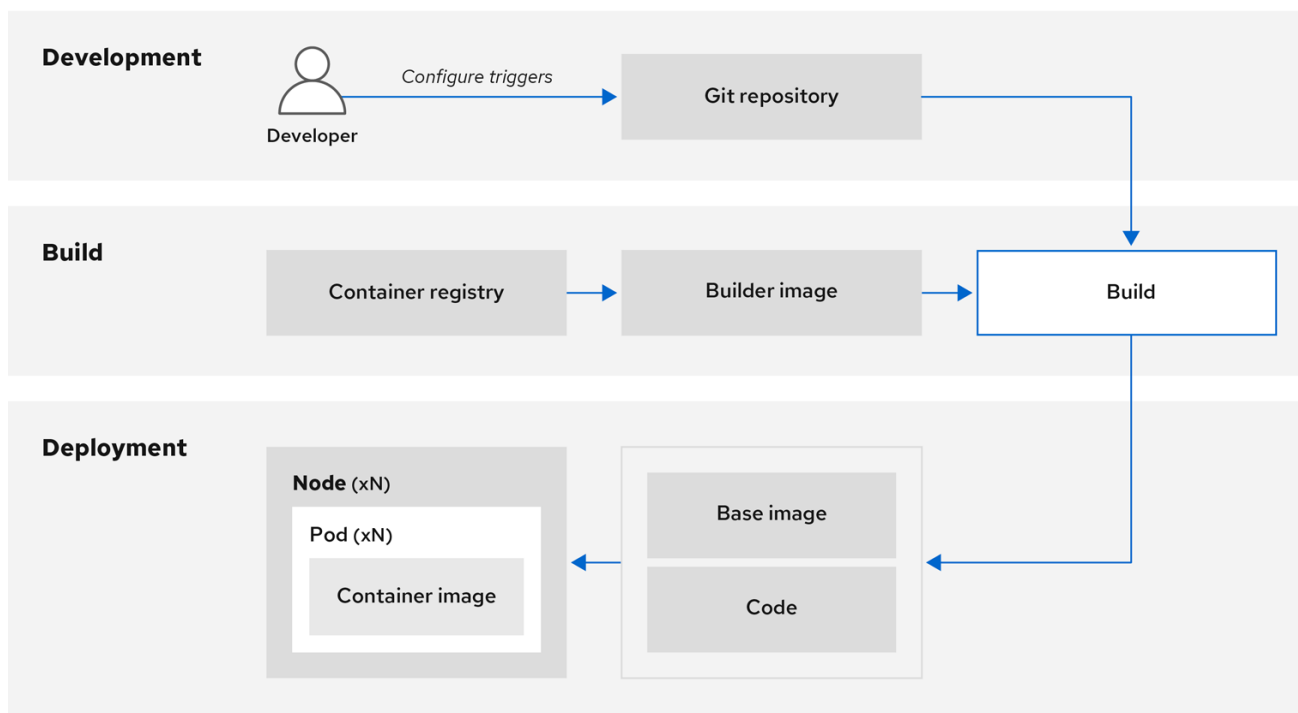
OpenShift Container Platform can be integrated with trusted code repositories (such as GitHub) and development platforms (such as Che) for creating and managing secure code. Unit testing could rely on [Cucumber](#) and [JUnit](#). You could inspect your containers for vulnerabilities and compliance issues with [Anchore](#) or [Twistlock](#), and use image scanning tools such as [AtomicScan](#) or [Clair](#). Tools such as [Sysdig](#) could provide ongoing monitoring of your containerized applications.

2.8.2. Managing builds

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. With Red Hat S2I images available as Universal Base Image (UBI) images, you can now freely redistribute your software with base images built from real RHEL RPM packages. Red Hat has removed subscription restrictions to allow this.

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

- Trigger, either by using 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 a CI process.



107_OpenShift_0720

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

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

2.8.3. 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 must 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** object:

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

```
$ oc create secret generic secret-npmrc --from-file=.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** object:

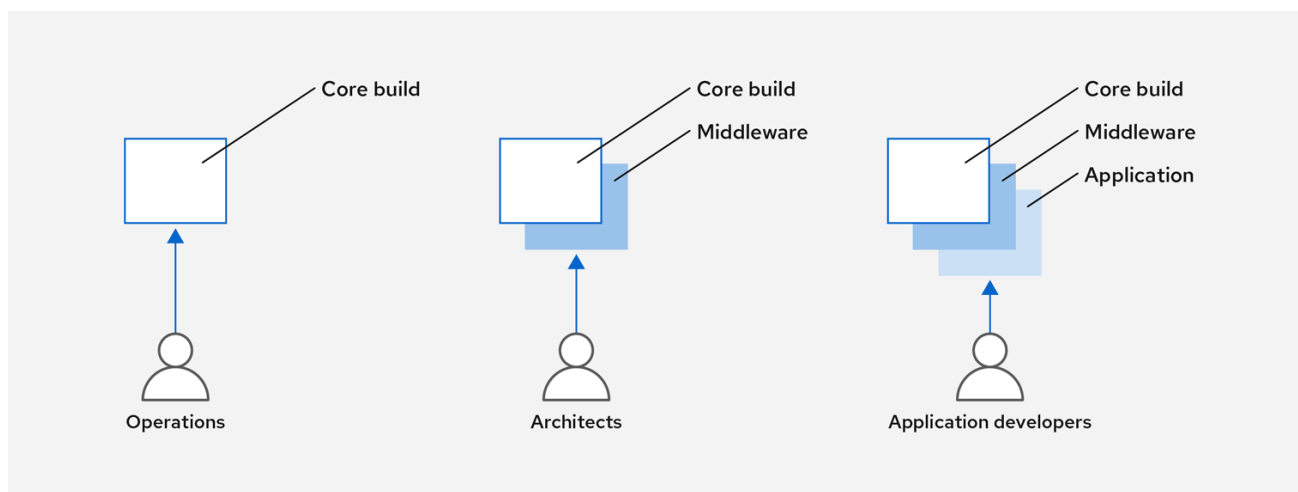
```
source:
  git:
    uri: https://github.com/sclorg/nodejs-ex.git
  secrets:
  - destinationDir: .
    secret:
      name: secret-npmrc
```

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

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

2.8.4. Designing your build process

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



107_OpenShift_0720

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 focus on writing 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.

Using GitOps methodology, you can use the same CI/CD mechanisms to manage not only your application configurations, but also your OpenShift Container Platform infrastructure.

2.8.5. Building Knative serverless applications

Relying on Kubernetes and Kourier, you can build, deploy, and manage serverless applications by using OpenShift Serverless in OpenShift Container Platform.

As with other builds, you can use S2I images to build your containers, then serve them using Knative services. View Knative application builds through the **Topology** view of the OpenShift Container Platform web console.

2.8.6. Additional resources

- [Understanding image builds](#)
- [Triggering and modifying builds](#)
- [Creating build inputs](#)
- [Input secrets and config maps](#)
- [The CI/CD methodology and practice](#)
- [About OpenShift Serverless](#)
- [Viewing application composition using the Topology view](#)

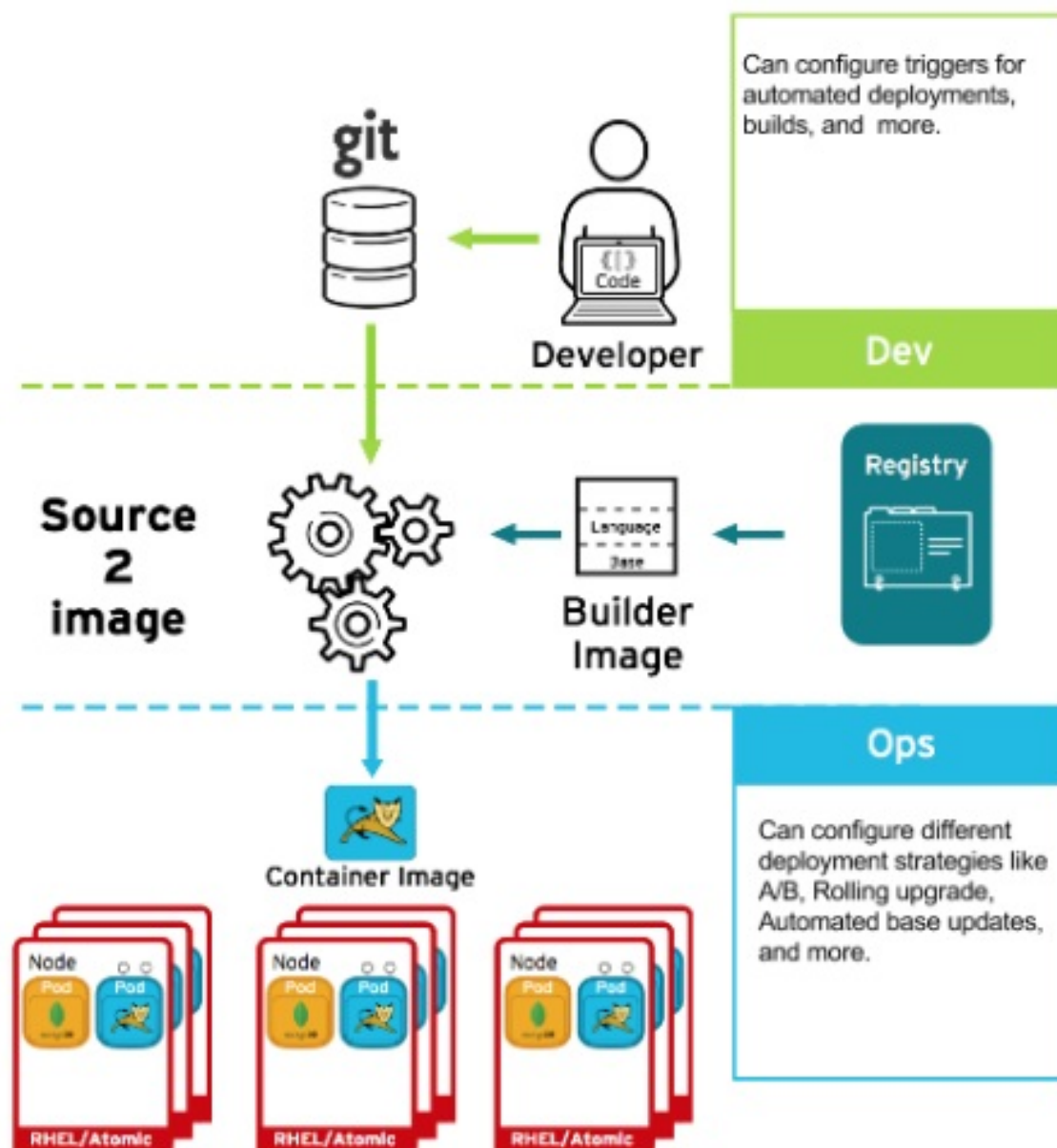
2.9. DEPLOYING CONTAINERS

You can use a variety of techniques to make sure that the containers you deploy hold the latest production-quality content and that they have not been tampered with. These techniques include setting up build triggers to incorporate the latest code and using signatures to ensure that the container comes from a trusted source and has not been modified.

2.9.1. Controlling container deployments with triggers

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 to remediate. You can use triggers to rebuild and replace images, ensuring the immutable containers process, 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 your 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.

You can use the **oc set triggers** command to set a deployment trigger. For example, to set a trigger for a deployment called deployment-example:

```
$ oc set triggers deploy/deployment-example \
  --from-image=example:latest \
  --containers=web
```

2.9.2. Controlling what image sources can be deployed

It is important that the intended images are actually being deployed, that the images including the contained content are from trusted sources, and they have not been altered. Cryptographic signing provides this assurance. OpenShift Container Platform enables cluster administrators to apply security policy that is broad or narrow, reflecting deployment environment and security requirements. Two parameters define this policy:

- one or more registries, with optional project namespace
- trust type, such as accept, reject, or require public key(s)

You can use these policy parameters to allow, deny, or require a trust relationship for entire registries, parts of registries, or individual images. Using trusted public keys, you can ensure that the source is cryptographically verified. The policy rules apply to nodes. Policy may be applied uniformly across all nodes or targeted for different node workloads (for example, build, zone, or environment).

Example image signature policy file

```
{
  "default": [{"type": "reject"}],
  "transports": {
    "docker": {
      "access.redhat.com": [
        {
          "type": "signedBy",
          "keyType": "GPGKeys",
          "keyPath": "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release"
        }
      ]
    },
    "atomic": {
      "172.30.1.1:5000/openshift": [
        {
          "type": "signedBy",
          "keyType": "GPGKeys",
          "keyPath": "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release"
        }
      ],
      "172.30.1.1:5000/production": [
        {
          "type": "signedBy",
          "keyType": "GPGKeys",
          "keyPath": "/etc/pki/example.com/pubkey"
        }
      ],
      "172.30.1.1:5000": [{"type": "reject"}]
    }
  }
}
```

The policy can be saved onto a node as **/etc/containers/policy.json**. Saving this file to a node is best accomplished using a new **MachineConfig** object. This example enforces the following rules:

- Require images from the Red Hat Registry (**registry.access.redhat.com**) to be signed by the Red Hat public key.

- Require images from your OpenShift Container Registry in the **openshift** namespace to be signed by the Red Hat public key.
- Require images from your OpenShift Container Registry in the **production** namespace to be signed by the public key for **example.com**.
- Reject all other registries not specified by the global **default** definition.

2.9.3. Using signature transports

A signature transport is a way to store and retrieve the binary signature blob. There are two types of signature transports.

- **atomic**: Managed by the OpenShift Container Platform API.
- **docker**: Served as a local file or by a web server.

The OpenShift Container Platform API manages signatures that use the **atomic** transport type. You must store the images that use this signature type in your OpenShift Container Registry. Because the `docker/distribution` **extensions** API auto-discovers the image signature endpoint, no additional configuration is required.

Signatures that use the **docker** transport type are served by local file or web server. These signatures are more flexible; you can serve images from any container image registry and use an independent server to deliver binary signatures.

However, the **docker** transport type requires additional configuration. You must configure the nodes with the URI of the signature server by placing arbitrarily-named YAML files into a directory on the host system, `/etc/containers/registries.d` by default. The YAML configuration files contain a registry URI and a signature server URI, or *sigstore*:

Example registries.d file

```
docker:
  access.redhat.com:
    sigstore: https://access.redhat.com/webassets/docker/content/sigstore
```

In this example, the Red Hat Registry, **access.redhat.com**, is the signature server that provides signatures for the **docker** transport type. Its URI is defined in the **sigstore** parameter. You might name this file `/etc/containers/registries.d/redhat.com.yaml` and use the Machine Config Operator to automatically place the file on each node in your cluster. No service restart is required since policy and **registries.d** files are dynamically loaded by the container runtime.

2.9.4. Creating secrets and config maps

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 so that it can access a private image repository, do the following:

Procedure

1. Log in to the OpenShift Container Platform web console.
2. Create a new project.
3. 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.
4. When creating a deployment configuration (for example, from the **Add to Project** → **Deploy Image** page), set the **Pull Secret** to your new secret.

Config maps 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.

2.9.5. Automating continuous deployment

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.

Additional resources

- [Input secrets and config maps](#)

2.10. SECURING THE CONTAINER PLATFORM

OpenShift Container Platform and Kubernetes 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.

Security-related features in OpenShift Container Platform that are based on Kubernetes include:

- Multitenancy, which combines Role-Based Access Controls and network policies to isolate containers at multiple levels.
- Admission plug-ins, which form boundaries between an API and those making requests to the API.

OpenShift Container Platform uses Operators to automate and simplify the management of Kubernetes-level security features.

2.10.1. Isolating containers with multitenancy

Multitenancy allows applications on an OpenShift Container Platform cluster that are owned by multiple users, and run across multiple hosts and namespaces, to remain isolated from each other and from outside attacks. You obtain multitenancy by applying role-based access control (RBAC) to Kubernetes namespaces.

In Kubernetes, *namespaces* are areas where applications can run in ways that are separate from other applications. OpenShift Container Platform uses and extends namespaces by adding extra annotations, including MCS labeling in SELinux, and identifying these extended namespaces as *projects*. Within the scope of a project, users can maintain their own cluster resources, including service accounts, policies, constraints, and various other objects.

RBAC objects are assigned to projects to authorize selected users to have access to those projects. That authorization takes the form of rules, roles, and bindings:

- Rules define what a user can create or access in a project.
- Roles are collections of rules that you can bind to selected users or groups.
- Bindings define the association between users or groups and roles.

Local RBAC roles and bindings attach a user or group to a particular project. Cluster RBAC can attach cluster-wide roles and bindings to all projects in a cluster. There are default cluster roles that can be assigned to provide **admin**, **basic-user**, **cluster-admin**, and **cluster-status** access.

2.10.2. Protecting control plane with admission plug-ins

While RBAC controls access rules between users and groups and available projects, *admission plug-ins* define access to the OpenShift Container Platform master API. Admission plug-ins form a chain of rules that consist of:

- Default admissions plug-ins: These implement a default set of policies and resources limits that are applied to components of the OpenShift Container Platform control plane.
- Mutating admission plug-ins: These plug-ins dynamically extend the admission chain. They call out to a webhook server and can both authenticate a request and modify the selected resource.
- Validating admission plug-ins: These validate requests for a selected resource and can both validate the request and ensure that the resource does not change again.

API requests go through admissions plug-ins in a chain, with any failure along the way causing the request to be rejected. Each admission plug-in is associated with particular resources and only responds to requests for those resources.

2.10.2.1. Security context constraints (SCCs)

You can use *security context constraints* (SCCs) to define a set of conditions that a pod 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, if required.

2.10.2.2. Granting roles to service accounts

You can assign roles to service accounts, in the same way that users are assigned role-based access. There are three default service accounts created for each project. A service account:

- is limited in scope to a particular project
- derives its name from its project
- is automatically assigned an API token and credentials to access the OpenShift Container Registry

Service accounts associated with platform components automatically have their keys rotated.

2.10.3. Authentication and authorization

2.10.3.1. 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 identity provider is used by default for new OpenShift Container Platform deployments, but you can configure this at initial installation time or post-installation.

2.10.3.2. 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 endpoints, 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](#) in the 3scale documentation.

2.10.3.3. Red Hat Single Sign-On

The Red Hat Single Sign-On server enables you to secure your applications by providing web single sign-on 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 Single Sign-On with LDAP-based directory services including Microsoft Active Directory and Red Hat Enterprise Linux Identity Management.

2.10.3.4. 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 multitenant 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
- The etcd service is not exposed directly to the cluster

2.10.4. 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 installer configures these certificates during installation. There are some primary components that generate this traffic:

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

2.10.4.1. Configuring custom certificates

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

Additional resources

- [Introduction to OpenShift Container Platform](#)
- [Using RBAC to define and apply permissions](#)
- [About admission plug-ins](#)
- [Managing security context constraints](#)
- [SCC reference commands](#)
- [Examples of granting roles to service accounts](#)
- [Configuring the internal OAuth server](#)
- [Understanding identity provider configuration](#)
- [Certificate types and descriptions](#)
- [Proxy certificates](#)

2.11. SECURING NETWORKS

Network security can be managed at several levels. At the pod level, network namespaces can prevent containers from seeing other pods or the host system by restricting network access. Network policies give you control over allowing and rejecting connections. You can manage ingress and egress traffic to and from your containerized applications.

2.11.1. Using network namespaces

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

Network policy mode, by default, makes all pods in a project accessible from other pods and network endpoints. To isolate one or more pods in a project, you can create **NetworkPolicy** objects in that project to indicate the allowed incoming connections. Using multitenant mode, you can provide project-level isolation for pods and services.

2.11.2. Isolating pods with network policies

Using *network policies*, you can isolate pods from each other in the same project. Network policies can deny all network access to a pod, only allow connections for the ingress controller, reject connections from pods in other projects, or set similar rules for how networks behave.

Additional resources

- [About network policy](#)

2.11.3. Using multiple pod networks

Each running container has only one network interface by default. The Multus CNI plug-in lets you create multiple CNI networks, and then attach any of those networks to your pods. In that way, you can do things like separate private data onto a more restricted network and have multiple network interfaces on each node.

Additional resources

- [Using multiple networks](#)

2.11.4. Isolating applications

OpenShift Container Platform enables you to segment network traffic on a single cluster to make multitenant clusters that isolate users, teams, applications, and environments from non-global resources.

Additional resources

- [Configuring network isolation using OpenShiftSDN](#)

2.11.5. Securing ingress traffic

There are many security implications related to how you configure access to your Kubernetes services from outside of your OpenShift Container Platform cluster. Besides exposing HTTP and HTTPS routes, ingress routing allows you to set up NodePort or LoadBalancer ingress types. NodePort exposes an

application's service API object from each cluster worker. LoadBalancer lets you assign an external load balancer to an associated service API object in your OpenShift Container Platform cluster.

Additional resources

- [Configuring ingress cluster traffic](#)

2.11.6. Securing egress traffic

OpenShift Container Platform 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. A cluster administrator can assign one or more egress IP addresses to a project in an OpenShift Container Platform SDN network provider. Likewise, a cluster administrator can prevent egress traffic from going outside of an OpenShift Container Platform cluster using an egress firewall.

By assigning a fixed egress IP address, you can have all outgoing traffic assigned to that IP address for a particular project. With the egress firewall, you can prevent a pod from connecting to an external network, prevent a pod from connecting to an internal network, or limit a pod's access to specific internal subnets.

Additional resources

- [Configuring an egress firewall to control access to external IP addresses](#)
- [Configuring egress IPs for a project](#)

2.12. SECURING ATTACHED STORAGE

OpenShift Container Platform supports multiple types of storage, both for on-premise and cloud providers. In particular, OpenShift Container Platform can use storage types that support the Container Storage Interface.

2.12.1. Persistent volume plug-ins

Containers are useful for both stateless and stateful applications. Protecting attached storage is a key element of securing stateful services. Using the Container Storage Interface (CSI), OpenShift Container Platform can incorporate storage from any storage back end that supports the CSI interface.

OpenShift Container Platform provides plug-ins for multiple types of storage, including:

- Red Hat OpenShift Container Storage *
- AWS Elastic Block Stores (EBS) *
- AWS Elastic File System (EFS) *
- Azure Disk *
- Azure File *
- OpenStack Cinder *
- GCE Persistent Disks *
- VMware vSphere *

- Network File System (NFS)
- FlexVolume
- Fibre Channel
- iSCSI

Plug-ins for those storage types with dynamic provisioning are marked with an asterisk (*). Data in transit is encrypted via HTTPS for all OpenShift Container Platform components communicating with each other.

You can mount a persistent volume (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**.

2.12.2. Shared storage

For shared storage providers like NFS, 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.

2.12.3. 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.

Additional resources

- [Understanding persistent storage](#)
- [Configuring CSI volumes](#)
- [Dynamic provisioning](#)
- [Persistent storage using NFS](#)
- [Persistent storage using AWS Elastic Block Store](#)
- [Persistent storage using GCE Persistent Disk](#)

2.13. MONITORING CLUSTER EVENTS AND LOGS

The ability to monitor and audit an OpenShift Container Platform cluster is an important part of safeguarding the cluster and its users against inappropriate usage.

There are two main sources of cluster-level information that are useful for this purpose: events and logging.

2.13.1. Watching cluster events

Cluster administrators are encouraged to familiarize themselves with the **Event** resource type and review the list of system events to determine which events are of interest. Events are associated with a namespace, either the namespace of the resource they are related to or, for cluster events, the **default** namespace. The default namespace holds relevant events for monitoring or auditing a cluster, such as node events and resource events related to infrastructure components.

The master API and **oc** command do not provide parameters to scope a listing of events to only those related to nodes. A simple approach would be to use **grep**:

```
$ oc get event -n default | grep Node
```

Example output

```
1h      20h      3      origin-node-1.example.local Node      Normal      NodeHasDiskPressure ...
```

A more flexible approach is to output the events in a form that other tools can process. For example, the following example uses the **jq** tool against JSON output to extract only **NodeHasDiskPressure** events:

```
$ oc get events -n default -o json \
  | jq '.items[] | select(.involvedObject.kind == "Node" and .reason == "NodeHasDiskPressure")'
```

Example output

```
{
  "apiVersion": "v1",
  "count": 3,
  "involvedObject": {
    "kind": "Node",
    "name": "origin-node-1.example.local",
    "uid": "origin-node-1.example.local"
  },
  "kind": "Event",
  "reason": "NodeHasDiskPressure",
  ...
}
```

Events related to resource creation, modification, or deletion can also be good candidates for detecting misuse of the cluster. The following query, for example, can be used to look for excessive pulling of images:

```
$ oc get events --all-namespaces -o json \
  | jq '[.items[] | select(.involvedObject.kind == "Pod" and .reason == "Pulling")] | length'
```

Example output

```
4
```

**NOTE**

When a namespace is deleted, its events are deleted as well. Events can also expire and are deleted to prevent filling up etcd storage. Events are not stored as a permanent record and frequent polling is necessary to capture statistics over time.

2.13.2. Logging

Using the **oc log** command, you can view container logs, build configs and deployments in real time. Different can users have access different access to logs:

- Users who have access to a project are able to see the logs for that project by default.
- Users with admin roles can access all container logs.

To save your logs for further audit and analysis, you can enable the **cluster-logging** add-on feature to collect, manage, and view system, container, and audit logs. You can deploy, manage, and upgrade cluster logging through the OpenShift Elasticsearch Operator and Cluster Logging Operator.

2.13.3. Audit logs

With *audit logs*, you can follow a sequence of activities associated with how a user, administrator, or other OpenShift Container Platform component is behaving. API audit logging is done on each server.

Additional resources

- [List of system events](#)
- [Understanding cluster logging](#)
- [Viewing audit logs](#)

CHAPTER 3. CONFIGURING CERTIFICATES

3.1. REPLACING THE DEFAULT INGRESS CERTIFICATE

3.1.1. Understanding the default ingress certificate

By default, OpenShift Container Platform uses the Ingress Operator to create an internal CA and issue a wildcard certificate that is valid for applications under the **.apps** sub-domain. Both the web console and CLI use this certificate as well.

The internal infrastructure CA certificates are self-signed. While this process might be perceived as bad practice by some security or PKI teams, any risk here is minimal. The only clients that implicitly trust these certificates are other components within the cluster. Replacing the default wildcard certificate with one that is issued by a public CA already included in the CA bundle as provided by the container userspace allows external clients to connect securely to applications running under the **.apps** sub-domain.

3.1.2. Replacing the default ingress certificate

You can replace the default ingress certificate for all applications under the **.apps** subdomain. After you replace the certificate, all applications, including the web console and CLI, will have encryption provided by specified certificate.

Prerequisites

- You must have a wildcard certificate for the fully qualified **.apps** subdomain and its corresponding private key. Each should be in a separate PEM format file.
- The private key must be unencrypted. If your key is encrypted, decrypt it before importing it into OpenShift Container Platform.
- The certificate must include the **subjectAltName** extension showing ***.apps.<clustername>.<domain>**.
- The certificate file can contain one or more certificates in a chain. The wildcard certificate must be the first certificate in the file. It can then be followed with any intermediate certificates, and the file should end with the root CA certificate.
- Copy the root CA certificate into an additional PEM format file.

Procedure

1. Create a config map that includes only the root CA certificate used to sign the wildcard certificate:

```
$ oc create configmap custom-ca \
  --from-file=ca-bundle.crt=</path/to/example-ca.crt> 1 \
  -n openshift-config
```

- 1** **</path/to/example-ca.crt>** is the path to the root CA certificate file on your local file system.

2. Update the cluster-wide proxy configuration with the newly created config map:

```
$ oc patch proxy/cluster \
  --type=merge \
  --patch='{"spec":{"trustedCA":{"name":"custom-ca}}}'
```

3. Create a secret that contains the wildcard certificate chain and key:

```
$ oc create secret tls <secret> \ 1
  --cert=</path/to/cert.crt> \ 2
  --key=</path/to/cert.key> \ 3
  -n openshift-ingress
```

- 1** **<secret>** is the name of the secret that will contain the certificate chain and private key.
- 2** **</path/to/cert.crt>** is the path to the certificate chain on your local file system.
- 3** **</path/to/cert.key>** is the path to the private key associated with this certificate.

4. Update the Ingress Controller configuration with the newly created secret:

```
$ oc patch ingresscontroller.operator default \
  --type=merge -p \
  '{"spec":{"defaultCertificate":{"name":"<secret>}}}' 1
  -n openshift-ingress-operator
```

- 1** Replace **<secret>** with the name used for the secret in the previous step.

Additional resources

- [Replacing the CA Bundle certificate](#)
- [Proxy certificate customization](#)

3.2. ADDING API SERVER CERTIFICATES

The default API server certificate is issued by an internal OpenShift Container Platform cluster CA. Clients outside of the cluster will not be able to verify the API server's certificate by default. This certificate can be replaced by one that is issued by a CA that clients trust.

3.2.1. Add an API server named certificate

The default API server certificate is issued by an internal OpenShift Container Platform cluster CA. You can add one or more alternative certificates that the API server will return based on the fully qualified domain name (FQDN) requested by the client, for example when a reverse proxy or load balancer is used.

Prerequisites

- You must have a certificate for the FQDN and its corresponding private key. Each should be in a separate PEM format file.
- The private key must be unencrypted. If your key is encrypted, decrypt it before importing it into OpenShift Container Platform.

- The certificate must include the **subjectAltName** extension showing the FQDN.
- The certificate file can contain one or more certificates in a chain. The certificate for the API server FQDN must be the first certificate in the file. It can then be followed with any intermediate certificates, and the file should end with the root CA certificate.



WARNING

Do not provide a named certificate for the internal load balancer (host name **api-int.<cluster_name>.<base_domain>**). Doing so will leave your cluster in a degraded state.

Procedure

1. Login to the new API as the **kubeadmin** user.

```
$ oc login -u kubeadmin -p <password> https://FQDN:6443
```

2. Get the **kubeconfig** file.

```
$ oc config view --flatten > kubeconfig-newapi
```

3. Create a secret that contains the certificate chain and private key in the **openshift-config** namespace.

```
$ oc create secret tls <secret> \ 1
  --cert=</path/to/cert.crt> 2
  --key=</path/to/cert.key> \ 3
  -n openshift-config
```

1 **<secret>** is the name of the secret that will contain the certificate chain and private key.

2 **</path/to/cert.crt>** is the path to the certificate chain on your local file system.

3 **</path/to/cert.key>** is the path to the private key associated with this certificate.

4. Update the API server to reference the created secret.

```
$ oc patch apiserver cluster \
  --type=merge -p \
  '{"spec":{"servingCerts":{"namedCertificates":
  [{"names":["<FQDN>"], 1
  "servingCertificate":{"name":"<secret>"}}]}}' 2
```

1 Replace **<FQDN>** with the FQDN that the API server should provide the certificate for.

2 Replace **<secret>** with the name used for the secret in the previous step.

- Examine the **apiserver/cluster** object and confirm the secret is now referenced.

```
$ oc get apiserver cluster -o yaml
```

Example output

```
...
spec:
  servingCerts:
    namedCertificates:
      - names:
        - <FQDN>
      servingCertificate:
        name: <secret>
...
```

- Check the **kube-apiserver** operator, and verify that a new revision of the Kubernetes API server rolls out. It may take a minute for the operator to detect the configuration change and trigger a new deployment. While the new revision is rolling out, **PROGRESSING** will report **True**.

```
$ oc get clusteroperators kube-apiserver
```

Do not continue to the next step until **PROGRESSING** is listed as **False**, as shown in the following output:

Example output

```
NAME          VERSION AVAILABLE PROGRESSING DEGRADED SINCE
kube-apiserver 4.6.0   True      False      False      145m
```

If **PROGRESSING** is showing **True**, wait a few minutes and try again.

3.3. SECURING SERVICE TRAFFIC USING SERVICE SERVING CERTIFICATE SECRETS

3.3.1. Understanding service serving certificates

Service serving certificates are intended to support complex middleware applications that require encryption. These certificates are issued as TLS web server certificates.

The **service-ca** controller uses the **x509.SHA256WithRSA** signature algorithm to generate service certificates.

The generated certificate and key are in PEM format, stored in **tls.crt** and **tls.key** respectively, within a created secret. The certificate and key are automatically replaced when they get close to expiration.

The service CA certificate, which issues the service certificates, is valid for 26 months and is automatically rotated when there is less than 13 months validity left. After rotation, the previous service CA configuration is still trusted until its expiration. This allows a grace period for all affected services to refresh their key material before the expiration. If you do not upgrade your cluster during this grace period, which restarts services and refreshes their key material, you might need to manually restart services to avoid failures after the previous service CA expires.

**NOTE**

You can use the following command to manually restart all pods in the cluster. Be aware that running this command causes a service interruption, because it deletes every running pod in every namespace. These pods will automatically restart after they are deleted.

```
$ for I in $(oc get ns -o jsonpath='{range .items[*]} {.metadata.name}{"\n"} {end}'); \
do oc delete pods --all -n $I; \
sleep 1; \
done
```

3.3.2. Add a service certificate

To secure communication to your service, generate a signed serving certificate and key pair into a secret in the same namespace as the service.

**IMPORTANT**

The generated certificate is only valid for the internal service DNS name **<service.name>.<service.namespace>.svc**, and are only valid for internal communications.

Prerequisites:

- You must have a service defined.

Procedure

1. Annotate the service with **service.beta.openshift.io/serving-cert-secret-name**:

```
$ oc annotate service <service_name> \ 1
service.beta.openshift.io/serving-cert-secret-name=<secret_name> 2
```

- 1** Replace **<service_name>** with the name of the service to secure.
- 2** **<secret_name>** will be the name of the generated secret containing the certificate and key pair. For convenience, it is recommended that this be the same as **<service_name>**.

For example, use the following command to annotate the service **test1**:

```
$ oc annotate service test1 service.beta.openshift.io/serving-cert-secret-name=test1
```

2. Examine the service to confirm that the annotations are present:

```
$ oc describe service <service_name>
```

Example output

```
...
Annotations:          service.beta.openshift.io/serving-cert-secret-name: <service_name>
                    service.beta.openshift.io/serving-cert-signed-by: openshift-service-serving-
```

```
signer@1556850837
```

```
...
```

3. After the cluster generates a secret for your service, your **Pod** spec can mount it, and the pod will run after it becomes available.

Additional resources

- You can use a service certificate to configure a secure route using reencrypt TLS termination. For more information, see [Creating a re-encrypt route with a custom certificate](#).

3.3.3. Add the service CA bundle to a config map

A pod can access the service CA certificate by mounting a **ConfigMap** object that is annotated with **service.beta.openshift.io/inject-cabundle=true**. Once annotated, the cluster automatically injects the service CA certificate into the **service-ca.crt** key on the config map. Access to this CA certificate allows TLS clients to verify connections to services using service serving certificates.



IMPORTANT

After adding this annotation to a config map all existing data in it is deleted. It is recommended to use a separate config map to contain the **service-ca.crt**, instead of using the same config map that stores your pod configuration.

Procedure

1. Annotate the config map with **service.beta.openshift.io/inject-cabundle=true**:

```
$ oc annotate configmap <config_map_name> \
  service.beta.openshift.io/inject-cabundle=true
```

- 1 Replace **<config_map_name>** with the name of the config map to annotate.



NOTE

Explicitly referencing the **service-ca.crt** key in a volume mount will prevent a pod from starting until the config map has been injected with the CA bundle. This behavior can be overridden by setting the **optional** field to **true** for the volume's serving certificate configuration.

For example, use the following command to annotate the config map **test1**:

```
$ oc annotate configmap test1 service.beta.openshift.io/inject-cabundle=true
```

2. View the config map to ensure that the service CA bundle has been injected:

```
$ oc get configmap <config_map_name> -o yaml
```

The CA bundle is displayed as the value of the **service-ca.crt** key in the YAML output:

```
apiVersion: v1
```

```
data:
  service-ca.crt: |
    -----BEGIN CERTIFICATE-----
  ...
```

3.3.4. Add the service CA bundle to an API service

You can annotate an **APIService** object with **service.beta.openshift.io/inject-cabundle=true** to have its **spec.caBundle** field populated with the service CA bundle. This allows the Kubernetes API server to validate the service CA certificate used to secure the targeted endpoint.

Procedure

1. Annotate the API service with **service.beta.openshift.io/inject-cabundle=true**:

```
$ oc annotate apiservice <api_service_name> \
  service.beta.openshift.io/inject-cabundle=true
```

- 1 Replace **<api_service_name>** with the name of the API service to annotate.

For example, use the following command to annotate the API service **test1**:

```
$ oc annotate apiservice test1 service.beta.openshift.io/inject-cabundle=true
```

2. View the API service to ensure that the service CA bundle has been injected:

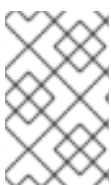
```
$ oc get apiservice <api_service_name> -o yaml
```

The CA bundle is displayed in the **spec.caBundle** field in the YAML output:

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  annotations:
    service.beta.openshift.io/inject-cabundle: "true"
  ...
spec:
  caBundle: <CA_BUNDLE>
  ...
```

3.3.5. Add the service CA bundle to a custom resource definition

You can annotate a **CustomResourceDefinition** (CRD) object with **service.beta.openshift.io/inject-cabundle=true** to have its **spec.conversion.webhook.clientConfig.caBundle** field populated with the service CA bundle. This allows the Kubernetes API server to validate the service CA certificate used to secure the targeted endpoint.



NOTE

The service CA bundle will only be injected into the CRD if the CRD is configured to use a webhook for conversion. It is only useful to inject the service CA bundle if a CRD's webhook is secured with a service CA certificate.

Procedure

1. Annotate the CRD with **service.beta.openshift.io/inject-cabundle=true**:

```
$ oc annotate crd <crd_name> \1
    service.beta.openshift.io/inject-cabundle=true
```

- 1 Replace **<crd_name>** with the name of the CRD to annotate.

For example, use the following command to annotate the CRD **test1**:

```
$ oc annotate crd test1 service.beta.openshift.io/inject-cabundle=true
```

2. View the CRD to ensure that the service CA bundle has been injected:

```
$ oc get crd <crd_name> -o yaml
```

The CA bundle is displayed in the **spec.conversion.webhook.clientConfig.caBundle** field in the YAML output:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    service.beta.openshift.io/inject-cabundle: "true"
  ...
spec:
  conversion:
    strategy: Webhook
    webhook:
      clientConfig:
        caBundle: <CA_BUNDLE>
  ...
```

3.3.6. Add the service CA bundle to a mutating webhook configuration

You can annotate a **MutatingWebhookConfiguration** object with **service.beta.openshift.io/inject-cabundle=true** to have the **clientConfig.caBundle** field of each webhook populated with the service CA bundle. This allows the Kubernetes API server to validate the service CA certificate used to secure the targeted endpoint.



NOTE

Do not set this annotation for admission webhook configurations that need to specify different CA bundles for different webhooks. If you do, then the service CA bundle will be injected for all webhooks.

Procedure

1. Annotate the mutating webhook configuration with **service.beta.openshift.io/inject-cabundle=true**:

```
$ oc annotate mutatingwebhookconfigurations <mutating_webhook_name> \1
```

```
service.beta.openshift.io/inject-cabundle=true
```

- 1 Replace **<mutating_webhook_name>** with the name of the mutating webhook configuration to annotate.

For example, use the following command to annotate the mutating webhook configuration **test1**:

```
$ oc annotate mutatingwebhookconfigurations test1 service.beta.openshift.io/inject-cabundle=true
```

2. View the mutating webhook configuration to ensure that the service CA bundle has been injected:

```
$ oc get mutatingwebhookconfigurations <mutating_webhook_name> -o yaml
```

The CA bundle is displayed in the **clientConfig.caBundle** field of all webhooks in the YAML output:

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  annotations:
    service.beta.openshift.io/inject-cabundle: "true"
  ...
webhooks:
- myWebhook:
  - v1beta1
  clientConfig:
    caBundle: <CA_BUNDLE>
  ...
```

3.3.7. Add the service CA bundle to a validating webhook configuration

You can annotate a **ValidatingWebhookConfiguration** object with **service.beta.openshift.io/inject-cabundle=true** to have the **clientConfig.caBundle** field of each webhook populated with the service CA bundle. This allows the Kubernetes API server to validate the service CA certificate used to secure the targeted endpoint.



NOTE

Do not set this annotation for admission webhook configurations that need to specify different CA bundles for different webhooks. If you do, then the service CA bundle will be injected for all webhooks.

Procedure

1. Annotate the validating webhook configuration with **service.beta.openshift.io/inject-cabundle=true**:

```
$ oc annotate validatingwebhookconfigurations <validating_webhook_name> \
service.beta.openshift.io/inject-cabundle=true
```

- 1 Replace `<validating_webhook_name>` with the name of the validating webhook configuration to annotate.

For example, use the following command to annotate the validating webhook configuration **test1**:

```
$ oc annotate validatingwebhookconfigurations test1 service.beta.openshift.io/inject-cabundle=true
```

2. View the validating webhook configuration to ensure that the service CA bundle has been injected:

```
$ oc get validatingwebhookconfigurations <validating_webhook_name> -o yaml
```

The CA bundle is displayed in the **clientConfig.caBundle** field of all webhooks in the YAML output:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  annotations:
    service.beta.openshift.io/inject-cabundle: "true"
  ...
webhooks:
- myWebhook:
  - v1beta1
  clientConfig:
    caBundle: <CA_BUNDLE>
  ...
```

3.3.8. Manually rotate the generated service certificate

You can rotate the service certificate by deleting the associated secret. Deleting the secret results in a new one being automatically created, resulting in a new certificate.

Prerequisites

- A secret containing the certificate and key pair must have been generated for the service.

Procedure

1. Examine the service to determine the secret containing the certificate. This is found in the **servicing-cert-secret-name** annotation, as seen below.

```
$ oc describe service <service_name>
```

Example output

```
...
service.beta.openshift.io/servicing-cert-secret-name: <secret>
...
```

2. Delete the generated secret for the service. This process will automatically recreate the secret.

```
$ oc delete secret <secret> 1
```

- 1 Replace **<secret>** with the name of the secret from the previous step.

3. Confirm that the certificate has been recreated by obtaining the new secret and examining the **AGE**.

```
$ oc get secret <service_name>
```

Example output

```
NAME          TYPE          DATA  AGE
<service.name>  kubernetes.io/tls  2     1s
```

3.3.9. Manually rotate the service CA certificate

The service CA is valid for 26 months and is automatically refreshed when there is less than 13 months validity left.

If necessary, you can manually refresh the service CA by using the following procedure.



WARNING

A manually-rotated service CA does not maintain trust with the previous service CA. You might experience a temporary service disruption until the pods in the cluster are restarted, which ensures that pods are using service serving certificates issued by the new service CA.

Prerequisites

- You must be logged in as a cluster admin.

Procedure

1. View the expiration date of the current service CA certificate by using the following command.

```
$ oc get secrets/signing-key -n openshift-service-ca \
  -o template='{{index .data "tls.crt"}}' \
  | base64 --decode \
  | openssl x509 -noout -enddate
```

2. Manually rotate the service CA. This process generates a new service CA which will be used to sign the new service certificates.

```
$ oc delete secret/signing-key -n openshift-service-ca
```

- To apply the new certificates to all services, restart all the pods in your cluster. This command ensures that all services use the updated certificates.

```
$ for I in $(oc get ns -o jsonpath='{range .items[*]} {.metadata.name}{"\n"} {end}'); \
do oc delete pods --all -n $I; \
sleep 1; \
done
```



WARNING

This command will cause a service interruption, as it goes through and deletes every running pod in every namespace. These pods will automatically restart after they are deleted.

3.4. UPDATING THE CA BUNDLE

3.4.1. Understanding the CA Bundle certificate

Proxy certificates allow users to specify one or more custom certificate authority (CA) used by platform components when making egress connections.

The **trustedCA** field of the Proxy object is a reference to a config map that contains a user-provided trusted certificate authority (CA) bundle. This bundle is merged with the Red Hat Enterprise Linux CoreOS (RHCOS) trust bundle and injected into the trust store of platform components that make egress HTTPS calls. For example, **image-registry-operator** calls an external image registry to download images. If **trustedCA** is not specified, only the RHCOS trust bundle is used for proxied HTTPS connections. Provide custom CA certificates to the RHCOS trust bundle if you want to use your own certificate infrastructure.

The **trustedCA** field should only be consumed by a proxy validator. The validator is responsible for reading the certificate bundle from required key **ca-bundle.crt** and copying it to a config map named **trusted-ca-bundle** in the **openshift-config-managed** namespace. The namespace for the config map referenced by **trustedCA** is **openshift-config**:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: user-ca-bundle
  namespace: openshift-config
data:
  ca-bundle.crt: |
    -----BEGIN CERTIFICATE-----
    Custom CA certificate bundle.
    -----END CERTIFICATE-----
```

3.4.2. Replacing the CA Bundle certificate

Procedure

1. Create a config map that includes the root CA certificate used to sign the wildcard certificate:

```
$ oc create configmap custom-ca \  
  --from-file=ca-bundle.crt=</path/to/example-ca.crt> \ 1  
  -n openshift-config
```

- 1 </path/to/example-ca.crt> is the path to the CA certificate bundle on your local file system.

2. Update the cluster-wide proxy configuration with the newly created config map:

```
$ oc patch proxy/cluster \  
  --type=merge \  
  --patch='{"spec":{"trustedCA":{"name":"custom-ca}}}'
```

Additional resources

- [Replacing the default ingress certificate](#)
- [Enabling the cluster-wide proxy](#)
- [Proxy certificate customization](#)

CHAPTER 4. CERTIFICATE TYPES AND DESCRIPTIONS

4.1. USER-PROVIDED CERTIFICATES FOR THE API SERVER

4.1.1. Purpose

The API server is accessible by clients external to the cluster at **api.<cluster_name>.<base_domain>**. You might want clients to access the API server at a different hostname or without the need to distribute the cluster-managed certificate authority (CA) certificates to the clients. The administrator must set a custom default certificate to be used by the API server when serving content.

4.1.2. Location

The user-provided certificates must be provided in a **kubernetes.io/tls** type **Secret** in the **openshift-config** namespace. Update the API server cluster configuration, the **apiserver/cluster** resource, to enable the use of the user-provided certificate.

4.1.3. Management

User-provided certificates are managed by the user.

4.1.4. Expiration

API server client certificate expiration is less than five minutes.

User-provided certificates are managed by the user.

4.1.5. Customization

Update the secret containing the user-managed certificate as needed.

Additional resources

- [Adding API server certificates](#)

4.2. PROXY CERTIFICATES

4.2.1. Purpose

Proxy certificates allow users to specify one or more custom certificate authority (CA) certificates used by platform components when making egress connections.

The **trustedCA** field of the Proxy object is a reference to a config map that contains a user-provided trusted certificate authority (CA) bundle. This bundle is merged with the Red Hat Enterprise Linux CoreOS (RHCOS) trust bundle and injected into the trust store of platform components that make egress HTTPS calls. For example, **image-registry-operator** calls an external image registry to download images. If **trustedCA** is not specified, only the RHCOS trust bundle is used for proxied HTTPS connections. Provide custom CA certificates to the RHCOS trust bundle if you want to use your own certificate infrastructure.

The **trustedCA** field should only be consumed by a proxy validator. The validator is responsible for reading the certificate bundle from required key **ca-bundle.crt** and copying it to a config map named

trusted-ca-bundle in the **openshift-config-managed** namespace. The namespace for the config map referenced by **trustedCA** is **openshift-config**:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: user-ca-bundle
  namespace: openshift-config
data:
  ca-bundle.crt: |
    -----BEGIN CERTIFICATE-----
    Custom CA certificate bundle.
    -----END CERTIFICATE-----
```

Additional resources

- [Configuring the cluster-wide proxy](#)

4.2.2. Managing proxy certificates during installation

The **additionalTrustBundle** value of the installer configuration is used to specify any proxy-trusted CA certificates during installation. For example:

```
$ cat install-config.yaml
```

Example output

```
...
proxy:
  httpProxy: http://<https://username:password@proxy.example.com:123/>
  httpsProxy: https://<https://username:password@proxy.example.com:123/>
  noProxy: <123.example.com,10.88.0.0/16>
  additionalTrustBundle: |
    -----BEGIN CERTIFICATE-----
    <MY_HTTPS_PROXY_TRUSTED_CA_CERT>
    -----END CERTIFICATE-----
...
```

4.2.3. Location

The user-provided trust bundle is represented as a config map. The config map is mounted into the file system of platform components that make egress HTTPS calls. Typically, Operators mount the config map to **/etc/pki/ca-trust/extracted/pem/tls-ca-bundle.pem**, but this is not required by the proxy. A proxy can modify or inspect the HTTPS connection. In either case, the proxy must generate and sign a new certificate for the connection.

Complete proxy support means connecting to the specified proxy and trusting any signatures it has generated. Therefore, it is necessary to let the user specify a trusted root, such that any certificate chain connected to that trusted root is also trusted.

If using the RHCOS trust bundle, place CA certificates in **/etc/pki/ca-trust/source/anchors**.

See [Using shared system certificates](#) in the Red Hat Enterprise Linux documentation for more information.

4.2.4. Expiration

The user sets the expiration term of the user-provided trust bundle.

The default expiration term is defined by the CA certificate itself. It is up to the CA administrator to configure this for the certificate before it can be used by OpenShift Container Platform or RHCOS.



NOTE

Red Hat does not monitor for when CAs expire. However, due to the long life of CAs, this is generally not an issue. However, you might need to periodically update the trust bundle.

4.2.5. Services

By default, all platform components that make egress HTTPS calls will use the RHCOS trust bundle. If **trustedCA** is defined, it will also be used.

Any service that is running on the RHCOS node is able to use the trust bundle of the node.

4.2.6. Management

These certificates are managed by the system and not the user.

4.2.7. Customization

Updating the user-provided trust bundle consists of either:

- updating the PEM-encoded certificates in the config map referenced by **trustedCA**, or
- creating a config map in the namespace **openshift-config** that contains the new trust bundle and updating **trustedCA** to reference the name of the new config map.

The mechanism for writing CA certificates to the RHCOS trust bundle is exactly the same as writing any other file to RHCOS, which is done through the use of machine configs. When the Machine Config Operator (MCO) applies the new machine config that contains the new CA certificates, the node is rebooted. During the next boot, the service **coreos-update-ca-trust.service** runs on the RHCOS nodes, which automatically update the trust bundle with the new CA certificates. For example:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 50-examplecorp-ca-cert
spec:
  config:
    ignition:
      version: 3.1.0
    storage:
      files:
      - contents:
          source: data:text/plain;charset=utf-
8;base64,LS0tLS1CRUdJTlBDRVJUSUZJQ0FURSU0tLS0tCk1JSUVVORENDQXh5Z0F3SUJBZ0lKQU5
1bkkwRDY2MmNuTUEwR0NTcUdTZWlzMFRFFQkN3VUFNSUdsTVFzd0NRWUQKV1FRR0V3SIZVek
VYTUJVR0ExVUVDQXdPVG05eWRHZ2dRMkZ5JjJ4cGJtRXhFREFPQmdOVkKJBY01CMUpoYkdWcA
```

```
pBMmd4RmpBVUJnTIZCQW9NRFZKbFpDQkIZWFFzSUVsdVI5NHhFekFSQmdOVkJBc01DbEpsWk
NCSVIYUWdTVIF4Ckh6QVpCZ05WQkFNTUVsSmxaQ0JJWVhRZ1NWUWdVbTI2ZENCRRFFURWhN
QjhHQ1NxR1NJYjNEUUVKQVJZU2FXNW0KWGpDQnBURUxNQWtHQTFVRUJoTUNWVnk14RnpBV
kJnTIZCQWdNRGs1dmNuUm9JRUs0Y205c2FXNWWhNUkF3RGdZRApXUVFIREFkU1IXeGxhV2RvTV
JZd0ZBWURWUUVFLREExU1pXUWdTR0YwTENCSSmJtTXVNUk13RVFZRFZRUUxEQXBTCkFXUWd
TR0YwSUVsVU1Sc3dHUUVIEVIFRRERCSINaV1FnU0dGMEIFbFVJRkp2YjNRZ1EwRXhJVEFmQmdrc
WhraUcKMhCwQkNRRVdFbWx1Wm05elpXTkFjbVZrYUdGMEExtTnZiVENDQVNjd0RRWUplb1pJaH
ZjTkFRRUJCUEFEZ2dFUApCRENDQVFvQ2dnRUJBTFF0OU9KUWg2R0M1TFQxZzgwU5oMHU1
MEJRNHNal3laOGFFVHh0KzVsbIBWWDZNSet6CmQvaTdsRHFUZIRjZkxMMm55VUJkMmZRRGsX
QjBmeHJza2hHSUlaM2ImUDFQczRsdFRrdjhoUINvYjNWdE5xU28KSHhrS2Z2RDJQS2pUUHhEUfDz
eXJ1eTlpcKxaaW9NZmZpM2kvZ0N1dDBaV3RBeU8zTVZINXFXRi9lbt3Z1BFUwpZOXBvK1RkQ3ZS
Qi9SVU9iQmFNNzYxRWNYTFNNMUdxSE51ZVNmcW5obzNBakxRNmRCbIBXbG82MzhabTFWZWJ
LCKnFTHloa0xXTVNGa0t3RG1uZTBqUTAyWTRnMDc1dkNLdkNzQ0F3RUFBYU5qTUdFd0hRWUR
WUjBPQkJZRUZIN1IKNXIDK1VlaEIJUGV1TDhacXczUHpiZ2NaTUI4R0ExVWRJd1FZTUJhQUZIN1I0
eUMrVWVoSUIQZXVMOFpxdzNQegpjZ2NaTUE4R0ExVWRfD0VCL3dRRk1BTUJBJh3RGdZRFZS
MFBBUUGvQkFRREFnR0dNQTBHQ1NxR1NJYjNEUUVCCkR3VUFBNEICQVFCRE52RDJWbTlZQT
VBOUFsT0pSOCTljbYVYejloWGN4Sk1cGh4Y1pROGpGb0cwNFZzaHkMGUKTUVuVXJNY2ZGZ0laN
G5qTUtUUUNNNFpGVVBBaWV5THg0ZjUySHVEb3BwM2U1SnJTWZk0tGY05JcEt3Q3NhawpwU2
9LdEIVT3NVSKs3cUJWWnhjckl5ZVFWMnFjWU9IWmh0UzV3QnFJd09BaEZ3bENFVDdaZTU4UUhtUz
Q4c2xqCjVIVGtSaml2QWxFeHJGektjbGpDNGF4S1Fsbk92Vkf6eitHbTMyVTB4UEJGNEJ5ZVBWeEN
KVUh3MVRzeVRtZWwKU3hORXA3eUhwWGN3bitmWG5hK3Q1SlDoMWd4VvP0eTMKLS0tLS1FTkQ
gQ0VSVEIGSUNBVEUtlS0tLQo=
```

```
mode: 0644
```

```
overwrite: true
```

```
path: /etc/pki/ca-trust/source/anchors/examplecorp-ca.crt
```

The trust store of machines must also support updating the trust store of nodes.

4.2.8. Renewal

There are no Operators that can auto-renew certificates on the RHCOS nodes.



NOTE

Red Hat does not monitor for when CAs expire. However, due to the long life of CAs, this is generally not an issue. However, you might need to periodically update the trust bundle.

4.3. SERVICE CA CERTIFICATES

4.3.1. Purpose

service-ca is an Operator that creates a self-signed CA when an OpenShift Container Platform cluster is deployed.

4.3.2. Expiration

A custom expiration term is not supported. The self-signed CA is stored in a secret with qualified name **service-ca/signing-key** in fields **tls.crt** (certificate(s)), **tls.key** (private key), and **ca-bundle.crt** (CA bundle).

Other services can request a service serving certificate by annotating a service resource with **service.beta.openshift.io/serving-cert-secret-name: <secret name>**. In response, the Operator generates a new certificate, as **tls.crt**, and private key, as **tls.key** to the named secret. The certificate is valid for two years.

Other services can request that the CA bundle for the service CA be injected into API service or config map resources by annotating with **service.beta.openshift.io/inject-cabundle: true** to support validating certificates generated from the service CA. In response, the Operator writes its current CA bundle to the **CABundle** field of an API service or as **service-ca.crt** to a config map.

As of OpenShift Container Platform 4.3.5, automated rotation is supported and is backported to some 4.2.z and 4.3.z releases. For any release supporting automated rotation, the service CA is valid for 26 months and is automatically refreshed when there is less than 13 months validity left. If necessary, you can manually refresh the service CA.

The service CA expiration of 26 months is longer than the expected upgrade interval for a supported OpenShift Container Platform cluster, such that non-control plane consumers of service CA certificates will be refreshed after CA rotation and prior to the expiration of the pre-rotation CA.



WARNING

A manually-rotated service CA does not maintain trust with the previous service CA. You might experience a temporary service disruption until the Pods in the cluster are restarted, which ensures that Pods are using service serving certificates issued by the new service CA.

4.3.3. Management

These certificates are managed by the system and not the user.

4.3.4. Services

Services that use service CA certificates include:

- cluster-autoscaler-operator
- cluster-monitoring-operator
- cluster-authentication-operator
- cluster-image-registry-operator
- cluster-ingress-operator
- cluster-kube-apiserver-operator
- cluster-kube-controller-manager-operator
- cluster-kube-scheduler-operator
- cluster-networking-operator
- cluster-openshift-apiserver-operator
- cluster-openshift-controller-manager-operator
- cluster-samples-operator

- `machine-config-operator`
- `console-operator`
- `insights-operator`
- `machine-api-operator`
- `operator-lifecycle-manager`

This is not a comprehensive list.

Additional resources

- [Manually rotate service serving certificates](#)
- [Securing service traffic using service serving certificate secrets](#)

4.4. NODE CERTIFICATES

4.4.1. Purpose

Node certificates are signed by the cluster; they come from a certificate authority (CA) that is generated by the bootstrap process. Once the cluster is installed, the node certificates are auto-rotated.

4.4.2. Management

These certificates are managed by the system and not the user.

Additional resources

- [Working with nodes](#)

4.5. BOOTSTRAP CERTIFICATES

4.5.1. Purpose

The kubelet, in OpenShift Container Platform 4 and later, uses the bootstrap certificate located in `/etc/kubernetes/kubeconfig` to initially bootstrap. This is followed by the [bootstrap initialization process](#) and [authorization of the kubelet to create a CSR](#) .

In that process, the kubelet generates a CSR while communicating over the bootstrap channel. The controller manager signs the CSR, resulting in a certificate that the kubelet manages.

4.5.2. Management

These certificates are managed by the system and not the user.

4.5.3. Expiration

This bootstrap CA is valid for 10 years.

The kubelet-managed certificate is valid for one year and rotates automatically at around the 80 percent mark of that one year.

4.5.4. Customization

You cannot customize the bootstrap certificates.

4.6. ETCD CERTIFICATES

4.6.1. Purpose

etcd certificates are signed by the etcd-signer; they come from a certificate authority (CA) that is generated by the bootstrap process.

4.6.2. Expiration

The CA certificates are valid for 10 years. The peer, client, and server certificates are valid for three years.

4.6.3. Management

These certificates are managed by the system and not the user.

4.6.4. Services

etcd certificates are used for encrypted communication between etcd member peers, as well as encrypted client traffic. The following certificates are generated and used by etcd and other processes that communicate with etcd:

- Peer certificates: Used for communication between etcd members.
- Client certificates: Used for encrypted server-client communication. Client certificates are currently used by the API server only, and no other service should connect to etcd directly except for the proxy. Client secrets (**etcd-client**, **etcd-metric-client**, **etcd-metric-signer**, and **etcd-signer**) are added to the **openshift-config**, **openshift-monitoring**, and **openshift-kube-apiserver** namespaces.
- Server certificates: Used by the etcd server for authenticating client requests.
- Metric certificates: All metric consumers connect to proxy with metric-client certificates.

Additional resources

- [Restoring to a previous cluster state](#)

4.7. OLM CERTIFICATES

4.7.1. Management

All certificates for OpenShift Lifecycle Manager (OLM) components (**olm-operator**, **catalog-operator**, **packageserver**, and **marketplace-operator**) are managed by the system.

When installing Operators that include webhooks or API services in their **ClusterServiceVersion** (CSV) object, OLM creates and rotates the certificates for these resources. Certificates for resources in the **openshift-operator-lifecycle-manager** namespace are managed by OLM.

OLM will not update the certificates of Operators that it manages in proxy environments. These certificates must be managed by the user using the subscription config.

4.8. USER-PROVIDED CERTIFICATES FOR DEFAULT INGRESS

4.8.1. Purpose

Applications are usually exposed at `<route_name>.apps.<cluster_name>.<base_domain>`. The `<cluster_name>` and `<base_domain>` come from the installation config file. `<route_name>` is the host field of the route, if specified, or the route name. For example, **hello-openshift-default.apps.username.devcluster.openshift.com**. **hello-openshift** is the name of the route and the route is in the default namespace. You might want clients to access the applications without the need to distribute the cluster-managed CA certificates to the clients. The administrator must set a custom default certificate when serving application content.



WARNING

The Ingress Operator generates a default certificate for an Ingress Controller to serve as a placeholder until you configure a custom default certificate. Do not use operator-generated default certificates in production clusters.

4.8.2. Location

The user-provided certificates must be provided in a **tls** type **Secret** resource in the **openshift-ingress** namespace. Update the **IngressController** CR in the **openshift-ingress-operator** namespace to enable the use of the user-provided certificate. For more information on this process, see [Setting a custom default certificate](#).

4.8.3. Management

User-provided certificates are managed by the user.

4.8.4. Expiration

User-provided certificates are managed by the user.

4.8.5. Services

Applications deployed on the cluster use user-provided certificates for default ingress.

4.8.6. Customization

Update the secret containing the user-managed certificate as needed.

Additional resources

- [Replacing the default ingress certificate](#)

4.9. INGRESS CERTIFICATES

4.9.1. Purpose

The Ingress Operator uses certificates for:

- Securing access to metrics for Prometheus.
- Securing access to routes.

4.9.2. Location

To secure access to Ingress Operator and Ingress Controller metrics, the Ingress Operator uses service serving certificates. The Operator requests a certificate from the **service-ca** controller for its own metrics, and the **service-ca** controller puts the certificate in a secret named **metrics-tls** in the **openshift-ingress-operator** namespace. Additionally, the Ingress Operator requests a certificate for each Ingress Controller, and the **service-ca** controller puts the certificate in a secret named **router-metrics-certs-`<name>`**, where **<name>** is the name of the Ingress Controller, in the **openshift-ingress** namespace.

Each Ingress Controller has a default certificate that it uses for secured routes that do not specify their own certificates. Unless you specify a custom certificate, the Operator uses a self-signed certificate by default. The Operator uses its own self-signed signing certificate to sign any default certificate that it generates. The Operator generates this signing certificate and puts it in a secret named **router-ca** in the **openshift-ingress-operator** namespace. When the Operator generates a default certificate, it puts the default certificate in a secret named **router-certs-`<name>`** (where **<name>** is the name of the Ingress Controller) in the **openshift-ingress** namespace.



WARNING

The Ingress Operator generates a default certificate for an Ingress Controller to serve as a placeholder until you configure a custom default certificate. Do not use Operator-generated default certificates in production clusters.

4.9.3. Workflow

Figure 4.1. Custom certificate workflow

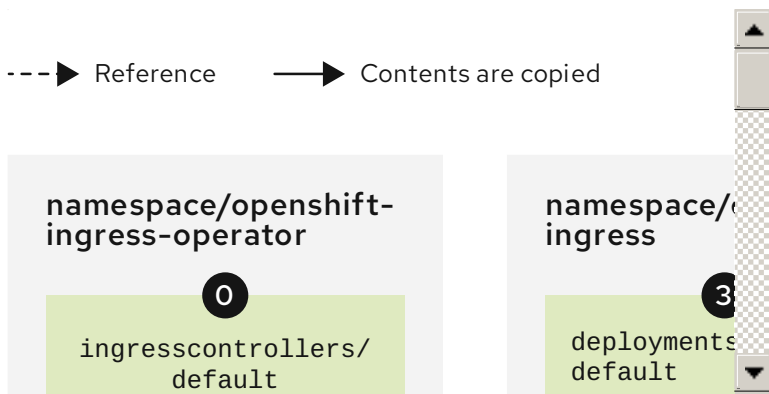
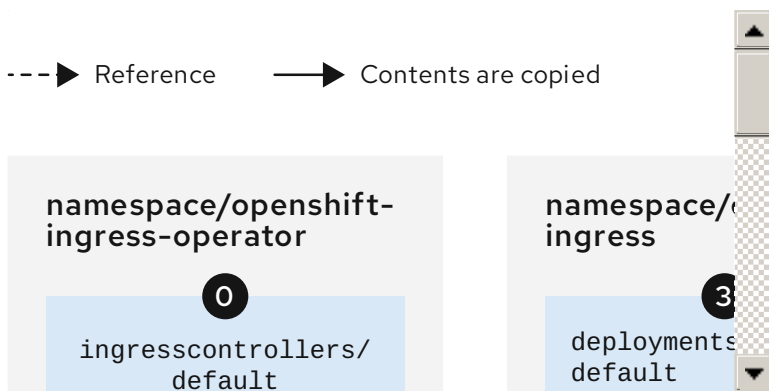


Figure 4.2. Default certificate workflow



- 0** An empty **defaultCertificate** field causes the Ingress Operator to use its self-signed CA to generate a serving certificate for the specified domain.
- 1** The default CA certificate and key generated by the Ingress Operator. Used to sign Operator-generated default serving certificates.
- 2** In the default workflow, the wildcard default serving certificate, created by the Ingress Operator and signed using the generated default CA certificate. In the custom workflow, this is the user-provided certificate.
- 3** The router deployment. Uses the certificate in **secrets/router-certs-default** as its default front-end server certificate.
- 4** In the default workflow, the contents of the wildcard default serving certificate (public and private parts) are copied here to enable OAuth integration. In the custom workflow, this is the user-provided certificate.
- 5** The public (certificate) part of the default serving certificate. Replaces the **configmaps/router-ca** resource.
- 6** The user updates the cluster proxy configuration with the CA certificate that signed the **ingresscontroller** serving certificate. This enables components like **auth**, **console**, and the registry to trust the serving certificate.

- 7 The cluster-wide trusted CA bundle containing the combined Red Hat Enterprise Linux CoreOS (RHCOS) and user-provided CA bundles or an RHCOS-only bundle if a user bundle is not provided.
- 8 The custom CA certificate bundle, which instructs other components (for example, **auth** and **console**) to trust an **ingresscontroller** configured with a custom certificate.
- 9 The **trustedCA** field is used to reference the user-provided CA bundle.
- 10 The Cluster Network Operator injects the trusted CA bundle into the **proxy-ca** config map.
- 11 OpenShift Container Platform 4.6 and newer use **default-ingress-cert**.

4.9.4. Expiration

The expiration terms for the Ingress Operator's certificates are as follows:

- The expiration date for metrics certificates that the **service-ca** controller creates is two years after the date of creation.
- The expiration date for the Operator's signing certificate is two years after the date of creation.
- The expiration date for default certificates that the Operator generates is two years after the date of creation.

You cannot specify custom expiration terms on certificates that the Ingress Operator or **service-ca** controller creates.

You cannot specify expiration terms when installing OpenShift Container Platform for certificates that the Ingress Operator or **service-ca** controller creates.

4.9.5. Services

Prometheus uses the certificates that secure metrics.

The Ingress Operator uses its signing certificate to sign default certificates that it generates for Ingress Controllers for which you do not set custom default certificates.

Cluster components that use secured routes may use the default Ingress Controller's default certificate.

Ingress to the cluster via a secured route uses the default certificate of the Ingress Controller by which the route is accessed unless the route specifies its own certificate.

4.9.6. Management

Ingress certificates are managed by the user. See [Replacing the default ingress certificate](#) for more information.

4.9.7. Renewal

The **service-ca** controller automatically rotates the certificates that it issues. However, it is possible to use **oc delete secret <secret>** to manually rotate service serving certificates.

The Ingress Operator does not rotate its own signing certificate or the default certificates that it generates. Operator-generated default certificates are intended as placeholders for custom default certificates that you configure.

4.10. MONITORING AND CLUSTER LOGGING OPERATOR COMPONENT CERTIFICATES

4.10.1. Expiration

Monitoring components secure their traffic with service CA certificates. These certificates are valid for 2 years and are replaced automatically on rotation of the service CA, which is every 13 months.

If the certificate lives in the **openshift-monitoring** or **openshift-logging** namespace, it is system managed and rotated automatically.

4.10.2. Management

These certificates are managed by the system and not the user.

4.11. CONTROL PLANE CERTIFICATES

4.11.1. Location

Control plane certificates are included in these namespaces:

- openshift-config-managed
- openshift-kube-apiserver
- openshift-kube-apiserver-operator
- openshift-kube-controller-manager
- openshift-kube-controller-manager-operator
- openshift-kube-scheduler

4.11.2. Management

Control plane certificates are managed by the system and rotated automatically.

In the rare case that your control plane certificates have expired, see [Recovering from expired control plane certificates](#).

CHAPTER 5. COMPLIANCE OPERATOR

5.1. COMPLIANCE OPERATOR RELEASE NOTES

The Compliance Operator lets OpenShift Container Platform administrators describe the required compliance state of a cluster and provides them with an overview of gaps and ways to remediate them.

These release notes track the development of the Compliance Operator in the OpenShift Container Platform.

For an overview of the Compliance Operator, see [Understanding the Compliance Operator](#).

5.1.1. OpenShift Compliance Operator 0.1.53

The following advisory is available for the OpenShift Compliance Operator 0.1.53:

- [RHBA-2022:5537 - OpenShift Compliance Operator bug fix update](#)

5.1.1.1. Bug fixes

- Previously, the **ocp4-kubelet-enable-streaming-connections** rule contained an incorrect variable comparison, resulting in false positive scan results. Now, the Compliance Operator provides accurate scan results when setting **streamingConnectionIdleTimeout**. ([BZ#2069891](#))
- Previously, group ownership for **/etc/openswitch/conf.db** was incorrect on IBM Z architectures, resulting in **ocp4-cis-node-worker-file-groupowner-ovs-conf-db** check failures. Now, the check is marked **NOT-APPLICABLE** on IBM Z architecture systems. ([BZ#2072597](#))
- Previously, the **ocp4-cis-scc-limit-container-allowed-capabilities** rule reported in a **FAIL** state due to incomplete data regarding the security context constraints (SCC) rules in the deployment. Now, the result is **MANUAL**, which is consistent with other checks that require human intervention. ([BZ#2077916](#))
- Previously, the following rules failed to account for additional configuration paths for API servers and TLS certificates and keys, resulting in reported failures even if the certificates and keys were set properly:
 - **ocp4-cis-api-server-kubelet-client-cert**
 - **ocp4-cis-api-server-kubelet-client-key**
 - **ocp4-cis-kubelet-configure-tls-cert**
 - **ocp4-cis-kubelet-configure-tls-key**

Now, the rules report accurately and observe legacy file paths specified in the kubelet configuration file. ([BZ#2079813](#))

- Previously, the **content_rule_oauth_or_oauthclient_inactivity_timeout** rule did not account for a configurable timeout set by the deployment when assessing compliance for timeouts. This resulted in the rule failing even if the timeout was valid. Now, the Compliance Operator uses the **var_oauth_inactivity_timeout** variable to set valid timeout length. ([BZ#2081952](#))
- Previously, the Compliance Operator used administrative permissions on namespaces not

labeled appropriately for privileged use, resulting in warning messages regarding pod security-level violations. Now, the Compliance Operator has appropriate namespace labels and permission adjustments to access results without violating permissions. ([BZ#2088202](#))

- Previously, applying auto remediations for **rhcos4-high-master-sysctl-kernel-yama-pttrace-scope** and **rhcos4-sysctl-kernel-core-pattern** resulted in subsequent failures of those rules in scan results, even though they were remediated. Now, the rules report **PASS** accurately, even after remediations are applied. ([BZ#2094382](#))
- Previously, the Compliance Operator would fail in a **CrashLoopBackoff** state because of out-of-memory exceptions. Now, the Compliance Operator is improved to handle large machine configuration data sets in memory and function correctly. ([BZ#2094854](#))

5.1.1.2. Known issue

- When **"debug":true** is set within the **ScanSettingBinding** object, the pods generated by the **ScanSettingBinding** object are not removed when that binding is deleted. As a workaround, run the following command to delete the remaining pods:

```
$ oc delete pods -l compliance.openshift.io/scan-name=ocp4-cis
```

([BZ#2092913](#))

5.1.2. OpenShift Compliance Operator 0.1.52

The following advisory is available for the OpenShift Compliance Operator 0.1.52:

- [RHBA-2022:4657 - OpenShift Compliance Operator bug fix update](#)

5.1.2.1. New features and enhancements

- The FedRAMP high SCAP profile is now available for use in OpenShift Container Platform environments. For more information, See [Supported compliance profiles](#).

5.1.2.2. Bug fixes

- Previously, the **OpenScap** container would crash due to a mount permission issue in a security environment where **DAC_OVERRIDE** capability is dropped. Now, executable mount permissions are applied to all users. ([BZ#2082151](#))
- Previously, the compliance rule **ocp4-configure-network-policies** could be configured as **MANUAL**. Now, compliance rule **ocp4-configure-network-policies** is set to **AUTOMATIC**. ([BZ#2072431](#))
- Previously, the Cluster Autoscaler would fail to scale down because the Compliance Operator scan pods were never removed after a scan. Now, the pods are removed from each node by default unless explicitly saved for debugging purposes. ([BZ#2075029](#))
- Previously, applying the Compliance Operator to the **KubeletConfig** would result in the node going into a **NotReady** state due to unpausing the Machine Config Pools too early. Now, the Machine Config Pools are unpaused appropriately and the node operates correctly. ([BZ#2071854](#))
- Previously, the Machine Config Operator used **base64** instead of **url-encoded** code in the latest release, causing Compliance Operator remediation to fail. Now, the Compliance Operator

checks encoding to handle both **base64** and **url-encoded** Machine Config code and the remediation applies correctly. ([BZ#2082431](#))

5.1.2.3. Known issue

- When "**debug**":**true** is set within the **ScanSettingBinding** object, the pods generated by the **ScanSettingBinding** object are not removed when that binding is deleted. As a workaround, run the following command to delete the remaining pods:

```
$ oc delete pods -l compliance.openshift.io/scan-name=ocp4-cis
```

([BZ#2092913](#))

5.1.3. OpenShift Compliance Operator 0.1.49

The following advisory is available for the OpenShift Compliance Operator 0.1.49:

- [RHBA-2022:1148 - OpenShift Compliance Operator bug fix and enhancement update](#)

5.1.3.1. Bug fixes

- Previously, the **openshift-compliance** content did not include platform-specific checks for network types. As a result, OVN- and SDN-specific checks would show as **failed** instead of **not-applicable** based on the network configuration. Now, new rules contain platform checks for networking rules, resulting in a more accurate assessment of network-specific checks. ([BZ#1994609](#))
- Previously, the **ocp4-moderate-routes-protected-by-tls** rule incorrectly checked TLS settings that results in the rule failing the check, even if the connection secure SSL TLS protocol. Now, the check will properly evaluate TLS settings that are consistent with the networking guidance and profile recommendations. ([BZ#2002695](#))
- Previously, **ocp-cis-configure-network-policies-namespace** used pagination when requesting namespaces. This caused the rule to fail because the deployments truncated lists of more than 500 namespaces. Now, the entire namespace list is requested, and the rule for checking configured network policies will work for deployments with more than 500 namespaces. ([BZ#2038909](#))
- Previously, remediations using the **sshd jinja** macros were hard-coded to specific sshd configurations. As a result, the configurations were inconsistent with the content the rules were checking for and the check would fail. Now, the sshd configuration is parameterized and the rules apply successfully. ([BZ#2049141](#))
- Previously, the **ocp4-cluster-version-operator-verify-integrity** always checked the first entry in the Cluster Version Operator (CVO) history. As a result, the upgrade would fail in situations where subsequent versions of {product-name} would be verified. Now, the compliance check result for **ocp4-cluster-version-operator-verify-integrity** is able to detect verified versions and is accurate with the CVO history. ([BZ#2053602](#))
- Previously, the **ocp4-api-server-no-adm-ctrl-plugins-disabled** rule did not check for a list of empty admission controller plug-ins. As a result, the rule would always fail, even if all admission plug-ins were enabled. Now, more robust checking of the **ocp4-api-server-no-adm-ctrl-plugins-disabled** rule will accurately pass with all admission controller plug-ins enabled. ([BZ#2058631](#))

- Previously, scans did not contain platform checks for running against Linux worker nodes. As a result, running scans against worker nodes that were not Linux-based resulted in a never ending scan loop. Now, the scan will schedule appropriately based on platform type and labels and will completely successfully. ([BZ#2056911](#))

5.1.4. OpenShift Compliance Operator 0.1.48

The following advisory is available for the OpenShift Compliance Operator 0.1.48:

- [RHBA-2022:0416 - OpenShift Compliance Operator bug fix and enhancement update](#)

5.1.4.1. Bug fixes

- Previously, some rules associated with extended Open Vulnerability and Assessment Language (OVAL) definitions had a **checkType** of **None**. This was because the Compliance Operator was not processing extended OVAL definitions when parsing rules. With this update, content from extended OVAL definitions is parsed so that these rules now have a **checkType** of either **Node** or **Platform**. ([BZ#2040282](#))
- Previously, a manually created **MachineConfig** object for **KubeletConfig** prevented a **KubeletConfig** object from being generated for remediation, leaving the remediation in the **Pending** state. With this release, a **KubeletConfig** object is created by the remediation, regardless if there is a manually created **MachineConfig** object for **KubeletConfig**. As a result, **KubeletConfig** remediations now work as expected. ([BZ#2040401](#))

5.1.5. OpenShift Compliance Operator 0.1.47

The following advisory is available for the OpenShift Compliance Operator 0.1.47:

- [RHBA-2022:0014 - OpenShift Compliance Operator bug fix and enhancement update](#)

5.1.5.1. New features and enhancements

- The Compliance Operator now supports the following compliance benchmarks for the Payment Card Industry Data Security Standard (PCI DSS):
 - ocp4-pci-dss
 - ocp4-pci-dss-node
- Additional rules and remediations for FedRAMP moderate impact level are added to the OCP4-moderate, OCP4-moderate-node, and rhcos4-moderate profiles.
- Remediations for KubeletConfig are now available in node-level profiles.

5.1.5.2. Bug fixes

- Previously, if your cluster was running OpenShift Container Platform 4.6 or earlier, remediations for USBGuard-related rules would fail for the moderate profile. This is because the remediations created by the Compliance Operator were based on an older version of USBGuard that did not support drop-in directories. Now, invalid remediations for USBGuard-related rules are not created for clusters running OpenShift Container Platform 4.6. If your cluster is using OpenShift Container Platform 4.6, you must manually create remediations for USBGuard-related rules.

Additionally, remediations are created only for rules that satisfy minimum version requirements. ([BZ#1965511](#))

- Previously, when rendering remediations, the compliance operator would check that the remediation was well-formed by using a regular expression that was too strict. As a result, some remediations, such as those that render **sshd_config**, would not pass the regular expression check and therefore, were not created. The regular expression was found to be unnecessary and removed. Remediations now render correctly. ([BZ#2033009](#))

5.1.6. OpenShift Compliance Operator 0.1.44

The following advisory is available for the OpenShift Compliance Operator 0.1.44:

- [RHBA-2021:4530 - OpenShift Compliance Operator bug fix and enhancement update](#)

5.1.6.1. New features and enhancements

- In this release, the **strictNodeScan** option is now added to the **ComplianceScan**, **ComplianceSuite** and **ScanSetting** CRs. This option defaults to **true** which matches the previous behavior, where an error occurred if a scan was not able to be scheduled on a node. Setting the option to **false** allows the Compliance Operator to be more permissive about scheduling scans. Environments with ephemeral nodes can set the **strictNodeScan** value to false, which allows a compliance scan to proceed, even if some of the nodes in the cluster are not available for scheduling.
- You can now customize the node that is used to schedule the result server workload by configuring the **nodeSelector** and **tolerations** attributes of the **ScanSetting** object. These attributes are used to place the **ResultServer** pod, the pod that is used to mount a PV storage volume and store the raw Asset Reporting Format (ARF) results. Previously, the **nodeSelector** and the **tolerations** parameters defaulted to selecting one of the control plane nodes and tolerating the **node-role.kubernetes.io/master** taint. This did not work in environments where control plane nodes are not permitted to mount PVs. This feature provides a way for you to select the node and tolerate a different taint in those environments.
- The Compliance Operator can now remediate **KubeletConfig** objects.
- A comment containing an error message is now added to help content developers differentiate between objects that do not exist in the cluster versus objects that cannot be fetched.
- Rule objects now contain two new attributes, **checkType** and **description**. These attributes allow you to determine if the rule pertains to a node check or platform check, and also allow you to review what the rule does.
- This enhancement removes the requirement that you have to extend an existing profile in order to create a tailored profile. This means the **extends** field in the **TailoredProfile** CRD is no longer mandatory. You can now select a list of rule objects to create a tailored profile. Note that you must select whether your profile applies to nodes or the platform by setting the **compliance.openshift.io/product-type:** annotation or by setting the **-node** suffix for the **TailoredProfile** CR.
- In this release, the Compliance Operator is now able to schedule scans on all nodes irrespective of their taints. Previously, the scan pods would only tolerate the **node-role.kubernetes.io/master** taint, meaning that they would either run on nodes with no taints or only on nodes with the **node-role.kubernetes.io/master** taint. In deployments that use custom taints for their nodes, this resulted in the scans not being scheduled on those nodes. Now, the scan pods tolerate all node taints.

- In this release, the Compliance Operator supports the following North American Electric Reliability Corporation (NERC) security profiles:
 - ocp4-nerc-cip
 - ocp4-nerc-cip-node
 - rhcos4-nerc-cip
- In this release, the Compliance Operator supports the NIST 800-53 Moderate-Impact Baseline for the Red Hat OpenShift - Node level, ocp4-moderate-node, security profile.

5.1.6.2. Templating and variable use

- In this release, the remediation template now allows multi-value variables.
- With this update, the Compliance Operator can change remediations based on variables that are set in the compliance profile. This is useful for remediations that include deployment-specific values such as time outs, NTP server host names, or similar. Additionally, the **ComplianceCheckResult** objects now use the label **compliance.openshift.io/check-has-value** that lists the variables a check has used.

5.1.6.3. Bug fixes

- Previously, while performing a scan, an unexpected termination occurred in one of the scanner containers of the pods. In this release, the Compliance Operator uses the latest OpenSCAP version 1.3.5 to avoid a crash.
- Previously, using **autoReplyRemediations** to apply remediations triggered an update of the cluster nodes. This was disruptive if some of the remediations did not include all of the required input variables. Now, if a remediation is missing one or more required input variables, it is assigned a state of **NeedsReview**. If one or more remediations are in a **NeedsReview** state, the machine config pool remains paused, and the remediations are not applied until all of the required variables are set. This helps minimize disruption to the nodes.
- The RBAC Role and Role Binding used for Prometheus metrics are changed to 'ClusterRole' and 'ClusterRoleBinding' to ensure that monitoring works without customization.
- Previously, if an error occurred while parsing a profile, rules or variables objects were removed and deleted from the profile. Now, if an error occurs during parsing, the **profileparser** annotates the object with a temporary annotation that prevents the object from being deleted until after parsing completes. ([BZ#1988259](#))
- Previously, an error occurred if titles or descriptions were missing from a tailored profile. Because the XCCDF standard requires titles and descriptions for tailored profiles, titles and descriptions are now required to be set in **TailoredProfile** CRs.
- Previously, when using tailored profiles, **TailoredProfile** variable values were allowed to be set using only a specific selection set. This restriction is now removed, and **TailoredProfile** variables can be set to any value.

5.1.7. Release Notes for Compliance Operator 0.1.39

The following advisory is available for the OpenShift Compliance Operator 0.1.39:

- [RHBA-2021:3214 - OpenShift Compliance Operator bug fix and enhancement update](#)

5.1.7.1. New features and enhancements

- Previously, the Compliance Operator was unable to parse Payment Card Industry Data Security Standard (PCI DSS) references. Now, the Operator can parse compliance content that ships with PCI DSS profiles.
- Previously, the Compliance Operator was unable to execute rules for AU-5 control in the moderate profile. Now, permission is added to the Operator so that it can read **Prometheusrules.monitoring.coreos.com** objects and run the rules that cover AU-5 control in the moderate profile.

5.1.8. Additional resources

- [Understanding the Compliance Operator](#)

5.2. SUPPORTED COMPLIANCE PROFILES

There are several profiles available as part of the Compliance Operator (CO) installation.

5.2.1. Compliance profiles

The Compliance Operator provides the following compliance profiles:

Table 5.1. Supported compliance profiles

Profile	Profile title	Compliance Operator version	Industry compliance benchmark	Supported architectures
ocp4-cis	CIS Red Hat OpenShift Container Platform 4 Benchmark	0.1.39 +	CIS Benchmarks™ footnote:cisbenchmark[To locate the CIS RedHat OpenShift Container Platform v4 Benchmark, go to CIS Benchmarks and type Kubernetes in the search box. Click on Kubernetes and then Download Latest CIS Benchmark , where you can then register to download the benchmark.]	x86_64 ppc64le s390x
ocp4-cis-node	CIS Red Hat OpenShift Container Platform 4 Benchmark	0.1.39 +	CIS Benchmarks™ footnote:cisbenchmark[]	x86_64 ppc64le s390x
ocp4-e8	Australian Cyber Security Centre (ACSC) Essential Eight	0.1.39 +	ACSC Hardening Linux Workstations and Servers	x86_64

Profile	Profile title	Compliance Operator version	Industry compliance benchmark	Supported architectures
ocp4-moderate	NIST 800-53 Moderate-Impact Baseline for Red Hat OpenShift - Platform level	0.139 +	NIST SP-800-53 Release Search	x86_64
rhcos4-e8	Australian Cyber Security Centre (ACSC) Essential Eight	0.139 +	ACSC Hardening Linux Workstations and Servers	x86_64
rhcos4-moderate	NIST 800-53 Moderate-Impact Baseline for Red Hat Enterprise Linux CoreOS	0.139 +	NIST SP-800-53 Release Search	x86_64
ocp4-moderate-node	NIST 800-53 Moderate-Impact Baseline for Red Hat OpenShift - Node level	0.144 +	NIST SP-800-53 Release Search	x86_64
ocp4-nerc-cip	North American Electric Reliability Corporation (NERC) Critical Infrastructure Protection (CIP) cybersecurity standards profile for the Red Hat OpenShift Container Platform - Platform level	0.144 +	NERC CIP Standards	x86_64
ocp4-nerc-cip-node	North American Electric Reliability Corporation (NERC) Critical Infrastructure Protection (CIP) cybersecurity standards profile for the Red Hat OpenShift Container Platform - Node level	0.144 +	NERC CIP Standards	x86_64
rhcos4-nerc-cip	North American Electric Reliability Corporation (NERC) Critical Infrastructure Protection (CIP) cybersecurity standards profile for Red Hat Enterprise Linux CoreOS	0.144 +	NERC CIP Standards	x86_64
ocp4-pci-dss	PCI-DSS v3.2.1 Control Baseline for Red Hat OpenShift Container Platform 4	0.147 +	PCI Security Standards® Council Document Library	x86_64

Profile	Profile title	Compliance Operator version	Industry compliance benchmark	Supported architectures
ocp4-pci-dss-node	PCI-DSS v3.2.1 Control Baseline for Red Hat OpenShift Container Platform 4	0.1.47 +	PCI Security Standards® Council Document Library	x86_64
ocp4-high	NIST 800-53 High-Impact Baseline for Red Hat OpenShift - Platform level	0.1.52 +	NIST SP-800-53 Release Search	x86_64
ocp4-high-node	NIST 800-53 High-Impact Baseline for Red Hat OpenShift - Node level	0.1.52 +	NIST SP-800-53 Release Search	x86_64
rhcos4-high	NIST 800-53 High-Impact Baseline for Red Hat Enterprise Linux CoreOS	0.1.52 +	NIST SP-800-53 Release Search	x86_64

5.2.2. Additional resources

- For more information about viewing the compliance profiles available in your system, see [Compliance Operator profiles](#) in Understanding the Compliance Operator.

5.3. INSTALLING THE COMPLIANCE OPERATOR

Before you can use the Compliance Operator, you must ensure it is deployed in the cluster.

5.3.1. Installing the Compliance Operator through the web console

Prerequisites

- You must have **admin** privileges.

Procedure

1. In the OpenShift Container Platform web console, navigate to **Operators** → **OperatorHub**.
2. Search for the Compliance Operator, then click **Install**.
3. Keep the default selection of **Installation mode** and **namespace** to ensure that the Operator will be installed to the **openshift-compliance** namespace.
4. Click **Install**.

Verification

To confirm that the installation is successful:

1. Navigate to the **Operators → Installed Operators** page.
2. Check that the Compliance Operator is installed in the **openshift-compliance** namespace and its status is **Succeeded**.

If the Operator is not installed successfully:

1. Navigate to the **Operators → Installed Operators** page and inspect the **Status** column for any errors or failures.
2. Navigate to the **Workloads → Pods** page and check the logs in any pods in the **openshift-compliance** project that are reporting issues.

5.3.2. Installing the Compliance Operator using the CLI

Prerequisites

- You must have **admin** privileges.

Procedure

1. Define a **Namespace** object:

Example namespace-object.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  labels:
    openshift.io/cluster-monitoring: "true"
  name: openshift-compliance
```

2. Create the **Namespace** object:

```
$ oc create -f namespace-object.yaml
```

3. Define an **OperatorGroup** object:

Example operator-group-object.yaml

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: compliance-operator
  namespace: openshift-compliance
spec:
  targetNamespaces:
    - openshift-compliance
```

4. Create the **OperatorGroup** object:

```
$ oc create -f operator-group-object.yaml
```

- 5. Define a **Subscription** object:

Example subscription-object.yaml

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: compliance-operator-sub
  namespace: openshift-compliance
spec:
  channel: "release-0.1"
  installPlanApproval: Automatic
  name: compliance-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

- 6. Create the **Subscription** object:

```
$ oc create -f subscription-object.yaml
```



NOTE

If you are setting the global scheduler feature and enable **defaultNodeSelector**, you must create the namespace manually and update the annotations of the **openshift-compliance** namespace, or the namespace where the Compliance Operator was installed, with **openshift.io/node-selector: ""**. This removes the default node selector and prevents deployment failures.

Verification

- 1. Verify the installation succeeded by inspecting the CSV file:

```
$ oc get csv -n openshift-compliance
```

- 2. Verify that the Compliance Operator is up and running:

```
$ oc get deploy -n openshift-compliance
```

5.3.3. Additional resources

- The Compliance Operator is supported in a restricted network environment. For more information, see [Using Operator Lifecycle Manager on restricted networks](#).

5.4. COMPLIANCE OPERATOR SCANS

The **ScanSetting** and **ScanSettingBinding** APIs are recommended to run compliance scans with the Compliance Operator. For more information on these API objects, run:

```
$ oc explain scansettings
```

or


```
$ oc explain scansettingsbindings
```

5.4.1. Running compliance scans

You can run a scan using the Center for Internet Security (CIS) profiles. For convenience, the Compliance Operator creates a **ScanSetting** object with reasonable defaults on startup. This **ScanSetting** object is named **default**.



NOTE

For all-in-one control plane and worker nodes, the compliance scan runs twice on the worker and control plane nodes. The compliance scan might generate inconsistent scan results. You can avoid inconsistent results by defining only a single role in the **ScanSetting** object.

Procedure

1. Inspect the **ScanSetting** object by running:

```
$ oc describe scansettings default -n openshift-compliance
```

Example output

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSetting
metadata:
  name: default
  namespace: openshift-compliance
rawResultStorage:
  pvAccessModes:
    - ReadWriteOnce 1
  rotation: 3 2
  size: 1Gi 3
  roles:
    - worker 4
    - master 5
  scanTolerations: 6
    default:
      - operator: Exists
  schedule: 0 1 * * * 7
```

- 1** The Compliance Operator creates a persistent volume (PV) that contains the results of the scans. By default, the PV will use access mode **ReadWriteOnce** because the Compliance Operator cannot make any assumptions about the storage classes configured on the cluster. Additionally, **ReadWriteOnce** access mode is available on most clusters. If you need to fetch the scan results, you can do so by using a helper pod, which also binds the volume. Volumes that use the **ReadWriteOnce** access mode can be mounted by only one pod at a time, so it is important to remember to delete the helper pods. Otherwise, the Compliance Operator will not be able to reuse the volume for subsequent scans.
- 2** The Compliance Operator keeps results of three subsequent scans in the volume; older scans are rotated.

- 3 The Compliance Operator will allocate one GB of storage for the scan results.
- 4 5 If the scan setting uses any profiles that scan cluster nodes, scan these node roles.
- 6 The default scan setting object also scans all the nodes.
- 7 The default scan setting object runs scans at 01:00 each day.

As an alternative to the default scan setting, you can use **default-auto-apply**, which has the following settings:

```

apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSetting
metadata:
  name: default-auto-apply
  namespace: openshift-compliance
autoUpdateRemediations: true 1
autoApplyRemediations: true 2
rawResultStorage:
  pvAccessModes:
    - ReadWriteOnce
  rotation: 3
  size: 1Gi
schedule: 0 1 * * *
roles:
  - worker
  - master
scanTolerations:
  default:
    - operator: Exists

```

- 1 2 Setting **autoUpdateRemediations** and **autoApplyRemediations** flags to **true** allows you to easily create **ScanSetting** objects that auto-remediate without extra steps.

2. Create a **ScanSettingBinding** object that binds to the default **ScanSetting** object and scans the cluster using the **cis** and **cis-node** profiles. For example:

```

apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSettingBinding
metadata:
  name: cis-compliance
  namespace: openshift-compliance
profiles:
  - name: ocp4-cis-node
    kind: Profile
    apiGroup: compliance.openshift.io/v1alpha1
  - name: ocp4-cis
    kind: Profile
    apiGroup: compliance.openshift.io/v1alpha1
settingsRef:
  name: default
  kind: ScanSetting
  apiGroup: compliance.openshift.io/v1alpha1

```

3. Create the **ScanSettingBinding** object by running:

```
$ oc create -f <file-name>.yaml -n openshift-compliance
```

At this point in the process, the **ScanSettingBinding** object is reconciled and based on the **Binding** and the **Bound** settings. The Compliance Operator creates a **ComplianceSuite** object and the associated **ComplianceScan** objects.

4. Follow the compliance scan progress by running:

```
$ oc get compliancescan -w -n openshift-compliance
```

The scans progress through the scanning phases and eventually reach the **DONE** phase when complete. In most cases, the result of the scan is **NON-COMPLIANT**. You can review the scan results and start applying remediations to make the cluster compliant. See [Managing Compliance Operator remediation](#) for more information.

5.4.2. Scheduling the result server pod on a worker node

The result server pod mounts the persistent volume (PV) that stores the raw Asset Reporting Format (ARF) scan results. The **nodeSelector** and **tolerations** attributes enable you to configure the location of the result server pod.

This is helpful for those environments where control plane nodes are not permitted to mount persistent volumes.

Procedure

- Create a **ScanSetting** custom resource (CR) for the Compliance Operator:
 - a. Define the **ScanSetting** CR, and save the YAML file, for example, **rs-workers.yaml**:

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSetting
metadata:
  name: rs-on-workers
  namespace: openshift-compliance
rawResultStorage:
  nodeSelector:
    node-role.kubernetes.io/worker: "" 1
  pvAccessModes:
  - ReadWriteOnce
  rotation: 3
  size: 1Gi
  tolerations:
  - operator: Exists 2
roles:
- worker
- master
scanTolerations:
- operator: Exists
schedule: 0 1 * * *
```

- 1** The Compliance Operator uses this node to store scan results in ARF format.

2 The result server pod tolerates all taints.

b. To create the **ScanSetting** CR, run the following command:

```
$ oc create -f rs-workers.yaml
```

Verification

- To verify that the **ScanSetting** object is created, run the following command:

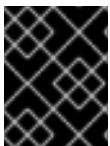
```
$ oc get scansettings rs-on-workers -n openshift-compliance -o yaml
```

Example output

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSetting
metadata:
  creationTimestamp: "2021-11-19T19:36:36Z"
  generation: 1
  name: rs-on-workers
  namespace: openshift-compliance
  resourceVersion: "48305"
  uid: 43fdcf5f-15a7-445a-8bbc-0e4a160cd46e
rawResultStorage:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  pvAccessModes:
  - ReadWriteOnce
  rotation: 3
  size: 1Gi
  tolerations:
  - operator: Exists
roles:
  - worker
  - master
scanTolerations:
  - operator: Exists
schedule: 0 1 * * *
strictNodeScan: true
```

5.5. UNDERSTANDING THE COMPLIANCE OPERATOR

The Compliance Operator lets OpenShift Container Platform administrators describe the required compliance state of a cluster and provides them with an overview of gaps and ways to remediate them. The Compliance Operator assesses compliance of both the Kubernetes API resources of OpenShift Container Platform, as well as the nodes running the cluster. The Compliance Operator uses OpenSCAP, a NIST-certified tool, to scan and enforce security policies provided by the content.



IMPORTANT

The Compliance Operator is available for Red Hat Enterprise Linux CoreOS (RHCOS) deployments only.

5.5.1. Compliance Operator profiles

There are several profiles available as part of the Compliance Operator installation. You can use the **oc get** command to view available profiles, profile details, and specific rules.

- View the available profiles:

```
$ oc get -n <namespace> profiles.compliance
```

This example displays the profiles in the default **openshift-compliance** namespace:

```
$ oc get -n openshift-compliance profiles.compliance
```

Example output

```
NAME          AGE
ocp4-cis      32m
ocp4-cis-node 32m
ocp4-e8       32m
ocp4-moderate 32m
ocp4-moderate-node 32m
ocp4-nerc-cip 32m
ocp4-nerc-cip-node 32m
ocp4-pci-dss  32m
ocp4-pci-dss-node 32m
rhcos4-e8     32m
rhcos4-moderate 32m
rhcos4-nerc-cip 32m
```

These profiles represent different compliance benchmarks. Each profile has the product name that it applies to added as a prefix to the profile's name. **ocp4-e8** applies the Essential 8 benchmark to the OpenShift Container Platform product, while **rhcos4-e8** applies the Essential 8 benchmark to the Red Hat Enterprise Linux CoreOS (RHCOS) product.

- View the details of a profile:

```
$ oc get -n <namespace> -oyaml profiles.compliance <profile name>
```

This example displays the details of the **rhcos4-e8** profile:

```
$ oc get -n openshift-compliance -oyaml profiles.compliance rhcos4-e8
```

Example output

```
apiVersion: compliance.openshift.io/v1alpha1
description: |-
  This profile contains configuration checks for Red Hat
  Enterprise Linux CoreOS that align to the Australian
  Cyber Security Centre (ACSC) Essential Eight.
  A copy of the Essential Eight in Linux Environments guide can
  be found at the ACSC website: ...
id: xccdf_org.ssgproject.content_profile_e8
kind: Profile
metadata:
```

```
annotations:  
  compliance.openshift.io/image-digest: pb-rhcos426smj  
  compliance.openshift.io/product: redhat_enterprise_linux_coreos_4  
  compliance.openshift.io/product-type: Node  
labels:  
  compliance.openshift.io/profile-bundle: rhcos4  
name: rhcos4-e8  
namespace: openshift-compliance  
ownerReferences:  
- apiVersion: compliance.openshift.io/v1alpha1  
  blockOwnerDeletion: true  
  controller: true  
  kind: ProfileBundle  
  name: rhcos4  
rules:  
- rhcos4-accounts-no-uid-except-zero  
- rhcos4-audit-rules-dac-modification-chmod  
- rhcos4-audit-rules-dac-modification-chown  
- rhcos4-audit-rules-execution-chcon  
- rhcos4-audit-rules-execution-restorecon  
- rhcos4-audit-rules-execution-semanage  
- rhcos4-audit-rules-execution-setfiles  
- rhcos4-audit-rules-execution-setsebool  
- rhcos4-audit-rules-execution-seunshare  
- rhcos4-audit-rules-kernel-module-loading-delete  
- rhcos4-audit-rules-kernel-module-loading-finit  
- rhcos4-audit-rules-kernel-module-loading-init  
- rhcos4-audit-rules-login-events  
- rhcos4-audit-rules-login-events-faillock  
- rhcos4-audit-rules-login-events-lastlog  
- rhcos4-audit-rules-login-events-tallylog  
- rhcos4-audit-rules-networkconfig-modification  
- rhcos4-audit-rules-sysadmin-actions  
- rhcos4-audit-rules-time-adjtimex  
- rhcos4-audit-rules-time-clock-settime  
- rhcos4-audit-rules-time-settimeofday  
- rhcos4-audit-rules-time-stime  
- rhcos4-audit-rules-time-watch-localtime  
- rhcos4-audit-rules-usergroup-modification  
- rhcos4-auditd-data-retention-flush  
- rhcos4-auditd-freq  
- rhcos4-auditd-local-events  
- rhcos4-auditd-log-format  
- rhcos4-auditd-name-format  
- rhcos4-auditd-write-logs  
- rhcos4-configure-crypto-policy  
- rhcos4-configure-ssh-crypto-policy  
- rhcos4-no-empty-passwords  
- rhcos4-selinux-policytype  
- rhcos4-selinux-state  
- rhcos4-service-auditd-enabled  
- rhcos4-sshd-disable-empty-passwords  
- rhcos4-sshd-disable-gssapi-auth  
- rhcos4-sshd-disable-rhosts  
- rhcos4-sshd-disable-root-login  
- rhcos4-sshd-disable-user-known-hosts
```

```

- rhcos4-sshd-do-not-permit-user-env
- rhcos4-sshd-enable-strictmodes
- rhcos4-sshd-print-last-log
- rhcos4-sshd-set-loglevel-info
- rhcos4-sysctl-kernel-dmesg-restrict
- rhcos4-sysctl-kernel-kptr-restrict
- rhcos4-sysctl-kernel-randomize-va-space
- rhcos4-sysctl-kernel-unprivileged-bpf-disabled
- rhcos4-sysctl-kernel-yama-pttrace-scope
- rhcos4-sysctl-net-core-bpf-jit-harden
title: Australian Cyber Security Centre (ACSC) Essential Eight

```

- View the rules within a desired profile:

```
$ oc get -n <namespace> -oyaml rules.compliance <rule_name>
```

This example displays the **rhcos4-audit-rules-login-events** rule in the **rhcos4** profile:

```
$ oc get -n openshift-compliance -oyaml rules.compliance rhcos4-audit-rules-login-events
```

Example output

```

apiVersion: compliance.openshift.io/v1alpha1
checkType: Node
description: |-

```

The audit system already collects login information for all users and root. If the auditd daemon is configured to use the augenrules program to read audit rules during daemon startup (the default), add the following lines to a file with suffix.rules in the directory /etc/audit/rules.d in order to watch for attempted manual edits of files involved in storing logon events:

```

-w /var/log/tallylog -p wa -k logins
-w /var/run/faillock -p wa -k logins
-w /var/log/lastlog -p wa -k logins

```

If the auditd daemon is configured to use the auditctl utility to read audit rules during daemon startup, add the following lines to /etc/audit/audit.rules file in order to watch for unattempted manual edits of files involved in storing logon events:

```

-w /var/log/tallylog -p wa -k logins
-w /var/run/faillock -p wa -k logins
-w /var/log/lastlog -p wa -k logins
id: xccdf_org.ssgproject.content_rule_audit_rules_login_events
kind: Rule
metadata:
  annotations:
    compliance.openshift.io/image-digest: pb-rhcos426smj
    compliance.openshift.io/rule: audit-rules-login-events
    control.compliance.openshift.io/NIST-800-53: AU-2(d);AU-12(c);AC-6(9);CM-6(a)
    control.compliance.openshift.io/PCI-DSS: Req-10.2.3
    policies.open-cluster-management.io/controls: AU-2(d),AU-12(c),AC-6(9),CM-6(a),Req-
10.2.3
    policies.open-cluster-management.io/standards: NIST-800-53,PCI-DSS
  labels:
    compliance.openshift.io/profile-bundle: rhcos4

```

```

name: rhcos4-audit-rules-login-events
namespace: openshift-compliance
ownerReferences:
- apiVersion: compliance.openshift.io/v1alpha1
  blockOwnerDeletion: true
  controller: true
  kind: ProfileBundle
  name: rhcos4
rationale: Manual editing of these files may indicate nefarious activity, such as
  an attacker attempting to remove evidence of an intrusion.
severity: medium
title: Record Attempts to Alter Logon and Logout Events
warning: Manual editing of these files may indicate nefarious activity, such as an attacker
  attempting to remove evidence of an intrusion.

```

5.6. MANAGING THE COMPLIANCE OPERATOR

This section describes the lifecycle of security content, including how to use an updated version of compliance content and how to create a custom **ProfileBundle** object.

5.6.1. Updating security content

Security content is shipped as container images that the **ProfileBundle** objects refer to. To accurately track updates to **ProfileBundles** and the custom resources parsed from the bundles such as rules or profiles, identify the container image with the compliance content using a digest instead of a tag:

Example output

```

apiVersion: compliance.openshift.io/v1alpha1
kind: ProfileBundle
metadata:
  name: rhcos4
spec:
  contentImage: quay.io/user/ocp4-openscap-
content@sha256:a1749f5150b19a9560a5732fe48a89f07bffc79c0832aa8c49ee5504590ae687 1
  contentFile: ssg-rhcos4-ds.xml

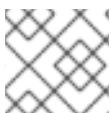
```

1 Security container image.

Each **ProfileBundle** is backed by a deployment. When the Compliance Operator detects that the container image digest has changed, the deployment is updated to reflect the change and parse the content again. Using the digest instead of a tag ensures that you use a stable and predictable set of profiles.

5.6.2. Using image streams

The **contentImage** reference points to a valid **ImageStreamTag**, and the Compliance Operator ensures that the content stays up to date automatically.



NOTE

ProfileBundle objects also accept **ImageStream** references.

Example image stream

```
$ oc get is -n openshift-compliance
```

Example output

NAME	IMAGE REPOSITORY	TAGS
UPDATED		
openscap-ocp4-ds	image-registry.openshift-image-registry.svc:5000/openshift-compliance/openscap-ocp4-ds	latest 32 seconds ago

Procedure

1. Ensure that the lookup policy is set to local:

```
$ oc patch is openscap-ocp4-ds \
  -p '{"spec":{"lookupPolicy":{"local":true}}}' \
  --type=merge
imagestream.image.openshift.io/openscap-ocp4-ds patched
-n openshift-compliance
```

2. Use the name of the **ImageStreamTag** for the **ProfileBundle** by retrieving the **istag** name:

```
$ oc get istag -n openshift-compliance
```

Example output

NAME	IMAGE REFERENCE
UPDATED	
openscap-ocp4-ds:latest	image-registry.openshift-image-registry.svc:5000/openshift-compliance/openscap-ocp4-ds@sha256:46d7ca9b7055fe56ade818ec3e62882cfcc2d27b9bf0d1cbae9f4b6df2710c96 3 minutes ago

3. Create the **ProfileBundle**:

```
$ cat << EOF | oc create -f -
apiVersion: compliance.openshift.io/v1alpha1
kind: ProfileBundle
metadata:
  name: mybundle
spec:
  contentImage: openscap-ocp4-ds:latest
  contentFile: ssg-rhcos4-ds.xml
EOF
```

This **ProfileBundle** will track the image and any changes that are applied to it, such as updating the tag to point to a different hash, will immediately be reflected in the **ProfileBundle**.

5.6.3. ProfileBundle CR example

The bundle object needs two pieces of information: the URL of a container image that contains the

contentImage and the file that contains the compliance content. The **contentFile** parameter is relative to the root of the file system. The built-in **rhcos4 ProfileBundle** object can be defined in the example below:

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ProfileBundle
metadata:
  name: rhcos4
spec:
  contentImage: quay.io/complianceascode/ocp4:latest 1
  contentFile: ssg-rhcos4-ds.xml 2
```

- 1** Content image location.
- 2** Location of the file containing the compliance content.



IMPORTANT

The base image used for the content images must include **coreutils**.

5.6.4. Additional resources

- The Compliance Operator is supported in a restricted network environment. For more information, see [Using Operator Lifecycle Manager on restricted networks](#).

5.7. TAILORING THE COMPLIANCE OPERATOR

While the Compliance Operator comes with ready-to-use profiles, they must be modified in order to fit the organizations' needs and requirements. The process of modifying a profile is called *tailoring*.

The Compliance Operator provides an object to easily tailor profiles called a **TailoredProfile**. This assumes that you are extending a pre-existing profile, and allows you to enable and disable rules and values which come from the **ProfileBundle**.



NOTE

You will only be able to use rules and variables that are available as part of the **ProfileBundle** that the profile you want to extend belongs to.

5.7.1. Using tailored profiles

While the **TailoredProfile** CR enables the most common tailoring operations, the XCCDF standard allows even more flexibility in tailoring OpenSCAP profiles. In addition, if your organization has been using OpenScap previously, you may have an existing XCCDF tailoring file and can reuse it.

The **ComplianceSuite** object contains an optional **TailoringConfigMap** attribute that you can point to a custom tailoring file. The value of the **TailoringConfigMap** attribute is a name of a config map, which must contain a key called **tailoring.xml** and the value of this key is the tailoring contents.

Procedure

1. Browse the available rules for the Red Hat Enterprise Linux CoreOS (RHCOS) **ProfileBundle**:

■

```
$ oc get rules.compliance -n openshift-compliance -l compliance.openshift.io/profile-bundle=rhcos4
```

2. Browse the available variables in the same **ProfileBundle**:

```
$ oc get variables.compliance -n openshift-compliance -l compliance.openshift.io/profile-bundle=rhcos4
```

3. Create a tailored profile named **nist-moderate-modified**:

- a. Choose which rules you want to add to the **nist-moderate-modified** tailored profile. This example extends the **rhcos4-moderate** profile by disabling two rules and changing one value. Use the **rationale** value to describe why these changes were made:

Example new-profile-node.yaml

```
apiVersion: compliance.openshift.io/v1alpha1
kind: TailoredProfile
metadata:
  name: nist-moderate-modified
spec:
  extends: rhcos4-moderate
  description: NIST moderate profile
  title: My modified NIST moderate profile
  disableRules:
    - name: rhcos4-file-permissions-var-log-messages
      rationale: The file contains logs of error messages in the system
    - name: rhcos4-account-disable-post-pw-expiration
      rationale: No need to check this as it comes from the IdP
  setValues:
    - name: rhcos4-var-selinux-state
      rationale: Organizational requirements
      value: permissive
```

Table 5.2. Attributes for spec variables

Attribute	Description
extends	Name of the Profile object upon which this TailoredProfile is built.
title	Human-readable title of the TailoredProfile .
disableRules	A list of name and rationale pairs. Each name refers to a name of a rule object that is to be disabled. The rationale value is human-readable text describing why the rule is disabled.
enableRules	A list of name and rationale pairs. Each name refers to a name of a rule object that is to be enabled. The rationale value is human-readable text describing why the rule is enabled.
description	Human-readable text describing the TailoredProfile .

Attribute	Description
setValues	A list of name, rationale, and value groupings. Each name refers to a name of the value set. The rationale is human-readable text describing the set. The value is the actual setting.

- b. Create the **TailoredProfile** object:

```
$ oc create -n openshift-compliance -f new-profile-node.yaml 1
```

- 1** The **TailoredProfile** object is created in the default **openshift-compliance** namespace.

Example output

```
tailoredprofile.compliance.openshift.io/nist-moderate-modified created
```

4. Define the **ScanSettingBinding** object to bind the new **nist-moderate-modified** tailored profile to the default **ScanSetting** object.

Example new-scansettingbinding.yaml

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSettingBinding
metadata:
  name: nist-moderate-modified
profiles:
  - apiGroup: compliance.openshift.io/v1alpha1
    kind: Profile
    name: ocp4-moderate
  - apiGroup: compliance.openshift.io/v1alpha1
    kind: TailoredProfile
    name: nist-moderate-modified
settingsRef:
  apiGroup: compliance.openshift.io/v1alpha1
  kind: ScanSetting
  name: default
```

5. Create the **ScanSettingBinding** object:

```
$ oc create -n openshift-compliance -f new-scansettingbinding.yaml
```

Example output

```
scansettingbinding.compliance.openshift.io/nist-moderate-modified created
```

5.8. RETRIEVING COMPLIANCE OPERATOR RAW RESULTS

When proving compliance for your OpenShift Container Platform cluster, you might need to provide the scan results for auditing purposes.

5.8.1. Obtaining Compliance Operator raw results from a persistent volume

Procedure

The Compliance Operator generates and stores the raw results in a persistent volume. These results are in Asset Reporting Format (ARF).

1. Explore the **ComplianceSuite** object:

```
$ oc get compliancesuites nist-moderate-modified -o json \
  | jq '.status.scanStatuses[].resultsStorage'
{
  "name": "rhcos4-moderate-worker",
  "namespace": "openshift-compliance"
}
{
  "name": "rhcos4-moderate-master",
  "namespace": "openshift-compliance"
}
```

This shows the persistent volume claims where the raw results are accessible.

2. Verify the raw data location by using the name and namespace of one of the results:

```
$ oc get pvc -n openshift-compliance rhcos4-moderate-worker
```

Example output

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES
STORAGECLASS	AGE			
rhcos4-moderate-worker	Bound	pvc-548f6cfe-164b-42fe-ba13-a07cfbc77f3a	1Gi	RWO
gp2	92m			

3. Fetch the raw results by spawning a pod that mounts the volume and copying the results:

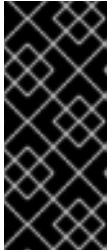
Example pod

```
apiVersion: "v1"
kind: Pod
metadata:
  name: pv-extract
spec:
  containers:
    - name: pv-extract-pod
      image: registry.access.redhat.com/ubi8/ubi
      command: ["sleep", "3000"]
      volumeMounts:
        - mountPath: "/workers-scan-results"
          name: workers-scan-vol
  volumes:
```

```
- name: workers-scan-vol
  persistentVolumeClaim:
    claimName: rhcos4-moderate-worker
```

4. After the pod is running, download the results:

```
$ oc cp pv-extract:/workers-scan-results .
```



IMPORTANT

Spawning a pod that mounts the persistent volume will keep the claim as **Bound**. If the volume's storage class in use has permissions set to **ReadWriteOnce**, the volume is only mountable by one pod at a time. You must delete the pod upon completion, or it will not be possible for the Operator to schedule a pod and continue storing results in this location.

5. After the extraction is complete, the pod can be deleted:

```
$ oc delete pod pv-extract
```

5.9. MANAGING COMPLIANCE OPERATOR RESULT AND REMEDIATION

Each **ComplianceCheckResult** represents a result of one compliance rule check. If the rule can be remediated automatically, a **ComplianceRemediation** object with the same name, owned by the **ComplianceCheckResult** is created. Unless requested, the remediations are not applied automatically, which gives an OpenShift Container Platform administrator the opportunity to review what the remediation does and only apply a remediation once it has been verified.

5.9.1. Filters for compliance check results

By default, the **ComplianceCheckResult** objects are labeled with several useful labels that allow you to query the checks and decide on the next steps after the results are generated.

List checks that belong to a specific suite:

```
$ oc get compliancecheckresults -l compliance.openshift.io/suite=example-compliancesuite
```

List checks that belong to a specific scan:

```
$ oc get compliancecheckresults -l compliance.openshift.io/scan=example-compliancescan
```

Not all **ComplianceCheckResult** objects create **ComplianceRemediation** objects. Only **ComplianceCheckResult** objects that can be remediated automatically do. A **ComplianceCheckResult** object has a related remediation if it is labeled with the **compliance.openshift.io/automated-remediation** label. The name of the remediation is the same as the name of the check.

List all failing checks that can be remediated automatically:

```
$ oc get compliancecheckresults -l 'compliance.openshift.io/check-status=FAIL,compliance.openshift.io/automated-remediation'
```

List all failing checks that must be remediated manually:

```
$ oc get compliancecheckresults -l 'compliance.openshift.io/check-status=FAIL,!compliance.openshift.io/automated-remediation'
```

The manual remediation steps are typically stored in the **description** attribute in the **ComplianceCheckResult** object.

Table 5.3. ComplianceCheckResult Status

ComplianceCheckResult Status	Description
PASS	Compliance check ran to completion and passed.
FAIL	Compliance check ran to completion and failed.
INFO	Compliance check ran to completion and found something not severe enough to be considered an error.
MANUAL	Compliance check does not have a way to automatically assess the success or failure and must be checked manually.
INCONSISTENT	Compliance check reports different results from different sources, typically cluster nodes.
ERROR	Compliance check ran, but could not complete properly.
NOT-APPLICABLE	Compliance check did not run because it is not applicable or not selected.

5.9.2. Reviewing a remediation

Review both the **ComplianceRemediation** object and the **ComplianceCheckResult** object that owns the remediation. The **ComplianceCheckResult** object contains human-readable descriptions of what the check does and the hardening trying to prevent, as well as other **metadata** like the severity and the associated security controls. The **ComplianceRemediation** object represents a way to fix the problem described in the **ComplianceCheckResult**. After first scan, check for remediations with the state **MissingDependencies**.

Below is an example of a check and a remediation called **sysctl-net-ipv4-conf-all-accept-redirects**. This example is redacted to only show **spec** and **status** and omits **metadata**:

```
spec:
  apply: false
  current:
  object:
    apiVersion: machineconfiguration.openshift.io/v1
    kind: MachineConfig
  spec:
```

```

config:
  ignition:
    version: 3.1.0
  storage:
    files:
      - path: /etc/sysctl.d/75-sysctl_net_ipv4_conf_all_accept_redirects.conf
        mode: 0644
        contents:
          source: data:.,net.ipv4.conf.all.accept_redirects%3D0
    outdated: {}
  status:
    applicationState: NotApplied

```

The remediation payload is stored in the **spec.current** attribute. The payload can be any Kubernetes object, but because this remediation was produced by a node scan, the remediation payload in the above example is a **MachineConfig** object. For Platform scans, the remediation payload is often a different kind of an object (for example, a **ConfigMap** or **Secret** object), but typically applying that remediation is up to the administrator, because otherwise the Compliance Operator would have required a very broad set of permissions in order to manipulate any generic Kubernetes object. An example of remediating a Platform check is provided later in the text.

To see exactly what the remediation does when applied, the **MachineConfig** object contents use the Ignition objects for the configuration. Refer to the [Ignition specification](#) for further information about the format. In our example, the **spec.config.storage.files[0].path** attribute specifies the file that is being create by this remediation (`/etc/sysctl.d/75-sysctl_net_ipv4_conf_all_accept_redirects.conf`) and the **spec.config.storage.files[0].contents.source** attribute specifies the contents of that file.



NOTE

The contents of the files are URL-encoded.

Use the following Python script to view the contents:

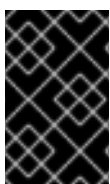
```
$ echo "net.ipv4.conf.all.accept_redirects%3D0" | python3 -c "import sys, urllib.parse;
print(urllib.parse.unquote(''.join(sys.stdin.readlines())))"
```

Example output

```
net.ipv4.conf.all.accept_redirects=0
```

5.9.3. Applying remediation when using customized machine config pools

When you create a custom **MachineConfigPool**, add a label to the **MachineConfigPool** so that **machineConfigPoolSelector** present in the **KubeletConfig** can match the label with **MachineConfigPool**.



IMPORTANT

Do not set **protectKernelDefaults: false** in the **KubeletConfig** file, because the **MachineConfigPool** object might fail to unpaue unexpectedly after the Compliance Operator finishes applying remediation.

Procedure

1. List the nodes.

```
$ oc get nodes
```

Example output

```
NAME                                STATUS ROLES AGE  VERSION
ip-10-0-128-92.us-east-2.compute.internal Ready  master 5h21m v1.23.3+d99c04f
ip-10-0-158-32.us-east-2.compute.internal Ready  worker 5h17m v1.23.3+d99c04f
ip-10-0-166-81.us-east-2.compute.internal Ready  worker 5h17m v1.23.3+d99c04f
ip-10-0-171-170.us-east-2.compute.internal Ready  master 5h21m v1.23.3+d99c04f
ip-10-0-197-35.us-east-2.compute.internal Ready  master 5h22m v1.23.3+d99c04f
```

2. Add a label to nodes.

```
$ oc label node ip-10-0-166-81.us-east-2.compute.internal node-
role.kubernetes.io/<machine_config_pool_name>=
```

Example output

```
node/ip-10-0-166-81.us-east-2.compute.internal labeled
```

3. Create custom **MachineConfigPool** CR.

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  name: <machine_config_pool_name>
  labels:
    pools.operator.machineconfiguration.openshift.io/<machine_config_pool_name>: " 1
spec:
  machineConfigSelector:
    matchExpressions:
      - {key: machineconfiguration.openshift.io/role, operator: In, values: [worker,
<machine_config_pool_name>]}
  nodeSelector:
    matchLabels:
      node-role.kubernetes.io/<machine_config_pool_name>: ""
```

- 1 The **labels** field defines label name to add for Machine config pool(MCP).

4. Verify MCP created successfully.

```
$ oc get mcp -w
```

5.9.4. Applying a remediation

The boolean attribute **spec.apply** controls whether the remediation should be applied by the Compliance Operator. You can apply the remediation by setting the attribute to **true**:

```
$ oc patch complianceremediations/<scan_name>-sysctl-net-ipv4-conf-all-accept-redirects --patch
'{"spec":{"apply":true}}' --type=merge
```

■

After the Compliance Operator processes the applied remediation, the **status.ApplicationState** attribute would change to **Applied** or to **Error** if incorrect. When a machine config remediation is applied, that remediation along with all other applied remediations are rendered into a **MachineConfig** object named **75-\$scan-name-\$suite-name**. That **MachineConfig** object is subsequently rendered by the Machine Config Operator and finally applied to all the nodes in a machine config pool by an instance of the machine control daemon running on each node.

Note that when the Machine Config Operator applies a new **MachineConfig** object to nodes in a pool, all the nodes belonging to the pool are rebooted. This might be inconvenient when applying multiple remediations, each of which re-renders the composite **75-\$scan-name-\$suite-name MachineConfig** object. To prevent applying the remediation immediately, you can pause the machine config pool by setting the **.spec.paused** attribute of a **MachineConfigPool** object to **true**.

The Compliance Operator can apply remediations automatically. Set **autoApplyRemediations: true** in the **ScanSetting** top-level object.



WARNING

Applying remediations automatically should only be done with careful consideration.

5.9.5. Remediating a platform check manually

Checks for Platform scans typically have to be remediated manually by the administrator for two reasons:

- It is not always possible to automatically determine the value that must be set. One of the checks requires that a list of allowed registries is provided, but the scanner has no way of knowing which registries the organization wants to allow.
- Different checks modify different API objects, requiring automated remediation to possess **root** or superuser access to modify objects in the cluster, which is not advised.

Procedure

1. The example below uses the **ocp4-ocp-allowed-registries-for-import** rule, which would fail on a default OpenShift Container Platform installation. Inspect the rule **oc get rule.compliance/ocp4-ocp-allowed-registries-for-import -oyaml**, the rule is to limit the registries the users are allowed to import images from by setting the **allowedRegistriesForImport** attribute, The **warning** attribute of the rule also shows the API object checked, so it can be modified and remediate the issue:

```
$ oc edit image.config.openshift.io/cluster
```

Example output

```
apiVersion: config.openshift.io/v1
kind: Image
metadata:
  annotations:
```

```

    release.openshift.io/create-only: "true"
    creationTimestamp: "2020-09-10T10:12:54Z"
    generation: 2
    name: cluster
    resourceVersion: "363096"
    selfLink: /apis/config.openshift.io/v1/images/cluster
    uid: 2dcb614e-2f8a-4a23-ba9a-8e33cd0ff77e
  spec:
    allowedRegistriesForImport:
    - domainName: registry.redhat.io
  status:
    externalRegistryHostnames:
    - default-route-openshift-image-registry.apps.user-cluster-09-10-12-
      07.devcluster.openshift.com
    internalRegistryHostname: image-registry.openshift-image-registry.svc:5000

```

2. Re-run the scan:

```
$ oc annotate compliancescans/<scan_name> compliance.openshift.io/rescan=
```

5.9.6. Updating remediations

When a new version of compliance content is used, it might deliver a new and different version of a remediation than the previous version. The Compliance Operator will keep the old version of the remediation applied. The OpenShift Container Platform administrator is also notified of the new version to review and apply. A `ComplianceRemediation` object that had been applied earlier, but was updated changes its status to **Outdated**. The outdated objects are labeled so that they can be searched for easily.

The previously applied remediation contents would then be stored in the **spec.outdated** attribute of a **ComplianceRemediation** object and the new updated contents would be stored in the **spec.current** attribute. After updating the content to a newer version, the administrator then needs to review the remediation. As long as the **spec.outdated** attribute exists, it would be used to render the resulting **MachineConfig** object. After the **spec.outdated** attribute is removed, the Compliance Operator re-renders the resulting **MachineConfig** object, which causes the Operator to push the configuration to the nodes.

Procedure

1. Search for any outdated remediations:

```
$ oc get complianceremediations -lcomplianceoperator.openshift.io/outdated-remediation=
```

Example output

```

NAME                                STATE
workers-scan-no-empty-passwords  Outdated

```

The currently applied remediation is stored in the **Outdated** attribute and the new, unapplied remediation is stored in the **Current** attribute. If you are satisfied with the new version, remove the **Outdated** field. If you want to keep the updated content, remove the **Current** and **Outdated** attributes.

2. Apply the newer version of the remediation:

```
$ oc patch complianceremediations workers-scan-no-empty-passwords --type json -p
'{"op":"remove", "path":"/spec/outdated}'
```

- The remediation state will switch from **Outdated** to **Applied**:

```
$ oc get complianceremediations workers-scan-no-empty-passwords
```

Example output

```
NAME                                STATE
workers-scan-no-empty-passwords    Applied
```

- The nodes will apply the newer remediation version and reboot.

5.9.7. Unapplying a remediation

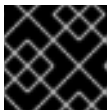
It might be required to unapply a remediation that was previously applied.

Procedure

- Set the **apply** flag to **false**:

```
$ oc patch complianceremediations/<scan_name>-sysctl-net-ipv4-conf-all-accept-redirects -p
'{"spec":{"apply":false}}' --type=merge
```

- The remediation status will change to **NotApplied** and the composite **MachineConfig** object would be re-rendered to not include the remediation.



IMPORTANT

All affected nodes with the remediation will be rebooted.

5.9.8. Removing a KubeletConfig remediation

KubeletConfig remediations are included in node-level profiles. In order to remove a KubeletConfig remediation, you must manually remove it from the **KubeletConfig** objects. This example demonstrates how to remove the compliance check for the **one-rule-tp-node-master-kubelet-eviction-thresholds-set-hard-imagefs-available** remediation.

Procedure

- Locate the **scan-name** and compliance check for the **one-rule-tp-node-master-kubelet-eviction-thresholds-set-hard-imagefs-available** remediation:

```
$ oc get remediation one-rule-tp-node-master-kubelet-eviction-thresholds-set-hard-imagefs-
available -o yaml
```

Example output

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ComplianceRemediation
metadata:
```

```

annotations:
  compliance.openshift.io/xccdf-value-used: var-kubelet-evictionhard-imagefs-available
creationTimestamp: "2022-01-05T19:52:27Z"
generation: 1
labels:
  compliance.openshift.io/scan-name: one-rule-tp-node-master 1
  compliance.openshift.io/suite: one-rule-ssb-node
name: one-rule-tp-node-master-kubelet-eviction-thresholds-set-hard-imagefs-available
namespace: openshift-compliance
ownerReferences:
- apiVersion: compliance.openshift.io/v1alpha1
  blockOwnerDeletion: true
  controller: true
  kind: ComplianceCheckResult
  name: one-rule-tp-node-master-kubelet-eviction-thresholds-set-hard-imagefs-available
  uid: fe8e1577-9060-4c59-95b2-3e2c51709adc
resourceVersion: "84820"
uid: 5339d21a-24d7-40cb-84d2-7a2ebb015355
spec:
  apply: true
  current:
    object:
      apiVersion: machineconfiguration.openshift.io/v1
      kind: KubeletConfig
      spec:
        kubeletConfig:
          evictionHard:
            imagefs.available: 10% 2
    outdated: {}
  type: Configuration
status:
  applicationState: Applied

```

- 1** The scan name of the remediation.
- 2** The remediation that was added to the **KubeletConfig** objects.



NOTE

If the remediation invokes an **evictionHard** kubelet configuration, you must specify all of the **evictionHard** parameters: **memory.available**, **nodefs.available**, **nodefs.inodesFree**, **imagefs.available**, and **imagefs.inodesFree**. If you do not specify all parameters, only the specified parameters are applied and the remediation will not function properly.

2. Remove the remediation:
 - a. Set **apply** to false for the remediation object:


```
$ oc patch complianceremediations/one-rule-tp-node-master-kubelet-eviction-thresholds-set-hard-imagefs-available -p '{"spec":{"apply":false}}' --type=merge
```
 - b. Using the **scan-name**, find the **KubeletConfig** object that the remediation was applied to:

```
$ oc get kubeletconfig --selector compliance.openshift.io/scan-name=one-rule-tp-node-master
```

Example output

```
NAME                                AGE
compliance-operator-kubelet-master  2m34s
```

- c. Manually remove the remediation, **imagefs.available: 10%**, from the **KubeletConfig** object:

```
$ oc edit KubeletConfig compliance-operator-kubelet-master
```



IMPORTANT

All affected nodes with the remediation will be rebooted.

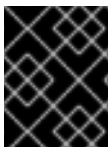


NOTE

You must also exclude the rule from any scheduled scans in your tailored profiles that auto-applies the remediation, otherwise, the remediation will be re-applied during the next scheduled scan.

5.9.9. Inconsistent ComplianceScan

The **ScanSetting** object lists the node roles that the compliance scans generated from the **ScanSetting** or **ScanSettingBinding** objects would scan. Each node role usually maps to a machine config pool.



IMPORTANT

It is expected that all machines in a machine config pool are identical and all scan results from the nodes in a pool should be identical.

If some of the results are different from others, the Compliance Operator flags a **ComplianceCheckResult** object where some of the nodes will report as **INCONSISTENT**. All **ComplianceCheckResult** objects are also labeled with **compliance.openshift.io/inconsistent-check**.

Because the number of machines in a pool might be quite large, the Compliance Operator attempts to find the most common state and list the nodes that differ from the common state. The most common state is stored in the **compliance.openshift.io/most-common-status** annotation and the annotation **compliance.openshift.io/inconsistent-source** contains pairs of **hostname:status** of check statuses that differ from the most common status. If no common state can be found, all the **hostname:status** pairs are listed in the **compliance.openshift.io/inconsistent-source** annotation.

If possible, a remediation is still created so that the cluster can converge to a compliant status. However, this might not always be possible and correcting the difference between nodes must be done manually. The compliance scan must be re-run to get a consistent result by annotating the scan with the **compliance.openshift.io/rescan=** option:

```
$ oc annotate compliancescans/<scan_name> compliance.openshift.io/rescan=
```

5.9.10. Additional resources

- [Modifying nodes.](#)

5.10. PERFORMING ADVANCED COMPLIANCE OPERATOR TASKS

The Compliance Operator includes options for advanced users for the purpose of debugging or integration with existing tooling.

5.10.1. Using the ComplianceSuite and ComplianceScan objects directly

While it is recommended that users take advantage of the **ScanSetting** and **ScanSettingBinding** objects to define the suites and scans, there are valid use cases to define the **ComplianceSuite** objects directly:

- Specifying only a single rule to scan. This can be useful for debugging together with the **debug: true** attribute which increases the OpenSCAP scanner verbosity, as the debug mode tends to get quite verbose otherwise. Limiting the test to one rule helps to lower the amount of debug information.
- Providing a custom nodeSelector. In order for a remediation to be applicable, the nodeSelector must match a pool.
- Pointing the Scan to a bespoke config map with a tailoring file.
- For testing or development when the overhead of parsing profiles from bundles is not required.

The following example shows a **ComplianceSuite** that scans the worker machines with only a single rule:

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ComplianceSuite
metadata:
  name: workers-compliancesuite
spec:
  scans:
    - name: workers-scan
      profile: xccdf_org.ssgproject.content_profile_moderate
      content: ssg-rhcos4-ds.xml
      contentImage: quay.io/complianceascode/ocp4:latest
      debug: true
      rule: xccdf_org.ssgproject.content_rule_no_direct_root_logins
      nodeSelector:
        node-role.kubernetes.io/worker: ""
```

The **ComplianceSuite** object and the **ComplianceScan** objects referred to above specify several attributes in a format that OpenSCAP expects.

To find out the profile, content, or rule values, you can start by creating a similar Suite from **ScanSetting** and **ScanSettingBinding** or inspect the objects parsed from the **ProfileBundle** objects like rules or profiles. Those objects contain the **xccdf_org** identifiers you can use to refer to them from a **ComplianceSuite**.

5.10.2. Using raw tailored profiles

While the **TailoredProfile** CR enables the most common tailoring operations, the XCCDF standard allows even more flexibility in tailoring OpenSCAP profiles. In addition, if your organization has been using OpenScap previously, you may have an existing XCCDF tailoring file and can reuse it.

The **ComplianceSuite** object contains an optional **TailoringConfigMap** attribute that you can point to a custom tailoring file. The value of the **TailoringConfigMap** attribute is a name of a config map which must contain a key called **tailoring.xml** and the value of this key is the tailoring contents.

Procedure

1. Create the **ConfigMap** object from a file:

```
$ oc create configmap <scan_name> --from-file=tailoring.xml=/path/to/the/tailoringFile.xml
```

2. Reference the tailoring file in a scan that belongs to a suite:

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ComplianceSuite
metadata:
  name: workers-compliancesuite
spec:
  debug: true
  scans:
    - name: workers-scan
      profile: xccdf_org.ssgproject.content_profile_moderate
      content: ssg-rhcos4-ds.xml
      contentImage: quay.io/complianceascode/ocp4:latest
      debug: true
  tailoringConfigMap:
    name: <scan_name>
  nodeSelector:
    node-role.kubernetes.io/worker: ""
```

5.10.3. Performing a rescan

Typically you will want to re-run a scan on a defined schedule, like every Monday or daily. It can also be useful to re-run a scan once after fixing a problem on a node. To perform a single scan, annotate the scan with the **compliance.openshift.io/rescan=** option:

```
$ oc annotate compliancescans/<scan_name> compliance.openshift.io/rescan=
```

A rescan generates four additional **mc** for **rhcos-moderate** profile:

```
$ oc get mc
```

Example output

```
75-worker-scan-chronyd-or-ntpd-specify-remote-server
75-worker-scan-configure-usbguard-auditbackend
75-worker-scan-service-usbguard-enabled
75-worker-scan-usbguard-allow-hid-and-hub
```




IMPORTANT

When the scan setting **default-auto-apply** label is applied, remediations are applied automatically and outdated remediations automatically update. If there are remediations that were not applied due to dependencies, or remediations that had been outdated, rescanning applies the remediations and might trigger a reboot. Only remediations that use **MachineConfig** objects trigger reboots. If there are no updates or dependencies to be applied, no reboot occurs.

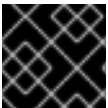
5.10.4. Setting custom storage size for results

While the custom resources such as **ComplianceCheckResult** represent an aggregated result of one check across all scanned nodes, it can be useful to review the raw results as produced by the scanner. The raw results are produced in the ARF format and can be large (tens of megabytes per node), it is impractical to store them in a Kubernetes resource backed by the **etcd** key-value store. Instead, every scan creates a persistent volume (PV) which defaults to 1GB size. Depending on your environment, you may want to increase the PV size accordingly. This is done using the **rawResultStorage.size** attribute that is exposed in both the **ScanSetting** and **ComplianceScan** resources.

A related parameter is **rawResultStorage.rotation** which controls how many scans are retained in the PV before the older scans are rotated. The default value is 3, setting the rotation policy to 0 disables the rotation. Given the default rotation policy and an estimate of 100MB per a raw ARF scan report, you can calculate the right PV size for your environment.

5.10.4.1. Using custom result storage values

Because OpenShift Container Platform can be deployed in a variety of public clouds or bare metal, the Compliance Operator cannot determine available storage configurations. By default, the Compliance Operator will try to create the PV for storing results using the default storage class of the cluster, but a custom storage class can be configured using the **rawResultStorage.StorageClassName** attribute.



IMPORTANT

If your cluster does not specify a default storage class, this attribute must be set.

Configure the **ScanSetting** custom resource to use a standard storage class and create persistent volumes that are 10GB in size and keep the last 10 results:

Example **ScanSetting** CR

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSetting
metadata:
  name: default
  namespace: openshift-compliance
rawResultStorage:
  storageClassName: standard
  rotation: 10
  size: 10Gi
roles:
- worker
- master
scanTolerations:
- effect: NoSchedule
```

```
key: node-role.kubernetes.io/master
operator: Exists
schedule: '0 1 * * *'
```

5.10.5. Applying remediations generated by suite scans

Although you can use the **autoApplyRemediations** boolean parameter in a **ComplianceSuite** object, you can alternatively annotate the object with **compliance.openshift.io/apply-remediations**. This allows the Operator to apply all of the created remediations.

Procedure

- Apply the **compliance.openshift.io/apply-remediations** annotation by running:

```
$ oc annotate compliancesuites/<suite-_name> compliance.openshift.io/apply-remediations=
```

5.10.6. Automatically update remediations

In some cases, a scan with newer content might mark remediations as **OUTDATED**. As an administrator, you can apply the **compliance.openshift.io/remove-outdated** annotation to apply new remediations and remove the outdated ones.

Procedure

- Apply the **compliance.openshift.io/remove-outdated** annotation:

```
$ oc annotate compliancesuites/<suite_name> compliance.openshift.io/remove-outdated=
```

Alternatively, set the **autoUpdateRemediations** flag in a **ScanSetting** or **ComplianceSuite** object to update the remediations automatically.

5.11. TROUBLESHOOTING THE COMPLIANCE OPERATOR

This section describes how to troubleshoot the Compliance Operator. The information can be useful either to diagnose a problem or provide information in a bug report. Some general tips:

- The Compliance Operator emits Kubernetes events when something important happens. You can either view all events in the cluster using the command:

```
$ oc get events -n openshift-compliance
```

Or view events for an object like a scan using the command:

```
$ oc describe compliancescan/<scan_name>
```

- The Compliance Operator consists of several controllers, approximately one per API object. It could be useful to filter only those controllers that correspond to the API object having issues. If a **ComplianceRemediation** cannot be applied, view the messages from the **remediationctrl** controller. You can filter the messages from a single controller by parsing with **jq**:

```
$ oc logs compliance-operator-775d7bddbd-gj58f | jq -c 'select(.logger == "profilebundlectrl")'
```

- The timestamps are logged as seconds since UNIX epoch in UTC. To convert them to a human-readable date, use **date -d @timestamp --utc**, for example:

```
$ date -d @1596184628.955853 --utc
```

- Many custom resources, most importantly **ComplianceSuite** and **ScanSetting**, allow the **debug** option to be set. Enabling this option increases verbosity of the OpenSCAP scanner pods, as well as some other helper pods.
- If a single rule is passing or failing unexpectedly, it could be helpful to run a single scan or a suite with only that rule to find the rule ID from the corresponding **ComplianceCheckResult** object and use it as the **rule** attribute value in a **Scan** CR. Then, together with the **debug** option enabled, the **scanner** container logs in the scanner pod would show the raw OpenSCAP logs.

5.11.1. Anatomy of a scan

The following sections outline the components and stages of Compliance Operator scans.

5.11.1.1. Compliance sources

The compliance content is stored in **Profile** objects that are generated from a **ProfileBundle** object. The Compliance Operator creates a **ProfileBundle** object for the cluster and another for the cluster nodes.

```
$ oc get profilebundle.compliance
```

```
$ oc get profile.compliance
```

The **ProfileBundle** objects are processed by deployments labeled with the **Bundle** name. To troubleshoot an issue with the **Bundle**, you can find the deployment and view logs of the pods in a deployment:

```
$ oc logs -lprofile-bundle=ocp4 -c profileparser
```

```
$ oc get deployments,pods -lprofile-bundle=ocp4
```

```
$ oc logs pods/<pod-name>
```

```
$ oc describe pod/<pod-name> -c profileparser
```

5.11.1.2. The ScanSetting and ScanSettingBinding objects lifecycle and debugging

With valid compliance content sources, the high-level **ScanSetting** and **ScanSettingBinding** objects can be used to generate **ComplianceSuite** and **ComplianceScan** objects:

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSetting
metadata:
  name: my-companys-constraints
debug: true
# For each role, a separate scan will be created pointing
```

```
# to a node-role specified in roles
roles:
- worker
---
apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSettingBinding
metadata:
  name: my-companys-compliance-requirements
profiles:
  # Node checks
  - name: rhcos4-e8
    kind: Profile
    apiGroup: compliance.openshift.io/v1alpha1
  # Cluster checks
  - name: ocp4-e8
    kind: Profile
    apiGroup: compliance.openshift.io/v1alpha1
settingsRef:
  name: my-companys-constraints
  kind: ScanSetting
  apiGroup: compliance.openshift.io/v1alpha1
```

Both **ScanSetting** and **ScanSettingBinding** objects are handled by the same controller tagged with **logger=scansettingbindingctrl**. These objects have no status. Any issues are communicated in form of events:

```
Events:
  Type    Reason      Age   From          Message
  ----    -
Normal   SuiteCreated 9m52s scansettingbindingctrl ComplianceSuite openshift-compliance/my-companys-compliance-requirements created
```

Now a **ComplianceSuite** object is created. The flow continues to reconcile the newly created **ComplianceSuite**.

5.11.1.3. ComplianceSuite custom resource lifecycle and debugging

The **ComplianceSuite** CR is a wrapper around **ComplianceScan** CRs. The **ComplianceSuite** CR is handled by controller tagged with **logger=suitectrl**. This controller handles creating scans from a suite, reconciling and aggregating individual Scan statuses into a single Suite status. If a suite is set to execute periodically, the **suitectrl** also handles creating a **CronJob** CR that re-runs the scans in the suite after the initial run is done:

```
$ oc get cronjobs
```

Example output

```
NAME                                SCHEDULE    SUSPEND    ACTIVE    LAST SCHEDULE    AGE
<cron_name>                        0 1 * * *    False     0         <none>          151m
```

For the most important issues, events are emitted. View them with **oc describe compliancesuites/<name>**. The **Suite** objects also have a **Status** subresource that is updated when any of **Scan** objects that belong to this suite update their **Status** subresource. After all expected scans are created, control is passed to the scan controller.

5.11.1.4. ComplianceScan custom resource lifecycle and debugging

The **ComplianceScan** CRs are handled by the **scanctrl** controller. This is also where the actual scans happen and the scan results are created. Each scan goes through several phases:

5.11.1.4.1. Pending phase

The scan is validated for correctness in this phase. If some parameters like storage size are invalid, the scan transitions to DONE with ERROR result, otherwise proceeds to the Launching phase.

5.11.1.4.2. Launching phase

In this phase, several config maps that contain either environment for the scanner pods or directly the script that the scanner pods will be evaluating. List the config maps:

```
$ oc get cm -lcompliance.openshift.io/scan-name=rhcos4-e8-
worker,complianceoperator.openshift.io/scan-script=
```

These config maps will be used by the scanner pods. If you ever needed to modify the scanner behavior, change the scanner debug level or print the raw results, modifying the config maps is the way to go. Afterwards, a persistent volume claim is created per scan in order to store the raw ARF results:

```
$ oc get pvc -lcompliance.openshift.io/scan-name=<scan_name>
```

The PVCs are mounted by a per-scan **ResultServer** deployment. A **ResultServer** is a simple HTTP server where the individual scanner pods upload the full ARF results to. Each server can run on a different node. The full ARF results might be very large and you cannot presume that it would be possible to create a volume that could be mounted from multiple nodes at the same time. After the scan is finished, the **ResultServer** deployment is scaled down. The PVC with the raw results can be mounted from another custom pod and the results can be fetched or inspected. The traffic between the scanner pods and the **ResultServer** is protected by mutual TLS protocols.

Finally, the scanner pods are launched in this phase; one scanner pod for a **Platform** scan instance and one scanner pod per matching node for a **node** scan instance. The per-node pods are labeled with the node name. Each pod is always labeled with the **ComplianceScan** name:

```
$ oc get pods -lcompliance.openshift.io/scan-name=rhcos4-e8-worker,workload=scanner --show-
labels
```

Example output

```
NAME                                READY STATUS    RESTARTS AGE LABELS
rhcos4-e8-worker-ip-10-0-169-90.eu-north-1.compute.internal-pod 0/2 Completed 0 39m
compliance.openshift.io/scan-name=rhcos4-e8-worker,targetNode=ip-10-0-169-90.eu-north-
1.compute.internal,workload=scanner
```

At this point, the scan proceeds to the Running phase.

5.11.1.4.3. Running phase

The running phase waits until the scanner pods finish. The following terms and processes are in use in the running phase:

- **init container**: There is one init container called **content-container**. It runs the **contentImage** container and executes a single command that copies the **contentFile** to the **/content** directory shared with the other containers in this pod.
- **scanner**: This container runs the scan. For node scans, the container mounts the node filesystem as **/host** and mounts the content delivered by the init container. The container also mounts the **entrypoint ConfigMap** created in the Launching phase and executes it. The default script in the **entrypoint ConfigMap** executes OpenSCAP and stores the result files in the **/results** directory shared between the pod's containers. Logs from this pod can be viewed to determine what the OpenSCAP scanner checked. More verbose output can be viewed with the **debug** flag.
- **logcollector**: The logcollector container waits until the scanner container finishes. Then, it uploads the full ARF results to the **ResultServer** and separately uploads the XCCDF results along with scan result and OpenSCAP result code as a **ConfigMap**. These result config maps are labeled with the scan name (**compliance.openshift.io/scan-name=<scan_name>**):

```
$ oc describe cm/rhcos4-e8-worker-ip-10-0-169-90.eu-north-1.compute.internal-pod
```

Example output

```
Name:      rhcos4-e8-worker-ip-10-0-169-90.eu-north-1.compute.internal-pod
Namespace: openshift-compliance
Labels:    compliance.openshift.io/scan-name-scan=rhcos4-e8-worker
           complianceoperator.openshift.io/scan-result=
Annotations: compliance-remediations/processed:
             compliance.openshift.io/scan-error-msg:
             compliance.openshift.io/scan-result: NON-COMPLIANT
             OpenSCAP-scan-result/node: ip-10-0-169-90.eu-north-1.compute.internal

Data
====
exit-code:
----
2
results:
----
<?xml version="1.0" encoding="UTF-8"?>
...
```

Scanner pods for **Platform** scans are similar, except:

- There is one extra init container called **api-resource-collector** that reads the OpenSCAP content provided by the **content-container** init container, figures out which API resources the content needs to examine and stores those API resources to a shared directory where the **scanner** container would read them from.
- The **scanner** container does not need to mount the host file system.

When the scanner pods are done, the scans move on to the Aggregating phase.

5.11.1.4.4. Aggregating phase

In the aggregating phase, the scan controller spawns yet another pod called the aggregator pod. Its purpose is to take the result **ConfigMap** objects, read the results and for each check result create the corresponding Kubernetes object. If the check failure can be automatically remediated, a

ComplianceRemediation object is created. To provide human-readable metadata for the checks and remediations, the aggregator pod also mounts the OpenSCAP content using an init container.

When a config map is processed by an aggregator pod, it is labeled the **compliance-remediations/processed** label. The result of this phase are **ComplianceCheckResult** objects:

```
$ oc get compliancecheckresults -lcompliance.openshift.io/scan-name=rhcos4-e8-worker
```

Example output

NAME	STATUS	SEVERITY
rhcos4-e8-worker-accounts-no-uid-except-zero	PASS	high
rhcos4-e8-worker-audit-rules-dac-modification-chmod	FAIL	medium

and **ComplianceRemediation** objects:

```
$ oc get complianceremediations -lcompliance.openshift.io/scan-name=rhcos4-e8-worker
```

Example output

NAME	STATE
rhcos4-e8-worker-audit-rules-dac-modification-chmod	NotApplied
rhcos4-e8-worker-audit-rules-dac-modification-chown	NotApplied
rhcos4-e8-worker-audit-rules-execution-chcon	NotApplied
rhcos4-e8-worker-audit-rules-execution-restorecon	NotApplied
rhcos4-e8-worker-audit-rules-execution-semanage	NotApplied
rhcos4-e8-worker-audit-rules-execution-setfiles	NotApplied

After these CRs are created, the aggregator pod exits and the scan moves on to the Done phase.

5.11.1.4.5. Done phase

In the final scan phase, the scan resources are cleaned up if needed and the **ResultServer** deployment is either scaled down (if the scan was one-time) or deleted if the scan is continuous; the next scan instance would then recreate the deployment again.

It is also possible to trigger a re-run of a scan in the Done phase by annotating it:

```
$ oc annotate compliancescans/<scan_name> compliance.openshift.io/rescan=
```

After the scan reaches the Done phase, nothing else happens on its own unless the remediations are set to be applied automatically with **autoApplyRemediations: true**. The OpenShift Container Platform administrator would now review the remediations and apply them as needed. If the remediations are set to be applied automatically, the **ComplianceSuite** controller takes over in the Done phase, pauses the machine config pool to which the scan maps to and applies all the remediations in one go. If a remediation is applied, the **ComplianceRemediation** controller takes over.

5.11.1.5. ComplianceRemediation controller lifecycle and debugging

The example scan has reported some findings. One of the remediations can be enabled by toggling its **apply** attribute to **true**:

```
$ oc patch complianceremediations/rhcos4-e8-worker-audit-rules-dac-modification-chmod --patch '{"spec":{"apply":true}}' --type=merge
```

The **ComplianceRemediation** controller (**logger=remediationctrl**) reconciles the modified object. The result of the reconciliation is change of status of the remediation object that is reconciled, but also a change of the rendered per-suite **MachineConfig** object that contains all the applied remediations.

The **MachineConfig** object always begins with **75-** and is named after the scan and the suite:

```
$ oc get mc | grep 75-
```

Example output

```
75-rhcos4-e8-worker-my-companys-compliance-requirements          2.2.0
2m46s
```

The remediations the **mc** currently consists of are listed in the machine config's annotations:

```
$ oc describe mc/75-rhcos4-e8-worker-my-companys-compliance-requirements
```

Example output

```
Name:      75-rhcos4-e8-worker-my-companys-compliance-requirements
Labels:    machineconfiguration.openshift.io/role=worker
Annotations: remediation/rhcos4-e8-worker-audit-rules-dac-modification-chmod:
```

The **ComplianceRemediation** controller's algorithm works like this:

- All currently applied remediations are read into an initial remediation set.
- If the reconciled remediation is supposed to be applied, it is added to the set.
- A **MachineConfig** object is rendered from the set and annotated with names of remediations in the set. If the set is empty (the last remediation was unapplied), the rendered **MachineConfig** object is removed.
- If and only if the rendered machine config is different from the one already applied in the cluster, the applied MC is updated (or created, or deleted).
- Creating or modifying a **MachineConfig** object triggers a reboot of nodes that match the **machineconfiguration.openshift.io/role** label - see the Machine Config Operator documentation for more details.

The remediation loop ends once the rendered machine config is updated, if needed, and the reconciled remediation object status is updated. In our case, applying the remediation would trigger a reboot. After the reboot, annotate the scan to re-run it:

```
$ oc annotate compliancescans/<scan_name> compliance.openshift.io/rescan=
```

The scan will run and finish. Check for the remediation to pass:

```
$ oc get compliancecheckresults/rhcos4-e8-worker-audit-rules-dac-modification-chmod
```


Example output

```

NAME                                STATUS SEVERITY
rhcos4-e8-worker-audit-rules-dac-modification-chmod PASS  medium

```

5.11.1.6. Useful labels

Each pod that is spawned by the Compliance Operator is labeled specifically with the scan it belongs to and the work it does. The scan identifier is labeled with the **compliance.openshift.io/scan-name** label. The workload identifier is labeled with the **workload** label.

The Compliance Operator schedules the following workloads:

- **scanner**: Performs the compliance scan.
- **resultserver**: Stores the raw results for the compliance scan.
- **aggregator**: Aggregates the results, detects inconsistencies and outputs result objects (checkresults and remediations).
- **suitererunner**: Will tag a suite to be re-run (when a schedule is set).
- **profileparser**: Parses a datastream and creates the appropriate profiles, rules and variables.

When debugging and logs are required for a certain workload, run:

```
$ oc logs -l workload=<workload_name> -c <container_name>
```

5.11.2. Getting support

If you experience difficulty with a procedure described in this documentation, or with OpenShift Container Platform in general, visit the [Red Hat Customer Portal](#). From the Customer Portal, you can:

- Search or browse through the Red Hat Knowledgebase of articles and solutions relating to Red Hat products.
- Submit a support case to Red Hat Support.
- Access other product documentation.

To identify issues with your cluster, you can use Insights in [OpenShift Cluster Manager](#). Insights provides details about issues and, if available, information on how to solve a problem.

If you have a suggestion for improving this documentation or have found an error, submit a [Jira issue](#) for the most relevant documentation component. Please provide specific details, such as the section name and OpenShift Container Platform version.

5.12. UNINSTALLING THE COMPLIANCE OPERATOR

You can remove the OpenShift Compliance Operator from your cluster by using the OpenShift Container Platform web console.

5.12.1. Uninstalling the OpenShift Compliance Operator from OpenShift Container Platform

To remove the Compliance Operator, you must first delete the Compliance Operator custom resource definitions (CRDs). After the CRDs are removed, you can then remove the Operator and its namespace by deleting the **openshift-compliance** project.

Prerequisites


- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- The OpenShift Compliance Operator must be installed.

Procedure

To remove the Compliance Operator by using the OpenShift Container Platform web console:

1. Remove CRDs that were installed by the Compliance Operator:


- a. Switch to the **Administration** → **Custom Resource Definitions** page.
- b. Search for **compliance.openshift.io** in the **Name** field.

c. Click the Options menu  next to each of the following CRDs, and select **Delete Custom Resource Definition**:

- **ComplianceCheckResult**
- **ComplianceRemediation**
- **ComplianceScan**
- **ComplianceSuite**
- **ProfileBundle**
- **Profile**
- **Rule**
- **ScanSettingBinding**
- **ScanSetting**
- **TailoredProfile**
- **Variable**

2. Remove the OpenShift Compliance project:

a. Switch to the **Home** → **Projects** page.

b. Click the Options menu  next to the **openshift-compliance** project, and select **Delete Project**.

c. Confirm the deletion by typing **openshift-compliance** in the dialog box, and click **Delete**.

5.13. UNDERSTANDING THE CUSTOM RESOURCE DEFINITIONS

The Compliance Operator in the OpenShift Container Platform provides you with several Custom Resource Definitions (CRDs) to accomplish the compliance scans. To run a compliance scan, it leverages the predefined security policies, which are derived from the [ComplianceAsCode](#) community project. The Compliance Operator converts these security policies into CRDs, which you can use to run compliance scans and get remediations for the issues found.

5.13.1. CRDs workflow

The CRD provides you the following workflow to complete the compliance scans:

1. Define your compliance scan requirements
2. Configure the compliance scan settings
3. Process compliance requirements with compliance scans settings
4. Monitor the compliance scans
5. Check the compliance scan results

5.13.2. Defining the compliance scan requirements

By default, the Compliance Operator CRDs include **ProfileBundle** and **Profile** objects, in which you can define and set the rules for your compliance scan requirements. You can also customize the default profiles by using a **TailoredProfile** object.

5.13.2.1. ProfileBundle object

When you install the Compliance Operator, it includes ready-to-run **ProfileBundle** object. The Compliance Operator parses the **ProfileBundle** object and creates a **Profile** object for each profile in the bundle. It also parses **Rule** and **Variable** objects, which are used by the **Profile** object.

Example ProfileBundle object

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ProfileBundle
  name: <profile bundle name>
  namespace: openshift-compliance
spec:
  contentFile: ssg-ocp4-ds.xml 1
  contentImage: quay.io/complianceascode/ocp4:latest 2
status:
  dataStreamStatus: VALID 3
```

- 1** Specify a path from the root directory (/) where the profile file is located.
- 2** Specify the container image that encapsulates the profile files.
- 3** Indicates whether the Compliance Operator was able to parse the content files.

**NOTE**

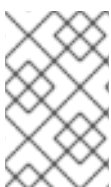
When the **contentFile** fails, an **errorMessage** attribute appears, which provides details of the error that occurred.

Troubleshooting

When you roll back to a known content image from an invalid image, the **ProfileBundle** object stops responding and displays **PENDING** state. As a workaround, you can move to a different image than the previous one. Alternatively, you can delete and re-create the **ProfileBundle** object to return to the working state.

5.13.2.2. Profile object

The **Profile** object defines the rules and variables that can be evaluated for a certain compliance standard. It contains parsed out details about an OpenSCAP profile, such as its XCCDF identifier and profile checks for a **Node** or **Platform** type. You can either directly use the **Profile** object or further customize it using a **TailorProfile** object.

**NOTE**

You cannot create or modify the **Profile** object manually because it is derived from a single **ProfileBundle** object. Typically, a single **ProfileBundle** object can include several **Profile** objects.

Example Profile object

```

apiVersion: compliance.openshift.io/v1alpha1
description: <description of the profile>
id: xccdf_org.ssgproject.content_profile_moderate 1
kind: Profile
metadata:
  annotations:
    compliance.openshift.io/product: <product name>
    compliance.openshift.io/product-type: Node 2
  creationTimestamp: "YYYY-MM-DDTMM:HH:SSZ"
  generation: 1
  labels:
    compliance.openshift.io/profile-bundle: <profile bundle name>
name: rhcos4-moderate
namespace: openshift-compliance
ownerReferences:
- apiVersion: compliance.openshift.io/v1alpha1
  blockOwnerDeletion: true
  controller: true
  kind: ProfileBundle
  name: <profile bundle name>
  uid: <uid string>
resourceVersion: "<version number>"
selfLink: /apis/compliance.openshift.io/v1alpha1/namespaces/openshift-compliance/profiles/rhcos4-
moderate
  uid: <uid string>
rules: 3
- rhcos4-account-disable-post-pw-expiration

```

```
- rhcos4-accounts-no-uid-except-zero
- rhcos4-audit-rules-dac-modification-chmod
- rhcos4-audit-rules-dac-modification-chown
title: <title of the profile>
```

- 1 Specify the XCCDF name of the profile. Use this identifier when you define a **ComplianceScan** object as the value of the profile attribute of the scan.
- 2 Specify either a **Node** or **Platform**. Node profiles scan the cluster nodes and platform profiles scan the Kubernetes platform.
- 3 Specify the list of rules for the profile. Each rule corresponds to a single check.

5.13.2.3. Rule object

The **Rule** object, which forms the profiles, are also exposed as objects. Use the **Rule** object to define your compliance check requirements and specify how it could be fixed.

Example Rule object

```
apiVersion: compliance.openshift.io/v1alpha1
checkType: Platform 1
description: <description of the rule>
id: xccdf_org.ssgproject.content_rule_configure_network_policies_namespaces 2
instructions: <manual instructions for the scan>
kind: Rule
metadata:
  annotations:
    compliance.openshift.io/rule: configure-network-policies-namespaces
    control.compliance.openshift.io/CIS-OCP: 5.3.2
    control.compliance.openshift.io/NERC-CIP: CIP-003-3 R4;CIP-003-3 R4.2;CIP-003-3
      R5;CIP-003-3 R6;CIP-004-3 R2.2.4;CIP-004-3 R3;CIP-007-3 R2;CIP-007-3 R2.1;CIP-007-3
      R2.2;CIP-007-3 R2.3;CIP-007-3 R5.1;CIP-007-3 R6.1
    control.compliance.openshift.io/NIST-800-53: AC-4;AC-4(21);CA-3(5);CM-6;CM-6(1);CM-7;CM-
      7(1);SC-7;SC-7(3);SC-7(5);SC-7(8);SC-7(12);SC-7(13);SC-7(18)
  labels:
    compliance.openshift.io/profile-bundle: ocp4
    name: ocp4-configure-network-policies-namespaces
    namespace: openshift-compliance
  rationale: <description of why this rule is checked>
  severity: high 3
  title: <summary of the rule>
```

- 1 Specify the type of check this rule executes. **Node** profiles scan the cluster nodes and **Platform** profiles scan the Kubernetes platform. An empty value indicates there is no automated check.
- 2 Specify the XCCDF name of the rule, which is parsed directly from the datastream.
- 3 Specify the severity of the rule when it fails.

**NOTE**

The **Rule** object gets an appropriate label for an easy identification of the associated **ProfileBundle** object. The **ProfileBundle** also gets specified in the **OwnerReferences** of this object.

5.13.2.4. TailoredProfile object

Use the **TailoredProfile** object to modify the default **Profile** object based on your organization requirements. You can enable or disable rules, set variable values, and provide justification for the customization. After validation, the **TailoredProfile** object creates a **ConfigMap**, which can be referenced by a **ComplianceScan** object.

TIP

You can use the **TailoredProfile** object by referencing it in a **ScanSettingBinding** object. For more information about **ScanSettingBinding**, see [ScanSettingBinding](#) object.

Example TailoredProfile object

```
apiVersion: compliance.openshift.io/v1alpha1
kind: TailoredProfile
metadata:
  name: rhcos4-with-usb
spec:
  extends: rhcos4-moderate 1
  title: <title of the tailored profile>
  disableRules:
    - name: <name of a rule object to be disabled>
      rationale: <description of why this rule is checked>
status:
  id: xccdf_compliance.openshift.io_profile_rhcos4-with-usb 2
  outputRef:
    name: rhcos4-with-usb-tp 3
    namespace: openshift-compliance
  state: READY 4
```

- 1** This is optional. Name of the **Profile** object upon which the **TailoredProfile** is built. If no value is set, a new profile is created from the **enableRules** list.
- 2** Specifies the XCCDF name of the tailored profile.
- 3** Specifies the **ConfigMap** name, which can be used as the value of the **tailoringConfigMap.name** attribute of a **ComplianceScan**.
- 4** Shows the state of the object such as **READY**, **PENDING**, and **FAILURE**. If the state of the object is **ERROR**, then the attribute **status.errorMessage** provides the reason for the failure.

With the **TailoredProfile** object, it is possible to create a new **Profile** object using the **TailoredProfile** construct. To create a new **Profile**, set the following configuration parameters :

- an appropriate title
- **extends** value must be empty

- scan type annotation on the **TailoredProfile** object:

```
compliance.openshift.io/product-type: <scan type>
```



NOTE

If you have not set the **product-type** annotation, the Compliance Operator defaults to **Platform** scan type. Adding the **-node** suffix to the name of the **TailoredProfile** object results in **node** scan type.

5.13.3. Configuring the compliance scan settings

After you have defined the requirements of the compliance scan, you can configure it by specifying the type of the scan, occurrence of the scan, and location of the scan. To do so, Compliance Operator provides you with a **ScanSetting** object.

5.13.3.1. ScanSetting object

Use the **ScanSetting** object to define and reuse the operational policies to run your scans. By default, the Compliance Operator creates the following **ScanSetting** objects:

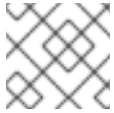
- **default** - it runs a scan every day at 1 AM on both master and worker nodes using a 1Gi Persistent Volume (PV) and keeps the last three results. Remediation is neither applied nor updated automatically.
- **default-auto-apply** - it runs a scan every day at 1AM on both control plane and worker nodes using a 1Gi Persistent Volume (PV) and keeps the last three results. Both **autoApplyRemediations** and **autoUpdateRemediations** are set to true.

Example ScanSetting object

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSetting
metadata:
  name: <name of the scan>
autoApplyRemediations: false 1
autoUpdateRemediations: false 2
schedule: "0 1 * * *" 3
rawResultStorage:
  size: "2Gi" 4
  rotation: 10 5
roles: 6
  - worker
  - master
```

- 1 Set to **true** to enable auto remediations. Set to **false** to disable auto remediations.
- 2 Set to **true** to enable auto remediations for content updates. Set to **false** to disable auto remediations for content updates.
- 3 Specify how often the scan should be run in cron format.
- 4 Specify the storage size that should be created for the scan to store the raw results. The default value is **1Gi**

- 5 Specify the amount of scans for which the raw results will be stored. The default value is **3**. As the older results get rotated, the administrator has to store the results elsewhere before the rotation

**NOTE**

To disable the rotation policy, set the value to **0**.

- 6 Specify the **node-role.kubernetes.io** label value to schedule the scan for **Node** type. This value has to match the name of a **MachineConfigPool**.

5.13.4. Processing the compliance scan requirements with compliance scans settings

When you have defined the compliance scan requirements and configured the settings to run the scans, then the Compliance Operator processes it using the **ScanSettingBinding** object.

5.13.4.1. ScanSettingBinding object

Use the **ScanSettingBinding** object to specify your compliance requirements with reference to the **Profile** or **TailoredProfile** object. It is then linked to a **ScanSetting** object, which provides the operational constraints for the scan. Then the Compliance Operator generates the **ComplianceSuite** object based on the **ScanSetting** and **ScanSettingBinding** objects.

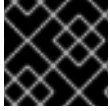
Example ScanSettingBinding object

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ScanSettingBinding
metadata:
  name: <name of the scan>
profiles: 1
  # Node checks
  - name: rhcos4-with-usb
    kind: TailoredProfile
    apiGroup: compliance.openshift.io/v1alpha1
  # Cluster checks
  - name: ocp4-moderate
    kind: Profile
    apiGroup: compliance.openshift.io/v1alpha1
settingsRef: 2
  name: my-companys-constraints
  kind: ScanSetting
  apiGroup: compliance.openshift.io/v1alpha1
```

- 1 Specify the details of **Profile** or **TailoredProfile** object to scan your environment.
- 2 Specify the operational constraints, such as schedule and storage size.

The creation of **ScanSetting** and **ScanSettingBinding** objects results in the compliance suite. To get the list of compliance suite, run the following command:

```
$ oc get compliancesuites
```

IMPORTANT

If you delete **ScanSettingBinding**, then compliance suite also is deleted.

5.13.5. Tracking the compliance scans

After the creation of compliance suite, you can monitor the status of the deployed scans using the **ComplianceSuite** object.

5.13.5.1. ComplianceSuite object

The **ComplianceSuite** object helps you keep track of the state of the scans. It contains the raw settings to create scans and the overall result.

For **Node** type scans, you should map the scan to the **MachineConfigPool**, since it contains the remediations for any issues. If you specify a label, ensure it directly applies to a pool.

Example ComplianceSuite object

```

apiVersion: compliance.openshift.io/v1alpha1
kind: ComplianceSuite
metadata:
  name: <name of the scan>
spec:
  autoApplyRemediations: false 1
  schedule: "0 1 * * *" 2
  scans: 3
  - name: workers-scan
    scanType: Node
    profile: xccdf_org.ssgproject.content_profile_moderate
    content: ssg-rhcos4-ds.xml
    contentImage: quay.io/complianceascode/ocp4:latest
    rule: "xccdf_org.ssgproject.content_rule_no_netrc_files"
    nodeSelector:
      node-role.kubernetes.io/worker: ""
status:
  Phase: DONE 4
  Result: NON-COMPLIANT 5
  scanStatuses:
  - name: workers-scan
    phase: DONE
    result: NON-COMPLIANT

```

- 1** Set to **true** to enable auto remediations. Set to **false** to disable auto remediations.
- 2** Specify how often the scan should be run in cron format.
- 3** Specify a list of scan specifications to run in the cluster.
- 4** Indicates the progress of the scans.
- 5** Indicates the overall verdict of the suite.

The suite in the background creates the **ComplianceScan** object based on the **scans** parameter. You can programmatically fetch the **ComplianceSuites** events. To get the events for the suite, run the following command:

```
$ oc get events --field-selector involvedObject.kind=ComplianceSuite,involvedObject.name=<name of the suite>
```



IMPORTANT

You might create errors when you manually define the **ComplianceSuite**, since it contains the XCCDF attributes.

5.13.5.2. Advanced ComplianceScan Object

The Compliance Operator includes options for advanced users for debugging or integrating with existing tooling. While it is recommended that you not create a **ComplianceScan** object directly, you can instead manage it using a **ComplianceSuite** object.

Example Advanced ComplianceScan object

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ComplianceScan
metadata:
  name: <name of the scan>
spec:
  scanType: Node 1
  profile: xccdf_org.ssgproject.content_profile_moderate 2
  content: ssg-ocp4-ds.xml
  contentImage: quay.io/complianceascode/ocp4:latest 3
  rule: "xccdf_org.ssgproject.content_rule_no_netrc_files" 4
  nodeSelector: 5
    node-role.kubernetes.io/worker: ""
status:
  phase: DONE 6
  result: NON-COMPLIANT 7
```

- 1 Specify either **Node** or **Platform**. Node profiles scan the cluster nodes and platform profiles scan the Kubernetes platform.
- 2 Specify the XCCDF identifier of the profile that you want to run.
- 3 Specify the container image that encapsulates the profile files.
- 4 It is optional. Specify the scan to run a single rule. This rule has to be identified with the XCCDF ID, and has to belong to the specified profile.



NOTE

If you skip the **rule** parameter, then scan runs for all the available rules of the specified profile.

- 5 If you are on the OpenShift Container Platform and wants to generate a remediation, then nodeSelector label has to match the **MachineConfigPool** label.

**NOTE**

If you do not specify **nodeSelector** parameter or match the **MachineConfig** label, scan will still run, but it will not create remediation.

- 6 Indicates the current phase of the scan.
- 7 Indicates the verdict of the scan.

**IMPORTANT**

If you delete a **ComplianceSuite** object, then all the associated scans get deleted.

When the scan is complete, it generates the result as Custom Resources of the **ComplianceCheckResult** object. However, the raw results are available in ARF format. These results are stored in a Persistent Volume (PV), which has a Persistent Volume Claim (PVC) associated with the name of the scan. You can programmatically fetch the **ComplianceScans** events. To generate events for the suite, run the following command:

```
oc get events --field-selector involvedObject.kind=ComplianceScan,involvedObject.name=<name of the suite>
```

5.13.6. Viewing the compliance results

When the compliance suite reaches the **DONE** phase, you can view the scan results and possible remediations.

5.13.6.1. ComplianceCheckResult object

When you run a scan with a specific profile, several rules in the profiles are verified. For each of these rules, a **ComplianceCheckResult** object is created, which provides the state of the cluster for a specific rule.

Example ComplianceCheckResult object

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ComplianceCheckResult
metadata:
  labels:
    compliance.openshift.io/check-severity: medium
    compliance.openshift.io/check-status: FAIL
    compliance.openshift.io/suite: example-compliancesuite
    compliance.openshift.io/scan-name: workers-scan
name: workers-scan-no-direct-root-logins
namespace: openshift-compliance
ownerReferences:
- apiVersion: compliance.openshift.io/v1alpha1
  blockOwnerDeletion: true
  controller: true
  kind: ComplianceScan
  name: workers-scan
description: <description of scan check>
```

```
instructions: <manual instructions for the scan>
id: xccdf_org.ssgproject.content_rule_no_direct_root_logins
severity: medium 1
status: FAIL 2
```

1 Describes the severity of the scan check.

2 Describes the result of the check. The possible values are:

- PASS: check was successful.
- FAIL: check was unsuccessful.
- INFO: check was successful and found something not severe enough to be considered an error.
- MANUAL: check cannot automatically assess the status and manual check is required.
- INCONSISTENT: different nodes report different results.
- ERROR: check run successfully, but could not complete.
- NOTAPPLICABLE: check did not run as it is not applicable.

To get all the check results from a suite, run the following command:

```
oc get compliancecheckresults -l compliance.openshift.io/suite=<suit name>
```

5.13.6.2. ComplianceRemediation object

For a specific check you can have a datastream specified fix. However, if a Kubernetes fix is available, then the Compliance Operator creates a **ComplianceRemediation** object.

Example ComplianceRemediation object

```
apiVersion: compliance.openshift.io/v1alpha1
kind: ComplianceRemediation
metadata:
  labels:
    compliance.openshift.io/suite: example-compliancesuite
    compliance.openshift.io/scan-name: workers-scan
    machineconfiguration.openshift.io/role: worker
  name: workers-scan-disable-users-coredumps
  namespace: openshift-compliance
  ownerReferences:
  - apiVersion: compliance.openshift.io/v1alpha1
    blockOwnerDeletion: true
    controller: true
    kind: ComplianceCheckResult
    name: workers-scan-disable-users-coredumps
    uid: <UID>
spec:
  apply: false 1
  object:
```

```

current: ❷
  apiVersion: machineconfiguration.openshift.io/v1
  kind: MachineConfig
  spec:
    config:
      ignition:
        version: 2.2.0
      storage:
        files:
          - contents:
              source: data:,%2A%20%20%20%20%20hard%20%20%20core%20%20%20%20
              filesystem: root
              mode: 420
              path: /etc/security/limits.d/75-disable_users_coredumps.conf
    outdated: {} ❸

```

- ❶ **true** indicates the remediation was applied. **false** indicates the remediation was not applied.
- ❷ Includes the definition of the remediation.
- ❸ Indicates remediation that was previously parsed from an earlier version of the content. The Compliance Operator still retains the outdated objects to give the administrator a chance to review the new remediations before applying them.

To get all the remediations from a suite, run the following command:

```
oc get complianceremediations -l compliance.openshift.io/suite=<suite name>
```

To list all failing checks that can be remediated automatically, run the following command:

```
oc get compliancecheckresults -l 'compliance.openshift.io/check-status in (FAIL),compliance.openshift.io/automated-remediation'
```

To list all failing checks that can be remediated manually, run the following command:

```
oc get compliancecheckresults -l 'compliance.openshift.io/check-status in (FAIL),!compliance.openshift.io/automated-remediation'
```

CHAPTER 6. FILE INTEGRITY OPERATOR

6.1. FILE INTEGRITY OPERATOR RELEASE NOTES

The File Integrity Operator for OpenShift Container Platform continually runs file integrity checks on RHCOS nodes.

These release notes track the development of the File Integrity Operator in the OpenShift Container Platform.

For an overview of the File Integrity Operator, see [Understanding the File Integrity Operator](#).

6.1.1. OpenShift File Integrity Operator 0.1.30

The following advisory is available for the OpenShift File Integrity Operator 0.1.30:

- [RHBA-2022:5538 OpenShift File Integrity Operator Bug Fix and Enhancement Update](#)

6.1.1.1. Bug fixes

- Previously, alerts issued by the File Integrity Operator did not set a namespace, making it difficult to understand where the alert originated. Now, the Operator sets the appropriate namespace, increasing understanding of the alert. ([BZ#2101393](#))

6.1.2. OpenShift File Integrity Operator 0.1.24

The following advisory is available for the OpenShift File Integrity Operator 0.1.24:

- [RHBA-2022:1331 OpenShift File Integrity Operator Bug Fix](#)

6.1.2.1. New features and enhancements

- You can now configure the maximum number of backups stored in the **FileIntegrity** Custom Resource (CR) with the **config.maxBackups** attribute. This attribute specifies the number of AIDE database and log backups left over from the **re-init** process to keep on the node. Older backups beyond the configured number are automatically pruned. The default is set to five backups.

6.1.2.2. Bug fixes

- Previously, upgrading the Operator from versions older than 0.1.21 to 0.1.22 could cause the **re-init** feature to fail. This was a result of the Operator failing to update **configMap** resource labels. Now, upgrading to the latest version fixes the resource labels. ([BZ#2049206](#))
- Previously, when enforcing the default **configMap** script contents, the wrong data keys were compared. This resulted in the **aide-reinit** script not being updated properly after an Operator upgrade, and caused the **re-init** process to fail. Now, **daemonSets** run to completion and the AIDE database **re-init** process executes successfully. ([BZ#2072058](#))

6.1.3. OpenShift File Integrity Operator 0.1.22

The following advisory is available for the OpenShift File Integrity Operator 0.1.22:

- [RHBA-2022:0142 OpenShift File Integrity Operator Bug Fix](#)

6.1.3.1. Bug fixes

- Previously, a system with a File Integrity Operator installed might interrupt the OpenShift Container Platform update, due to the `/etc/kubernetes/aide.reinit` file. This occurred if the `/etc/kubernetes/aide.reinit` file was present, but later removed prior to the `ostree` validation. With this update, `/etc/kubernetes/aide.reinit` is moved to the `/run` directory so that it does not conflict with the OpenShift Container Platform update. ([BZ#2033311](#))

6.1.4. OpenShift File Integrity Operator 0.1.21

The following advisory is available for the OpenShift File Integrity Operator 0.1.21:

- [RHBA-2021:4631 OpenShift File Integrity Operator Bug Fix and Enhancement Update](#)

6.1.4.1. New features and enhancements

- The metrics related to **FileIntegrity** scan results and processing metrics are displayed on the monitoring dashboard on the web console. The results are labeled with the prefix of `file_integrity_operator_`.
- If a node has an integrity failure for more than 1 second, the default **PrometheusRule** provided in the operator namespace alerts with a warning.
- The following dynamic Machine Config Operator and Cluster Version Operator related filepaths are excluded from the default AIDE policy to help prevent false positives during node updates:
 - `/etc/machine-config-daemon/currentconfig`
 - `/etc/pki/ca-trust/extracted/java/cacerts`
 - `/etc/cvo/updatepayloads`
 - `/root/.kube`
- The AIDE daemon process has stability improvements over v0.1.16, and is more resilient to errors that might occur when the AIDE database is initialized.

6.1.4.2. Bug fixes

- Previously, when the Operator automatically upgraded, outdated daemon sets were not removed. With this release, outdated daemon sets are removed during the automatic upgrade.

6.1.5. Additional resources

- [Understanding the File Integrity Operator](#)

6.2. INSTALLING THE FILE INTEGRITY OPERATOR

6.2.1. Installing the File Integrity Operator using the web console

Prerequisites

- You must have **admin** privileges.

Procedure

1. In the OpenShift Container Platform web console, navigate to **Operators** → **OperatorHub**.
2. Search for the File Integrity Operator, then click **Install**.
3. Keep the default selection of **Installation mode** and **namespace** to ensure that the Operator will be installed to the **openshift-file-integrity** namespace.
4. Click **Install**.

Verification

To confirm that the installation is successful:

1. Navigate to the **Operators** → **Installed Operators** page.
2. Check that the Operator is installed in the **openshift-file-integrity** namespace and its status is **Succeeded**.

If the Operator is not installed successfully:

1. Navigate to the **Operators** → **Installed Operators** page and inspect the **Status** column for any errors or failures.
2. Navigate to the **Workloads** → **Pods** page and check the logs in any pods in the **openshift-file-integrity** project that are reporting issues.

6.2.2. Installing the File Integrity Operator using the CLI

Prerequisites

- You must have **admin** privileges.

Procedure

1. Create a **Namespace** object YAML file by running:

```
$ oc create -f <file-name>.yaml
```

Example output

```
apiVersion: v1
kind: Namespace
metadata:
  labels:
    openshift.io/cluster-monitoring: "true"
  name: openshift-file-integrity
```

2. Create the **OperatorGroup** object YAML file:

```
$ oc create -f <file-name>.yaml
```


Example output

```

apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: file-integrity-operator
  namespace: openshift-file-integrity
spec:
  targetNamespaces:
  - openshift-file-integrity

```

3. Create the **Subscription** object YAML file:

```
$ oc create -f <file-name>.yaml
```

Example output

```

apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: file-integrity-operator
  namespace: openshift-file-integrity
spec:
  channel: "release-0.1"
  installPlanApproval: Automatic
  name: file-integrity-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace

```

Verification

1. Verify the installation succeeded by inspecting the CSV file:

```
$ oc get csv -n openshift-file-integrity
```

2. Verify that the File Integrity Operator is up and running:

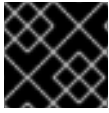
```
$ oc get deploy -n openshift-file-integrity
```

6.2.3. Additional resources

- The File Integrity Operator is supported in a restricted network environment. For more information, see [Using Operator Lifecycle Manager on restricted networks](#).

6.3. UNDERSTANDING THE FILE INTEGRITY OPERATOR

The File Integrity Operator is an OpenShift Container Platform Operator that continually runs file integrity checks on the cluster nodes. It deploys a daemon set that initializes and runs privileged advanced intrusion detection environment (AIDE) containers on each node, providing a status object with a log of files that are modified during the initial run of the daemon set pods.



IMPORTANT

Currently, only Red Hat Enterprise Linux CoreOS (RHCOS) nodes are supported.

6.3.1. Creating the FileIntegrity custom resource

An instance of a **FileIntegrity** custom resource (CR) represents a set of continuous file integrity scans for one or more nodes.

Each **FileIntegrity** CR is backed by a daemon set running AIDE on the nodes matching the **FileIntegrity** CR specification.

Procedure

1. Create the following example **FileIntegrity** CR named **worker-fileintegrity.yaml** to enable scans on worker nodes:

Example FileIntegrity CR

```
apiVersion: fileintegrity.openshift.io/v1alpha1
kind: FileIntegrity
metadata:
  name: worker-fileintegrity
  namespace: openshift-file-integrity
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  config: {}
```

2. Apply the YAML file to the **openshift-file-integrity** namespace:

```
$ oc apply -f worker-fileintegrity.yaml -n openshift-file-integrity
```

Verification

- Confirm the **FileIntegrity** object was created successfully by running the following command:

```
$ oc get fileintegrities -n openshift-file-integrity
```

Example output

```
NAME             AGE
worker-fileintegrity 14s
```

6.3.2. Checking the FileIntegrity custom resource status

The **FileIntegrity** custom resource (CR) reports its status through the **.status.phase** subresource.

Procedure

- To query the **FileIntegrity** CR status, run:

```
$ oc get fileintegrities/worker-fileintegrity -o jsonpath="{.status.phase}"
```

▪

Example output

Active

6.3.3. FileIntegrity custom resource phases

- **Pending** - The phase after the custom resource (CR) is created.
- **Active** - The phase when the backing daemon set is up and running.
- **Initializing** - The phase when the AIDE database is being reinitialized.

6.3.4. Understanding the FileIntegrityNodeStatuses object

The scan results of the **FileIntegrity** CR are reported in another object called **FileIntegrityNodeStatuses**.

```
$ oc get fileintegritynodestatuses
```

Example output

NAME	AGE
worker-fileintegrity-ip-10-0-130-192.ec2.internal	101s
worker-fileintegrity-ip-10-0-147-133.ec2.internal	109s
worker-fileintegrity-ip-10-0-165-160.ec2.internal	102s

**NOTE**

It might take some time for the **FileIntegrityNodeStatus** object results to be available.

There is one result object per node. The **nodeName** attribute of each **FileIntegrityNodeStatus** object corresponds to the node being scanned. The status of the file integrity scan is represented in the **results** array, which holds scan conditions.

```
$ oc get fileintegritynodestatuses.fileintegrity.openshift.io -ojsonpath='{.items[*].results}' | jq
```

The **fileintegritynodestatus** object reports the latest status of an AIDE run and exposes the status as **Failed**, **Succeeded**, or **Errored** in a **status** field.

```
$ oc get fileintegritynodestatuses -w
```

Example output

NAME	NODE	STATUS
example-fileintegrity-ip-10-0-134-186.us-east-2.compute.internal	ip-10-0-134-186.us-east-2.compute.internal	Succeeded
example-fileintegrity-ip-10-0-150-230.us-east-2.compute.internal	ip-10-0-150-230.us-east-2.compute.internal	Succeeded
example-fileintegrity-ip-10-0-169-137.us-east-2.compute.internal	ip-10-0-169-137.us-east-2.compute.internal	Succeeded

```

example-fileintegrity-ip-10-0-180-200.us-east-2.compute.internal ip-10-0-180-200.us-east-2.compute.internal Succeeded
example-fileintegrity-ip-10-0-194-66.us-east-2.compute.internal ip-10-0-194-66.us-east-2.compute.internal Failed
example-fileintegrity-ip-10-0-222-188.us-east-2.compute.internal ip-10-0-222-188.us-east-2.compute.internal Succeeded
example-fileintegrity-ip-10-0-134-186.us-east-2.compute.internal ip-10-0-134-186.us-east-2.compute.internal Succeeded
example-fileintegrity-ip-10-0-222-188.us-east-2.compute.internal ip-10-0-222-188.us-east-2.compute.internal Succeeded
example-fileintegrity-ip-10-0-194-66.us-east-2.compute.internal ip-10-0-194-66.us-east-2.compute.internal Failed
example-fileintegrity-ip-10-0-150-230.us-east-2.compute.internal ip-10-0-150-230.us-east-2.compute.internal Succeeded
example-fileintegrity-ip-10-0-180-200.us-east-2.compute.internal ip-10-0-180-200.us-east-2.compute.internal Succeeded

```

6.3.5. FileIntegrityNodeStatus CR status types

These conditions are reported in the results array of the corresponding **FileIntegrityNodeStatus** CR status:

- **Succeeded** - The integrity check passed; the files and directories covered by the AIDE check have not been modified since the database was last initialized.
- **Failed** - The integrity check failed; some files or directories covered by the AIDE check have been modified since the database was last initialized.
- **Errored** - The AIDE scanner encountered an internal error.

6.3.5.1. FileIntegrityNodeStatus CR success example

Example output of a condition with a success status

```

[
  {
    "condition": "Succeeded",
    "lastProbeTime": "2020-09-15T12:45:57Z"
  }
]
[
  {
    "condition": "Succeeded",
    "lastProbeTime": "2020-09-15T12:46:03Z"
  }
]
[
  {
    "condition": "Succeeded",
    "lastProbeTime": "2020-09-15T12:45:48Z"
  }
]

```

In this case, all three scans succeeded and so far there are no other conditions.

6.3.5.2. FileIntegrityNodeStatus CR failure status example

To simulate a failure condition, modify one of the files AIDE tracks. For example, modify `/etc/resolv.conf` on one of the worker nodes:

```
$ oc debug node/ip-10-0-130-192.ec2.internal
```

Example output

```
Creating debug namespace/openshift-debug-node-ldfbj ...
Starting pod/ip-10-0-130-192ec2internal-debug ...
To use host binaries, run `chroot /host`
Pod IP: 10.0.130.192
If you don't see a command prompt, try pressing enter.
sh-4.2# echo "# integrity test" >> /host/etc/resolv.conf
sh-4.2# exit

Removing debug pod ...
Removing debug namespace/openshift-debug-node-ldfbj ...
```

After some time, the **Failed** condition is reported in the results array of the corresponding **FileIntegrityNodeStatus** object. The previous **Succeeded** condition is retained, which allows you to pinpoint the time the check failed.

```
$ oc get fileintegritynodestatuses.fileintegrity.openshift.io/worker-fileintegrity-ip-10-0-130-192.ec2.internal -ojsonpath='{.results}' | jq -r
```

Alternatively, if you are not mentioning the object name, run:

```
$ oc get fileintegritynodestatuses.fileintegrity.openshift.io -ojsonpath='{.items[*].results}' | jq
```

Example output

```
[
  {
    "condition": "Succeeded",
    "lastProbeTime": "2020-09-15T12:54:14Z"
  },
  {
    "condition": "Failed",
    "filesChanged": 1,
    "lastProbeTime": "2020-09-15T12:57:20Z",
    "resultConfigMapName": "aide-ds-worker-fileintegrity-ip-10-0-130-192.ec2.internal-failed",
    "resultConfigMapNamespace": "openshift-file-integrity"
  }
]
```

The **Failed** condition points to a config map that gives more details about what exactly failed and why:

```
$ oc describe cm aide-ds-worker-fileintegrity-ip-10-0-130-192.ec2.internal-failed
```

Example output

■

```
Name:      aide-ds-worker-fileintegrity-ip-10-0-130-192.ec2.internal-failed
Namespace: openshift-file-integrity
Labels:    file-integrity.openshift.io/node=ip-10-0-130-192.ec2.internal
           file-integrity.openshift.io/owner=worker-fileintegrity
           file-integrity.openshift.io/result-log=
Annotations: file-integrity.openshift.io/files-added: 0
             file-integrity.openshift.io/files-changed: 1
             file-integrity.openshift.io/files-removed: 0
```

Data

```
integritylog:
```

```
-----
```

```
AIDE 0.15.1 found differences between database and filesystem!!
```

```
Start timestamp: 2020-09-15 12:58:15
```

Summary:

```
Total number of files: 31553
```

```
Added files:          0
```

```
Removed files:       0
```

```
Changed files:       1
```

```
-----
Changed files:
```

```
-----
changed: /hostroot/etc/resolv.conf
```

```
-----
Detailed information about changes:
```

```
-----
File: /hostroot/etc/resolv.conf
```

```
SHA512  : sTQYpB/AL7FeoGtu/1g7opv6C+KT1C BJ , qAeM+a8yTgHPnIHMaRIS+so61EN8VOpg
```

```
Events: <none>
```

Due to the config map data size limit, AIDE logs over 1 MB are added to the failure config map as a base64-encoded gzip archive. In this case, you want to pipe the output of the above command to **base64 --decode | gunzip**. Compressed logs are indicated by the presence of a **file-integrity.openshift.io/compressed** annotation key in the config map.

6.3.6. Understanding events

Transitions in the status of the **FileIntegrity** and **FileIntegrityNodeStatus** objects are logged by *events*. The creation time of the event reflects the latest transition, such as **Initializing** to **Active**, and not necessarily the latest scan result. However, the newest event always reflects the most recent status.

```
$ oc get events --field-selector reason=FileIntegrityStatus
```

Example output

LAST SEEN	TYPE	REASON	OBJECT	MESSAGE
-----------	------	--------	--------	---------

```

97s    Normal FileIntegrityStatus fileintegrity/example-fileintegrity Pending
67s    Normal FileIntegrityStatus fileintegrity/example-fileintegrity Initializing
37s    Normal FileIntegrityStatus fileintegrity/example-fileintegrity Active

```

When a node scan fails, an event is created with the **add/changed/removed** and config map information.

```
$ oc get events --field-selector reason=NodeIntegrityStatus
```

Example output

```

LAST SEEN   TYPE      REASON          OBJECT                                     MESSAGE
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-134-173.ec2.internal
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-168-238.ec2.internal
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-169-175.ec2.internal
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-152-92.ec2.internal
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-158-144.ec2.internal
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-131-30.ec2.internal
87m         Warning   NodeIntegrityStatus fileintegrity/example-fileintegrity node ip-10-0-152-92.ec2.internal has changed! a:1,c:1,r:0 \ log:openshift-file-integrity/aide-ds-example-fileintegrity-ip-10-0-152-92.ec2.internal-failed

```

Changes to the number of added, changed, or removed files results in a new event, even if the status of the node has not transitioned.

```
$ oc get events --field-selector reason=NodeIntegrityStatus
```

Example output

```

LAST SEEN   TYPE      REASON          OBJECT                                     MESSAGE
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-134-173.ec2.internal
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-168-238.ec2.internal
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-169-175.ec2.internal
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-152-92.ec2.internal
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-158-144.ec2.internal
114m        Normal    NodeIntegrityStatus fileintegrity/example-fileintegrity no changes to node ip-10-0-131-30.ec2.internal
87m         Warning   NodeIntegrityStatus fileintegrity/example-fileintegrity node ip-10-0-152-92.ec2.internal has changed! a:1,c:1,r:0 \ log:openshift-file-integrity/aide-ds-example-fileintegrity-ip-10-0-152-92.ec2.internal-failed
40m         Warning   NodeIntegrityStatus fileintegrity/example-fileintegrity node ip-10-0-152-92.ec2.internal has changed! a:3,c:1,r:0 \ log:openshift-file-integrity/aide-ds-example-fileintegrity-ip-10-0-152-92.ec2.internal-failed

```

6.4. CONFIGURING THE CUSTOM FILE INTEGRITY OPERATOR

6.4.1. Viewing FileIntegrity object attributes

As with any Kubernetes custom resources (CRs), you can run **oc explain fileintegrity**, and then look at the individual attributes using:

```
$ oc explain fileintegrity.spec
```

```
$ oc explain fileintegrity.spec.config
```

6.4.2. Important attributes

Table 6.1. Important **spec** and **spec.config** attributes

Attribute	Description
spec.nodeSelector	A map of key-values pairs that must match with node's labels in order for the AIDE pods to be schedulable on that node. The typical use is to set only a single key-value pair where node-role.kubernetes.io/worker: "" schedules AIDE on all worker nodes, node.openshift.io/os_id: "rhcos" schedules on all Red Hat Enterprise Linux CoreOS (RHCOS) nodes.
spec.debug	A boolean attribute. If set to true , the daemon running in the AIDE daemon set's pods would output extra information.
spec.tolerations	Specify tolerations to schedule on nodes with custom taints. When not specified, a default toleration is applied, which allows tolerations to run on control plane nodes (also known as the master nodes).
spec.config.gracePeriod	The number of seconds to pause in between AIDE integrity checks. Frequent AIDE checks on a node can be resource intensive, so it can be useful to specify a longer interval. Defaults to 900 , or 15 minutes.
maxBackups	The maximum number of AIDE database and log backups leftover from the re-init process to keep on a node. Older backups beyond this number are automatically pruned by the daemon.

Attribute	Description
spec.config.name	Name of a configMap that contains custom AIDE configuration. If omitted, a default configuration is created.
spec.config.namespace	Namespace of a configMap that contains custom AIDE configuration. If unset, the FIO generates a default configuration suitable for RHCOS systems.
spec.config.key	Key that contains actual AIDE configuration in a config map specified by name and namespace . The default value is aide.conf .

6.4.3. Examine the default configuration

The default File Integrity Operator configuration is stored in a config map with the same name as the **FileIntegrity** CR.

Procedure

- To examine the default config, run:

```
$ oc describe cm/worker-fileintegrity
```

6.4.4. Understanding the default File Integrity Operator configuration

Below is an excerpt from the **aide.conf** key of the config map:

```
@@define DBDIR /hostroot/etc/kubernetes
@@define LOGDIR /hostroot/etc/kubernetes
database=file:@@{DBDIR}/aide.db.gz
database_out=file:@@{DBDIR}/aide.db.gz
gzip_dbout=yes
verbose=5
report_url=file:@@{LOGDIR}/aide.log
report_url=stdout
PERMS = p+u+g+acl+selinux+xattrs
CONTENT_EX = sha512+ftype+p+u+g+n+acl+selinux+xattrs

/hostroot/boot/ CONTENT_EX
/hostroot/root/.* PERMS
/hostroot/root/ CONTENT_EX
```

The default configuration for a **FileIntegrity** instance provides coverage for files under the following directories:

- /root**
- /boot**
- /usr**

- **/etc**

The following directories are not covered:

- **/var**
- **/opt**
- Some OpenShift-specific excludes under **/etc/**

6.4.5. Supplying a custom AIDE configuration

Any entries that configure AIDE internal behavior such as **DBDIR**, **LOGDIR**, **database**, and **database_out** are overwritten by the Operator. The Operator would add a prefix to **/hostroot/** before all paths to be watched for integrity changes. This makes reusing existing AIDE configs that might often not be tailored for a containerized environment and start from the root directory easier.



NOTE

/hostroot is the directory where the pods running AIDE mount the host's file system. Changing the configuration triggers a reinitializing of the database.

6.4.6. Defining a custom File Integrity Operator configuration

This example focuses on defining a custom configuration for a scanner that runs on the control plane nodes (also known as the master nodes) based on the default configuration provided for the **worker-fileintegrity** CR. This workflow might be useful if you are planning to deploy a custom software running as a daemon set and storing its data under **/opt/mydaemon** on the control plane nodes.

Procedure

1. Make a copy of the default configuration.
2. Edit the default configuration with the files that must be watched or excluded.
3. Store the edited contents in a new config map.
4. Point the **FileIntegrity** object to the new config map through the attributes in **spec.config**.
5. Extract the default configuration:

```
$ oc extract cm/worker-fileintegrity --keys=aide.conf
```

This creates a file named **aide.conf** that you can edit. To illustrate how the Operator post-processes the paths, this example adds an exclude directory without the prefix:

```
$ vim aide.conf
```

Example output

```
/hostroot/etc/kubernetes/static-pod-resources
!/hostroot/etc/kubernetes/aide.*
!/hostroot/etc/kubernetes/manifests
```

```
! /hostroot/etc/docker/certs.d
! /hostroot/etc/selinux/targeted
! /hostroot/etc/openvswitch/conf.db
```

Exclude a path specific to control plane nodes:

```
! /opt/mydaemon/
```

Store the other content in **/etc**:

```
/hostroot/etc/ CONTENT_EX
```

6. Create a config map based on this file:

```
$ oc create cm master-aide-conf --from-file=aide.conf
```

7. Define a **FileIntegrity** CR manifest that references the config map:

```
apiVersion: fileintegrity.openshift.io/v1alpha1
kind: FileIntegrity
metadata:
  name: master-fileintegrity
  namespace: openshift-file-integrity
spec:
  nodeSelector:
    node-role.kubernetes.io/master: ""
  config:
    name: master-aide-conf
    namespace: openshift-file-integrity
```

The Operator processes the provided config map file and stores the result in a config map with the same name as the **FileIntegrity** object:

```
$ oc describe cm/master-fileintegrity | grep /opt/mydaemon
```

Example output

```
! /hostroot/opt/mydaemon
```

6.4.7. Changing the custom File Integrity configuration

To change the File Integrity configuration, never change the generated config map. Instead, change the config map that is linked to the **FileIntegrity** object through the **spec.name**, **namespace**, and **key** attributes.

6.5. PERFORMING ADVANCED CUSTOM FILE INTEGRITY OPERATOR TASKS

6.5.1. Reinitializing the database

If the File Integrity Operator detects a change that was planned, it might be required to reinitialize the database.

Procedure

- Annotate the **FileIntegrity** custom resource (CR) with **file-integrity.openshift.io/re-init**:

```
$ oc annotate fileintegrities/worker-fileintegrity file-integrity.openshift.io/re-init=
```

The old database and log files are backed up and a new database is initialized. The old database and logs are retained on the nodes under **/etc/kubernetes**, as seen in the following output from a pod spawned using **oc debug**:

Example output

```
ls -lR /host/etc/kubernetes/aide.*
-rw-----. 1 root root 1839782 Sep 17 15:08 /host/etc/kubernetes/aide.db.gz
-rw-----. 1 root root 1839783 Sep 17 14:30 /host/etc/kubernetes/aide.db.gz.backup-
20200917T15_07_38
-rw-----. 1 root root 73728 Sep 17 15:07 /host/etc/kubernetes/aide.db.gz.backup-
20200917T15_07_55
-rw-r--r--. 1 root root 0 Sep 17 15:08 /host/etc/kubernetes/aide.log
-rw-----. 1 root root 613 Sep 17 15:07 /host/etc/kubernetes/aide.log.backup-
20200917T15_07_38
-rw-r--r--. 1 root root 0 Sep 17 15:07 /host/etc/kubernetes/aide.log.backup-
20200917T15_07_55
```

To provide some permanence of record, the resulting config maps are not owned by the **FileIntegrity** object, so manual cleanup is necessary. As a result, any previous integrity failures would still be visible in the **FileIntegrityNodeStatus** object.

6.5.2. Machine config integration

In OpenShift Container Platform 4, the cluster node configuration is delivered through **MachineConfig** objects. You can assume that the changes to files that are caused by a **MachineConfig** object are expected and should not cause the file integrity scan to fail. To suppress changes to files caused by **MachineConfig** object updates, the File Integrity Operator watches the node objects; when a node is being updated, the AIDE scans are suspended for the duration of the update. When the update finishes, the database is reinitialized and the scans resume.

This pause and resume logic only applies to updates through the **MachineConfig** API, as they are reflected in the node object annotations.

6.5.3. Exploring the daemon sets

Each **FileIntegrity** object represents a scan on a number of nodes. The scan itself is performed by pods managed by a daemon set.

To find the daemon set that represents a **FileIntegrity** object, run:

```
$ oc -n openshift-file-integrity get ds/aide-worker-fileintegrity
```

To list the pods in that daemon set, run:

```
$ oc -n openshift-file-integrity get pods -lapp=aide-worker-fileintegrity
```

To view logs of a single AIDE pod, call **oc logs** on one of the pods.

```
$ oc -n openshift-file-integrity logs pod/aide-worker-fileintegrity-mr8x6
```

Example output

```
Starting the AIDE runner daemon
initializing AIDE db
initialization finished
running aide check
...
```

The config maps created by the AIDE daemon are not retained and are deleted after the File Integrity Operator processes them. However, on failure and error, the contents of these config maps are copied to the config map that the **FileIntegrityNodeStatus** object points to.

6.6. TROUBLESHOOTING THE FILE INTEGRITY OPERATOR

6.6.1. General troubleshooting

Issue

You want to generally troubleshoot issues with the File Integrity Operator.

Resolution

Enable the debug flag in the **FileIntegrity** object. The **debug** flag increases the verbosity of the daemons that run in the **DaemonSet** pods and run the AIDE checks.

6.6.2. Checking the AIDE configuration

Issue

You want to check the AIDE configuration.

Resolution

The AIDE configuration is stored in a config map with the same name as the **FileIntegrity** object. All AIDE configuration config maps are labeled with **file-integrity.openshift.io/aide-conf**.

6.6.3. Determining the FileIntegrity object's phase

Issue

You want to determine if the **FileIntegrity** object exists and see its current status.

Resolution

To see the **FileIntegrity** object's current status, run:

```
$ oc get fileintegrities/worker-fileintegrity -o jsonpath="{ .status }"
```

Once the **FileIntegrity** object and the backing daemon set are created, the status should switch to **Active**. If it does not, check the Operator pod logs.

6.6.4. Determining that the daemon set's pods are running on the expected nodes

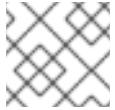
Issue

You want to confirm that the daemon set exists and that its pods are running on the nodes you expect them to run on.

Resolution

Run:

```
$ oc -n openshift-file-integrity get pods -lapp=aide-worker-fileintegrity
```



NOTE

Adding **-owide** includes the IP address of the node that the pod is running on.

To check the logs of the daemon pods, run **oc logs**.

Check the return value of the AIDE command to see if the check passed or failed.

CHAPTER 7. VIEWING AUDIT LOGS

OpenShift Container Platform auditing provides a security-relevant chronological set of records documenting the sequence of activities that have affected the system by individual users, administrators, or other components of the system.

7.1. ABOUT THE API AUDIT LOG

Audit works at the API server level, logging all requests coming to the server. Each audit log contains the following information:

Table 7.1. Audit log fields

Field	Description
level	The audit level at which the event was generated.
auditID	A unique audit ID, generated for each request.
stage	The stage of the request handling when this event instance was generated.
requestURI	The request URI as sent by the client to a server.
verb	The Kubernetes verb associated with the request. For non-resource requests, this is the lowercase HTTP method.
user	The authenticated user information.
impersonatedUser	Optional. The impersonated user information, if the request is impersonating another user.
sourceIPs	Optional. The source IPs, from where the request originated and any intermediate proxies.
userAgent	Optional. The user agent string reported by the client. Note that the user agent is provided by the client, and must not be trusted.
objectRef	Optional. The object reference this request is targeted at. This does not apply for List -type requests, or non-resource requests.
responseStatus	Optional. The response status, populated even when the ResponseObject is not a Status type. For successful responses, this will only include the code. For non-status type error responses, this will be auto-populated with the error message.

Field	Description
requestObject	Optional. The API object from the request, in JSON format. The RequestObject is recorded as is in the request (possibly re-encoded as JSON), prior to version conversion, defaulting, admission or merging. It is an external versioned object type, and might not be a valid object on its own. This is omitted for non-resource requests and is only logged at request level and higher.
responseObject	Optional. The API object returned in the response, in JSON format. The ResponseObject is recorded after conversion to the external type, and serialized as JSON. This is omitted for non-resource requests and is only logged at response level.
requestReceivedTimestamp	The time that the request reached the API server.
stageTimestamp	The time that the request reached the current audit stage.
annotations	Optional. An unstructured key value map stored with an audit event that may be set by plug-ins invoked in the request serving chain, including authentication, authorization and admission plug-ins. Note that these annotations are for the audit event, and do not correspond to the metadata.annotations of the submitted object. Keys should uniquely identify the informing component to avoid name collisions, for example podsecuritypolicy.admission.k8s.io/policy . Values should be short. Annotations are included in the metadata level.

Example output for the Kubernetes API server:

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "Metadata",
  "auditID": "ad209ce1-fec7-4130-8192-c4cc63f1d8cd",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/openshift-kube-controller-manager/configmaps/cert-recovery-controller-lock?timeout=35s",
  "verb": "update",
  "user": {
    "username": "system:serviceaccount:openshift-kube-controller-manager:localhost-recovery-client",
    "uid": "dd4997e3-d565-4e37-80f8-7fc122ccd785",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:openshift-kube-controller-manager",
      "system:authenticated"
    ],
    "sourceIPs": ["::1"],
    "userAgent": "cluster-kube-controller-manager-operator/v0.0.0 (linux/amd64) kubernetes/$Format",
    "objectRef": {
      "resource": "configmaps",
      "namespace": "openshift-kube-controller-manager",
      "name": "cert-recovery-controller-lock",
      "uid": "5c57190b-6993-425d-8101-8337e48c7548",
      "apiVersion": "v1",
      "resourceVersion": "574307"
    },
    "responseStatus": {
      "metadata": {},
      "code": 200
    },
    "requestReceivedTimestamp": "2020-04-02T08:27:20.200962Z",
    "stageTimestamp": "2020-04-02T08:27:20.206710Z",
    "annotations": {
      "authorization.k8s.io/decision": "allow",
      "authorization.k8s.io/reason": "RBAC: allowed by ClusterRoleBinding 'system:openshift:operator:kube-controller-manager-recovery' of ClusterRole 'cluster-admin' to ServiceAccount 'localhost-recovery-client/openshift-kube-controller-manager'"
    }
  }
}
```

7.2. VIEWING THE AUDIT LOGS

You can view the logs for the OpenShift API server, Kubernetes API server, and OpenShift OAuth API server for each control plane node (also known as the master node).

Procedure

To view the audit logs:

- View the OpenShift API server logs:
 - a. List the OpenShift API server logs that are available for each control plane node:

```
$ oc adm node-logs --role=master --path=openshift-apiserver/
```

Example output

```
ci-ln-m0wpfjb-f76d1-vnb5x-master-0 audit-2021-03-09T00-12-19.834.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-0 audit.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-1 audit-2021-03-09T00-11-49.835.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-1 audit.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-2 audit-2021-03-09T00-13-00.128.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-2 audit.log
```

- b. View a specific OpenShift API server log by providing the node name and the log name:

```
$ oc adm node-logs <node_name> --path=openshift-apiserver/<log_name>
```

For example:

```
$ oc adm node-logs ci-ln-m0wpfjb-f76d1-vnb5x-master-0 --path=openshift-apiserver/audit-2021-03-09T00-12-19.834.log
```

Example output

```
{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Metadata","auditID":"381acf6d-5f30-4c7d-8175-c9c317ae5893","stage":"ResponseComplete","requestURI":"/metrics","verb":"get","user":{"username":"system:serviceaccount:openshift-monitoring:prometheus-k8s","uid":"825b60a0-3976-4861-a342-3b2b561e8f82","groups":["system:serviceaccounts","system:serviceaccounts:openshift-monitoring","system:authenticated"]},"sourceIPs":["10.129.2.6"],"userAgent":"Prometheus/2.23.0","responseStatus":{"metadata":{"code":200},"requestReceivedTimestamp":"2021-03-08T18:02:04.086545Z","stageTimestamp":"2021-03-08T18:02:04.107102Z"},"annotations":{"authorization.k8s.io/decision":"allow","authorization.k8s.io/reason":"RBAC: allowed by ClusterRoleBinding \"prometheus-k8s\" of ClusterRole \"prometheus-k8s\" to ServiceAccount \"prometheus-k8s/openshift-monitoring\"\"}}
```

- View the Kubernetes API server logs:
 - a. List the Kubernetes API server logs that are available for each control plane node:

```
$ oc adm node-logs --role=master --path=kube-apiserver/
```

Example output

```
ci-ln-m0wpfjb-f76d1-vnb5x-master-0 audit-2021-03-09T14-07-27.129.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-0 audit.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-1 audit-2021-03-09T19-24-22.620.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-1 audit.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-2 audit-2021-03-09T18-37-07.511.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-2 audit.log
```

- b. View a specific Kubernetes API server log by providing the node name and the log name:

```
$ oc adm node-logs <node_name> --path=kube-apiserver/<log_name>
```

For example:

```
$ oc adm node-logs ci-ln-m0wpfjb-f76d1-vnb5x-master-0 --path=kube-apiserver/audit-2021-03-09T14-07-27.129.log
```

Example output

```
{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Metadata","auditID":"cfce8a0b-b5f5-4365-8c9f-79c1227d10f9","stage":"ResponseComplete","requestURI":"/api/v1/namespaces/openshift-kube-scheduler/serviceaccounts/openshift-kube-scheduler-sa","verb":"get","user":{"username":"system:serviceaccount:openshift-kube-scheduler-operator:openshift-kube-scheduler-operator","uid":"2574b041-f3c8-44e6-a057-baef7aa81516","groups":["system:serviceaccounts","system:serviceaccounts:openshift-kube-scheduler-operator","system:authenticated"],"sourceIPs":["10.128.0.8"],"userAgent":"cluster-kube-scheduler-operator/v0.0.0 (linux/amd64) kubernetes/$Format","objectRef":{"resource":"serviceaccounts","namespace":"openshift-kube-scheduler","name":"openshift-kube-scheduler-sa","apiVersion":"v1"},"responseStatus":{"metadata":{"code":200},"requestReceivedTimestamp":"2021-03-08T18:06:42.512619Z","stageTimestamp":"2021-03-08T18:06:42.516145Z","annotations":{"authentication.k8s.io/legacy-token":"system:serviceaccount:openshift-kube-scheduler-operator:openshift-kube-scheduler-operator","authorization.k8s.io/decision":"allow","authorization.k8s.io/reason":"RBAC: allowed by ClusterRoleBinding \"system:openshift:operator:cluster-kube-scheduler-operator\" of ClusterRole \"cluster-admin\" to ServiceAccount \"openshift-kube-scheduler-operator/openshift-kube-scheduler-operator\"}}}}
```

- View the OpenShift OAuth API server logs:
 - a. List the OpenShift OAuth API server logs that are available for each control plane node:

```
$ oc adm node-logs --role=master --path=oauth-apiserver/
```

Example output

```
ci-ln-m0wpfjb-f76d1-vnb5x-master-0 audit-2021-03-09T13-06-26.128.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-0 audit.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-1 audit-2021-03-09T18-23-21.619.log
```

```
ci-ln-m0wpfjb-f76d1-vnb5x-master-1 audit.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-2 audit-2021-03-09T17-36-06.510.log
ci-ln-m0wpfjb-f76d1-vnb5x-master-2 audit.log
```

- b. View a specific OpenShift OAuth API server log by providing the node name and the log name:

```
$ oc adm node-logs <node_name> --path=oauth-apiserver/<log_name>
```

For example:

```
$ oc adm node-logs ci-ln-m0wpfjb-f76d1-vnb5x-master-0 --path=oauth-apiserver/audit-2021-03-09T13-06-26.128.log
```

Example output

```
{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Metadata","auditID":"dd4c44e2-3ea1-4830-9ab7-c91a5f1388d6","stage":"ResponseComplete","requestURI":"/apis/user.openshift.io/v1/users/~","verb":"get","user":{"username":"system:serviceaccount:openshift-monitoring:prometheus-k8s","groups":["system:serviceaccounts","system:serviceaccounts:openshift-monitoring","system:authenticated"],"sourceIPs":["10.0.32.4","10.128.0.1"],"userAgent":"dockerregistry/v0.0.0 (linux/amd64) kubernetes/$Format","objectRef":{"resource":"users","name":"~","apiGroup":"user.openshift.io","apiVersion":"v1"},"responseStatus":{"metadata":{},"code":200},"requestReceivedTimestamp":"2021-03-08T17:47:43.653187Z","stageTimestamp":"2021-03-08T17:47:43.660187Z","annotations":{"authorization.k8s.io/decision":"allow","authorization.k8s.io/reason":"RBAC: allowed by ClusterRoleBinding \"basic-users\" of ClusterRole \"basic-user\" to Group \"system:authenticated\"}}
```

7.3. FILTERING AUDIT LOGS

You can use **jq** or another JSON parsing tool to filter the API server audit logs.



NOTE

The amount of information logged to the API server audit logs is controlled by the audit log policy that is set.

The following procedure provides examples of using **jq** to filter audit logs on control plane node **node-1.example.com**. See the [jq Manual](#) for detailed information on using **jq**.

Prerequisites

- You have access to the cluster as a user with the **cluster-admin** role.
- You have installed **jq**.

Procedure

- Filter OpenShift API server audit logs by user:

```
$ oc adm node-logs node-1.example.com \
  --path=openshift-apiserver/audit.log \
  | jq 'select(.user.username == "myusername")'
```

- Filter OpenShift API server audit logs by user agent:

```
$ oc adm node-logs node-1.example.com \
  --path=openshift-apiserver/audit.log \
  | jq 'select(.userAgent == "cluster-version-operator/v0.0.0 (linux/amd64)
  kubernetes/$Format")'
```

- Filter Kubernetes API server audit logs by a certain API version and only output the user agent:

```
$ oc adm node-logs node-1.example.com \
  --path=kube-apiserver/audit.log \
  | jq 'select(.requestURI | startswith("/apis/apiextensions.k8s.io/v1beta1")) | .userAgent'
```

- Filter OpenShift OAuth API server audit logs by excluding a verb:

```
$ oc adm node-logs node-1.example.com \
  --path=oauth-apiserver/audit.log \
  | jq 'select(.verb != "get")'
```

7.4. GATHERING AUDIT LOGS

You can use the `must-gather` tool to collect the audit logs for debugging your cluster, which you can review or send to Red Hat Support.

Procedure

1. Run the `oc adm must-gather` command with the `-- /usr/bin/gather_audit_logs` flag:

```
$ oc adm must-gather -- /usr/bin/gather_audit_logs
```

2. Create a compressed file from the `must-gather` directory that was just created in your working directory. For example, on a computer that uses a Linux operating system, run the following command:

```
$ tar cvaf must-gather.tar.gz must-gather.local.472290403699006248 1
```

- 1** Replace `must-gather-local.472290403699006248` with the actual directory name.

3. Attach the compressed file to your support case on the [Red Hat Customer Portal](#).

7.5. ADDITIONAL RESOURCES

- [Must-gather tool](#)
- [API audit log event structure](#)

- [Configuring the audit log policy](#)
- [Forwarding logs to third party systems](#)

CHAPTER 8. CONFIGURING THE AUDIT LOG POLICY

You can control the amount of information that is logged to the API server audit logs by choosing the audit log policy profile to use.

8.1. ABOUT AUDIT LOG POLICY PROFILES

Audit log profiles define how to log requests that come to the OpenShift API server, the Kubernetes API server, and the OAuth API server.

OpenShift Container Platform provides the following predefined audit policy profiles:

Profile	Description
Default	Logs only metadata for read and write requests; does not log request bodies except for OAuth access token creation (login) requests. This is the default policy.
WriteRequestBodies	In addition to logging metadata for all requests, logs request bodies for every write request to the API servers (create, update, patch). This profile has more resource overhead than the Default profile. ^[1]
AllRequestBodies	In addition to logging metadata for all requests, logs request bodies for every read and write request to the API servers (get, list, create, update, patch). This profile has the most resource overhead. ^[1]

1. Sensitive resources, such as **Secret**, **Route**, and **OAuthClient** objects, are never logged past the metadata level. OAuth tokens are not logged at all if your cluster was upgraded from OpenShift Container Platform 4.5, because their object names might contain secret information.

By default, OpenShift Container Platform uses the **Default** audit log profile. You can use another audit policy profile that also logs request bodies, but be aware of the increased resource usage (CPU, memory, and I/O).

8.2. CONFIGURING THE AUDIT LOG POLICY

You can configure the audit log policy to use when logging requests that come to the API servers.

Prerequisites

- You have access to the cluster as a user with the **cluster-admin** role.

Procedure

1. Edit the **APIServer** resource:

```
$ oc edit apiserver cluster
```

2. Update the **spec.audit.profile** field:

```
apiVersion: config.openshift.io/v1
```

```
kind: APIServer
metadata:
...
spec:
  audit:
    profile: WriteRequestBodies ❶
```

❶ Set to **Default**, **WriteRequestBodies**, or **AllRequestBodies**. The default profile is **Default**.

3. Save the file to apply the changes.
4. Verify that a new revision of the Kubernetes API server pods has rolled out. This will take several minutes.

```
$ oc get kubeapiserver -o=jsonpath='{range .items[0].status.conditions[?(@.type=="NodeInstallerProgressing")]}{.reason}\n'}{.message}\n}'
```

Review the **NodeInstallerProgressing** status condition for the Kubernetes API server to verify that all nodes are at the latest revision. The output shows **AllNodesAtLatestRevision** upon successful update:

```
AllNodesAtLatestRevision
3 nodes are at revision 12 ❶
```

❶ In this example, the latest revision number is **12**.

If the output shows a message similar to one of the following, this means that the update is still in progress. Wait a few minutes and try again.

- **3 nodes are at revision 11; 0 nodes have achieved new revision 12**
- **2 nodes are at revision 11; 1 nodes are at revision 12**

CHAPTER 9. CONFIGURING TLS SECURITY PROFILES

TLS security profiles provide a way for servers to regulate which ciphers a client can use when connecting to the server. This ensures that OpenShift Container Platform components use cryptographic libraries that do not allow known insecure protocols, ciphers, or algorithms.

Cluster administrators can choose which TLS security profile to use for each of the following components:

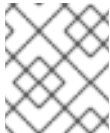
- the Ingress Controller
- the control plane
This includes the Kubernetes API server, OpenShift API server, OpenShift OAuth API server, and OpenShift OAuth server.





9.1. UNDERSTANDING TLS SECURITY PROFILES

You can use a TLS (Transport Layer Security) security profile to define which TLS ciphers are required by various OpenShift Container Platform components. The OpenShift Container Platform TLS security profiles are based on [Mozilla recommended configurations](#).

You can specify one of the following TLS security profiles for each component:

Table 9.1. TLS security profiles

Profile	Description
Old	<p>This profile is intended for use with legacy clients or libraries. The profile is based on the Old backward compatibility recommended configuration.</p> <p>The Old profile requires a minimum TLS version of 1.0.</p> <div style="display: flex; align-items: flex-start;">  <div> <p>NOTE</p> <p>For the Ingress Controller, the minimum TLS version is converted from 1.0 to 1.1.</p> </div> </div>
Intermediate	<p>This profile is the recommended configuration for the majority of clients. It is the default TLS security profile for the Ingress Controller and control plane. The profile is based on the Intermediate compatibility recommended configuration.</p> <p>The Intermediate profile requires a minimum TLS version of 1.2.</p>

Profile	Description
<p>Modern</p>	<p>This profile is intended for use with modern clients that have no need for backwards compatibility. This profile is based on the Modern compatibility recommended configuration.</p> <p>The Modern profile requires a minimum TLS version of 1.3.</p> <div data-bbox="595 439 703 602">  <p>NOTE</p> <p>In OpenShift Container Platform 4.6, 4.7, and 4.8, the Modern profile is unsupported. If selected, the Intermediate profile is enabled.</p> </div> <div data-bbox="595 651 703 752">  <p>IMPORTANT</p> <p>The Modern profile is currently not supported.</p> </div>
<p>Custom</p>	<p>This profile allows you to define the TLS version and ciphers to use.</p> <div data-bbox="595 1070 1428 1391">  <p>WARNING</p> <p>Use caution when using a Custom profile, because invalid configurations can cause problems.</p> </div> <div data-bbox="595 1440 703 1664">  <p>NOTE</p> <p>OpenShift Container Platform router enables Red Hat-distributed OpenSSL default set of TLS 1.3 cipher suites. Your cluster might accept TLS 1.3 connections and cipher suites, even though TLS 1.3 is unsupported in OpenShift Container Platform 4.6, 4.7, and 4.8.</p> </div>



NOTE

When using one of the predefined profile types, the effective profile configuration is subject to change between releases. For example, given a specification to use the Intermediate profile deployed on release X.Y.Z, an upgrade to release X.Y.Z+1 might cause a new profile configuration to be applied, resulting in a rollout.

9.2. VIEWING TLS SECURITY PROFILE DETAILS

You can view the minimum TLS version and ciphers for the predefined TLS security profiles for each of the following components: Ingress Controller and control plane.



IMPORTANT

The effective configuration of minimum TLS version and list of ciphers for a profile might differ between components.

Procedure

- View details for a specific TLS security profile:

```
$ oc explain <component>.spec.tlsSecurityProfile.<profile> 1
```

- 1** For **<component>**, specify **ingresscontroller** or **apiserver**. For **<profile>**, specify **old**, **intermediate**, or **custom**.

For example, to check the ciphers included for the **intermediate** profile for the control plane:

```
$ oc explain apiserver.spec.tlsSecurityProfile.intermediate
```

Example output

```
KIND: APIServer
VERSION: config.openshift.io/v1

DESCRIPTION:
  intermediate is a TLS security profile based on:

  https://wiki.mozilla.org/Security/Server_Side_TLS#Intermediate_compatibility_.28recommended_.29
  and looks like this (yaml):
  ciphers: - TLS_AES_128_GCM_SHA256 - TLS_AES_256_GCM_SHA384 -
  TLS_CHACHA20_POLY1305_SHA256 - ECDHE-ECDSA-AES128-GCM-SHA256 -
  ECDHE-RSA-AES128-GCM-SHA256 - ECDHE-ECDSA-AES256-GCM-SHA384 -
  ECDHE-RSA-AES256-GCM-SHA384 - ECDHE-ECDSA-CHACHA20-POLY1305 -
  ECDHE-RSA-CHACHA20-POLY1305 - DHE-RSA-AES128-GCM-SHA256 -
  DHE-RSA-AES256-GCM-SHA384 minTLSVersion: TLSv1.2
```

- View all details for the **tlsSecurityProfile** field of a component:

```
$ oc explain <component>.spec.tlsSecurityProfile 1
```

- 1** For **<component>**, specify **ingresscontroller** or **apiserver**.

For example, to check all details for the **tlsSecurityProfile** field for the Ingress Controller:

```
$ oc explain ingresscontroller.spec.tlsSecurityProfile
```

Example output

```

KIND: IngressController
VERSION: operator.openshift.io/v1

RESOURCE: tlsSecurityProfile <Object>

DESCRIPTION:
  ...

FIELDS:
  custom <>
    custom is a user-defined TLS security profile. Be extremely careful using a
    custom profile as invalid configurations can be catastrophic. An example
    custom profile looks like this:
    ciphers: - ECDHE-ECDSA-CHACHA20-POLY1305 - ECDHE-RSA-CHACHA20-
    POLY1305 -
    ECDHE-RSA-AES128-GCM-SHA256 - ECDHE-ECDSA-AES128-GCM-SHA256
  minTLSVersion:
    TLSv1.1

  intermediate <>
    intermediate is a TLS security profile based on:
    https://wiki.mozilla.org/Security/Server_Side_TLS#Intermediate_compatibility_.28recommended_.29
    and looks like this (yaml):
    ... 1

  modern <>
    modern is a TLS security profile based on:
    https://wiki.mozilla.org/Security/Server_Side_TLS#Modern_compatibility and
    looks like this (yaml):
    ... 2
    NOTE: Currently unsupported.

  old <>
    old is a TLS security profile based on:
    https://wiki.mozilla.org/Security/Server_Side_TLS#Old_backward_compatibility
    and looks like this (yaml):
    ... 3

  type <string>
  ...

```

- 1** Lists ciphers and minimum version for the **intermediate** profile here.
- 2** Lists ciphers and minimum version for the **modern** profile here.
- 3** Lists ciphers and minimum version for the **old** profile here.

9.3. CONFIGURING THE TLS SECURITY PROFILE FOR THE INGRESS CONTROLLER

To configure a TLS security profile for an Ingress Controller, edit the **IngressController** custom resource (CR) to specify a predefined or custom TLS security profile. If a TLS security profile is not configured, the default value is based on the TLS security profile set for the API server.

Sample IngressController CR that configures the Old TLS security profile

```
apiVersion: operator.openshift.io/v1
kind: IngressController
...
spec:
  tlsSecurityProfile:
    old: {}
    type: Old
  ...
```

The TLS security profile defines the minimum TLS version and the TLS ciphers for TLS connections for Ingress Controllers.

You can see the ciphers and the minimum TLS version of the configured TLS security profile in the **IngressController** custom resource (CR) under **Status.Tls Profile** and the configured TLS security profile under **Spec.Tls Security Profile**. For the **Custom** TLS security profile, the specific ciphers and minimum TLS version are listed under both parameters.



IMPORTANT

The HAProxy Ingress Controller image does not support TLS **1.3** and because the **Modern** profile requires TLS **1.3**, it is not supported. The Ingress Operator converts the **Modern** profile to **Intermediate**. The Ingress Operator also converts the TLS **1.0** of an **Old** or **Custom** profile to **1.1**, and TLS **1.3** of a **Custom** profile to **1.2**.

Prerequisites

- You have access to the cluster as a user with the **cluster-admin** role.

Procedure

- Edit the **IngressController** CR in the **openshift-ingress-operator** project to configure the TLS security profile:

```
$ oc edit IngressController default -n openshift-ingress-operator
```

- Add the **spec.tlsSecurityProfile** field:

Sample IngressController CR for a Custom profile

```
apiVersion: operator.openshift.io/v1
kind: IngressController
...
spec:
  tlsSecurityProfile:
    type: Custom 1
    custom: 2
      ciphers: 3
        - ECDHE-ECDSA-CHACHA20-POLY1305
```

```
- ECDHE-RSA-CHACHA20-POLY1305
- ECDHE-RSA-AES128-GCM-SHA256
- ECDHE-ECDSA-AES128-GCM-SHA256
minTLSVersion: VersionTLS11
...
```

- 1 Specify the TLS security profile type (**Old**, **Intermediate**, or **Custom**). The default is **Intermediate**.
- 2 Specify the appropriate field for the selected type:
 - **old:** {}
 - **intermediate:** {}
 - **custom:**
- 3 For the **custom** type, specify a list of TLS ciphers and minimum accepted TLS version.

3. Save the file to apply the changes.

Verification

- Verify that the profile is set in the **IngressController** CR:

```
$ oc describe IngressController default -n openshift-ingress-operator
```

Example output

```
Name:      default
Namespace: openshift-ingress-operator
Labels:    <none>
Annotations: <none>
API Version: operator.openshift.io/v1
Kind:      IngressController
...
Spec:
...
Tls Security Profile:
  Custom:
    Ciphers:
      ECDHE-ECDSA-CHACHA20-POLY1305
      ECDHE-RSA-CHACHA20-POLY1305
      ECDHE-RSA-AES128-GCM-SHA256
      ECDHE-ECDSA-AES128-GCM-SHA256
    Min TLS Version: VersionTLS11
  Type:      Custom
...
```

9.4. CONFIGURING THE TLS SECURITY PROFILE FOR THE CONTROL PLANE

To configure a TLS security profile for the control plane, edit the **APIServer** custom resource (CR) to specify a predefined or custom TLS security profile. Setting the TLS security profile in the **APIServer** CR propagates the setting to the following control plane components:

- Kubernetes API server
- OpenShift API server
- OpenShift OAuth API server
- OpenShift OAuth server

If a TLS security profile is not configured, the default TLS security profile is **Intermediate**.



NOTE

The default TLS security profile for the Ingress Controller is based on the TLS security profile set for the API server.

Sample APIServer CR that configures the Old TLS security profile

```
apiVersion: config.openshift.io/v1
kind: APIServer
...
spec:
  tlsSecurityProfile:
    old: {}
    type: Old
  ...
```

The TLS security profile defines the minimum TLS version and the TLS ciphers required to communicate with the control plane components.

You can see the configured TLS security profile in the **APIServer** custom resource (CR) under **Spec.Tls Security Profile**. For the **Custom** TLS security profile, the specific ciphers and minimum TLS version are listed.



NOTE

The control plane does not support TLS **1.3** as the minimum TLS version; the **Modern** profile is not supported because it requires TLS **1.3**.

Prerequisites

- You have access to the cluster as a user with the **cluster-admin** role.

Procedure

1. Edit the default **APIServer** CR to configure the TLS security profile:

```
$ oc edit APIServer cluster
```

2. Add the **spec.tlsSecurityProfile** field:

Sample APIServer CR for a Custom profile

Sample APIServer CR for a Custom profile

```

apiVersion: config.openshift.io/v1
kind: APIServer
metadata:
  name: cluster
spec:
  tlsSecurityProfile:
    type: Custom 1
    custom: 2
      ciphers: 3
        - ECDHE-ECDSA-CHACHA20-POLY1305
        - ECDHE-RSA-CHACHA20-POLY1305
        - ECDHE-RSA-AES128-GCM-SHA256
        - ECDHE-ECDSA-AES128-GCM-SHA256
      minTLSVersion: VersionTLS11

```

- 1** Specify the TLS security profile type (**Old**, **Intermediate**, or **Custom**). The default is **Intermediate**.
- 2** Specify the appropriate field for the selected type:
 - **old:** {}
 - **intermediate:** {}
 - **custom:**
- 3** For the **custom** type, specify a list of TLS ciphers and minimum accepted TLS version.

3. Save the file to apply the changes.

Verification

- Verify that the TLS security profile is set in the **APIServer** CR:

```
$ oc describe apiserver cluster
```

Example output

```

Name:      cluster
Namespace:
...
API Version: config.openshift.io/v1
Kind:      APIServer
...
Spec:
  Audit:
    Profile: Default
  Tls Security Profile:
    Custom:
      Ciphers:
        ECDHE-ECDSA-CHACHA20-POLY1305
        ECDHE-RSA-CHACHA20-POLY1305

```

ECDHE-RSA-AES128-GCM-SHA256
ECDHE-ECDSA-AES128-GCM-SHA256
Min TLS Version: VersionTLS11
Type: Custom
...

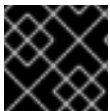
CHAPTER 10. CONFIGURING SECCOMP PROFILES

An OpenShift Container Platform container or a pod runs a single application that performs one or more well-defined tasks. The application usually requires only a small subset of the underlying operating system kernel APIs. Seccomp, secure computing mode, is a Linux kernel feature that can be used to limit the process running in a container to only call a subset of the available system calls. These system calls can be configured by creating a profile that is applied to a container or pod. Seccomp profiles are stored as JSON files on the disk.



IMPORTANT

OpenShift workloads run unconfined by default, without any seccomp profile applied.



IMPORTANT

Seccomp profiles cannot be applied to privileged containers.

10.1. ENABLING THE DEFAULT SECCOMP PROFILE FOR ALL PODS

OpenShift Container Platform ships with a default seccomp profile that is referenced as **runtime/default**. You can enable the default seccomp profile for a pod or container workload by creating a custom Security Context Constraint (SCC).



NOTE

There is a requirement to create a custom SCC. Do not edit the default SCCs. Editing the default SCCs can lead to issues when some of the platform pods deploy or OpenShift Container Platform is upgraded. For more information, see the section entitled "Default security context constraints".

Follow these steps to enable the default seccomp profile for all pods:

1. Export the available **restricted** SCC to a yaml file:

```
$ oc get scc restricted -o yaml > restricted-seccomp.yaml
```

2. Edit the created **restricted** SCC yaml file:

```
$ vi restricted-seccomp.yaml
```

3. Update as shown in this example:

```
kind: SecurityContextConstraints
metadata:
  name: restricted ①
<..snip..>
seccompProfiles: ②
- runtime/default ③
```

① Change to **restricted-seccomp**

② Add **seccompProfiles:**

3 Add - runtime/default

4. Create the custom SCC:

```
$ oc create -f restricted-seccomp.yaml
```

Expected output

```
securitycontextconstraints.security.openshift.io/restricted-seccomp created
```

5. Add the custom SCC to the ServiceAccount:

```
$ oc adm policy add-scc-to-user restricted-seccomp -z default
```



NOTE

The default service account is the ServiceAccount that is applied unless the user configures a different one. OpenShift Container Platform configures the seccomp profile of the pod based on the information in the SCC.

Expected output

```
clusterrole.rbac.authorization.k8s.io/system:openshift:scc:restricted-seccomp added:
"default"
```

In OpenShift Container Platform 4.6 the ability to add the pod annotations **seccomp.security.alpha.kubernetes.io/pod: runtime/default** and **container.seccomp.security.alpha.kubernetes.io/<container_name>: runtime/default** is deprecated.

10.2. CONFIGURING A CUSTOM SECCOMP PROFILE

You can configure a custom seccomp profile, which allows you to update the filters based on the application requirements. This allows cluster administrators to have greater control over the security of workloads running in OpenShift Container Platform.

10.2.1. Setting up the custom seccomp profile

Prerequisite

- You have cluster administrator permissions.
- You have created a custom security context constraints (SCC). For more information, see "Additional resources".
- You have created a custom seccomp profile.

Procedure

1. Upload your custom seccomp profile to **/var/lib/kubelet/seccomp/<custom-name>.json** by using the Machine Config. See "Additional resources" for detailed steps.

- Update the custom SCC by providing reference to the created custom seccomp profile:

```
seccompProfiles:
- localhost/<custom-name>.json 1
```

- Provide the name of your custom seccomp profile.

10.2.2. Applying the custom seccomp profile to the workload

Prerequisite

- The cluster administrator has set up the custom seccomp profile. For more details, see "Setting up the custom seccomp profile".

Procedure

- Apply the seccomp profile to the workload by setting the **securityContext.seccompProfile.type** field as following:

Example

```
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: <custom-name>.json 1
```

- Provide the name of your custom seccomp profile.

Alternatively, you can use the pod annotations **seccomp.security.alpha.kubernetes.io/pod: localhost/<custom-name>.json**. However, this method is deprecated in OpenShift Container Platform 4.6.

During deployment, the admission controller validates the following:

- The annotations against the current SCCs allowed by the user role.
- The SCC, which includes the seccomp profile, is allowed for the pod.

If the SCC is allowed for the pod, the kubelet runs the pod with the specified seccomp profile.



IMPORTANT

Ensure that the seccomp profile is deployed to all worker nodes.



NOTE

The custom SCC must have the appropriate priority to be automatically assigned to the pod or meet other conditions required by the pod, such as allowing CAP_NET_ADMIN.

10.3. ADDITIONAL RESOURCES

- [Managing security context constraints](#)
- [Post-installation machine configuration tasks](#)

CHAPTER 11. ALLOWING JAVASCRIPT-BASED ACCESS TO THE API SERVER FROM ADDITIONAL HOSTS

11.1. ALLOWING JAVASCRIPT-BASED ACCESS TO THE API SERVER FROM ADDITIONAL HOSTS

The default OpenShift Container Platform configuration only allows the OpenShift web console to send requests to the API server.

If you need to access the API server or OAuth server from a JavaScript application using a different hostname, you can configure additional hostnames to allow.

Prerequisites

- Access to the cluster as a user with the **cluster-admin** role.

Procedure

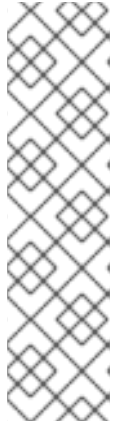
1. Edit the **APIServer** resource:

```
$ oc edit apiserver.config.openshift.io cluster
```

2. Add the **additionalCORSAAllowedOrigins** field under the **spec** section and specify one or more additional hostnames:

```
apiVersion: config.openshift.io/v1
kind: APIServer
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: "2019-07-11T17:35:37Z"
  generation: 1
  name: cluster
  resourceVersion: "907"
  selfLink: /apis/config.openshift.io/v1/apiservers/cluster
  uid: 4b45a8dd-a402-11e9-91ec-0219944e0696
spec:
  additionalCORSAAllowedOrigins:
    - (?i)//my\.subdomain\.domain\.com(:|z) 1
```

- 1 The hostname is specified as a [Golang regular expression](#) that matches against CORS headers from HTTP requests against the API server and OAuth server.

**NOTE**

This example uses the following syntax:

- The **(?i)** makes it case-insensitive.
- The **//** pins to the beginning of the domain and matches the double slash following **http:** or **https:**.
- The **\.** escapes dots in the domain name.
- The **(:|z)** matches the end of the domain name (**z**) or a port separator (**:**).

3. Save the file to apply the changes.

CHAPTER 12. ENCRYPTING ETCD DATA

12.1. ABOUT ETCD ENCRYPTION

By default, etcd data is not encrypted in OpenShift Container Platform. You can enable etcd encryption for your cluster to provide an additional layer of data security. For example, it can help protect the loss of sensitive data if an etcd backup is exposed to the incorrect parties.

When you enable etcd encryption, the following OpenShift API server and Kubernetes API server resources are encrypted:

- Secrets
- Config maps
- Routes
- OAuth access tokens
- OAuth authorize tokens

When you enable etcd encryption, encryption keys are created. These keys are rotated on a weekly basis. You must have these keys in order to restore from an etcd backup.



NOTE

Keep in mind that etcd encryption only encrypts values, not keys. This means that resource types, namespaces, and object names are unencrypted.

12.2. ENABLING ETCD ENCRYPTION

You can enable etcd encryption to encrypt sensitive resources in your cluster.



WARNING

It is not recommended to take a backup of etcd until the initial encryption process is complete. If the encryption process has not completed, the backup might be only partially encrypted.

Prerequisites

- Access to the cluster as a user with the **cluster-admin** role.

Procedure

1. Modify the **APIServer** object:

```
$ oc edit apiserver
```

- Set the **encryption** field type to **aescbc**:

```
spec:
  encryption:
    type: aescbc 1
```

- The **aescbc** type means that AES-CBC with PKCS#7 padding and a 32 byte key is used to perform the encryption.

- Save the file to apply the changes.

The encryption process starts. It can take 20 minutes or longer for this process to complete, depending on the size of your cluster.

- Verify that etcd encryption was successful.

- Review the **Encrypted** status condition for the OpenShift API server to verify that its resources were successfully encrypted:

```
$ oc get openshiftapiserver -o=jsonpath='{range .items[0].status.conditions[?(@.type=="Encrypted")]}{.reason}{"\n"}{.message}{"\n"}'
```

The output shows **EncryptionCompleted** upon successful encryption:

```
EncryptionCompleted
All resources encrypted: routes.route.openshift.io, oauthtokens.oauth.openshift.io,
oauthtokens.oauth.openshift.io
```

If the output shows **EncryptionInProgress**, this means that encryption is still in progress. Wait a few minutes and try again.

- Review the **Encrypted** status condition for the Kubernetes API server to verify that its resources were successfully encrypted:

```
$ oc get kubeapiserver -o=jsonpath='{range .items[0].status.conditions[?(@.type=="Encrypted")]}{.reason}{"\n"}{.message}{"\n"}'
```

The output shows **EncryptionCompleted** upon successful encryption:

```
EncryptionCompleted
All resources encrypted: secrets, configmaps
```

If the output shows **EncryptionInProgress**, this means that encryption is still in progress. Wait a few minutes and try again.

12.3. DISABLING ETCD ENCRYPTION

You can disable encryption of etcd data in your cluster.

Prerequisites

- Access to the cluster as a user with the **cluster-admin** role.

Procedure

1. Modify the **APIServer** object:

```
$ oc edit apiserver
```

2. Set the **encryption** field type to **identity**:

```
spec:
  encryption:
    type: identity 1
```

- 1 The **identity** type is the default value and means that no encryption is performed.

3. Save the file to apply the changes.

The decryption process starts. It can take 20 minutes or longer for this process to complete, depending on the size of your cluster.

4. Verify that etcd decryption was successful.

- a. Review the **Encrypted** status condition for the OpenShift API server to verify that its resources were successfully decrypted:

```
$ oc get openshiftapiserver -o=jsonpath='{range .items[0].status.conditions[?(@.type=="Encrypted")]}{.reason}{"\n"}{.message}{"\n"}'
```

The output shows **DecryptionCompleted** upon successful decryption:

```
DecryptionCompleted
Encryption mode set to identity and everything is decrypted
```

If the output shows **DecryptionInProgress**, this means that decryption is still in progress. Wait a few minutes and try again.

- b. Review the **Encrypted** status condition for the Kubernetes API server to verify that its resources were successfully decrypted:

```
$ oc get kubeapiserver -o=jsonpath='{range .items[0].status.conditions[?(@.type=="Encrypted")]}{.reason}{"\n"}{.message}{"\n"}'
```

The output shows **DecryptionCompleted** upon successful decryption:

```
DecryptionCompleted
Encryption mode set to identity and everything is decrypted
```

If the output shows **DecryptionInProgress**, this means that decryption is still in progress. Wait a few minutes and try again.

CHAPTER 13. SCANNING PODS FOR VULNERABILITIES

Using the Red Hat Quay Container Security Operator, you can access vulnerability scan results from the OpenShift Container Platform web console for container images used in active pods on the cluster. The Red Hat Quay Container Security Operator:

- Watches containers associated with pods on all or specified namespaces
- Queries the container registry where the containers came from for vulnerability information, provided an image's registry is running image scanning (such as [Quay.io](#) or a [Red Hat Quay](#) registry with Clair scanning)
- Exposes vulnerabilities via the **ImageManifestVuln** object in the Kubernetes API

Using the instructions here, the Red Hat Quay Container Security Operator is installed in the **openshift-operators** namespace, so it is available to all namespaces on your OpenShift cluster.

13.1. RUNNING THE RED HAT QUAY CONTAINER SECURITY OPERATOR

You can start the Red Hat Quay Container Security Operator from the OpenShift Container Platform web console by selecting and installing that Operator from the Operator Hub, as described here.

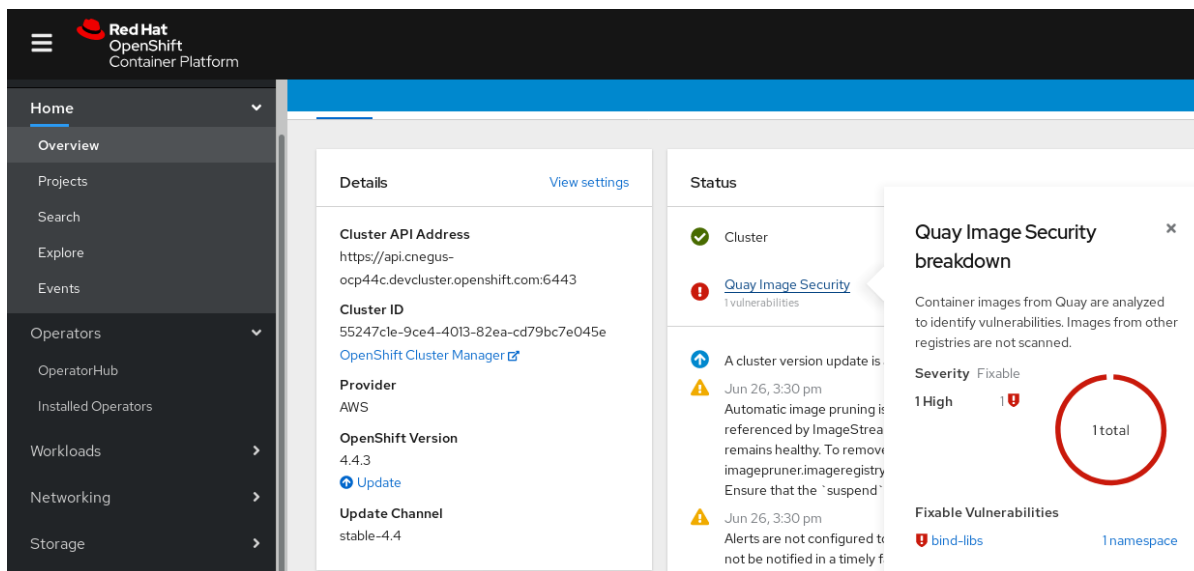
Prerequisites

- Have administrator privileges to the OpenShift Container Platform cluster
- Have containers that come from a Red Hat Quay or Quay.io registry running on your cluster

Procedure

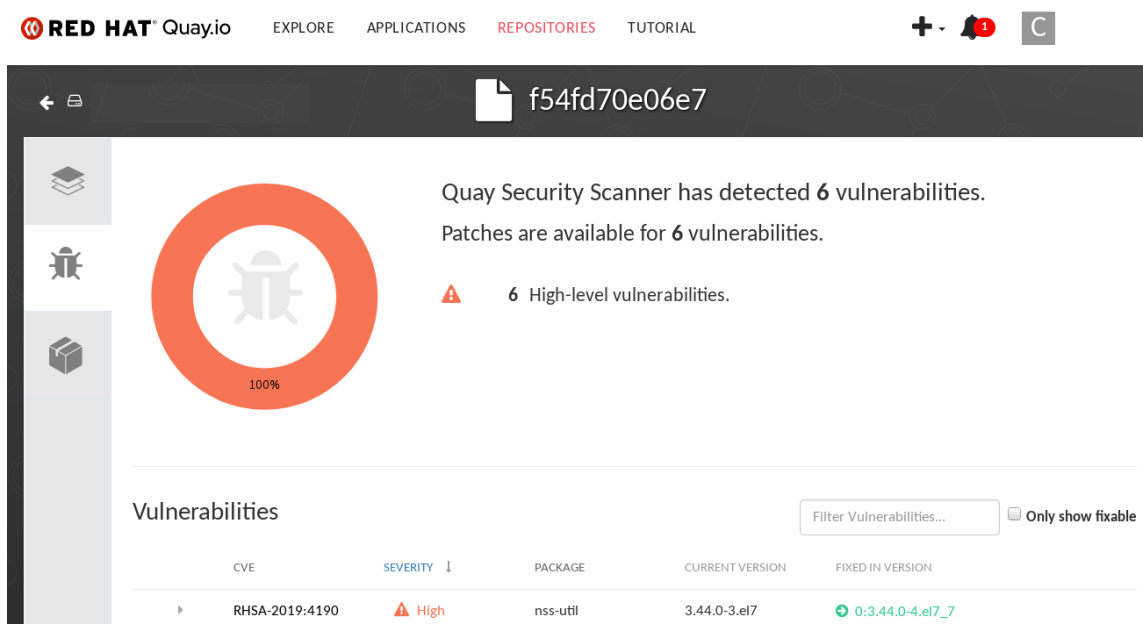
1. Navigate to **Operators** → **OperatorHub** and select **Security**.
2. Select the **Container Security** Operator, then select **Install** to go to the Create Operator Subscription page.
3. Check the settings. All namespaces and automatic approval strategy are selected, by default.
4. Select **Install**. The **Container Security** Operator appears after a few moments on the **Installed Operators** screen.
5. Optionally, you can add custom certificates to the Red Hat Quay Container Security Operator. In this example, create a certificate named **quay.crt** in the current directory. Then run the following command to add the cert to the Red Hat Quay Container Security Operator:

```
$ oc create secret generic container-security-operator-extra-certs --from-file=quay.crt -n openshift-operators
```
6. If you added a custom certificate, restart the Operator pod for the new certs to take effect.
7. Open the OpenShift Dashboard (**Home** → **Overview**). A link to **Quay Image Security** appears under the status section, with a listing of the number of vulnerabilities found so far. Select the link to see a **Quay Image Security breakdown**, as shown in the following figure:

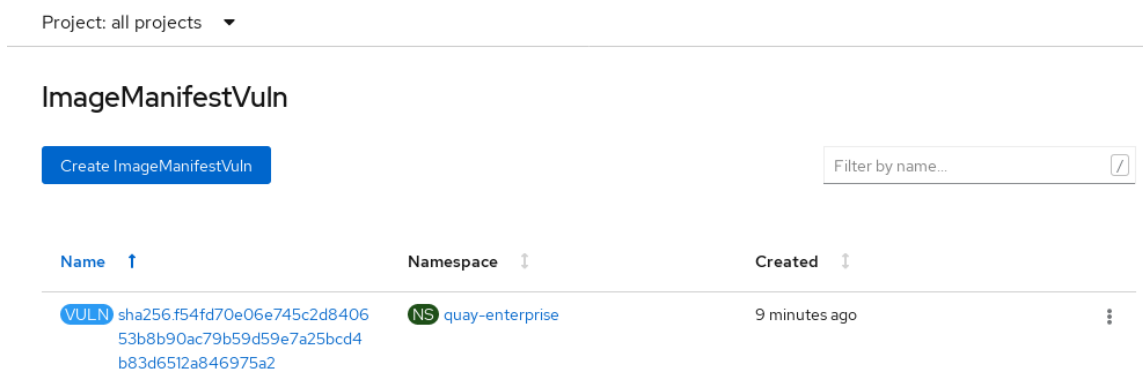


8. You can do one of two things at this point to follow up on any detected vulnerabilities:

- Select the link to the vulnerability. You are taken to the container registry that the container came from, where you can see information about the vulnerability. The following figure shows an example of detected vulnerabilities from a Quay.io registry:



- Select the namespaces link to go to the **ImageManifestVuln** screen, where you can see the name of the selected image and all namespaces where that image is running. The following figure indicates that a particular vulnerable image is running in the **quay-enterprise** namespace:



At this point, you know what images are vulnerable, what you need to do to fix those vulnerabilities, and every namespace that the image was run in. So you can:

- Alert anyone running the image that they need to correct the vulnerability
- Stop the images from running by deleting the deployment or other object that started the pod that the image is in

Note that if you do delete the pod, it may take several minutes for the vulnerability to reset on the dashboard.

13.2. QUERYING IMAGE VULNERABILITIES FROM THE CLI

Using the **oc** command, you can display information about vulnerabilities detected by the Red Hat Quay Container Security Operator.

Prerequisites

- Be running the Red Hat Quay Container Security Operator on your OpenShift Container Platform instance

Procedure

- To query for detected container image vulnerabilities, type:

```
$ oc get vuln --all-namespaces
```

Example output

```
NAMESPACE  NAME                AGE
default    sha256.ca90...     6m56s
skynet     sha256.ca90...     9m37s
```

- To display details for a particular vulnerability, provide the vulnerability name and its namespace to the **oc describe** command. This example shows an active container whose image includes an RPM package with a vulnerability:

```
$ oc describe vuln --namespace mynamespace sha256.ac50e3752...
```

Example output

```
Name:      sha256.ac50e3752...
Namespace: quay-enterprise
...
Spec:
Features:
  Name:      nss-util
  Namespace Name: centos:7
  Version:   3.44.0-3.el7
  Versionformat: rpm
Vulnerabilities:
  Description: Network Security Services (NSS) is a set of libraries...
```

