OpenShift Container Platform 4.5

Security

Learning about and managing security for OpenShift Container Platform

Last Updated: 2021-06-09

OpenShift Container Platform 4.5 Security

Learning about and managing security for OpenShift Container Platform

Legal Notice

Abstract

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

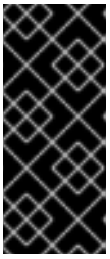
Table of Contents

CHAPTER 1. CONTAINER SECURITY

1.1. 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.

1.1.1. 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.

1.1.2. 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

1.2. 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.

1.2.1. 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 4.5 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.

- *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

1.2.2. 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.

1.2.3. 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.

1.3. 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.

1.3.1. 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.

1.3.2. 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 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.

1.3.2.1. 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 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.

1.3.2.2. 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.

1.3.2.3. 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

1.4. 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.

1.4.1. 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
```

1.4.2. 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-d9b3f4ffcfd65c30dcf591a0e8cf9b2e
    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
```

1.5. 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.

1.5.1. 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.

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

1.6. 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.

1.6.1. 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 Container Security Operator can be added to check container images used in OpenShift Container Platform.

1.6.2. 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-mimimal**). 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.

1.6.3. 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.

1.6.3.1. 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 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.

1.6.4. 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.

1.6.4.1. 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 1.1. Annotation key format

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

1.6.4.1.1. 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 1.2. 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

Field	Required?	Description	Type
summary	No	Summary of issues found	List (see table below)

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

Table 1.3. 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

1.6.4.1.2. 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 }
  ]
}
```

1.6.4.2. 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.

1.6.4.2.1. 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 } ]" }'
```

1.6.4.3. Controlling pod execution

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

1.6.4.3.1. Example annotation

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

1.6.4.4. 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 4.5 REST APIs to write the annotation. See OpenShift Container Platform REST APIs for general information on the REST APIs.

1.6.4.4.1. 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>:8443/oapi/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

1.7. 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.

1.7.1. 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.

1.7.2. 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.

1.7.3. 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.

1.7.4. 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

1.7.5. 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 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 Container Security Operator lets you check vulnerabilities of images running in OpenShift Container Platform that were pulled from Red Hat Quay registries.

1.8. SECURING THE BUILD PROCESS

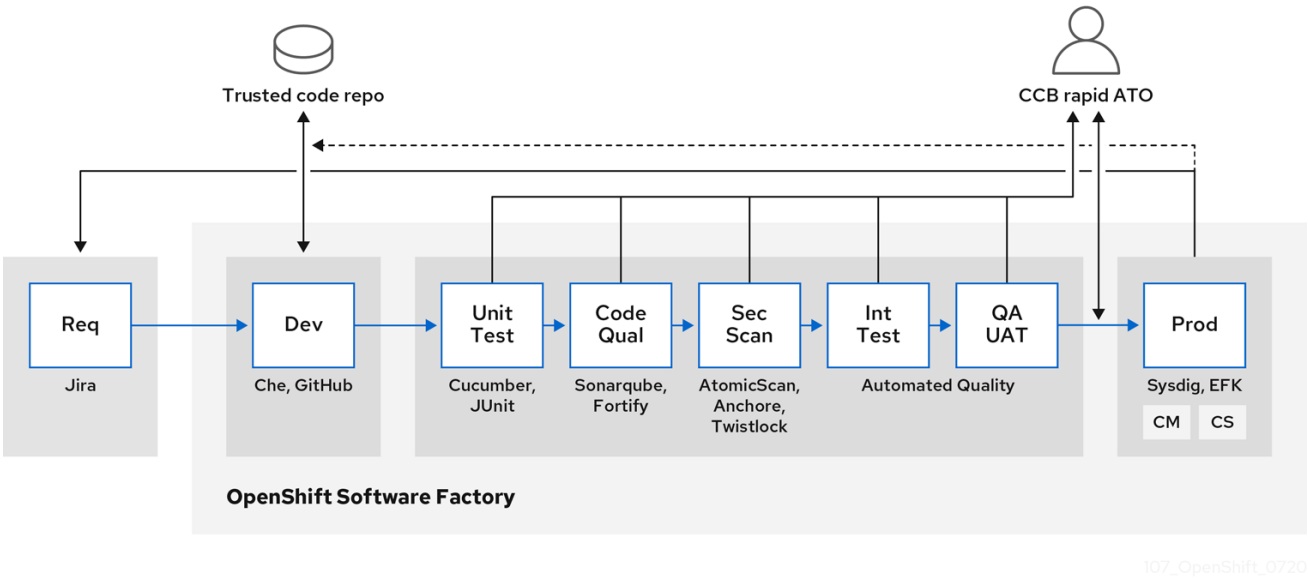
In a container environment, the software build process is the stage in the life cycle where application code is integrated with the required runtime libraries. Managing this build process is key to securing the software stack.

1.8.1. 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:



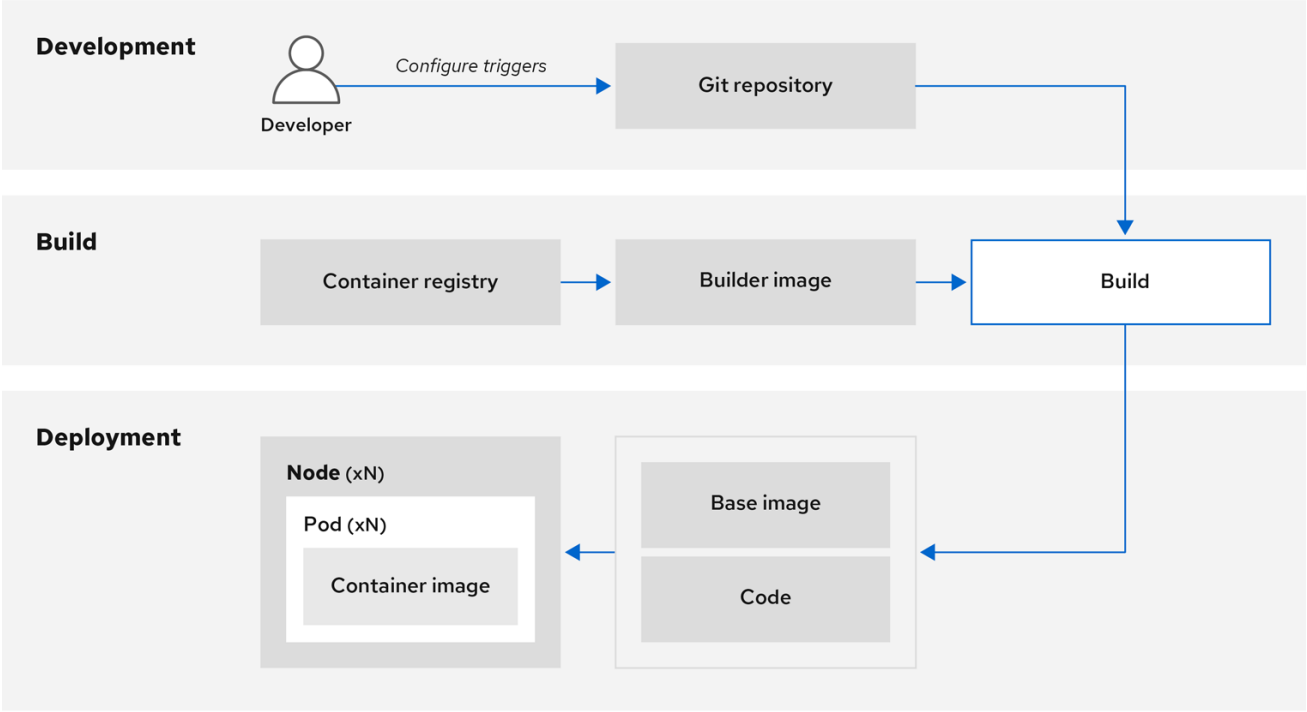
OpenShift Container Platform can be integrated with trusted code repositories (such as GitHub) and development platforms (such as Che) for creating and managing secure code. Unit testing could rely on Cucumber and JUnit. You could inspect your containers for vulnerabilities and compliance issues with 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.

1.8.2. 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.



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.

1.8.3. 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:
```

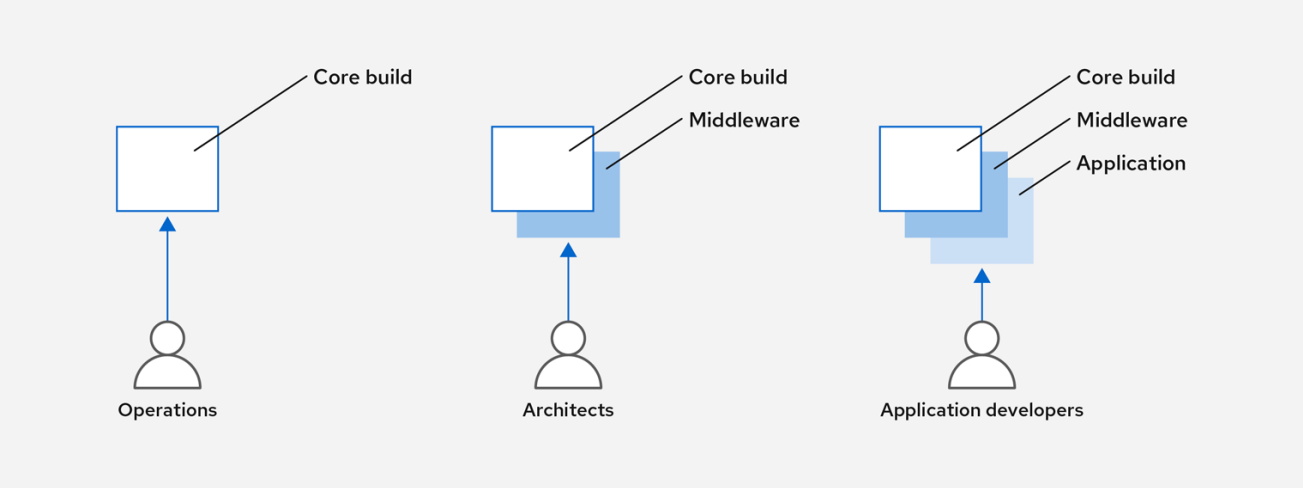
```
- destinationDir: .
  secret:
    name: secret-npmrc
```

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
```

1.8.4. Designing your build process

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



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.

1.8.5. Building Knative serverless applications

Relying on Kubernetes and Kourier, you can build, deploy and manage serverless applications using Knative 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.

Additional resources

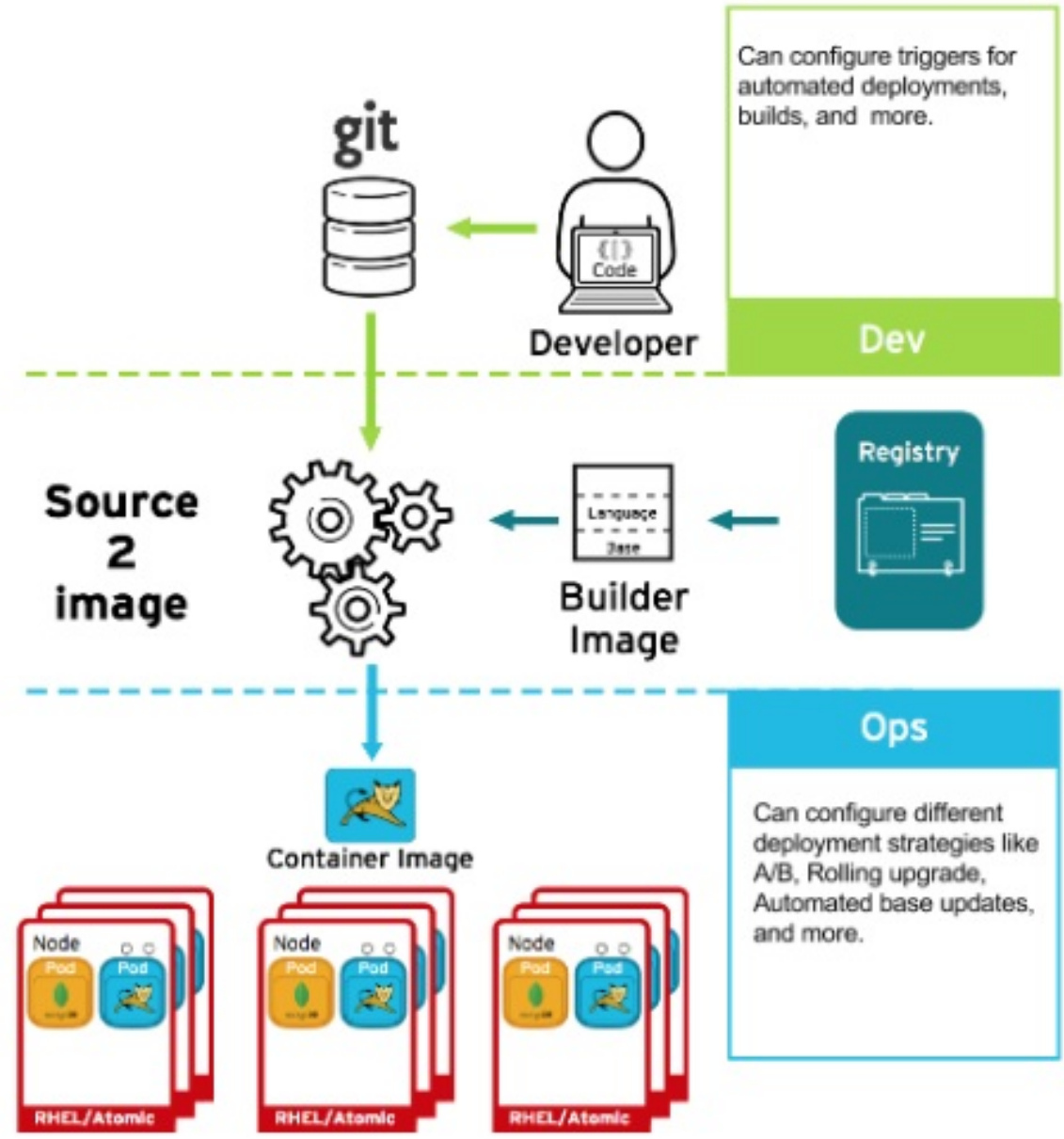
- Understanding image builds
- Triggering and modifying builds
- Creating build inputs
- Input secrets and config maps
- The CI/CD methodology and practice
- Knative Serving architecture
- Viewing application composition using the Topology view

1.9. 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.

1.9.1. 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
```

1.9.2. 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.

1.9.3. 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.

1.9.4. 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.

1.9.5. 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

1.10. 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.

1.10.1. 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.

1.10.2. 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.

1.10.2.1. 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.

1.10.2.2. 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.

1.10.3. Authentication and authorization

1.10.3.1. 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.

1.10.3.2. 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.

1.10.3.3. 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.

1.10.3.4. 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

1.10.4. 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

1.10.4.1. Configuring custom certificates

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

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

1.11. 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.

1.11.1. 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.

1.11.2. 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

1.11.3. 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

1.11.4. 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

1.11.5. 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

1.11.6. 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

1.12. 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.

1.12.1. 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**.

1.12.2. 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.

1.12.3. 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

1.13. 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.

1.13.1. 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.

1.13.2. 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 Elasticsearch Operator and Cluster Logging Operator.

1.13.3. 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 2. CONFIGURING CERTIFICATES

2.1. REPLACING THE DEFAULT INGRESS CERTIFICATE

2.1.1. 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.

2.1.2. 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.

2.2. 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.

2.2.1. 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. 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.

2. 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.

3. 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>
...
```

2.3. SECURING SERVICE TRAFFIC USING SERVICE SERVING CERTIFICATE SECRETS

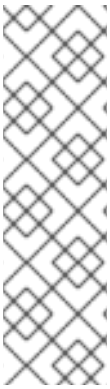
2.3.1. Understanding service serving certificates

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.



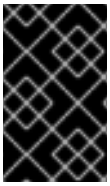
NOTE

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
```

2.3.2. 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>
```

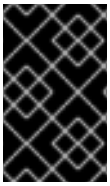
Example output

```
...
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.

2.3.3. 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

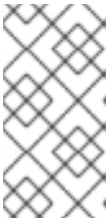
After adding this annotation to a config map all existing data in it is deleted. It is recommended to use a separate config map to contain the **service-ca.crt**, instead of using the same config map that stores your pod configuration.

Procedure

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

```
$ oc annotate configmap <config_map_name> \ (1)
    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-----
...
```

2.3.4. 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> \ (1)
    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
```

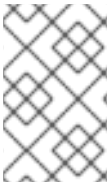
The CA bundle is displayed in the **spec.caBundle** field in the YAML output:

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
```

```
  annotations:
    service.beta.openshift.io/inject-cabundle: "true"
...
spec:
  caBundle: <CA_BUNDLE>
...
```

2.3.5. 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:

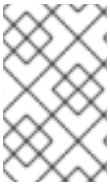
```
$ oc get crd <crd_name> -o yaml
```

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>
...
```

2.3.6. 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:

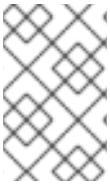
```
$ oc get mutatingwebhookconfigurations <mutating_webhook_name> -o yaml
```

   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>
...
```

2.3.7. 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> \ 1
    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>
...
```

2.3.8. 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 **serving-cert-secret-name** annotation, as seen below.

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

Example output

```
...
service.beta.openshift.io/serving-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
```

2.3.9. 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
```

3. 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.

CHAPTER 3. CERTIFICATE TYPES AND DESCRIPTIONS

3.1. USER-PROVIDED CERTIFICATES FOR THE API SERVER

3.1.1. 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 host name 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.

3.1.2. 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.

3.1.3. Management

User-provided certificates are managed by the user.

3.1.4. Expiration

API server client certificate expiration is less than five minutes.

User-provided certificates are managed by the user.

3.1.5. Customization

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

Additional resources

- Adding API server certificates

3.2. PROXY CERTIFICATES

3.2.1. 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

3.2.2. 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-----
...
```

3.2.3. 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.

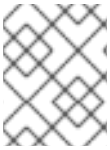
If using the RHCOS trust bundle, place CA certificates in **/etc/pki/ca-trust/source/anchors**.

See Using shared system certificates in the Red Hat Enterprise Linux documentation for more information.

3.2.4. 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.

3.2.5. 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.

3.2.6. Management

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

3.2.7. 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:

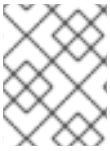
```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 50-examplecorp-ca-cert
spec:
  config:
    ignition:
      version: 2.2.0
    storage:
      files:
      - contents:
          source: data:text/plain;charset=utf-
8;base64,LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUVORENDQXh5Z0F3SUJBZ0lKQU5
1bkkwRDY2MmNuTUEwR0NTcUdTSWIzRFFFQkN3VUFNSUdsTVFzd0NRWUQKV1FRR0V3SlZVek
VYTUJVR0ExVUVDQXdPVG05eWRHZ2dRMkZ5YjJ4cGJtRXhFREFPQmdOVkJBY01CMUpoYkdWcA
```

```
pBMmd4RmpBVUJnTlZCQW9NRFZKbFpDQklZWFFzSUVsdVl5NHhFekFSQmdOVkJBc01DbEpsWk
NCSVlYUWdTVlF4Ckh6QVpCZ05WQkFNTUVsSmxaQ0JJWVhRZ1NWUWdVbTl2ZENCRFFURWhN
QjhHQ1NxR1NJYjNEUUVKQVJZU2FXNW0KWGpDQnBURUxNQWtHQTFVRUJoTUNWVk14RnpBV
kJnTlZCQWdNRGs1dmNuUm9JRU5oY205c2FXNWhNUkF3RGdZRApXUVFIREFkU1lXeGxhV2RvTV
JZd0ZBWURWUVFLREExU1pXUWdTR0YwTENCSmJtTXVNUk13RVFZRFZRUUxEQXBTCkFXUWd
TR0YwSUVsVU1Sc3dHUVlEVlFRRERCSlNaV1FnU0dGMElFbFVJRkp2YjNRZ1EwRXhJVEFmQmdrc
WhraUcKMHcwQkNRRVdFbWx1Wm05elpXTkFjbVZrYUdGMExtTnZiVENDQVNJd0RRWUpLb1pJaH
ZjTkFRRUJCUUFEZ2dFUApCRENDQVFvQ2dnRUJBTFF0OU9KUWg2R0M1TFQxZzgwcU5oMHU1
MEJRNHNaL3laOGFFVHh0KzVsblBWWDZNSEt6CmQvaTdsRHFUZlRjZkxMMm55VUJkMmZRRGsx
QjBmeHJza2hHSUlaM2lmUDFQczRsdFRrdjhoUlNvYjNWdE5xU28KSHhrS2Z2RDJQS2pUUHhEUFdZ
eXJ1eTlpckxaaW9NZmZpM2kvZ0N1dDBaV3RBeU8zTVZINXFXRi9lbkt3Z1BFUwpZOXBvK1RkQ3ZS
Qi9SVU9iQmFNNzYxRWNyTFNNMUdxSE51ZVNmcW5obzNBakxRNmRCblBXbG82MzhabTFWZWJ
LCkNFTHloa0xXTVNGa0t3RG1uZTBqUTAyWTRnMDc1dkNLdkNzQ0F3RUFBYU5qTUdFd0hRWUR
WUjBPQkJZRUZIN1IKNXlDK1VlaElJUGV1TDhacXczUHpiZ2NaTUI4R0ExVWRJd1FZTUJhQUZIN1I0
eUMrVWVoSUlQZXVMOFpxdzNQegpjZ2NaTUE4R0ExVWRFd0VCL3dRRk1BTUJBZjh3RGdZRFZS
MFBBUUgvQkFRREFnR0dNQTBHQ1NxR1NJYjNEUUVCCkR3VUFBNElCQVFCRE52RDJWbTlzQT
VBOUFsT0pSOCtlbjVYejloWGN4SkI1cGh4Y1pROGpGb0cwNFZzaHZkMGUKTUVuVXJNY2ZGZ0laN
G5qTUtUUUNNNFpGVVBBaWV5THg0ZjUySHVEb3BwM2U1SnlJTWZXK0tGY05JcEt3Q3NhawpwU2
9LdElVT3NVSks3cUJWWnhjckl5ZVFWMnFjWU9lWmh0UzV3QnFJd09BaEZ3bENFVDdaZTU4UUhtUz
Q4c2xqCjVlVGtSaml2QWxFeHJGektjbGpDNGF4S1Fsbk92VkF6eitHbTMyVTB4UEJGNEJ5ZVBWeEN
KVUh3MVRzeVRtZWwKU3hORXA3eUhvWGN3bitmWG5hK3Q1SldoMWd4VVp0eTMKLS0tLS1FTkQ
gQ0VSVElGSUNBVEUtLS0tLQo=
      filesystem: root
      mode: 0644
      path: /etc/pki/ca-trust/source/anchors/examplecorp-ca.crt
```

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

3.2.8. 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.

3.3. SERVICE CA CERTIFICATES

3.3.1. Purpose

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

3.3.2. 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.

3.3.3. Management

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

3.3.4. 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

3.4. NODE CERTIFICATES

3.4.1. 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.

3.4.2. Management

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

Additional resources

- Working with nodes

3.5. BOOTSTRAP CERTIFICATES

3.5.1. 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.

3.5.2. Management

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

3.5.3. 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.

3.5.4. Customization

You cannot customize the bootstrap certificates.

3.6. ETCD CERTIFICATES

3.6.1. Purpose

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

3.6.2. Expiration

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

3.6.3. Management

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

3.6.4. 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

- Recovering from lost master hosts

3.7. OLM CERTIFICATES

3.7.1. 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.

3.8. USER-PROVIDED CERTIFICATES FOR DEFAULT INGRESS

3.8.1. 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.

3.8.2. 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.

3.8.3. Management

User-provided certificates are managed by the user.

3.8.4. Expiration

User-provided certificates are managed by the user.

3.8.5. Services

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

3.8.6. Customization

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

Additional resources

- Replacing the default ingress certificate

3.9. INGRESS CERTIFICATES

3.9.1. Purpose

The Ingress Operator uses certificates for:

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

3.9.2. 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.

3.9.3. Workflow

Figure 3.1. Custom certificate workflow

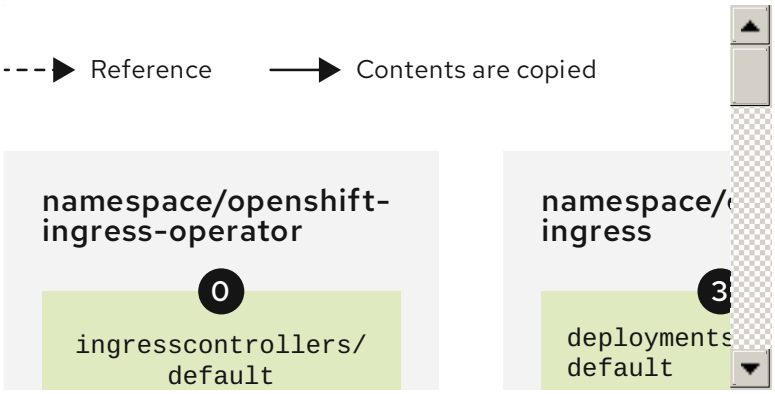
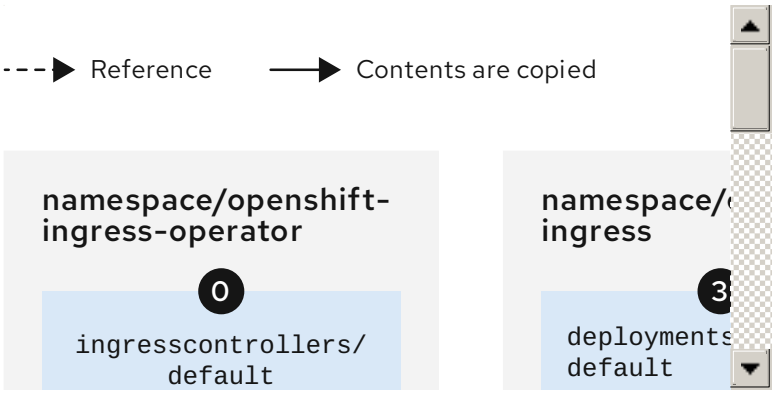


Figure 3.2. Default certificate workflow



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 4.5 and newer use **default-ingress-cert**.

3.9.4. 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.

3.9.5. 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.

3.9.6. Management

Ingress certificates are managed by the user. See Replacing the default ingress certificate for more information.

3.9.7. 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.

3.10. MONITORING AND CLUSTER LOGGING OPERATOR COMPONENT CERTIFICATES

3.10.1. 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.

3.10.2. Management

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

3.11. CONTROL PLANE CERTIFICATES

3.11.1. 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

3.11.2. Management

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

In the rare case that your control plane certificates expired, see Recovering from expired control plane certificates

CHAPTER 4. VIEWING AUDIT LOGS

Audit 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.

4.1. 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 4.1. 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\""}}
```


4.2. VIEWING THE AUDIT LOG

You can view logs for the OpenShift Container Platform API server or the Kubernetes API server for each master node.

Procedure

To view the audit log:

1. View the OpenShift Container Platform API server logs

   a. If necessary, get the node name of the log you want to view:

   ```
   $ oc adm node-logs --role=master --path=openshift-apiserver/
   ```

   Example output

   ```
   ip-10-0-140-97.ec2.internal audit-2019-04-09T00-12-19.834.log
   ip-10-0-140-97.ec2.internal audit-2019-04-09T11-13-00.469.log
   ip-10-0-140-97.ec2.internal audit.log
   ip-10-0-153-35.ec2.internal audit-2019-04-09T00-11-49.835.log
   ip-10-0-153-35.ec2.internal audit-2019-04-09T11-08-30.469.log
   ip-10-0-153-35.ec2.internal audit.log
   ip-10-0-170-165.ec2.internal audit-2019-04-09T00-13-00.128.log
   ip-10-0-170-165.ec2.internal audit-2019-04-09T11-10-04.082.log
   ip-10-0-170-165.ec2.internal audit.log
   ```

   b. View the OpenShift Container Platform API server log for a specific master node and timestamp or view all the logs for that master:

   ```
   $ oc adm node-logs <node-name> --path=openshift-apiserver/<log-name>
   ```

   For example:

   ```
   $ oc adm node-logs ip-10-0-140-97.ec2.internal --path=openshift-apiserver/audit-2019-04-08T13-09-01.227.log
   ```

   ```
   $ oc adm node-logs ip-10-0-140-97.ec2.internal --path=openshift-apiserver/audit.log
   ```

   The output appears similar to the following:

   Example output

   ```
   {"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\""}}
```

2. View the Kubernetes API server logs:

   a. If necessary, get the node name of the log you want to view:

   ```
   $ oc adm node-logs --role=master --path=kube-apiserver/
   ```

   Example output

   ```
   ip-10-0-140-97.ec2.internal audit-2019-04-09T14-07-27.129.log
   ip-10-0-140-97.ec2.internal audit-2019-04-09T19-18-32.542.log
   ip-10-0-140-97.ec2.internal audit.log
   ip-10-0-153-35.ec2.internal audit-2019-04-09T19-24-22.620.log
   ip-10-0-153-35.ec2.internal audit-2019-04-09T19-51-30.905.log
   ip-10-0-153-35.ec2.internal audit.log
   ip-10-0-170-165.ec2.internal audit-2019-04-09T18-37-07.511.log
   ip-10-0-170-165.ec2.internal audit-2019-04-09T19-21-14.371.log
   ip-10-0-170-165.ec2.internal audit.log
   ```

   b. View the Kubernetes API server log for a specific master node and timestamp or view all the logs for that master:

   ```
   $ oc adm node-logs <node-name> --path=kube-apiserver/<log-name>
   ```

   For example:

   ```
   $ oc adm node-logs ip-10-0-140-97.ec2.internal --path=kube-apiserver/audit-2019-04-
   09T14-07-27.129.log
   ```

   ```
   $ oc adm node-logs ip-10-0-170-165.ec2.internal --path=kube-apiserver/audit.log
   ```

   The output appears similar to the following:

   Example output

   ```
   {"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\""}}
```

CHAPTER 5. ALLOWING JAVASCRIPT-BASED ACCESS TO THE API SERVER FROM ADDITIONAL HOSTS

5.1. 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 host name, you can configure additional host names 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 **additionalCORSAllowedOrigins** field under the **spec** section and specify one or more additional host names:

```
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:
  additionalCORSAllowedOrigins:
  - (?i)//my\.subdomain\.domain\.com(:|\z) 1
```

1 The host name 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 6. ENCRYPTING ETCD DATA

6.1. 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.

6.2. 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
```

2. Set the **encryption** field type to **aescbc**:

```
spec:
  encryption:
    type: aescbc (1)
```

1 The **aescbc** type means that AES-CBC with PKCS#7 padding and a 32 byte key is used to perform the encryption.

3. 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.

4. Verify that etcd encryption was successful.

   a. 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, oauthaccesstokens.oauth.openshift.io,
   oauthauthorizetokens.oauth.openshift.io
   ```

   If the output shows **EncryptionInProgress**, this means that encryption 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 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.

6.3. 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 7. SCANNING PODS FOR VULNERABILITIES

Using the Container Security Operator (CSO), you can access vulnerability scan results from the OpenShift Container Platform web console for container images used in active pods on the cluster. The CSO:

- 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 CSO is installed in the **openshift-operators** namespace, so it is available to all namespaces on your OpenShift cluster.

7.1. RUNNING THE CONTAINER SECURITY OPERATOR

You can start the Container Security Operator from the OpenShift Container Platform web console by selecting and installing that Operator from the Operator Hub, as described here.

Prerequisites

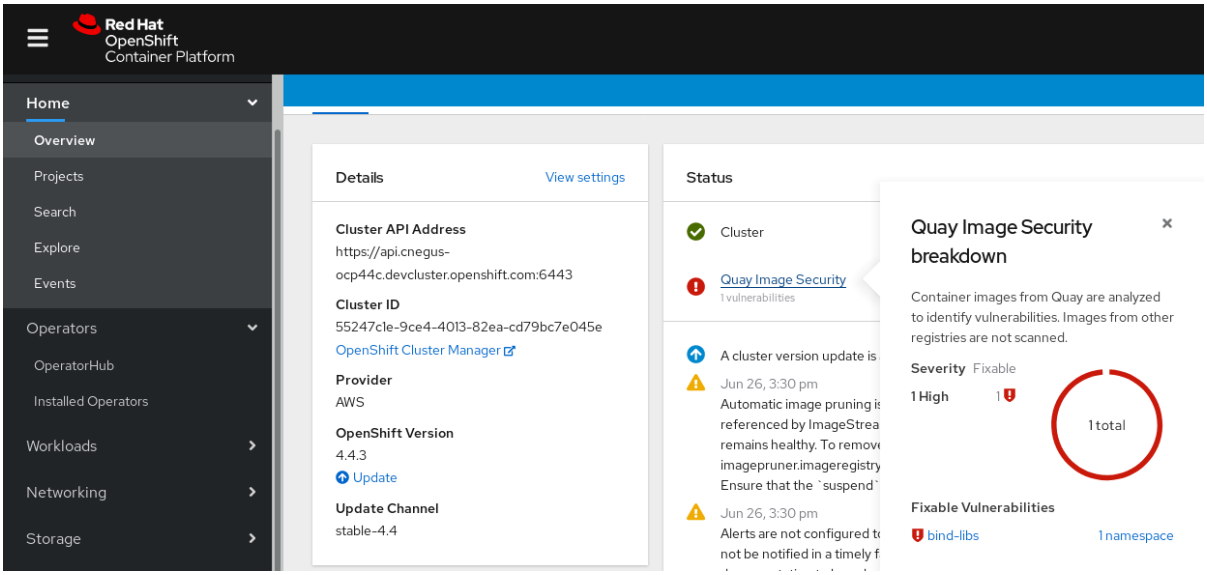
- Have administrator privileges to the OpenShift Container Platform cluster
- Have containers that come from a Red Hat Quay or Quay.io registry running on your cluster

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 CSO. In this example, create a certificate named **quay.crt** in the current directory. Then run the following command to add the cert to the CSO:

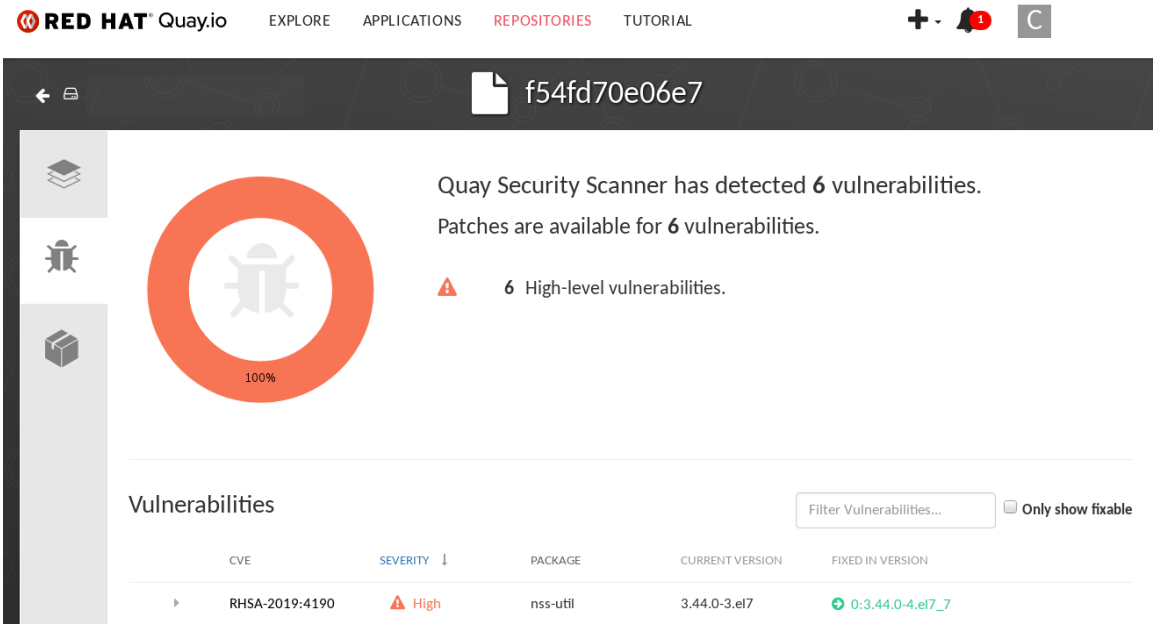
```
$ oc create secret generic container-security-operator-extra-certs --from-file=quay.crt -n
openshift-operators
```

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:

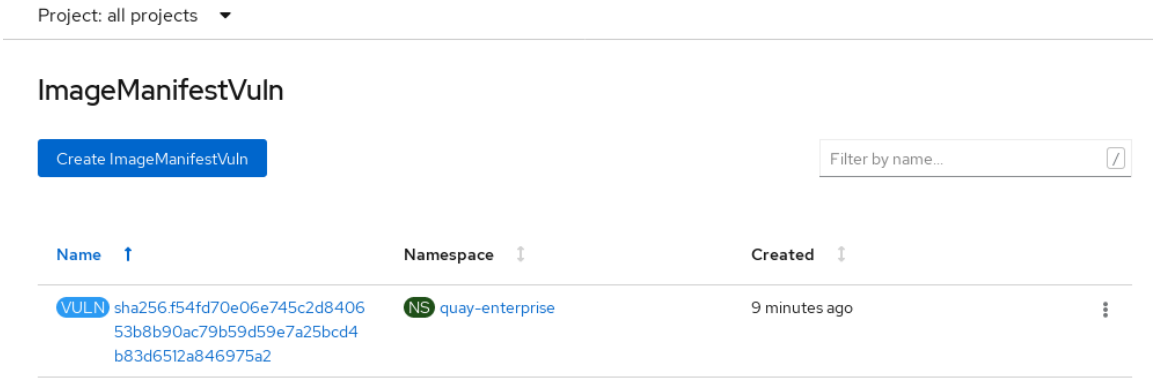


8. You can do one of two things at this point to follow up on any detected vulnerabilities:

   - 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.

7.2. QUERYING IMAGE VULNERABILITIES FROM THE CLI

Using the **oc** command, you can display information about vulnerabilities detected by the Container Security Operator.

Prerequisites

- Be running the 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...
```