# OpenShift Container Platform 4.13

## Specialized hardware and driver enablement

Learn about hardware enablement on OpenShift Container Platform

Last Updated: 2023-10-11

# OpenShift Container Platform 4.13 Specialized hardware and driver enablement

Learn about hardware enablement on OpenShift Container Platform

## Legal Notice



## Abstract

This document provides an overview of hardware enablement in OpenShift Container Platform.

# Table of Contents

# CHAPTER 1. ABOUT SPECIALIZED HARDWARE AND DRIVER ENABLEMENT

The Driver Toolkit (DTK) is a container image in the OpenShift Container Platform payload which is meant to be used as a base image on which to build driver containers. The Driver Toolkit image contains the kernel packages commonly required as dependencies to build or install kernel modules as well as a few tools needed in driver containers. The version of these packages will match the kernel version running on the RHCOS nodes in the corresponding OpenShift Container Platform release.

Driver containers are container images used for building and deploying out-of-tree kernel modules and drivers on container operating systems such as Red Hat Enterprise Linux CoreOS (RHCOS). Kernel modules and drivers are software libraries running with a high level of privilege in the operating system kernel. They extend the kernel functionalities or provide the hardware-specific code required to control new devices. Examples include hardware devices like field-programmable gate arrays (FPGA) or graphics processing units (GPU), and software-defined storage solutions, which all require kernel modules on client machines. Driver containers are the first layer of the software stack used to enable these technologies on OpenShift Container Platform deployments.

# CHAPTER 2. DRIVER TOOLKIT

Learn about the Driver Toolkit and how you can use it as a base image for driver containers for enabling special software and hardware devices on OpenShift Container Platform deployments.

## 2.1. ABOUT THE DRIVER TOOLKIT

### Background

The Driver Toolkit is a container image in the OpenShift Container Platform payload used as a base image on which you can build driver containers. The Driver Toolkit image includes the kernel packages commonly required as dependencies to build or install kernel modules, as well as a few tools needed in driver containers. The version of these packages will match the kernel version running on the Red Hat Enterprise Linux CoreOS (RHCOS) nodes in the corresponding OpenShift Container Platform release.

Driver containers are container images used for building and deploying out-of-tree kernel modules and drivers on container operating systems like RHCOS. Kernel modules and drivers are software libraries running with a high level of privilege in the operating system kernel. They extend the kernel functionalities or provide the hardware-specific code required to control new devices. Examples include hardware devices like Field Programmable Gate Arrays (FPGA) or GPUs, and software-defined storage (SDS) solutions, such as Lustre parallel file systems, which require kernel modules on client machines. Driver containers are the first layer of the software stack used to enable these technologies on Kubernetes.

The list of kernel packages in the Driver Toolkit includes the following and their dependencies:

- **kernel-core**
- **kernel-devel**
- **kernel-headers**
- **kernel-modules**
- **kernel-modules-extra**

In addition, the Driver Toolkit also includes the corresponding real-time kernel packages:

- **kernel-rt-core**
- **kernel-rt-devel**
- **kernel-rt-modules**
- **kernel-rt-modules-extra**

The Driver Toolkit also has several tools that are commonly needed to build and install kernel modules, including:

- **elfutils-libelf-devel**
- **kmod**
- **binutilskabi-dw**
- **kernel-abi-whitelists**

- dependencies for the above

**Purpose**
Prior to the Driver Toolkit's existence, users would install kernel packages in a pod or build config on OpenShift Container Platform using entitled builds or by installing from the kernel RPMs in the hosts **machine-os-content**. The Driver Toolkit simplifies the process by removing the entitlement step, and avoids the privileged operation of accessing the machine-os-content in a pod. The Driver Toolkit can also be used by partners who have access to pre-released OpenShift Container Platform versions to prebuild driver-containers for their hardware devices for future OpenShift Container Platform releases.

The Driver Toolkit is also used by the Kernel Module Management (KMM), which is currently available as a community Operator on OperatorHub. KMM supports out-of-tree and third-party kernel drivers and the support software for the underlying operating system. Users can create modules for KMM to build and deploy a driver container, as well as support software like a device plugin, or metrics. Modules can include a build config to build a driver container-based on the Driver Toolkit, or KMM can deploy a prebuilt driver container.

## 2.2. PULLING THE DRIVER TOOLKIT CONTAINER IMAGE

The **driver-toolkit** image is available from the Container images section of the Red Hat Ecosystem Catalog and in the OpenShift Container Platform release payload. The image corresponding to the most recent minor release of OpenShift Container Platform will be tagged with the version number in the catalog. The image URL for a specific release can be found using the **oc adm** CLI command.

### 2.2.1. Pulling the Driver Toolkit container image from registry.redhat.io

Instructions for pulling the **driver-toolkit** image from **registry.redhat.io** with **podman** or in OpenShift Container Platform can be found on the Red Hat Ecosystem Catalog . The driver-toolkit image for the latest minor release are tagged with the minor release version on **registry.redhat.io**, for example: **registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.13**.

### 2.2.2. Finding the Driver Toolkit image URL in the payload

**Prerequisites**

- You obtained the image pull secret from the Red Hat OpenShift Cluster Manager .
- You installed the OpenShift CLI (**oc**).

**Procedure**

1. Use the **oc adm** command to extract the image URL of the **driver-toolkit** corresponding to a certain release:
   - For an x86 image, the command is as follows:

     ```
     $ oc adm release info quay.io/openshift-release-dev/ocp-release:4.13.z-x86_64 --image-for=driver-toolkit
     ```

   - For an ARM image, the command is as follows:

     ```
     $ oc adm release info quay.io/openshift-release-dev/ocp-release:4.13.z-aarch64 --image-for=driver-toolkit
     ```

#### Example output

The output for the **ocp-release:4.13.0-x86_64** image is as follows:

```
quay.io/openshift-release-dev/ocp-v4.0-art-dev@sha256:b53883ca2bac5925857148c4a1abc300ced96c222498e3bc134fe7ce3a1dd404
```

2. Obtain this image using a valid pull secret, such as the pull secret required to install OpenShift Container Platform:

```
$ podman pull --authfile=path/to/pullsecret.json quay.io/openshift-release-dev/ocp-v4.0-art-dev@sha256:<SHA>
```

## 2.3. USING THE DRIVER TOOLKIT

As an example, the Driver Toolkit can be used as the base image for building a very simple kernel module called **simple-kmod**.

NOTE

The Driver Toolkit includes the necessary dependencies, **openssl**, **mokutil**, and **keyutils**, needed to sign a kernel module. However, in this example, the **simple-kmod** kernel module is not signed and therefore cannot be loaded on systems with **Secure Boot** enabled.

### 2.3.1. Build and run the simple-kmod driver container on a cluster

#### Prerequisites

- You have a running OpenShift Container Platform cluster.
- You set the Image Registry Operator state to **Managed** for your cluster.
- You installed the OpenShift CLI (**oc**).
- You are logged into the OpenShift CLI as a user with **cluster-admin** privileges.

#### Procedure

Create a namespace. For example:

```
$ oc new-project simple-kmod-demo
```

1. The YAML defines an **ImageStream** for storing the **simple-kmod** driver container image, and a **BuildConfig** for building the container. Save this YAML as **0000-buildconfig.yaml.template**.

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  labels:
    app: simple-kmod-driver-container
  name: simple-kmod-driver-container
  namespace: simple-kmod-demo
spec: {}
```

```
---
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  labels:
    app: simple-kmod-driver-build
  name: simple-kmod-driver-build
  namespace: simple-kmod-demo
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  runPolicy: "Serial"
  triggers:
    - type: "ConfigChange"
    - type: "ImageChange"
  source:
    dockerfile: |
      ARG DTK
      FROM ${DTK} as builder

      ARG KVER

      WORKDIR /build/

      RUN git clone https://github.com/openshift-psap/simple-kmod.git

      WORKDIR /build/simple-kmod

      RUN make all install KVER=${KVER}

      FROM registry.redhat.io/ubi8/ubi-minimal

      ARG KVER

      # Required for installing `modprobe`
      RUN microdnf install kmod

      COPY --from=builder /lib/modules/${KVER}/simple-kmod.ko /lib/modules/${KVER}/
      COPY --from=builder /lib/modules/${KVER}/simple-procfs-kmod.ko
/lib/modules/${KVER}/
      RUN depmod ${KVER}
  strategy:
    dockerStrategy:
      buildArgs:
        - name: KMODVER
          value: DEMO
          # $ oc adm release info quay.io/openshift-release-dev/ocp-release:<cluster version>-
x86_64 --image-for=driver-toolkit
        - name: DTK
          value: quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:34864ccd2f4b6e385705a730864c04a40908e57acede44457a783d739e377cae
        - name: KVER
          value: 4.18.0-372.26.1.el8_6.x86_64
  output:
```

```
    to:
      kind: ImageStreamTag
      name: simple-kmod-driver-container:demo
```

2. Substitute the correct driver toolkit image for the OpenShift Container Platform version you are running in place of “DRIVER_TOOLKIT_IMAGE” with the following commands.

```
$ OCP_VERSION=$(oc get clusterversion/version -ojsonpath={.status.desired.version})
```

```
$ DRIVER_TOOLKIT_IMAGE=$(oc adm release info $OCP_VERSION --image-for=driver-toolkit)
```

```
$ sed "s#DRIVER_TOOLKIT_IMAGE#${DRIVER_TOOLKIT_IMAGE}#" 0000-buildconfig.yaml.template > 0000-buildconfig.yaml
```

3. Create the image stream and build config with

```
$ oc create -f 0000-buildconfig.yaml
```

4. After the builder pod completes successfully, deploy the driver container image as a **DaemonSet**.

   a. The driver container must run with the privileged security context in order to load the kernel modules on the host. The following YAML file contains the RBAC rules and the **DaemonSet** for running the driver container. Save this YAML as **1000-drivercontainer.yaml**.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: simple-kmod-driver-container
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: simple-kmod-driver-container
rules:
- apiGroups:
  - security.openshift.io
  resources:
  - securitycontextconstraints
  verbs:
  - use
  resourceNames:
  - privileged
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: simple-kmod-driver-container
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: simple-kmod-driver-container
subjects:
```

```
- kind: ServiceAccount
  name: simple-kmod-driver-container
userNames:
- system:serviceaccount:simple-kmod-demo:simple-kmod-driver-container
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: simple-kmod-driver-container
spec:
  selector:
    matchLabels:
      app: simple-kmod-driver-container
  template:
    metadata:
      labels:
        app: simple-kmod-driver-container
    spec:
      serviceAccount: simple-kmod-driver-container
      serviceAccountName: simple-kmod-driver-container
      containers:
      - image: image-registry.openshift-image-registry.svc:5000/simple-kmod-
demo/simple-kmod-driver-container:demo
        name: simple-kmod-driver-container
        imagePullPolicy: Always
        command: [sleep, infinity]
        lifecycle:
          postStart:
            exec:
              command: ["modprobe", "-v", "-a" , "simple-kmod", "simple-procfs-kmod"]
          preStop:
            exec:
              command: ["modprobe", "-r", "-a" , "simple-kmod", "simple-procfs-kmod"]
        securityContext:
          privileged: true
      nodeSelector:
        node-role.kubernetes.io/worker: ""
```

b. Create the RBAC rules and daemon set:

```
$ oc create -f 1000-drivercontainer.yaml
```

5. After the pods are running on the worker nodes, verify that the **simple_kmod** kernel module is loaded successfully on the host machines with **lsmod**.

a. Verify that the pods are running:

```
$ oc get pod -n simple-kmod-demo
```

**Example output**

```
NAME                                READY   STATUS      RESTARTS   AGE
simple-kmod-driver-build-1-build    0/1     Completed   0          6m
simple-kmod-driver-container-b22fd  1/1     Running     0          40s
```

```
simple-kmod-driver-container-jz9vn   1/1     Running     0          40s
simple-kmod-driver-container-p45cc   1/1     Running     0          40s
```

b. Execute the **lsmod** command in the driver container pod:

```
$ oc exec -it pod/simple-kmod-driver-container-p45cc -- lsmod | grep simple
```

**Example output**

```
simple_procfs_kmod     16384  0
simple_kmod            16384  0
```

## 2.4. ADDITIONAL RESOURCES

- For more information about configuring registry storage for your cluster, see Image Registry Operator in OpenShift Container Platform.

# CHAPTER 3. NODE FEATURE DISCOVERY OPERATOR

Learn about the Node Feature Discovery (NFD) Operator and how you can use it to expose node-level information by orchestrating Node Feature Discovery, a Kubernetes add-on for detecting hardware features and system configuration.

## 3.1. ABOUT THE NODE FEATURE DISCOVERY OPERATOR

The Node Feature Discovery Operator (NFD) manages the detection of hardware features and configuration in an OpenShift Container Platform cluster by labeling the nodes with hardware-specific information. NFD labels the host with node-specific attributes, such as PCI cards, kernel, operating system version, and so on.

The NFD Operator can be found on the Operator Hub by searching for “Node Feature Discovery”.

## 3.2. INSTALLING THE NODE FEATURE DISCOVERY OPERATOR

The Node Feature Discovery (NFD) Operator orchestrates all resources needed to run the NFD daemon set. As a cluster administrator, you can install the NFD Operator by using the OpenShift Container Platform CLI or the web console.

### 3.2.1. Installing the NFD Operator using the CLI

As a cluster administrator, you can install the NFD Operator using the CLI.

**Prerequisites**

- An OpenShift Container Platform cluster
- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.

**Procedure**

1. Create a namespace for the NFD Operator.
   a. Create the following **Namespace** custom resource (CR) that defines the **openshift-nfd** namespace, and then save the YAML in the **nfd-namespace.yaml** file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-nfd
```

   b. Create the namespace by running the following command:

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

2. Install the NFD Operator in the namespace you created in the previous step by creating the following objects:
   a. Create the following **OperatorGroup** CR and save the YAML in the **nfd-operatorgroup.yaml** file:

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

b. Create the **OperatorGroup** CR by running the following command:

```
$ oc create -f nfd-operatorgroup.yaml
```

c. Create the following **Subscription** CR and save the YAML in the **nfd-sub.yaml** file:

**Example Subscription**

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: nfd
  namespace: openshift-nfd
spec:
  channel: "stable"
  installPlanApproval: Automatic
  name: nfd
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

d. Create the subscription object by running the following command:

```
$ oc create -f nfd-sub.yaml
```

e. Change to the **openshift-nfd** project:

```
$ oc project openshift-nfd
```

**Verification**

- To verify that the Operator deployment is successful, run:

```
$ oc get pods
```

**Example output**

```
NAME                                  READY   STATUS    RESTARTS   AGE
nfd-controller-manager-7f86ccfb58-vgr4x   2/2     Running   0          10m
```

A successful deployment shows a **Running** status.

### 3.2.2. Installing the NFD Operator using the web console

As a cluster administrator, you can install the NFD Operator using the web console.

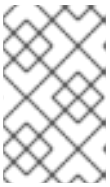
**Procedure**

1. In the OpenShift Container Platform web console, click **Operators → OperatorHub**.
2. Choose **Node Feature Discovery** from the list of available Operators, and then click **Install**.
3. On the **Install Operator** page, select **A specific namespace on the cluster**, and then click **Install**. You do not need to create a namespace because it is created for you.

### Verification

To verify that the NFD Operator installed successfully:

1. Navigate to the **Operators → Installed Operators** page.
2. Ensure that **Node Feature Discovery** is listed in the **openshift-nfd** project with a **Status** of **InstallSucceeded**.



NOTE

During installation an Operator might display a **Failed** status. If the installation later succeeds with an **InstallSucceeded** message, you can ignore the **Failed** message.

### Troubleshooting

If the Operator does not appear as installed, troubleshoot further:

1. Navigate to the **Operators → Installed Operators** page and inspect the **Operator Subscriptions** and **Install Plans** tabs for any failure or errors under **Status**.
2. Navigate to the **Workloads → Pods** page and check the logs for pods in the **openshift-nfd** project.

## 3.3. USING THE NODE FEATURE DISCOVERY OPERATOR

The Node Feature Discovery (NFD) Operator orchestrates all resources needed to run the Node-Feature-Discovery daemon set by watching for a **NodeFeatureDiscovery** CR. Based on the **NodeFeatureDiscovery** CR, the Operator will create the operand (NFD) components in the desired namespace. You can edit the CR to choose another **namespace**, **image**, **imagePullPolicy**, and **nfd-worker-conf**, among other options.

As a cluster administrator, you can create a **NodeFeatureDiscovery** instance using the OpenShift Container Platform CLI or the web console.

### 3.3.1. Create a NodeFeatureDiscovery instance using the CLI

As a cluster administrator, you can create a **NodeFeatureDiscovery** CR instance using the CLI.

**Prerequisites**

- An OpenShift Container Platform cluster

- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.
- Install the NFD Operator.

**Procedure**

1. Create the following **NodeFeatureDiscovery** Custom Resource (CR), and then save the YAML in the **NodeFeatureDiscovery.yaml** file:

```
apiVersion: nfd.openshift.io/v1
kind: NodeFeatureDiscovery
metadata:
  name: nfd-instance
  namespace: openshift-nfd
spec:
  instance: "" # instance is empty by default
  topologyupdater: false # False by default
  operand:
    image: registry.redhat.io/openshift4/ose-node-feature-discovery:v4.13
    imagePullPolicy: Always
  workerConfig:
    configData: |
      core:
      #  labelWhiteList:
      #  noPublish: false
        sleepInterval: 60s
      #  sources: [all]
      #  klog:
      #    addDirHeader: false
      #    alsologtostderr: false
      #    logBacktraceAt:
      #    logtostderr: true
      #    skipHeaders: false
      #    stderrthreshold: 2
      #    v: 0
      #    vmodule:
      ##   NOTE: the following options are not dynamically run-time configurable
      ##         and require a nfd-worker restart to take effect after being changed
      #    logDir:
      #    logFile:
      #    logFileMaxSize: 1800
      #    skipLogHeaders: false
      sources:
        cpu:
          cpuid:
      #     NOTE: whitelist has priority over blacklist
            attributeBlacklist:
              - "BMI1"
              - "BMI2"
              - "CLMUL"
              - "CMOV"
              - "CX16"
              - "ERMS"
              - "F16C"
```

```
            - "HTT"
            - "LZCNT"
            - "MMX"
            - "MMXEXT"
            - "NX"
            - "POPCNT"
            - "RDRAND"
            - "RDSEED"
            - "RDTSCP"
            - "SGX"
            - "SSE"
            - "SSE2"
            - "SSE3"
            - "SSE4.1"
            - "SSE4.2"
            - "SSSE3"
          attributeWhitelist:
      kernel:
        kconfigFile: "/path/to/kconfig"
        configOpts:
          - "NO_HZ"
          - "X86"
          - "DMI"
      pci:
        deviceClassWhitelist:
          - "0200"
          - "03"
          - "12"
        deviceLabelFields:
          - "class"
  customConfig:
    configData: |
        - name: "more.kernel.features"
          matchOn:
          - loadedKMod: ["example_kmod3"]
```

For more details on how to customize NFD workers, refer to the Configuration file reference of nfd-worker.

1. Create the **NodeFeatureDiscovery** CR instance by running the following command:

```
$ oc create -f NodeFeatureDiscovery.yaml
```

### Verification

- To verify that the instance is created, run:

```
$ oc get pods
```

#### Example output

```
NAME                                  READY   STATUS    RESTARTS   AGE
nfd-controller-manager-7f86ccfb58-vgr4x   2/2     Running   0          11m
nfd-master-hcn64                      1/1     Running   0          60s
```

```
nfd-master-lnnxx                1/1     Running   0          60s
nfd-master-mp6hr                 1/1     Running   0          60s
nfd-worker-vgcz9                1/1     Running   0          60s
nfd-worker-xqbws                 1/1     Running   0          60s
```

A successful deployment shows a **Running** status.

### 3.3.2. Create a NodeFeatureDiscovery CR using the web console

**Procedure**

1. Navigate to the **Operators → Installed Operators** page.
2. Find **Node Feature Discovery** and see a box under **Provided APIs**.
3. Click **Create instance**.
4. Edit the values of the **NodeFeatureDiscovery** CR.
5. Click **Create**.

## 3.4. CONFIGURING THE NODE FEATURE DISCOVERY OPERATOR

### 3.4.1. core

The **core** section contains common configuration settings that are not specific to any particular feature source.

**core.sleepInterval**
**core.sleepInterval** specifies the interval between consecutive passes of feature detection or re-detection, and thus also the interval between node re-labeling. A non-positive value implies infinite sleep interval; no re-detection or re-labeling is done.

This value is overridden by the deprecated **--sleep-interval** command line flag, if specified.

#### Example usage

```
core:
  sleepInterval: 60s 1
```

The default value is **60s**.

**core.sources**
**core.sources** specifies the list of enabled feature sources. A special value **all** enables all feature sources.

This value is overridden by the deprecated **--sources** command line flag, if specified.

Default: **[all]**

#### Example usage

```
core:
  sources:
```

```
    - system
    - custom
```

**core.labelWhiteList**
**core.labelWhiteList** specifies a regular expression for filtering feature labels based on the label name. Non-matching labels are not published.

The regular expression is only matched against the basename part of the label, the part of the name after '/'. The label prefix, or namespace, is omitted.

This value is overridden by the deprecated **--label-whitelist** command line flag, if specified.

Default: **null**

### Example usage

```
core:
  labelWhiteList: '^cpu-cpuid'
```

**core.noPublish**
Setting **core.noPublish** to **true** disables all communication with the **nfd-master**. It is effectively a dry run flag; **nfd-worker** runs feature detection normally, but no labeling requests are sent to **nfd-master**.

This value is overridden by the **--no-publish** command line flag, if specified.

Example:

### Example usage

```
core:
  noPublish: true (1)
```

The default value is **false**.

**core.klog**
The following options specify the logger configuration, most of which can be dynamically adjusted at run-time.

The logger options can also be specified using command line flags, which take precedence over any corresponding config file options.

**core.klog.addDirHeader**
If set to **true**, **core.klog.addDirHeader** adds the file directory to the header of the log messages.

Default: **false**

Run-time configurable: yes

**core.klog.alsologtostderr**
Log to standard error as well as files.

Default: **false**

Run-time configurable: yes

**core.klog.logBacktraceAt**

When logging hits line file:N, emit a stack trace.

Default: **empty**

Run-time configurable: yes

**core.klog.logDir**
If non-empty, write log files in this directory.

Default: **empty**

Run-time configurable: no

**core.klog.logFile**
If not empty, use this log file.

Default: **empty**

Run-time configurable: no

**core.klog.logFileMaxSize**
**core.klog.logFileMaxSize** defines the maximum size a log file can grow to. Unit is megabytes. If the value is **0**, the maximum file size is unlimited.

Default: **1800**

Run-time configurable: no

**core.klog.logtostderr**
Log to standard error instead of files

Default: **true**

Run-time configurable: yes

**core.klog.skipHeaders**
If **core.klog.skipHeaders** is set to **true**, avoid header prefixes in the log messages.

Default: **false**

Run-time configurable: yes

**core.klog.skipLogHeaders**
If **core.klog.skipLogHeaders** is set to **true**, avoid headers when opening log files.

Default: **false**

Run-time configurable: no

**core.klog.stderrthreshold**
Logs at or above this threshold go to stderr.

Default: **2**

Run-time configurable: yes

**core.klog.v**
**core.klog.v** is the number for the log level verbosity.

Default: **0**

Run-time configurable: yes

**core.klog.vmodule**
**core.klog.vmodule** is a comma-separated list of **pattern=N** settings for file-filtered logging.

Default: **empty**

Run-time configurable: yes

### 3.4.2. sources

The **sources** section contains feature source specific configuration parameters.

**sources.cpu.cpuid.attributeBlacklist**
Prevent publishing **cpuid** features listed in this option.

This value is overridden by **sources.cpu.cpuid.attributeWhitelist**, if specified.

Default: **[BMI1, BMI2, CLMUL, CMOV, CX16, ERMS, F16C, HTT, LZCNT, MMX, MMXEXT, NX, POPCNT, RDRAND, RDSEED, RDTSCP, SGX, SGXLC, SSE, SSE2, SSE3, SSE4.1, SSE4.2, SSSE3]**

#### Example usage

```
sources:
  cpu:
    cpuid:
      attributeBlacklist: [MMX, MMXEXT]
```

**sources.cpu.cpuid.attributeWhitelist**
Only publish the **cpuid** features listed in this option.

**sources.cpu.cpuid.attributeWhitelist** takes precedence over **sources.cpu.cpuid.attributeBlacklist**.

Default: **empty**

#### Example usage

```
sources:
  cpu:
    cpuid:
      attributeWhitelist: [AVX512BW, AVX512CD, AVX512DQ, AVX512F, AVX512VL]
```

**sources.kernel.kconfigFile**
**sources.kernel.kconfigFile** is the path of the kernel config file. If empty, NFD runs a search in the well-known standard locations.

Default: **empty**

#### Example usage

```
sources:
  kernel:
    kconfigFile: "/path/to/kconfig"
```

**sources.kernel.configOpts**
**sources.kernel.configOpts** represents kernel configuration options to publish as feature labels.

Default: **[NO_HZ, NO_HZ_IDLE, NO_HZ_FULL, PREEMPT]**

### Example usage

```
sources:
  kernel:
    configOpts: [NO_HZ, X86, DMI]
```

**sources.pci.deviceClassWhitelist**
**sources.pci.deviceClassWhitelist** is a list of PCI device class IDs for which to publish a label. It can be specified as a main class only (for example, **03**) or full class-subclass combination (for example **0300**). The former implies that all subclasses are accepted. The format of the labels can be further configured with **deviceLabelFields**.

Default: **["03", "0b40", "12"]**

### Example usage

```
sources:
  pci:
    deviceClassWhitelist: ["0200", "03"]
```

**sources.pci.deviceLabelFields**
**sources.pci.deviceLabelFields** is the set of PCI ID fields to use when constructing the name of the feature label. Valid fields are **class**, **vendor**, **device**, **subsystem_vendor** and **subsystem_device**.

Default: **[class, vendor]**

### Example usage

```
sources:
  pci:
    deviceLabelFields: [class, vendor, device]
```

With the example config above, NFD would publish labels such as **feature.node.kubernetes.io/pci-<class-id>_<vendor-id>_<device-id>.present=true**

**sources.usb.deviceClassWhitelist**
**sources.usb.deviceClassWhitelist** is a list of USB device class IDs for which to publish a feature label. The format of the labels can be further configured with **deviceLabelFields**.

Default: **["0e", "ef", "fe", "ff"]**

### Example usage

```
sources:
  usb:
    deviceClassWhitelist: ["ef", "ff"]
```

**sources.usb.deviceLabelFields**

**sources.usb.deviceLabelFields** is the set of USB ID fields from which to compose the name of the feature label. Valid fields are **class**, **vendor**, and **device**.

Default: **[class, vendor, device]**

#### Example usage

```
sources:
  pci:
    deviceLabelFields: [class, vendor]
```

With the example config above, NFD would publish labels like: **feature.node.kubernetes.io/usb-<class-id>_<vendor-id>.present=true**.

**sources.custom**
**sources.custom** is the list of rules to process in the custom feature source to create user-specific labels.

Default: **empty**

#### Example usage

```
source:
  custom:
  - name: "my.custom.feature"
    matchOn:
    - loadedKMod: ["e1000e"]
    - pciId:
        class: ["0200"]
        vendor: ["8086"]
```

## 3.5. USING THE NFD TOPOLOGY UPDATER

The Node Feature Discovery (NFD) Topology Updater is a daemon responsible for examining allocated resources on a worker node. It accounts for resources that are available to be allocated to new pod on a per-zone basis, where a zone can be a Non-Uniform Memory Access (NUMA) node. The NFD Topology Updater communicates the information to nfd-master, which creates a **NodeResourceTopology** custom resource (CR) corresponding to all of the worker nodes in the cluster. One instance of the NFD Topology Updater runs on each node of the cluster.

To enable the Topology Updater workers in NFD, set the **topologyupdater** variable to **true** in the **NodeFeatureDiscovery** CR, as described in the section **Using the Node Feature Discovery Operator**.

### 3.5.1. NodeResourceTopology CR

When run with NFD Topology Updater, NFD creates custom resource instances corresponding to the node resource hardware topology, such as:

```
apiVersion: topology.node.k8s.io/v1alpha1
kind: NodeResourceTopology
metadata:
  name: node1
topologyPolicies: ["SingleNUMANodeContainerLevel"]
zones:
```

```
- name: node-0
  type: Node
  resources:
    - name: cpu
      capacity: 20
      allocatable: 16
      available: 10
    - name: vendor/nic1
      capacity: 3
      allocatable: 3
      available: 3
- name: node-1
  type: Node
  resources:
    - name: cpu
      capacity: 30
      allocatable: 30
      available: 15
    - name: vendor/nic2
      capacity: 6
      allocatable: 6
      available: 6
- name: node-2
  type: Node
  resources:
    - name: cpu
      capacity: 30
      allocatable: 30
      available: 15
    - name: vendor/nic1
      capacity: 3
      allocatable: 3
      available: 3
```

### 3.5.2. NFD Topology Updater command line flags

To view available command line flags, run the **nfd-topology-updater -help** command. For example, in a podman container, run the following command:

```
$ podman run gcr.io/k8s-staging-nfd/node-feature-discovery:master nfd-topology-updater -help
```

**-ca-file**
The **-ca-file** flag is one of the three flags, together with the **-cert-file** and `-key-file` flags, that controls the mutual TLS authentication on the NFD Topology Updater. This flag specifies the TLS root certificate that is used for verifying the authenticity of nfd-master.

Default: empty

IMPORTANT

The **-ca-file** flag must be specified together with the **-cert-file** and **-key-file** flags.

**Example**

```
$ nfd-topology-updater -ca-file=/opt/nfd/ca.crt -cert-file=/opt/nfd/updater.crt -key-
file=/opt/nfd/updater.key
```

**-cert-file**
The **-cert-file** flag is one of the three flags, together with the **-ca-file** and **-key-file flags**, that controls mutual TLS authentication on the NFD Topology Updater. This flag specifies the TLS certificate presented for authenticating outgoing requests.

Default: empty

IMPORTANT

The **-cert-file** flag must be specified together with the **-ca-file** and **-key-file** flags.

### Example

```
$ nfd-topology-updater -cert-file=/opt/nfd/updater.crt -key-file=/opt/nfd/updater.key -ca-
file=/opt/nfd/ca.crt
```

**-h, -help**
Print usage and exit.

**-key-file**
The **-key-file** flag is one of the three flags, together with the **-ca-file** and **-cert-file** flags, that controls the mutual TLS authentication on the NFD Topology Updater. This flag specifies the private key corresponding the given certificate file, or **-cert-file**, that is used for authenticating outgoing requests.

Default: empty

IMPORTANT

The **-key-file** flag must be specified together with the **-ca-file** and **-cert-file** flags.

### Example

```
$ nfd-topology-updater -key-file=/opt/nfd/updater.key -cert-file=/opt/nfd/updater.crt -ca-
file=/opt/nfd/ca.crt
```

**-kubelet-config-file**
The **-kubelet-config-file** specifies the path to the Kubelet's configuration file.

Default: **/host-var/lib/kubelet/config.yaml**

### Example

```
$ nfd-topology-updater -kubelet-config-file=/var/lib/kubelet/config.yaml
```

**-no-publish**
The **-no-publish** flag disables all communication with the nfd-master, making it a dry run flag for nfd-topology-updater. NFD Topology Updater runs resource hardware topology detection normally, but no CR requests are sent to nfd-master.

Default: **false**

#### Example

```
$ nfd-topology-updater -no-publish
```

#### 3.5.2.1. -oneshot

The **-oneshot** flag causes the NFD Topology Updater to exit after one pass of resource hardware topology detection.

Default: **false**

#### Example

```
$ nfd-topology-updater -oneshot -no-publish
```

**-podresources-socket**
The **-podresources-socket** flag specifies the path to the Unix socket where kubelet exports a gRPC service to enable discovery of in-use CPUs and devices, and to provide metadata for them.

Default: **/host-var/liblib/kubelet/pod-resources/kubelet.sock**

#### Example

```
$ nfd-topology-updater -podresources-socket=/var/lib/kubelet/pod-resources/kubelet.sock
```

**-server**
The **-server** flag specifies the address of the nfd-master endpoint to connect to.

Default: **localhost:8080**

#### Example

```
$ nfd-topology-updater -server=nfd-master.nfd.svc.cluster.local:443
```

**-server-name-override**
The **-server-name-override** flag specifies the common name (CN) which to expect from the nfd-master TLS certificate. This flag is mostly intended for development and debugging purposes.

Default: empty

#### Example

```
$ nfd-topology-updater -server-name-override=localhost
```

**-sleep-interval**
The **-sleep-interval** flag specifies the interval between resource hardware topology re-examination and custom resource updates. A non-positive value implies infinite sleep interval and no re-detection is done.

Default: **60s**

### Example

```
$ nfd-topology-updater -sleep-interval=1h
```

**-version**
Print version and exit.

**-watch-namespace**
The **-watch-namespace** flag specifies the namespace to ensure that resource hardware topology examination only happens for the pods running in the specified namespace. Pods that are not running in the specified namespace are not considered during resource accounting. This is particularly useful for testing and debugging purposes. A * value means that all of the pods across all namespaces are considered during the accounting process.

Default: *

### Example

```
$ nfd-topology-updater -watch-namespace=rte
```

# CHAPTER 4. KERNEL MODULE MANAGEMENT OPERATOR

Learn about the Kernel Module Management (KMM) Operator and how you can use it to deploy out-of-tree kernel modules and device plugins on OpenShift Container Platform clusters.

## 4.1. ABOUT THE KERNEL MODULE MANAGEMENT OPERATOR

The Kernel Module Management (KMM) Operator manages, builds, signs, and deploys out-of-tree kernel modules and device plugins on OpenShift Container Platform clusters.

KMM adds a new **Module** CRD which describes an out-of-tree kernel module and its associated device plugin. You can use **Module** resources to configure how to load the module, define **ModuleLoader** images for kernel versions, and include instructions for building and signing modules for specific kernel versions.

KMM is designed to accommodate multiple kernel versions at once for any kernel module, allowing for seamless node upgrades and reduced application downtime.

## 4.2. INSTALLING THE KERNEL MODULE MANAGEMENT OPERATOR

As a cluster administrator, you can install the Kernel Module Management (KMM) Operator by using the OpenShift CLI or the web console.

The KMM Operator is supported on OpenShift Container Platform 4.12 and later. Installing KMM on version 4.11 does not require specific additional steps. For details on installing KMM on version 4.10 and earlier, see the section "Installing the Kernel Module Management Operator on earlier versions of OpenShift Container Platform".

### 4.2.1. Installing the Kernel Module Management Operator using the web console

As a cluster administrator, you can install the Kernel Module Management (KMM) Operator using the OpenShift Container Platform web console.

**Procedure**

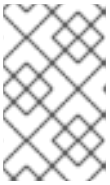
1. Log in to the OpenShift Container Platform web console.
2. Install the Kernel Module Management Operator:
    a. In the OpenShift Container Platform web console, click **Operators → OperatorHub**.
    b. Select **Kernel Module Management Operator** from the list of available Operators, and then click **Install**.
    c. From the **Installed Namespace** list, select the **openshift-kmm** namespace.
    d. Click **Install**.

**Verification**

To verify that KMM Operator installed successfully:

1. Navigate to the **Operators → Installed Operators** page.

2. Ensure that **Kernel Module Management Operator** is listed in the **openshift-kmm** project with a **Status** of **InstallSucceeded**.



NOTE

During installation, an Operator might display a **Failed** status. If the installation later succeeds with an **InstallSucceeded** message, you can ignore the **Failed** message.

**Troubleshooting**

1. To troubleshoot issues with Operator installation:

   a. Navigate to the **Operators → Installed Operators** page and inspect the **Operator Subscriptions** and **Install Plans** tabs for any failure or errors under **Status**.

   b. Navigate to the **Workloads → Pods** page and check the logs for pods in the **openshift-kmm** project.

### 4.2.2. Installing the Kernel Module Management Operator by using the CLI

As a cluster administrator, you can install the Kernel Module Management (KMM) Operator by using the OpenShift CLI.

**Prerequisites**

- You have a running OpenShift Container Platform cluster.
- You installed the OpenShift CLI (**oc**).
- You are logged into the OpenShift CLI as a user with **cluster-admin** privileges.

**Procedure**

1. Install KMM in the **openshift-kmm** namespace:

   a. Create the following **Namespace** CR and save the YAML file, for example, **kmm-namespace.yaml**:

```
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-kmm
```

   b. Create the following **OperatorGroup** CR and save the YAML file, for example, **kmm-op-group.yaml**:

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: kernel-module-management
  namespace: openshift-kmm
```

   c. Create the following **Subscription** CR and save the YAML file, for example, **kmm-sub.yaml**:

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: kernel-module-management
  namespace: openshift-kmm
spec:
  channel: release-1.0
  installPlanApproval: Automatic
  name: kernel-module-management
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  startingCSV: kernel-module-management.v1.0.0
```

d. Create the subscription object by running the following command:

```
$ oc create -f kmm-sub.yaml
```

**Verification**

- To verify that the Operator deployment is successful, run the following command:

```
$ oc get -n openshift-kmm deployments.apps kmm-operator-controller-manager
```

**Example output**

```
NAME                              READY UP-TO-DATE  AVAILABLE AGE
kmm-operator-controller-manager   1/1   1           1         97s
```

The Operator is available.

### 4.2.3. Installing the Kernel Module Management Operator on earlier versions of OpenShift Container Platform

The KMM Operator is supported on OpenShift Container Platform 4.12 and later. For version 4.10 and earlier, you must create a new **SecurityContextConstraint** object and bind it to the Operator's **ServiceAccount**. As a cluster administrator, you can install the Kernel Module Management (KMM) Operator by using the OpenShift CLI.

**Prerequisites**

- You have a running OpenShift Container Platform cluster.
- You installed the OpenShift CLI (**oc**).
- You are logged into the OpenShift CLI as a user with **cluster-admin** privileges.

**Procedure**

1. Install KMM in the **openshift-kmm** namespace:

   a. Create the following **Namespace** CR and save the YAML file, for example, **kmm-namespace.yaml** file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-kmm
```

b. Create the following **SecurityContextConstraint** object and save the YAML file, for example, **kmm-security-constraint.yaml**:

```
allowHostDirVolumePlugin: false
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: false
allowPrivilegedContainer: false
allowedCapabilities:
  - NET_BIND_SERVICE
apiVersion: security.openshift.io/v1
defaultAddCapabilities: null
fsGroup:
  type: MustRunAs
groups: []
kind: SecurityContextConstraints
metadata:
  name: restricted-v2
priority: null
readOnlyRootFilesystem: false
requiredDropCapabilities:
  - ALL
runAsUser:
  type: MustRunAsRange
seLinuxContext:
  type: MustRunAs
seccompProfiles:
  - runtime/default
supplementalGroups:
  type: RunAsAny
users: []
volumes:
  - configMap
  - downwardAPI
  - emptyDir
  - persistentVolumeClaim
  - projected
  - secret
```

c. Bind the **SecurityContextConstraint** object to the Operator's **ServiceAccount** by running the following commands:

```
$ oc apply -f kmm-security-constraint.yaml
```

```
$ oc adm policy add-scc-to-user kmm-security-constraint -z kmm-operator-controller-manager -n openshift-kmm
```

d. Create the following **OperatorGroup** CR and save the YAML file, for example, **kmm-op-group.yaml**:

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: kernel-module-management
  namespace: openshift-kmm
```

e. Create the following **Subscription** CR and save the YAML file, for example, **kmm-sub.yaml**:

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: kernel-module-management
  namespace: openshift-kmm
spec:
  channel: release-1.0
  installPlanApproval: Automatic
  name: kernel-module-management
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  startingCSV: kernel-module-management.v1.0.0
```

f. Create the subscription object by running the following command:

```
$ oc create -f kmm-sub.yaml
```

**Verification**

- To verify that the Operator deployment is successful, run the following command:

```
$ oc get -n openshift-kmm deployments.apps kmm-operator-controller-manager
```

**Example output**

```
NAME                              READY UP-TO-DATE  AVAILABLE AGE
kmm-operator-controller-manager   1/1   1           1         97s
```

The Operator is available.

## 4.3. KERNEL MODULE DEPLOYMENT

For each **Module** resource, Kernel Module Management (KMM) can create a number of **DaemonSet** resources:

- One ModuleLoader **DaemonSet** per compatible kernel version running in the cluster.
- One device plugin **DaemonSet**, if configured.

The module loader daemon set resources run ModuleLoader images to load kernel modules. A module loader image is an OCI image that contains the **.ko** files and both the **modprobe** and **sleep** binaries.

When the module loader pod is created, the pod runs **modprobe** to insert the specified module into the kernel. It then enters a sleep state until it is terminated. When that happens, the **ExecPreStop** hook runs **modprobe -r** to unload the kernel module.

If the **.spec.devicePlugin** attribute is configured in a **Module** resource, then KMM creates a device plugin daemon set in the cluster. That daemon set targets:

- Nodes that match the **.spec.selector** of the **Module** resource.
- Nodes with the kernel module loaded (where the module loader pod is in the **Ready** condition).

### 4.3.1. The Module custom resource definition

The **Module** custom resource definition (CRD) represents a kernel module that can be loaded on all or select nodes in the cluster, through a module loader image. A **Module** custom resource (CR) specifies one or more kernel versions with which it is compatible, and a node selector.

The compatible versions for a **Module** resource are listed under **.spec.moduleLoader.container.kernelMappings**. A kernel mapping can either match a **literal** version, or use **regexp** to match many of them at the same time.

The reconciliation loop for the **Module** resource runs the following steps:

1. List all nodes matching **.spec.selector**.
2. Build a set of all kernel versions running on those nodes.
3. For each kernel version:
    a. Go through **.spec.moduleLoader.container.kernelMappings** and find the appropriate container image name. If the kernel mapping has **build** or **sign** defined and the container image does not already exist, run the build, the signing job, or both, as needed.
    b. Create a module loader daemon set with the container image determined in the previous step.
    c. If **.spec.devicePlugin** is defined, create a device plugin daemon set using the configuration specified under **.spec.devicePlugin.container**.
4. Run **garbage-collect** on:
    a. Existing daemon set resources targeting kernel versions that are not run by any node in the cluster.
    b. Successful build jobs.
    c. Successful signing jobs.

### 4.3.2. Set soft dependencies between kernel modules

Some configurations require that several kernel modules be loaded in a specific order to work properly, even though the modules do not directly depend on each other through symbols. These are called soft dependencies. **depmod** is usually not aware of these dependencies, and they do not appear in the files it produces. For example, if **mod_a** has a soft dependency on **mod_b**, **modprobe mod_a** will not load **mod_b**.

You can resolve these situations by declaring soft dependencies in the Module Custom Resource Definition (CRD) using the **modulesLoadingOrder** field.

```
# ...
spec:
  moduleLoader:
    container:
      modprobe:
        moduleName: mod_a
        dirName: /opt
        firmwarePath: /firmware
        parameters:
          - param=1
        modulesLoadingOrder:
          - mod_a
          - mod_b
```

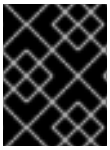
In the configuration above:

- The loading order is **mod_b**, then **mod_a**.
- The unloading order is **mod_a**, then **mod_b**.

NOTE

The first value in the list, to be loaded last, must be equivalent to the **moduleName**.

### 4.3.3. Security and permissions



IMPORTANT

Loading kernel modules is a highly sensitive operation. After they are loaded, kernel modules have all possible permissions to do any kind of operation on the node.

#### 4.3.3.1. ServiceAccounts and SecurityContextConstraints

Kernel Module Management (KMM) creates a privileged workload to load the kernel modules on nodes. That workload needs **ServiceAccounts** allowed to use the **privileged SecurityContextConstraint** (SCC) resource.

The authorization model for that workload depends on the namespace of the **Module** resource, as well as its spec.

- If the **.spec.moduleLoader.serviceAccountName** or **.spec.devicePlugin.serviceAccountName** fields are set, they are always used.
- If those fields are not set, then:
  - If the **Module** resource is created in the operator's namespace ( **openshift-kmm** by default), then KMM uses its default, powerful **ServiceAccounts** to run the daemon sets.
  - If the **Module** resource is created in any other namespace, then KMM runs the daemon sets as the namespace's **default ServiceAccount**. The **Module** resource cannot run a privileged workload unless you manually enable it to use the **privileged** SCC.

IMPORTANT

**openshift-kmm** is a trusted namespace.

When setting up RBAC permissions, remember that any user or **ServiceAccount** creating a **Module** resource in the **openshift-kmm** namespace results in KMM automatically running privileged workloads on potentially all nodes in the cluster.

To allow any **ServiceAccount** to use the **privileged** SCC and therefore to run module loader or device plugin pods, use the following command:

```
$ oc adm policy add-scc-to-user privileged -z "${serviceAccountName}" [ -n "${namespace}" ]
```

#### 4.3.3.2. Pod security standards

OpenShift runs a synchronization mechanism that sets the namespace Pod Security level automatically based on the security contexts in use. No action is needed.

**Additional resources**

- Understanding and managing pod security admission .

## 4.4. REPLACING IN-TREE MODULES WITH OUT-OF-TREE MODULES

You can use Kernel Module Management (KMM) to build kernel modules that can be loaded or unloaded into the kernel on demand. These modules extend the functionality of the kernel without the need to reboot the system. Modules can be configured as built-in or dynamically loaded.

Dynamically loaded modules include in-tree modules and out-of-tree (OOT) modules. In-tree modules are internal to the Linux kernel tree, that is, they are already part of the kernel. Out-of-tree modules are external to the Linux kernel tree. They are generally written for development and testing purposes, such as testing the new version of a kernel module that is shipped in-tree, or to deal with incompatibilities.

Some modules loaded by KMM could replace in-tree modules already loaded on the node. To unload an in-tree module before loading your module, set the **.spec.moduleLoader.container.inTreeModuleToRemove** field. The following is an example for module replacement for all kernel mappings:

```
# ...
spec:
  moduleLoader:
    container:
      modprobe:
        moduleName: mod_a

      inTreeModuleToRemove: mod_b
```

In this example, the **moduleLoader** pod uses **inTreeModuleToRemove** to unload the in-tree **mod_b** before loading **mod_a** from the **moduleLoader** image. When the **moduleLoader`pod is terminated and `mod_a** is unloaded, **mod_b** is not loaded again.

The following is an example for module replacement for specific kernel mappings:

```
# ...
```

```
spec:
  moduleLoader:
    container:
      kernelMappings:
        - literal: 6.0.15-300.fc37.x86_64
          containerImage: some.registry/org/my-kmod:6.0.15-300.fc37.x86_64
          inTreeModuleToRemove: <module_name>
```

**Additional resources**

- Building a linux kernel module

### 4.4.1. Example Module CR

The following is an annotated **Module** example:

```
apiVersion: kmm.sigs.x-k8s.io/v1beta1
kind: Module
metadata:
  name: <my_kmod>
spec:
  moduleLoader:
    container:
      modprobe:
        moduleName: <my_kmod> (1)
        dirName: /opt (2)
        firmwarePath: /firmware (3)
        parameters: (4)
          - param=1
      kernelMappings: (5)
        - literal: 6.0.15-300.fc37.x86_64
          containerImage: some.registry/org/my-kmod:6.0.15-300.fc37.x86_64
        - regexp: '^.+\fc37\.x86_64$' (6)
          containerImage: "some.other.registry/org/<my_kmod>:${KERNEL_FULL_VERSION}"
        - regexp: '^.+$' (7)
          containerImage: "some.registry/org/<my_kmod>:${KERNEL_FULL_VERSION}"
          build:
            buildArgs: (8)
              - name: ARG_NAME
                value: <some_value>
            secrets:
              - name: <some_kubernetes_secret> (9)
            baseImageRegistryTLS: (10)
              insecure: false
              insecureSkipTLSVerify: false (11)
            dockerfileConfigMap: (12)
              name: <my_kmod_dockerfile>
          sign:
            certSecret:
              name: <cert_secret> (13)
            keySecret:
              name: <key_secret> (14)
            filesToSign:
```

```
            - /opt/lib/modules/${KERNEL_FULL_VERSION}/<my_kmod>.ko
        registryTLS: 15
          insecure: false 16
          insecureSkipTLSVerify: false
  serviceAccountName: <sa_module_loader> 17
devicePlugin: 18
  container:
    image: some.registry/org/device-plugin:latest 19
    env:
      - name: MY_DEVICE_PLUGIN_ENV_VAR
        value: SOME_VALUE
    volumeMounts: 20
      - mountPath: /some/mountPath
        name: <device_plugin_volume>
  volumes: 21
    - name: <device_plugin_volume>
      configMap:
        name: <some_configmap>
  serviceAccountName: <sa_device_plugin> 22
imageRepoSecret: 23
  name: <secret_name>
selector:
  node-role.kubernetes.io/worker: ""
```

1 1 1 Required.

2 Optional.

3 Optional: Copies **/firmware/*** into **/var/lib/firmware/** on the node.

4 Optional.

5 At least one kernel item is required.

6 For each node running a kernel matching the regular expression, KMM creates a **DaemonSet** resource running the image specified in **containerImage** with **${KERNEL_FULL_VERSION}** replaced with the kernel version.

7 For any other kernel, build the image using the Dockerfile in the **my-kmod** ConfigMap.

8 Optional.

9 Optional: A value for **some-kubernetes-secret** can be obtained from the build environment at **/run/secrets/some-kubernetes-secret**.

10 Optional: Avoid using this parameter. If set to **true**, the build is allowed to pull the image in the Dockerfile **FROM** instruction using plain HTTP.

11 Optional: Avoid using this parameter. If set to **true**, the build will skip any TLS server certificate validation when pulling the image in the Dockerfile **FROM** instruction using plain HTTP.

12 Required.

13 Required: A secret holding the public secureboot key with the key 'cert'.

14 Required: A secret holding the private secureboot key with the key 'key'.

15 Optional: Avoid using this parameter. If set to **true**, KMM will be allowed to check if the container image already exists using plain HTTP.

16 Optional: Avoid using this parameter. If set to **true**, KMM will skip any TLS server certificate validation when checking if the container image already exists.

17 Optional.

18 Optional.

19 Required: If the device plugin section is present.

20 Optional.

21 Optional.

22 Optional.

23 Optional: Used to pull module loader and device plugin images.

## 4.5. USING A MODULELOADER IMAGE

Kernel Module Management (KMM) works with purpose-built module loader images. These are standard OCI images that must satisfy the following requirements:

- **.ko** files must be located in **/opt/lib/modules/${KERNEL_VERSION}**.
- **modprobe** and **sleep** binaries must be defined in the **$PATH** variable.

### 4.5.1. Running depmod

If your module loader image contains several kernel modules and if one of the modules depends on another module, it is best practice to run **depmod** at the end of the build process to generate dependencies and map files.

**NOTE**

You must have a Red Hat subscription to download the **kernel-devel** package.

**Procedure**

1. To generate **modules.dep** and **.map** files for a specific kernel version, run **depmod -b /opt ${KERNEL_VERSION}**.

#### 4.5.1.1. Example Dockerfile

If you are building your image on OpenShift Container Platform, consider using the Driver Tool Kit (DTK).

For further information, see using an entitled build .

```
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: kmm-ci-dockerfile
data:
  dockerfile: |
    ARG DTK_AUTO
    FROM ${DTK_AUTO} as builder
    ARG KERNEL_VERSION
    WORKDIR /usr/src
    RUN ["git", "clone", "https://github.com/rh-ecosystem-edge/kernel-module-management.git"]
    WORKDIR /usr/src/kernel-module-management/ci/kmm-kmod
    RUN KERNEL_SRC_DIR=/lib/modules/${KERNEL_VERSION}/build make all
    FROM registry.redhat.io/ubi9/ubi-minimal
    ARG KERNEL_VERSION
    RUN microdnf install kmod
    COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_a.ko
/opt/lib/modules/${KERNEL_VERSION}/
    COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_b.ko
/opt/lib/modules/${KERNEL_VERSION}/
    RUN depmod -b /opt ${KERNEL_VERSION}
```

**Additional resources**

- Driver Toolkit.

### 4.5.2. Building in the cluster

KMM can build module loader images in the cluster. Follow these guidelines:

- Provide build instructions using the **build** section of a kernel mapping.
- Copy the **Dockerfile** for your container image into a **ConfigMap** resource, under the **dockerfile** key.
- Ensure that the **ConfigMap** is located in the same namespace as the **Module**.

KMM checks if the image name specified in the **containerImage** field exists. If it does, the build is skipped.

Otherwise, KMM creates a **Build** resource to build your image. After the image is built, KMM proceeds with the **Module** reconciliation. See the following example.

```
# ...
- regexp: '^.+$'
  containerImage: "some.registry/org/<my_kmod>:${KERNEL_FULL_VERSION}"
  build:
    buildArgs: (1)
      - name: ARG_NAME
        value: <some_value>
    secrets: (2)
      - name: <some_kubernetes_secret> (3)
    baseImageRegistryTLS:
      insecure: false (4)
      insecureSkipTLSVerify: false (5)
    dockerfileConfigMap: (6)
```

```
    name: <my_kmod_dockerfile>
registryTLS:
  insecure: false 7
  insecureSkipTLSVerify: false 8
```

1. Optional.
2. Optional.
3. Will be mounted in the build pod as **/run/secrets/some-kubernetes-secret**.
4. Optional: Avoid using this parameter. If set to **true**, the build will be allowed to pull the image in the Dockerfile **FROM** instruction using plain HTTP.
5. Optional: Avoid using this parameter. If set to **true**, the build will skip any TLS server certificate validation when pulling the image in the Dockerfile **FROM** instruction using plain HTTP.
6. Required.
7. Optional: Avoid using this parameter. If set to **true**, KMM will be allowed to check if the container image already exists using plain HTTP.
8. Optional: Avoid using this parameter. If set to **true**, KMM will skip any TLS server certificate validation when checking if the container image already exists.

#### Additional resources

- Build configuration resources.

### 4.5.3. Using the Driver Toolkit

The Driver Toolkit (DTK) is a convenient base image for building build module loader images. It contains tools and libraries for the OpenShift version currently running in the cluster.

#### Procedure

Use DTK as the first stage of a multi-stage **Dockerfile**.

1. Build the kernel modules.
2. Copy the **.ko** files into a smaller end-user image such as **ubi-minimal**.
3. To leverage DTK in your in-cluster build, use the **DTK_AUTO** build argument. The value is automatically set by KMM when creating the **Build** resource. See the following example.

```
ARG DTK_AUTO
FROM ${DTK_AUTO} as builder
ARG KERNEL_VERSION
WORKDIR /usr/src
RUN ["git", "clone", "https://github.com/rh-ecosystem-edge/kernel-module-management.git"]
WORKDIR /usr/src/kernel-module-management/ci/kmm-kmod
RUN KERNEL_SRC_DIR=/lib/modules/${KERNEL_VERSION}/build make all
FROM registry.redhat.io/ubi9/ubi-minimal
ARG KERNEL_VERSION
RUN microdnf install kmod
```

```
COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_a.ko
/opt/lib/modules/${KERNEL_VERSION}/
COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_b.ko
/opt/lib/modules/${KERNEL_VERSION}/
RUN depmod -b /opt ${KERNEL_VERSION}
```

**Additional resources**

- Driver Toolkit.

## 4.6. USING SIGNING WITH KERNEL MODULE MANAGEMENT (KMM)

On a Secure Boot enabled system, all kernel modules (kmods) must be signed with a public/private key-pair enrolled into the Machine Owner's Key (MOK) database. Drivers distributed as part of a distribution should already be signed by the distribution's private key, but for kernel modules build out-of-tree, KMM supports signing kernel modules using the **sign** section of the kernel mapping.

For more details on using Secure Boot, see Generating a public and private key pair

**Prerequisites**

- A public private key pair in the correct (DER) format.
- At least one secure-boot enabled node with the public key enrolled in its MOK database.
- Either a pre-built driver container image, or the source code and **Dockerfile** needed to build one in-cluster.

## 4.7. ADDING THE KEYS FOR SECUREBOOT

To use KMM Kernel Module Management (KMM) to sign kernel modules, a certificate and private key are required. For details on how to create these, see Generating a public and private key pair .

For details on how to extract the public and private key pair, see Signing kernel modules with the private key. Use steps 1 through 4 to extract the keys into files.

**Procedure**

1. Create the **sb_cert.cer** file that contains the certificate and the **sb_cert.priv** file that contains the private key:

```
$ openssl req -x509 -new -nodes -utf8 -sha256 -days 36500 -batch -config
configuration_file.config -outform DER -out my_signing_key_pub.der -keyout
my_signing_key.priv
```

2. Add the files by using one of the following methods:
   - Add the files as secrets directly:

```
$ oc create secret generic my-signing-key --from-file=key=<my_signing_key.priv>
```

```
$ oc create secret generic my-signing-key-pub --from-file=key=
<my_signing_key_pub.der>
```

- Add the files by base64 encoding them:

```
$ cat sb_cert.priv | base64 -w 0 > my_signing_key2.base64
```

```
$ cat sb_cert.cer | base64 -w 0 > my_signing_key_pub.base64
```

3. Add the encoded text to a YAML file:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-signing-key-pub
  namespace: default 1
type: Opaque
data:
  cert: <base64_encoded_secureboot_public_key>

---
apiVersion: v1
kind: Secret
metadata:
  name: my-signing-key
  namespace: default 2
type: Opaque
data:
  key: <base64_encoded_secureboot_private_key>
```

1 2 **namespace** - Replace **default** with a valid namespace.

4. Apply the YAML file:

```
$ oc apply -f <yaml_filename>
```

### 4.7.1. Checking the keys

After you have added the keys, you must check them to ensure they are set correctly.

**Procedure**

1. Check to ensure the public key secret is set correctly:

```
$ oc get secret -o yaml <certificate secret name> | awk '/cert/{print $2; exit}' | base64 -d  |
openssl x509 -inform der -text
```

This should display a certificate with a Serial Number, Issuer, Subject, and more.

2. Check to ensure the private key secret is set correctly:

```
$ oc get secret -o yaml <private key secret name> | awk '/key/{print $2; exit}' | base64 -d
```

This should display the key enclosed in the **-----BEGIN PRIVATE KEY-----** and **-----END PRIVATE KEY-----** lines.

## 4.8. SIGNING A PRE-BUILT DRIVER CONTAINER

Use this procedure if you have a pre-built image, such as an image either distributed by a hardware vendor or built elsewhere.

The following YAML file adds the public/private key-pair as secrets with the required key names - **key** for the private key, **cert** for the public key. The cluster then pulls down the **unsignedImage** image, opens it, signs the kernel modules listed in **filesToSign**, adds them back, and pushes the resulting image as **containerImage**.

Kernel Module Management (KMM) should then deploy the DaemonSet that loads the signed kmods onto all the nodes that match the selector. The driver containers should run successfully on any nodes that have the public key in their MOK database, and any nodes that are not secure-boot enabled, which ignore the signature. They should fail to load on any that have secure-boot enabled but do not have that key in their MOK database.

### Prerequisites

- The **keySecret** and **certSecret** secrets have been created.

### Procedure

1. Apply the YAML file:

```
---
apiVersion: kmm.sigs.x-k8s.io/v1beta1
kind: Module
metadata:
  name: example-module
spec:
  moduleLoader:
    serviceAccountName: default
    container:
      modprobe: (1)
        moduleName: '<your module name>'
      kernelMappings:
        # the kmods will be deployed on all nodes in the cluster with a kernel that matches the
regexp
        - regexp: '^.*\.x86_64$'
          # the container to produce containing the signed kmods
          containerImage: <image name e.g. quay.io/myuser/my-driver:<kernelversion>-signed>
          sign:
            # the image containing the unsigned kmods (we need this because we are not
building the kmods within the cluster)
            unsignedImage: <image name e.g. quay.io/myuser/my-driver:<kernelversion> >
            keySecret: # a secret holding the private secureboot key with the key 'key'
              name: <private key secret name>
            certSecret: # a secret holding the public secureboot key with the key 'cert'
              name: <certificate secret name>
            filesToSign: # full path within the unsignedImage container to the kmod(s) to sign
              - /opt/lib/modules/4.18.0-348.2.1.el8_5.x86_64/kmm_ci_a.ko
  imageRepoSecret:
    # the name of a secret containing credentials to pull unsignedImage and push
containerImage to the registry
```

```
    name: repo-pull-secret
  selector:
    kubernetes.io/arch: amd64
```

1 **modprobe** - The name of the kmod to load.

## 4.9. BUILDING AND SIGNING A MODULELOADER CONTAINER IMAGE

Use this procedure if you have source code and must build your image first.

The following YAML file builds a new container image using the source code from the repository. The image produced is saved back in the registry with a temporary name, and this temporary image is then signed using the parameters in the **sign** section.

The temporary image name is based on the final image name and is set to be **<containerImage>:<tag>-<namespace>_<module name>_kmm_unsigned**.

For example, using the following YAML file, Kernel Module Management (KMM) builds an image named **example.org/repository/minimal-driver:final-default_example-module_kmm_unsigned** containing the build with unsigned kmods and push it to the registry. Then it creates a second image named **example.org/repository/minimal-driver:final** that contains the signed kmods. It is this second image that is loaded by the **DaemonSet** object and deploys the kmods to the cluster nodes.

After it is signed, the temporary image can be safely deleted from the registry. It will be rebuilt, if needed.

### Prerequisites

- The **keySecret** and **certSecret** secrets have been created.

### Procedure

1. Apply the YAML file:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-module-dockerfile
  namespace: default 1
data:
  Dockerfile: |
    ARG DTK_AUTO
    ARG KERNEL_VERSION
    FROM ${DTK_AUTO} as builder
    WORKDIR /build/
    RUN git clone -b main --single-branch https://github.com/rh-ecosystem-edge/kernel-module-management.git
    WORKDIR kernel-module-management/ci/kmm-kmod/
    RUN make
    FROM registry.access.redhat.com/ubi9/ubi:latest
    ARG KERNEL_VERSION
    RUN yum -y install kmod && yum clean all
    RUN mkdir -p /opt/lib/modules/${KERNEL_VERSION}
    COPY --from=builder /build/kernel-module-management/ci/kmm-kmod/*.ko
```

```
/opt/lib/modules/${KERNEL_VERSION}/
    RUN /usr/sbin/depmod -b /opt
---
apiVersion: kmm.sigs.x-k8s.io/v1beta1
kind: Module
metadata:
  name: example-module
  namespace: default (2)
spec:
  moduleLoader:
    serviceAccountName: default (3)
    container:
      modprobe:
        moduleName: simple_kmod
      kernelMappings:
        - regexp: '^.*\.x86_64$'
          containerImage: < the name of the final driver container to produce>
          build:
            dockerfileConfigMap:
              name: example-module-dockerfile
          sign:
            keySecret:
              name: <private key secret name>
            certSecret:
              name: <certificate secret name>
            filesToSign:
              - /opt/lib/modules/4.18.0-348.2.1.el8_5.x86_64/kmm_ci_a.ko
  imageRepoSecret: (4)
    name: repo-pull-secret
  selector: # top-level selector
    kubernetes.io/arch: amd64
```

(1) (2) **namespace** - Replace **default** with a valid namespace.

(3) **serviceAccountName** - The default **serviceAccountName** does not have the required permissions to run a module that is privileged. For information on creating a service account, see "Creating service accounts" in the "Additional resources" of this section.

(4) **imageRepoSecret** - Used as **imagePullSecrets** in the **DaemonSet** object and to pull and push for the build and sign features.

### Additional resources

For information on creating a service account, see Creating service accounts .

## 4.10. CUSTOMIZING UPGRADES FOR KERNEL MODULES

Use this procedure to upgrade the kernel module while running maintenance operations on the node, including rebooting the node, if needed. To minimize the impact on the workloads running in the cluster, run the kernel upgrade process sequentially, one node at a time.

NOTE

This procedure requires knowledge of the workload utilizing the kernel module and must be managed by the cluster administrator.

**Prerequisites**

- Before upgrading, set the **kmm.node.kubernetes.io/version-module.<module_namespace>.<module_name>=$moduleVersion** label on all the nodes that are used by the kernel module.
- Terminate all user application workloads on the node or move them to another node.
- Unload the currently loaded kernel module.
- Ensure that the user workload (the application running in the cluster that is accessing kernel module) is not running on the node prior to kernel module unloading and that the workload is back running on the node after the new kernel module version has been loaded.

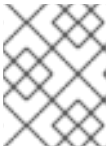
**Procedure**

1. Ensure that the device plugin managed by KMM on the node is unloaded.
2. Update the following fields in the **Module** custom resource (CR):
   - **containerImage** (to the appropriate kernel version)
   - **version**
     The update should be atomic; that is, both the **containerImage** and **version** fields must be updated simultaneously.
3. Terminate any workload using the kernel module on the node being upgraded.
4. Remove the **kmm.node.kubernetes.io/version-module.<module_namespace>.<module_name>** label on the node. Run the following command to unload the kernel module from the node:

   ```
   $ oc label node/<node_name> kmm.node.kubernetes.io/version-module.
   <module_namespace>.<module_name>-
   ```

5. If required, as the cluster administrator, perform any additional maintenance required on the node for the kernel module upgrade.
   If no additional upgrading is needed, you can skip Steps 3 through 6 by updating the **kmm.node.kubernetes.io/version-module.<module-namespace>.<module-name>** label value to the new **$moduleVersion** as set in the **Module**.
6. Run the following command to add the **kmm.node.kubernetes.io/version-module.<module_namespace>.<module_name>=$moduleVersion** label to the node. The **$moduleVersion** must be equal to the new value of the **version** field in the **Module** CR.

   ```
   $ oc label node/<node_name> kmm.node.kubernetes.io/version-module.
   <module_namespace>.<module_name>=<desired_version>
   ```



NOTE

Because of Kubernetes limitations in label names, the combined length of **Module** name and namespace must not exceed 39 characters.

7. Restore any workload that leverages the kernel module on the node.

8. Reload the device plugin managed by KMM on the node.

## 4.11. DAY 1 KERNEL MODULE LOADING

Kernel Module Management (KMM) is typically a Day 2 Operator. Kernel modules are loaded only after the complete initialization of a Linux (RHCOS) server. However, in some scenarios the kernel module must be loaded at an earlier stage. Day 1 functionality allows you to use the Machine Config Operator (MCO) to load kernel modules during the Linux **systemd** initialization stage.

**Additional resources**

- Machine Config Operator

### 4.11.1. Day 1 supported use cases

The Day 1 functionality supports a limited number of use cases. The main use case is to allow loading out-of-tree (OOT) kernel modules prior to NetworkManager service initialization. It does not support loading kernel module at the **initramfs** stage.

The following are the conditions needed for Day 1 functionality:

- The kernel module is not loaded in the kernel.
- The in-tree kernel module is loaded into the kernel, but can be unloaded and replaced by the OOT kernel module. This means that the in-tree module is not referenced by any other kernel modules.
- In order for Day 1 functionlity to work, the node must have a functional network interface, that is, an in-tree kernel driver for that interface. The OOT kernel module can be a network driver that will replace the functional network driver.

### 4.11.2. OOT kernel module loading flow

The loading of the out-of-tree (OOT) kernel module leverages the Machine Config Operator (MCO). The flow sequence is as follows:

**Procedure**

1. Apply a **MachineConfig** resource to the existing running cluster. In order to identify the necessary nodes that need to be updated, you must create an appropriate **MachineConfigPool** resource.

2. MCO applies the reboots node by node. On any rebooted node, two new **systemd** services are deployed: **pull** service and **load** service.

3. The **load** service is configured to run prior to the **NetworkConfiguration** service. The service tries to pull a predefined kernel module image and then, using that image, to unload an in-tree module and load an OOT kernel module.

4. The **pull** service is configured to run after NetworkManager service. The service checks if the preconfigured kernel module image is located on the node's filesystem. If it is, the service exists normally, and the server continues with the boot process. If not, it pulls the image onto the node and reboots the node afterwards.

### 4.11.3. The kernel module image

The Day 1 functionality uses the same DTK based image leveraged by Day 2 KMM builds. The out-of-tree kernel module should be located under **/opt/lib/modules/${kernelVersion}**.

#### Additional resources

- Driver Toolkit

### 4.11.4. In-tree module replacement

The Day 1 functionality always tries to replace the in-tree kernel module with the OOT version. If the in-tree kernel module is not loaded, the flow is not affected; the service proceeds and loads the OOT kernel module.

### 4.11.5. MCO yaml creation

KMM provides an API to create an MCO YAML manifest for the Day 1 functionality:

```
ProduceMachineConfig(machineConfigName, machineConfigPoolRef, kernelModuleImage,
kernelModuleName string) (string, error)
```

The returned output is a string representation of the MCO YAML manifest to be applied. It is up to the customer to apply this YAML.

The parameters are:

**machineConfigName**

The name of the MCO YAML manifest. This parameter is set as the **name** parameter of the metadata of the MCO YAML manifest.

**machineConfigPoolRef**

The **MachineConfigPool** name used to identify the targeted nodes.

**kernelModuleImage**

The name of the container image that includes the OOT kernel module.

**kernelModuleName**

The name of the OOT kernel module. This parameter is used both to unload the in-tree kernel module (if loaded into the kernel) and to load the OOT kernel module.

The API is located under **pkg/mcproducer** package of the KMM source code. The KMM operator does not need to be running to use the Day 1 functionality. You only need to import the **pkg/mcproducer** package into their operator/utility code, call the API, and apply the produced MCO YAML to the cluster.

### 4.11.6. The MachineConfigPool

The **MachineConfigPool** identifies a collection of nodes that are affected by the applied MCO.

```
kind: MachineConfigPool
metadata:
  name: sfc
spec:
  machineConfigSelector: 1
    matchExpressions:
      - {key: machineconfiguration.openshift.io/role, operator: In, values: [worker, sfc]}
  nodeSelector: 2
    matchLabels:
      node-role.kubernetes.io/sfc: ""
  paused: false
  maxUnavailable: 1
```

1 Matches the labels in the MachineConfig.

2 Matches the labels on the node.

There are predefined **MachineConfigPools** in the OCP cluster:

- **worker**: Targets all worker nodes in the cluster
- **master**: Targets all master nodes in the cluster

Define the following **MachineConfig** to target the master **MachineConfigPool**:

```
metadata:
  labels:
    machineconfiguration.opensfhit.io/role: master
```

Define the following **MachineConfig** to target the worker **MachineConfigPool**:

```
metadata:
  labels:
    machineconfiguration.opensfhit.io/role: worker
```

**Additional resources**

- About MachineConfigPool

## 4.12. DEBUGGING AND TROUBLESHOOTING

If the kmods in your driver container are not signed or are signed with the wrong key, then the container can enter a **PostStartHookError** or **CrashLoopBackOff** status. You can verify by running the **oc describe** command on your container, which displays the following message in this scenario:

```
modprobe: ERROR: could not insert '<your_kmod_name>': Required key not available
```

## 4.13. KMM FIRMWARE SUPPORT

Kernel modules sometimes need to load firmware files from the file system. KMM supports copying firmware files from the ModuleLoader image to the node's file system.

The contents of **.spec.moduleLoader.container.modprobe.firmwarePath** are copied into the **/var/lib/firmware** path on the node before running the **modprobe** command to insert the kernel module.

All files and empty directories are removed from that location before running the **modprobe -r** command to unload the kernel module, when the pod is terminated.

### Additional resources

- Creating a ModuleLoader image.

## 4.13.1. Configuring the lookup path on nodes

On OpenShift Container Platform nodes, the set of default lookup paths for firmwares does not include the **/var/lib/firmware** path.

### Procedure

1. Use the Machine Config Operator to create a **MachineConfig** custom resource (CR) that contains the **/var/lib/firmware** path:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker (1)
  name: 99-worker-kernel-args-firmware-path
spec:
  kernelArguments:
    - 'firmware_class.path=/var/lib/firmware'
```

(1) You can configure the label based on your needs. In the case of single-node OpenShift, use either **control-pane** or **master** objects.

2. By applying the **MachineConfig** CR, the nodes are automatically rebooted.

### Additional resources

- Machine Config Operator.

## 4.13.2. Building a ModuleLoader image

### Procedure

- In addition to building the kernel module itself, include the binary firmware in the builder image:

```
FROM registry.redhat.io/ubi9/ubi-minimal as builder

# Build the kmod

RUN ["mkdir", "/firmware"]
RUN ["curl", "-o", "/firmware/firmware.bin", "https://artifacts.example.com/firmware.bin"]
```

```
FROM registry.redhat.io/ubi9/ubi-minimal

# Copy the kmod, install modprobe, run depmod

COPY --from=builder /firmware /firmware
```

### 4.13.3. Tuning the Module resource

**Procedure**

- Set **.spec.moduleLoader.container.modprobe.firmwarePath** in the **Module** custom resource (CR):

```
apiVersion: kmm.sigs.x-k8s.io/v1beta1
kind: Module
metadata:
  name: my-kmod
spec:
  moduleLoader:
    container:
      modprobe:
        moduleName: my-kmod  # Required

        firmwarePath: /firmware (1)
```

(1) Optional: Copies **/firmware/*** into **/var/lib/firmware/** on the node.

## 4.14. TROUBLESHOOTING KMM

When troubleshooting KMM installation issues, you can monitor logs to determine at which stage issues occur. Then, retrieve diagnostic data relevant to that stage.

### 4.14.1. Using the must-gather tool

The **oc adm must-gather** command is the preferred way to collect a support bundle and provide debugging information to Red Hat Support. Collect specific information by running the command with the appropriate arguments as described in the following sections.

**Additional resources**

- About the must-gather tool

#### 4.14.1.1. Gathering data for KMM

**Procedure**

1. Gather the data for the KMM Operator controller manager:

   a. Set the **MUST_GATHER_IMAGE** variable:

```
$ export MUST_GATHER_IMAGE=$(oc get deployment -n openshift-kmm kmm-operator-controller-manager -ojsonpath='{.spec.template.spec.containers[?
```

```
(@.name=="manager")].env[?
(@.name=="RELATED_IMAGES_MUST_GATHER")].value}')
```

**NOTE**

Use the **-n <namespace>** switch to specify a namespace if you installed KMM in a custom namespace.

b. Run the **must-gather** tool:

```
$ oc adm must-gather --image="${MUST_GATHER_IMAGE}" -- /usr/bin/gather
```

2. View the Operator logs:

```
$ oc logs -fn openshift-kmm deployments/kmm-operator-controller-manager
```

**Example 4.1. Example output**

```
I0228 09:36:37.352405       1 request.go:682] Waited for 1.001998746s due to client-side
throttling, not priority and fairness, request:
GET:https://172.30.0.1:443/apis/machine.openshift.io/v1beta1?timeout=32s
I0228 09:36:40.767060       1 listener.go:44] kmm/controller-runtime/metrics
"msg"="Metrics server is starting to listen" "addr"="127.0.0.1:8080"
I0228 09:36:40.769483       1 main.go:234] kmm/setup "msg"="starting manager"
I0228 09:36:40.769907       1 internal.go:366] kmm "msg"="Starting server" "addr"=
{"IP":"127.0.0.1","Port":8080,"Zone":""} "kind"="metrics" "path"="/metrics"
I0228 09:36:40.770025       1 internal.go:366] kmm "msg"="Starting server" "addr"=
{"IP":"::","Port":8081,"Zone":""} "kind"="health probe"
I0228 09:36:40.770128       1 leaderelection.go:248] attempting to acquire leader lease
openshift-kmm/kmm.sigs.x-k8s.io...
I0228 09:36:40.784396       1 leaderelection.go:258] successfully acquired lease
openshift-kmm/kmm.sigs.x-k8s.io
I0228 09:36:40.784876       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module"
"source"="kind source: *v1beta1.Module"
I0228 09:36:40.784925       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module"
"source"="kind source: *v1.DaemonSet"
I0228 09:36:40.784968       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module"
"source"="kind source: *v1.Build"
I0228 09:36:40.785001       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module"
"source"="kind source: *v1.Job"
I0228 09:36:40.785025       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module"
"source"="kind source: *v1.Node"
I0228 09:36:40.785039       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module"
I0228 09:36:40.785458       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="PodNodeModule" "controllerGroup"="" "controllerKind"="Pod" "source"="kind
source: *v1.Pod"
I0228 09:36:40.786947       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="PreflightValidation" "controllerGroup"="kmm.sigs.x-k8s.io"
```

```
"controllerKind"="PreflightValidation" "source"="kind source: *v1beta1.PreflightValidation"
I0228 09:36:40.787406       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="PreflightValidation" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidation" "source"="kind source: *v1.Build"
I0228 09:36:40.787474       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="PreflightValidation" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidation" "source"="kind source: *v1.Job"
I0228 09:36:40.787488       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="PreflightValidation" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidation" "source"="kind source: *v1beta1.Module"
I0228 09:36:40.787603       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="NodeKernel" "controllerGroup"="" "controllerKind"="Node" "source"="kind
source: *v1.Node"
I0228 09:36:40.787634       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="NodeKernel" "controllerGroup"="" "controllerKind"="Node"
I0228 09:36:40.787680       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="PreflightValidation" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidation"
I0228 09:36:40.785607       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="imagestream" "controllerGroup"="image.openshift.io"
"controllerKind"="ImageStream" "source"="kind source: *v1.ImageStream"
I0228 09:36:40.787822       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="preflightvalidationocp" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidationOCP" "source"="kind source:
*v1beta1.PreflightValidationOCP"
I0228 09:36:40.787853       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="imagestream" "controllerGroup"="image.openshift.io"
"controllerKind"="ImageStream"
I0228 09:36:40.787879       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="preflightvalidationocp" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidationOCP" "source"="kind source:
*v1beta1.PreflightValidation"
I0228 09:36:40.787905       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="preflightvalidationocp" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidationOCP"
I0228 09:36:40.786489       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="PodNodeModule" "controllerGroup"="" "controllerKind"="Pod"
```

#### 4.14.1.2. Gathering data for KMM-Hub

**Procedure**

1. Gather the data for the KMM Operator hub controller manager:

   a. Set the **MUST_GATHER_IMAGE** variable:

   ```
   $ export MUST_GATHER_IMAGE=$(oc get deployment -n openshift-kmm-hub kmm-
   operator-hub-controller-manager -ojsonpath='{.spec.template.spec.containers[?
   (@.name=="manager")].env[?
   (@.name=="RELATED_IMAGES_MUST_GATHER")].value}')
   ```

NOTE

Use the **-n <namespace>** switch to specify a namespace if you installed KMM in a custom namespace.

b. Run the **must-gather** tool:

```
$ oc adm must-gather --image="${MUST_GATHER_IMAGE}" -- /usr/bin/gather -u
```

2. View the Operator logs:

```
$ oc logs -fn openshift-kmm-hub deployments/kmm-operator-hub-controller-manager
```

**Example 4.2. Example output**

```
I0417 11:34:08.807472       1 request.go:682] Waited for 1.023403273s due to client-side throttling, not priority and fairness, request: GET:https://172.30.0.1:443/apis/tuned.openshift.io/v1?timeout=32s
I0417 11:34:12.373413       1 listener.go:44] kmm-hub/controller-runtime/metrics "msg"="Metrics server is starting to listen" "addr"="127.0.0.1:8080"
I0417 11:34:12.376253       1 main.go:150] kmm-hub/setup "msg"="Adding controller" "name"="ManagedClusterModule"
I0417 11:34:12.376621       1 main.go:186] kmm-hub/setup "msg"="starting manager"
I0417 11:34:12.377690       1 leaderelection.go:248] attempting to acquire leader lease openshift-kmm-hub/kmm-hub.sigs.x-k8s.io...
I0417 11:34:12.378078       1 internal.go:366] kmm-hub "msg"="Starting server" "addr"={"IP":"127.0.0.1","Port":8080,"Zone":""} "kind"="metrics" "path"="/metrics"
I0417 11:34:12.378222       1 internal.go:366] kmm-hub "msg"="Starting server" "addr"={"IP":"::","Port":8081,"Zone":""} "kind"="health probe"
I0417 11:34:12.395703       1 leaderelection.go:258] successfully acquired lease openshift-kmm-hub/kmm-hub.sigs.x-k8s.io
I0417 11:34:12.396334       1 controller.go:185] kmm-hub "msg"="Starting EventSource" "controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io" "controllerKind"="ManagedClusterModule" "source"="kind source: *v1beta1.ManagedClusterModule"
I0417 11:34:12.396403       1 controller.go:185] kmm-hub "msg"="Starting EventSource" "controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io" "controllerKind"="ManagedClusterModule" "source"="kind source: *v1.ManifestWork"
I0417 11:34:12.396430       1 controller.go:185] kmm-hub "msg"="Starting EventSource" "controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io" "controllerKind"="ManagedClusterModule" "source"="kind source: *v1.Build"
I0417 11:34:12.396469       1 controller.go:185] kmm-hub "msg"="Starting EventSource" "controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io" "controllerKind"="ManagedClusterModule" "source"="kind source: *v1.Job"
I0417 11:34:12.396522       1 controller.go:185] kmm-hub "msg"="Starting EventSource" "controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io" "controllerKind"="ManagedClusterModule" "source"="kind source: *v1.ManagedCluster"
I0417 11:34:12.396543       1 controller.go:193] kmm-hub "msg"="Starting Controller" "controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io" "controllerKind"="ManagedClusterModule"
I0417 11:34:12.397175       1 controller.go:185] kmm-hub "msg"="Starting EventSource" "controller"="imagestream" "controllerGroup"="image.openshift.io" "controllerKind"="ImageStream" "source"="kind source: *v1.ImageStream"
I0417 11:34:12.397221       1 controller.go:193] kmm-hub "msg"="Starting Controller" "controller"="imagestream" "controllerGroup"="image.openshift.io"
```

```
"controllerKind"="ImageStream"
I0417 11:34:12.498335       1 filter.go:196] kmm-hub "msg"="Listing all
ManagedClusterModules" "managedcluster"="local-cluster"
I0417 11:34:12.498570       1 filter.go:205] kmm-hub "msg"="Listed
ManagedClusterModules" "count"=0 "managedcluster"="local-cluster"
I0417 11:34:12.498629       1 filter.go:238] kmm-hub "msg"="Adding reconciliation
requests" "count"=0 "managedcluster"="local-cluster"
I0417 11:34:12.498687       1 filter.go:196] kmm-hub "msg"="Listing all
ManagedClusterModules" "managedcluster"="sno1-0"
I0417 11:34:12.498750       1 filter.go:205] kmm-hub "msg"="Listed
ManagedClusterModules" "count"=0 "managedcluster"="sno1-0"
I0417 11:34:12.498801       1 filter.go:238] kmm-hub "msg"="Adding reconciliation
requests" "count"=0 "managedcluster"="sno1-0"
I0417 11:34:12.501947       1 controller.go:227] kmm-hub "msg"="Starting workers"
"controller"="imagestream" "controllerGroup"="image.openshift.io"
"controllerKind"="ImageStream" "worker count"=1
I0417 11:34:12.501948       1 controller.go:227] kmm-hub "msg"="Starting workers"
"controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io"
"controllerKind"="ManagedClusterModule" "worker count"=1
I0417 11:34:12.502285       1 imagestream_reconciler.go:50] kmm-hub "msg"="registered
imagestream info mapping" "ImageStream"={"name":"driver-
toolkit","namespace":"openshift"} "controller"="imagestream"
"controllerGroup"="image.openshift.io" "controllerKind"="ImageStream"
"dtkImage"="quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:df42b4785a7a662b30da53bdb0d206120cf4d24b45674227b16051ba4b7c393
4" "name"="driver-toolkit" "namespace"="openshift"
"osImageVersion"="412.86.202302211547-0" "reconcileID"="e709ff0a-5664-4007-8270-
49b5dff8bae9"
```