OpenShift Container Platform 4.9

Specialized hardware and driver enablement

Learn about hardware enablement on OpenShift Container Platform

Last Updated: 2022-01-31

OpenShift Container Platform 4.9 Specialized hardware and driver enablement

Learn about hardware enablement on OpenShift Container Platform

Abstract

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

Table of Contents

CHAPTER 1. ABOUT SPECIALIZED HARDWARE AND DRIVER ENABLEMENT

Many applications require specialized hardware or software that depends on kernel modules or drivers. You can use driver containers to load out-of-tree kernel modules on Red Hat Enterprise Linux CoreOS (RHCOS) nodes. To deploy out-of-tree drivers during cluster installation, use the **kmods-via-containers** framework. To load drivers or kernel modules on an existing OpenShift Container Platform cluster, OpenShift Container Platform offers several tools:

- The Driver Toolkit is a container image that is a part of every OpenShift Container Platform release. It contains the kernel packages and other common dependencies that are needed to build a driver or kernel module. The Driver Toolkit can be used as a base image for driver container image builds on OpenShift Container Platform.
- The Special Resource Operator (SRO) orchestrates the building and management of driver containers to load kernel modules and drivers on an existing OpenShift or Kubernetes cluster.
- The Node Feature Discovery (NFD) Operator adds node labels for CPU capabilities, kernel version, PCIe device vendor IDs, and more.

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 Kubernetes.

IMPORTANT

The Driver Toolkit is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

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 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 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 which 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, you could 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 Special Resource Operator (SRO), which is currently available as a community Operator on OperatorHub. SRO supports out-of-tree and third-party kernel drivers and the support software for the underlying operating system. Users can create *recipes* for SRO to build and deploy a driver container, as well as support software like a device plug-in, or metrics. Recipes can include a build config to build a driver container based on the Driver Toolkit, or SRO 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 will be tagged with the minor release version on registry.redhat.io for example **registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.9**.

2.2.2. Finding the Driver Toolkit image URL in the payload

Prerequisites

- You obtained the image pull secret needed to perform an installation of OpenShift Container Platform, from the Pull Secret page on the Red Hat OpenShift Cluster Manager site.
- You installed the OpenShift CLI (**oc**).

Procedure

1. The image URL of the **driver-toolkit** corresponding to a certain release can be extracted from the release image using the **oc adm** command:

```
$ oc adm release info 4.9.0 --image-for=driver-toolkit
```

Example output

```
quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:0fd84aee79606178b6561ac71f8540f404d518ae5deff45f6d6ac8f02636c7f4
```

2. This image can be pulled 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 contains 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:
    git:
      ref: "master"
      uri: "https://github.com/openshift-psap/kvc-simple-kmod.git"
    type: Git
    dockerfile: |
      FROM DRIVER_TOOLKIT_IMAGE

      WORKDIR /build/

      RUN yum -y install git make sudo gcc \
      && yum clean all \
      && rm -rf /var/cache/dnf

      # Expecting kmod software version as an input to the build
      ARG KMODVER

      # Grab the software from upstream
      RUN git clone https://github.com/openshift-psap/simple-kmod.git
      WORKDIR simple-kmod

      # Prep and build the module
      RUN make buildprep KVER=$(rpm -q --qf "%{VERSION}-%{RELEASE}.%{ARCH}"
kernel-core) KMODVER=${KMODVER} \
      && make all       KVER=$(rpm -q --qf "%{VERSION}-%{RELEASE}.%{ARCH}"  kernel-
core) KMODVER=${KMODVER} \
      && make install   KVER=$(rpm -q --qf "%{VERSION}-%{RELEASE}.%{ARCH}"  kernel-
core) KMODVER=${KMODVER}

      # Add the helper tools
      WORKDIR /root/kvc-simple-kmod
      ADD Makefile .
      ADD simple-kmod-lib.sh .
      ADD simple-kmod-wrapper.sh .
      ADD simple-kmod.conf .
      RUN mkdir -p /usr/lib/kvc/ \
      && mkdir -p /etc/kvc/ \
      && make install

      RUN systemctl enable kmods-via-containers@simple-kmod
```

```
strategy:
  dockerStrategy:
    buildArgs:
      - name: KMODVER
        value: DEMO
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: ["/sbin/init"]
        lifecycle:
          preStop:
            exec:
              command: ["/bin/sh", "-c", "systemctl stop kmods-via-containers@simple-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. SPECIAL RESOURCE OPERATOR

Learn about the Special Resource Operator (SRO) and how you can use it to build and manage driver containers for loading kernel modules and device drivers on nodes in an OpenShift Container Platform cluster.

IMPORTANT

The Special Resource Operator is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

3.1. ABOUT THE SPECIAL RESOURCE OPERATOR

The Special Resource Operator (SRO) helps you manage the deployment of kernel modules and drivers on an existing OpenShift Container Platform cluster. The SRO can be used for a case as simple as building and loading a single kernel module, or as complex as deploying the driver, device plug-in, and monitoring stack for a hardware accelerator.

For loading kernel modules, the SRO is designed around the use of driver containers. Driver containers are increasingly being used in cloud-native environments, especially when run on pure container operating systems, to deliver hardware drivers to the host. Driver containers extend the kernel stack beyond the out-of-the-box software and hardware features of a specific kernel. Driver containers work on various container-capable Linux distributions. With driver containers, the host operating system stays clean and there is no clash between different library versions or binaries on the host.

3.2. INSTALLING THE SPECIAL RESOURCE OPERATOR

As a cluster administrator, you can install the Special Resource Operator (SRO) by using the OpenShift CLI or the web console.

3.2.1. Installing the Special Resource Operator by using the CLI

As a cluster administrator, you can install the Special Resource Operator (SRO) 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.
- You installed the Node Feature Discovery (NFD) Operator.

Procedure

1. Create a namespace for the Special Resource Operator:

    a. Create the following **Namespace** custom resource (CR) that defines the **openshift-special-resource-operator** namespace, and then save the YAML in the **sro-namespace.yaml** file:

    ```
    apiVersion: v1
    kind: Namespace
    metadata:
      name: openshift-special-resource-operator
    ```

    b. Create the namespace by running the following command:

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

2. Install the SRO in the namespace you created in the previous step:

    a. Create the following **OperatorGroup** CR and save the YAML in the **sro-operatorgroup.yaml** file:

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

    b. Create the operator group by running the following command:

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

    c. Run the following **oc get** command to get the **channel** value required for the next step:

    ```
    $ oc get packagemanifest openshift-special-resource-operator -n openshift-marketplace -
    o jsonpath='{.status.defaultChannel}'
    ```

    Example output

    ```
    4.9
    ```

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

    Example Subscription CR

    ```
    apiVersion: operators.coreos.com/v1alpha1
    kind: Subscription
    metadata:
      name: openshift-special-resource-operator
      namespace: openshift-special-resource-operator
    spec:
    ```

```
channel: "4.9" (1)
installPlanApproval: Automatic
name: openshift-special-resource-operator
source: redhat-operators
sourceNamespace: openshift-marketplace
```

(1) Replace the channel value with the output from the previous command.

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

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

f. Switch to the **openshift-special-resource-operator** project:

```
$ oc project openshift-special-resource-operator
```

Verification

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

```
$ oc get pods
```

Example output

```
NAME                                                   READY   STATUS    RESTARTS   AGE
special-resource-controller-manager-7bfb544d45-xx62r   2/2     Running   0          2m28s
```

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

3.2.2. Installing the Special Resource Operator by using the web console

As a cluster administrator, you can install the Special Resource Operator (SRO) by using the OpenShift Container Platform web console.

Prerequisites

- You installed the Node Feature Discovery (NFD) Operator.

Procedure

1. Log in to the OpenShift Container Platform web console.
2. Create the required namespace for the Special Resource Operator:
   a. Navigate to **Administration** → **Namespaces** and click **Create Namespace**.
   b. Enter **openshift-special-resource-operator** in the **Name** field and click **Create**.
3. Install the Special Resource Operator:
   a. In the OpenShift Container Platform web console, click **Operators** → **OperatorHub**.

b. Choose **Special Resource Operator** from the list of available Operators, and then click **Install**.

c. On the **Install Operator** page, select **a specific namespace on the cluster**, select the namespace created in the previous section, and then click **Install**.

Verification

To verify that the Special Resource Operator installed successfully:

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

2. Ensure that **Special Resource Operator** is listed in the **openshift-special-resource-operator** 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.

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

   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-special-resource-operator** project.

> **NOTE**
>
> The Node Feature Discovery (NFD) Operator is a dependency of the Special Resource Operator (SRO). If the NFD Operator is not installed before installing the SRO, the Operator Lifecycle Manager will automatically install the NFD Operator. However, the required Node Feature Discovery operand will not be deployed automatically. The Node Feature Discovery Operator documentation provides details about how to deploy NFD by using the NFD Operator.

3.3. USING THE SPECIAL RESOURCE OPERATOR

The Special Resource Operator (SRO) is used to manage the build and deployment of a driver container. The objects required to build and deploy the container can be defined in a Helm chart.

The examples in this section use the simple-kmod kernel module to demonstrate how to use the SRO to build and run a driver container. In the first example, the SRO image contains a local repository of Helm charts including the templates for deploying the simple-kmod kernel module. In this case, a **SpecialResource** manifest is used to deploy the driver container. In the second example, the simple-kmod **SpecialResource** object points to a **ConfigMap** object that is created to store the Helm charts.

3.3.1. Building and running the simple-kmod SpecialResource by using the templates from the SRO image

The SRO image contains a local repository of Helm charts including the templates for deploying the simple-kmod kernel module. In this example, the simple-kmod kernel module is used to show how the SRO can manage a driver container that is defined in the internal SRO repository.

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.
- You installed the Node Feature Discovery (NFD) Operator.
- You installed the Special Resource Operator.

Procedure

1. To deploy the simple-kmod using the SRO image's local Helm repository, use the following **SpecialResource** manifest. Save this YAML as **simple-kmod-local.yaml**.

```
apiVersion: sro.openshift.io/v1beta1
kind: SpecialResource
metadata:
  name: simple-kmod
spec:
  namespace: simple-kmod
  chart:
    name: simple-kmod
    version: 0.0.1
    repository:
      name: example
      url: file:///charts/example
  set:
    kind: Values
    apiVersion: sro.openshift.io/v1beta1
    kmodNames: ["simple-kmod", "simple-procfs-kmod"]
    buildArgs:
    - name: "KMODVER"
      value: "SRO"
  driverContainer:
    source:
      git:
        ref: "master"
        uri: "https://github.com/openshift-psap/kvc-simple-kmod.git"
```

2. Create the **SpecialResource**:

```
$ oc create -f simple-kmod-local.yaml
```

The **simple-kmod** resources are deployed in the **simple-kmod** namespace as specified in the object manifest. After a short time, the build pod for the **simple-kmod** driver container starts running. The build completes after a few minutes, and then the driver container pods start running.

3. Use the **oc get pods** command to display the status of the pods:

```
$ oc get pods -n simple-kmod
```

Example output

```
NAME                                            READY   STATUS      RESTARTS   AGE
simple-kmod-driver-build-12813789169ac0ee-1-build     0/1     Completed   0          7m12s
simple-kmod-driver-container-12813789169ac0ee-mjsnh   1/1     Running     0          8m2s
simple-kmod-driver-container-12813789169ac0ee-qtkff   1/1     Running     0          8m2s
```

4. To display the logs of the simple-kmod driver container image build, use the **oc logs** command, along with the build pod name obtained above:

```
$ oc logs pod/simple-kmod-driver-build-12813789169ac0ee-1-build -n simple-kmod
```

5. To verify that the simple-kmod kernel modules are loaded, execute the **lsmod** command in one of the driver container pods that was returned from the **oc get pods** command above:

```
$ oc exec -n simple-kmod -it pod/simple-kmod-driver-container-12813789169ac0ee-mjsnh --
lsmod | grep simple
```

Example output

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

NOTE

If you want to remove the simple-kmod kernel module from the node, delete the simple-kmod **SpecialResource** API object using the **oc delete** command. The kernel module is unloaded when the driver container pod is deleted.

3.3.2. Building and running the simple-kmod SpecialResource by using a config map

In this example, the simple-kmod kernel module is used to show how the SRO can manage a driver container which is defined in Helm chart templates stored in a config map.

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.
- You installed the Node Feature Discovery (NFD) Operator.
- You installed the Special Resource Operator.
- You installed the Helm CLI (**helm**).

Procedure

1. To create a simple-kmod **SpecialResource** object, define an image stream and build config to build the image, and a service account, role, role binding, and daemon set to run the container. The service account, role, and role binding are required to run the daemon set with the privileged security context so that the kernel module can be loaded.

   a. Create a **templates** directory, and change into it:

   ```
   $ mkdir -p chart/simple-kmod-0.0.1/templates
   ```

   ```
   $ cd chart/simple-kmod-0.0.1/templates
   ```

   b. Save this YAML template for the image stream and build config in the **templates** directory as **0000-buildconfig.yaml**:

   ```
   apiVersion: image.openshift.io/v1
   kind: ImageStream
   metadata:
     labels:
       app: {{.Values.specialresource.metadata.name}}-
   {{.Values.groupName.driverContainer}} 1
     name: {{.Values.specialresource.metadata.name}}-
   {{.Values.groupName.driverContainer}} 2
   spec: {}
   ---
   apiVersion: build.openshift.io/v1
   kind: BuildConfig
   metadata:
     labels:
       app: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverBuild}}
   3
     name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverBuild}}
   4
     annotations:
       specialresource.openshift.io/wait: "true"
       specialresource.openshift.io/driver-container-vendor: simple-kmod
       specialresource.openshift.io/kernel-affine: "true"
   spec:
     nodeSelector:
       node-role.kubernetes.io/worker: ""
     runPolicy: "Serial"
     triggers:
       - type: "ConfigChange"
       - type: "ImageChange"
     source:
       git:
         ref: {{.Values.specialresource.spec.driverContainer.source.git.ref}}
         uri: {{.Values.specialresource.spec.driverContainer.source.git.uri}}
       type: Git
     strategy:
       dockerStrategy:
         dockerfilePath: Dockerfile.SRO
         buildArgs:
           - name: "IMAGE"
             value: {{ .Values.driverToolkitImage  }}
           {{- range $arg := .Values.buildArgs }}
   ```

```
        - name: {{ $arg.name }}
          value: {{ $arg.value }}
        {{- end }}
        - name: KVER
          value: {{ .Values.kernelFullVersion }}
  output:
    to:
      kind: ImageStreamTag
      name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}:v{{.Values.kernelFullVersion}} 5
```

1 2 3 4 5 The templates such as **{{.Values.specialresource.metadata.name}}** are filled in by the SRO, based on fields in the **SpecialResource** CR and variables known to the Operator such as **{{.Values.KernelFullVersion}}**.

c. Save the following YAML template for the RBAC resources and daemon set in the **templates** directory as **1000-driver-container.yaml**:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
rules:
- apiGroups:
  - security.openshift.io
  resources:
  - securitycontextconstraints
  verbs:
  - use
  resourceNames:
  - privileged
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
subjects:
- kind: ServiceAccount
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
  namespace: {{.Values.specialresource.spec.namespace}}
---
```

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    app: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
  annotations:
    specialresource.openshift.io/wait: "true"
    specialresource.openshift.io/state: "driver-container"
    specialresource.openshift.io/driver-container-vendor: simple-kmod
    specialresource.openshift.io/kernel-affine: "true"
    specialresource.openshift.io/from-configmap: "true"
spec:
  updateStrategy:
    type: OnDelete
  selector:
    matchLabels:
      app: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
  template:
    metadata:
      # Mark this pod as a critical add-on; when enabled, the critical add-on scheduler
      # reserves resources for critical add-on pods so that they can be rescheduled after
      # a failure.  This annotation works in tandem with the toleration below.
      annotations:
        scheduler.alpha.kubernetes.io/critical-pod: ""
      labels:
        app: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
    spec:
      serviceAccount: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
      serviceAccountName: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
      containers:
      - image: image-registry.openshift-image-
registry.svc:5000/{{.Values.specialresource.spec.namespace}}/{{.Values.specialresource.m
etadata.name}}-{{.Values.groupName.driverContainer}}:v{{.Values.kernelFullVersion}}
        name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
        imagePullPolicy: Always
        command: ["/sbin/init"]
        lifecycle:
          preStop:
            exec:
              command: ["/bin/sh", "-c", "systemctl stop kmods-via-
containers@{{.Values.specialresource.metadata.name}}"]
        securityContext:
          privileged: true
      nodeSelector:
        node-role.kubernetes.io/worker: ""
        feature.node.kubernetes.io/kernel-version.full: "{{.Values.KernelFullVersion}}"
```

d. Change into the **chart/simple-kmod-0.0.1** directory:

```
$ cd ..
```

e. Save the following YAML for the chart as **Chart.yaml** in the **chart/simple-kmod-0.0.1** directory:

```
apiVersion: v2
name: simple-kmod
description: Simple kmod will deploy a simple kmod driver-container
icon: https://avatars.githubusercontent.com/u/55542927
type: application
version: 0.0.1
appVersion: 1.0.0
```

2. From the **chart** directory, create the chart using the **helm package** command:

```
$ helm package simple-kmod-0.0.1/
```

Example output

```
Successfully packaged chart and saved it to:
/data/<username>/git/<github_username>/special-resource-operator/yaml-for-docs/chart/simple-kmod-0.0.1/simple-kmod-0.0.1.tgz
```

3. Create a config map to store the chart files:

a. Create a directory for the config map files:

```
$ mkdir cm
```

b. Copy the Helm chart into the **cm** directory:

```
$ cp simple-kmod-0.0.1.tgz cm/simple-kmod-0.0.1.tgz
```

c. Create an index file specifying the Helm repo that contains the Helm chart:

```
$ helm repo index cm --url=cm://simple-kmod/simple-kmod-chart
```

d. Create a namespace for the objects defined in the Helm chart:

```
$ oc create namespace simple-kmod
```

e. Create the config map object:

```
$ oc create cm simple-kmod-chart --from-file=cm/index.yaml --from-file=cm/simple-kmod-0.0.1.tgz -n simple-kmod
```

4. Use the following **SpecialResource** manifest to deploy the simple-kmod object using the Helm chart that you created in the config map. Save this YAML as **simple-kmod-configmap.yaml**:

```
apiVersion: sro.openshift.io/v1beta1
kind: SpecialResource
metadata:
```

```
  name: simple-kmod
spec:
  #debug: true 1
  namespace: simple-kmod
  chart:
    name: simple-kmod
    version: 0.0.1
    repository:
      name: example
      url: cm://simple-kmod/simple-kmod-chart 2
  set:
    kind: Values
    apiVersion: sro.openshift.io/v1beta1
    kmodNames: ["simple-kmod", "simple-procfs-kmod"]
    buildArgs:
    - name: "KMODVER"
      value: "SRO"
  driverContainer:
    source:
      git:
        ref: "master"
        uri: "https://github.com/openshift-psap/kvc-simple-kmod.git"
```

1. Optional: Uncomment the **#debug: true** line to have the YAML files in the chart printed in full in the Operator logs and to verify that the logs are created and templated properly.
2. The **spec.chart.repository.url** field tells the SRO to look for the chart in a config map.

5. From a command line, create the **SpecialResource** file:

```
$ oc create -f simple-kmod-configmap.yaml
```

The **simple-kmod** resources are deployed in the **simple-kmod** namespace as specified in the object manifest. After a short time, the build pod for the **simple-kmod** driver container starts running. The build completes after a few minutes, and then the driver container pods start running.

6. Use **oc get pods** command to display the status of the build pods:

```
$ oc get pods -n simple-kmod
```

Example output

```
NAME                                                   READY   STATUS      RESTARTS   AGE
simple-kmod-driver-build-12813789169ac0ee-1-build      0/1     Completed   0          7m12s
simple-kmod-driver-container-12813789169ac0ee-mjsnh    1/1     Running     0          8m2s
simple-kmod-driver-container-12813789169ac0ee-qtkff    1/1     Running     0          8m2s
```

7. Use the **oc logs** command, along with the build pod name obtained from the **oc get pods** command above, to display the logs of the simple-kmod driver container image build:

```
$ oc logs pod/simple-kmod-driver-build-12813789169ac0ee-1-build -n simple-kmod
```

8. To verify that the simple-kmod kernel modules are loaded, execute the **lsmod** command in one of the driver container pods that was returned from the **oc get pods** command above:

```
$ oc exec -n simple-kmod -it pod/simple-kmod-driver-container-12813789169ac0ee-mjsnh --
lsmod | grep simple
```

Example output

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

NOTE

If you want to remove the simple-kmod kernel module from the node, delete the simple-kmod **SpecialResource** API object using the **oc delete** command. The kernel module is unloaded when the driver container pod is deleted.

3.4. ADDITIONAL RESOURCES

- For information about restoring the Image Registry Operator state before using the Special Resource Operator, see Image registry removed during installation.
- For details about installing the NFD Operator see Node Feature Discovery (NFD) Operator.

CHAPTER 4. 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.

4.1. 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”.

4.2. 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.

4.2.1. 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. Run the following command to get the **channel** value required for the next step.

```
$ oc get packagemanifest nfd -n openshift-marketplace -o
jsonpath='{.status.defaultChannel}'
```

Example output

```
4.9
```

d. 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: "4.9"
  installPlanApproval: Automatic
  name: nfd
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

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

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

f. 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.

4.2.2. Installing the NFD Operator using the web console

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

> **NOTE**
>
> It is recommended to create the **Namespace** as mentioned in the previous section.

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**, select the namespace created in the previous section, and then click **Install**.

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.

4.3. 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.

4.3.1. 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
  operand:
    namespace: openshift-nfd
    image: registry.redhat.io/openshift4/ose-node-feature-discovery:v4.9
    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"
#     - "vendor"
#     - "device"
#     - "subsystem_vendor"
#     - "subsystem_device"
#  usb:
#   deviceClassWhitelist:
#     - "0e"
#     - "ef"
#     - "fe"
#     - "ff"
#   deviceLabelFields:
```

```
    #       - "class"
    #       - "vendor"
    #       - "device"
    #  custom:
    #    - name: "my.kernel.feature"
    #      matchOn:
    #        - loadedKMod: ["example_kmod1", "example_kmod2"]
    #    - name: "my.pci.feature"
    #      matchOn:
    #        - pciId:
    #            class: ["0200"]
    #            vendor: ["15b3"]
    #            device: ["1014", "1017"]
    #        - pciId :
    #            vendor: ["8086"]
    #            device: ["1000", "1100"]
    #    - name: "my.usb.feature"
    #      matchOn:
    #        - usbId:
    #          class: ["ff"]
    #          vendor: ["03e7"]
    #          device: ["2485"]
    #        - usbId:
    #          class: ["fe"]
    #          vendor: ["1a6e"]
    #          device: ["089a"]
    #    - name: "my.combined.feature"
    #      matchOn:
    #        - pciId:
    #            vendor: ["15b3"]
    #            device: ["1014", "1017"]
    #          loadedKMod : ["vendor_kmod1", "vendor_kmod2"]
  customConfig:
    configData: |
    #    - name: "more.kernel.features"
    #      matchOn:
    #      - loadedKMod: ["example_kmod3"]
    #    - name: "more.features.by.nodename"
    #      value: customValue
    #      matchOn:
    #      - nodename: ["special-.*-node-.*"]
```

2. 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.

4.3.2. 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**.

4.4. CONFIGURING THE NODE FEATURE DISCOVERY OPERATOR

4.4.1. 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

4.4.2. 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"]
```