OpenShift Container Platform 4.12

専用のハードウェアおよびドライバーの有効化

OpenShift Container Platform でのハードウェアの有効化について確認します。

Last Updated: 2023-03-31

OpenShift Container Platform 4.12 専用のハードウェアおよびドライバーの有効化

OpenShift Container Platform でのハードウェアの有効化について確認します。

法律上の通知

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

概要

本書では、OpenShift Container Platform でのハードウェアの有効化に関して解説します。

目次

第1章 専用のハードウェアおよびドライバーの有効化

Driver Toolkit (DTK) は、OpenShift Container Platform ペイロードのコンテナーイメージであり、ドライバーコンテナーを構築するベースイメージとして使用することが目的です。Driver Toolkit イメージには、カーネルモジュールをビルドまたはインストールするための依存関係として一般的に必要なカーネルパッケージと、ドライバーコンテナーに必要ないくつかのツールが含まれています。これらのパッケージのバージョンは、対応する OpenShift Container Platform リリースの RHCOS ノードで実行されているカーネルのバージョンと一致します。

ドライバーコンテナーは、コンテナーオペレーティングシステム (:op-system-first: など) でツリー外のカーネルモジュールとドライバーをビルドおよびデプロイメントするために使用されるコンテナーイメージです。カーネルモジュールおよびドライバーは、レベルの高い権限で、オペレーティングシステムカーネル内で実行されるソフトウェアライブラリーです。また、カーネル機能の拡張や、新しいデバイスの制御に必要なハードウェア固有のコードを提供します。例としては、field-programmable gate arrays (FPGA) や graphics processing units (GPU) などのハードウェアデバイスや、ソフトウェア定義のストレージソリューションなどがあります。これらはすべて、クライアントマシンでカーネルモジュールを必要とします。ドライバーコンテナーは、OpenShift Container Platform デプロイメントでこれらのテクノロジーを有効にするために使用されるソフトウェアスタックの最初の階層です。

第2章 ドライバーツールキット

Driver Toolkit について、およびドライバーコンテナーのベースイメージとしてそれを使用して、OpenShift Container Platform デプロイメントで特別なソフトウェアおよびハードウェアデバイスを有効にする方法について説明します。

2.1. DRIVER TOOLKIT について

背景情報
Driver Toolkit は、ドライバーコンテナーをビルドできるベースイメージとして使用する OpenShift Container Platform ペイロードのコンテナーイメージです。Driver Toolkit イメージには、カーネルモジュールをビルドまたはインストールするための依存関係として一般的に必要なカーネルパッケージと、ドライバーコンテナーに必要ないくつかのツールが含まれています。これらのパッケージのバージョンは、対応する OpenShift Container Platform リリースの Red Hat Enterprise Linux CoreOS (RHCOS) ノードで実行されているカーネルバージョンと同じです。

ドライバーコンテナーは、RHCOS などのコンテナーオペレーティングシステムで out-of-tree カーネルモジュールをビルドしてデプロイするのに使用するコンテナーイメージです。カーネルモジュールおよびドライバーは、レベルの高い権限で、オペレーティングシステムカーネル内で実行されるソフトウェアライブラリーです。また、カーネル機能の拡張や、新しいデバイスの制御に必要なハードウェア固有のコードを提供します。例として、Field Programmable Gate Arrays (FPGA) または GPU などのハードウェアデバイスや、クライアントマシンでカーネルモジュールを必要とする Lustre parallel ファイルシステムなどのソフトウェア定義のストレージ (SDS) ソリューションなどがあります。ドライバーコンテナーは、Kubernetes でこれらの技術を有効にするために使用されるソフトウェアスタックの最初の層です。

Driver Toolkit のカーネルパッケージの一覧には、以下とその依存関係が含まれます。

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

また、Driver Toolkit には、対応するリアルタイムカーネルパッケージも含まれています。

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

Driver Toolkit には、カーネルモジュールのビルドおよびインストールに一般的に必要となるツールが複数あります。たとえば、以下が含まれます。

- **elfutils-libelf-devel**
- **kmod**
- **binutilskabi-dw**

- **kernel-abi-whitelists**
- 上記の依存関係

目的
Driver Toolkit がリリースされる前は、エンタイトルメントのあるビルド を使用するか、または ホストの **machine-os-content** のカーネル RPM からインストールして、Pod またはビルド設定のカーネルパッケージを OpenShift Container Platform にインストールすることができていました。Driver Toolkit を使用すると、エンタイトルメントステップがなくなりプロセスが単純化され、Pod で machine-os-content にアクセスする特権操作を回避できます。Driver Toolkit は、プレリリース済みの OpenShift Container Platform バージョンにアクセスできるパートナーも使用でき、今後の OpenShift Container Platform リリース用にハードウェアデバイスのドライバーコンテナーを事前にビルドできます。

Driver Toolkit は Kernel Module Management (KMM) でも使用されます。Kernel Module Management (KMM) は、現在 OperatorHub でコミュニティー Operator として利用できます。KMM は、out-of-tree およびサードパーティーのカーネルドライバー、および基礎となるオペレーティングシステムのサポートソフトウェアをサポートします。ユーザーは、KMM のレシピを作成してドライバーコンテナーを構築してデプロイしたり、デバイスプラグインやメトリックなどのソフトウェアをサポートしたりできます。モジュールには、ビルド設定を追加して、Driver Toolkit をベースにドライバーコンテナーをビルドできます。または KMM で事前ビルドされたドライバーコンテナーをデプロイできます。

2.2. DRIVER TOOLKIT コンテナーイメージのプル

driver-toolkit イメージは、Red Hat Ecosystem Catalog および OpenShift Container Platform リリースペイロードのコンテナーイメージ セクションから入手できます。OpenShift Container Platform の最新のマイナーリリースに対応するイメージは、カタログのバージョン番号でタグ付けされます。特定のリリースのイメージ URL は、**oc adm** CLI コマンドを使用して確認できます。

2.2.1. registry.redhat.io からの Driver Toolkit コンテナーイメージのプル

podman または OpenShift Container Platform を使用して **registry.redhat.io** から **driver-toolkit** イメージをプルする手順は、Red Hat Ecosystem Catalog を参照してください。最新のマイナーリリースのドライバーツールキットイメージは、**registry.redhat.io** のマイナーリリースバージョンでタグ付けされます (例: **registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.12**)。

2.2.2. ペイロードでの Driver Toolkit イメージ URL の検索

前提条件

- Red Hat OpenShift Cluster Manager からイメージプルシークレットを 取得している。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. **oc adm** コマンドを使用して、特定のリリースに対応する **driver-toolkit** のイメージ URL を抽出します。
   - x86 イメージの場合は、以下のコマンドを入力します。

     ```
     $ oc adm release info quay.io/openshift-release-dev/ocp-release:4.12.z-x86_64 --image-for=driver-toolkit
     ```

   - ARM イメージの場合は、以下のコマンドを入力します。

```
$ oc adm release info quay.io/openshift-release-dev/ocp-release:4.12.z-aarch64 --
image-for=driver-toolkit
```

出力例

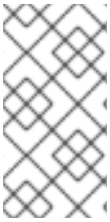
```
quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:0fd84aee79606178b6561ac71f8540f404d518ae5deff45f6d6ac8f02636c7f4
```

2. OpenShift Container Platform のインストールに必要なプルシークレットなどの有効なプルシークレットを使用して、このイメージを取得します。

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

2.3. DRIVER TOOLKIT の使用

たとえば、Driver Toolkit は **simple-kmod** と呼ばれる単純なカーネルモジュールを構築するベースイメージとして使用できます。



注記

Driver Toolkit には、カーネルモジュールに署名するために必要な依存関係である **openssl**、**mokutil**、および **keyutils** が含まれています。ただし、この例では、**simple-kmod** カーネルモジュールは署名されていないため、**Secure Boot** が有効になっているシステムにはロードできません。

2.3.1. クラスターでの simple-kmod ドライバーコンテナーをビルドし、実行します。

前提条件

- OpenShift Container Platform クラスターが実行中である。
- クラスターのイメージレジストリー Operator の状態を **Managed** に設定します。
- OpenShift CLI (**oc**) がインストールされている。
- **cluster-admin** 権限があるユーザーとして OpenShift CLI にログインしている。

手順

namespace を作成します。以下は例になります。

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

1. YAML は、**simple-kmod** ドライバーコンテナーイメージを保存する **ImageStream** と、コンテナーをビルドする **BuildConfig** を定義します。この YAML を **0000-buildconfig.yaml.template** として保存します。

```
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. 以下のコマンドで、DRIVER_TOOLKIT_IMAGE の代わりに、実行中の OpenShift Container Platform バージョンのドライバーツールキットイメージを置き換えます。

```
$ 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. 以下でイメージストリームおよびビルド設定を作成します。

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

4. ビルダー Pod が正常に完了したら、ドライバーコンテナーイメージを **DaemonSet** としてデプロイします。

   a. ホスト上でカーネルモジュールを読み込むには、特権付きセキュリティーコンテキストでドライバーコンテナーを実行する必要があります。以下の YAML ファイルには、ドライバーコンテナーを実行するための RBAC ルールおよび **DaemonSet** が含まれます。この YAML を **1000-drivercontainer.yaml** として保存し ます。

```
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. RBAC ルールおよびデーモンセットを作成します。

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

5. Pod がワーカーノードで実行された後に、**simple_kmod** カーネルモジュールが **lsmod** のホストマシンで正常に読み込まれることを確認します。

a. Pod が実行されていることを確認します。

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

出力例

```
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. ドライバーコンテナー Pod で **lsmod** コマンドを実行します。

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

出力例

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

2.4. 関連情報

- クラスターのレジストリーストレージの設定に関する詳細は、OpenShift Container Platform のイメージレジストリー Operator を参照してください。

第3章 NODE FEATURE DISCOVERY OPERATOR

Node Feature Discovery (NFD) Operator および、これを使用して Node Feature Discovery (ハードウェア機能やシステム設定を検出するための Kubernetes アドオン) をオーケストレーションしてノードレベルの情報を公開する方法を説明します。

3.1. NODE FEATURE DISCOVERY OPERATOR について

Node Feature Discovery Operator (NFD) は、ハードウェア固有の情報でノードにラベルを付け、OpenShift Container Platform クラスターのハードウェア機能と設定の検出を管理します。NFD は、PCI カード、カーネル、オペレーティングシステムのバージョンなど、ノード固有の属性でホストにラベルを付けます。

NFD Operator は、Node Feature Discovery と検索して Operator Hub で確認できます。

3.2. NODE FEATURE DISCOVERY OPERATOR のインストール

Node Feature Discovery (NFD) Operator は、NFD デーモンセットの実行に必要なすべてのリソースをオーケストレーションします。クラスター管理者は、OpenShift Container Platform CLI または Web コンソールを使用して NFD Operator をインストールできます。

3.2.1. CLI を使用した NFD Operator のインストール

クラスター管理者は、CLI を使用して NFD Operator をインストールできます。

前提条件

- OpenShift Container Platform クラスター
- OpenShift CLI (**oc**) をインストールすること。
- **cluster-admin** 権限を持つユーザーとしてログインすること。

手順

1. NFD Operator の namespace を作成します。
   a. **openshift-nfd** namespace を定義する以下の **Namespace** カスタムリソース (CR) を作成し、YAML を **nfd-namespace.yaml** ファイルに保存します。

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

   b. 以下のコマンドを実行して namespace を作成します。

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

2. 以下のオブジェクトを作成して、直前の手順で作成した namespace に NFD Operator をインストールします。
   a. 以下の **OperatorGroup** CR を作成し、YAML を **nfd-operatorgroup.yaml** ファイルに保存します。

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

b. 以下のコマンドを実行して **OperatorGroup** CR を作成します。

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

c. 以下の **Subscription** CR を作成し、YAML を **nfd-sub.yaml** ファイルに保存します。

Subscription の例

```
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. 以下のコマンドを実行して Subscription オブジェクトを作成します。

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

e. **openshift-nfd** プロジェクトに切り替えます。

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

検証

- Operator のデプロイメントが正常に行われたことを確認するには、以下を実行します。

```
$ oc get pods
```

出力例

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

正常にデプロイされると、**Running** ステータスが表示されます。

3.2.2. Web コンソールでの NFD Operator のインストール

クラスター管理者は、Web コンソールを使用して NFD Operator をインストールできます。

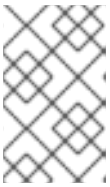
手順

1. OpenShift Container Platform Web コンソールで、**Operators → OperatorHub** をクリックします。
2. 利用可能な Operator の一覧から **Node Feature Discovery** を選択してから **Install** をクリックします。
3. **Install Operator** ページで **A specific namespace on the cluster**を選択し、**Install** をクリックします。namespace が作成されるため、これを作成する必要はありません。

検証

以下のように、NFD Operator が正常にインストールされていることを確認します。

1. **Operators → Installed Operators** ページに移動します。
2. **Status** が **InstallSucceeded** の **Node Feature Discovery** が **openshift-nfd** プロジェクトに一覧表示され ていることを確認します。



注記

インストール時に、 Operator は **Failed** ステータスを表示する可能性があります。インストールが後に **InstallSucceeded** メッセージを出して正常に実行される場合は、**Failed** メッセージを無視できます。

トラブルシューティング

Operator がインストール済みとして表示されない場合に、さらにトラブルシューティングを実行します。

1. **Operators → Installed Operators** ページに移動し、**Operator Subscriptions** および **Install Plans** タブで **Status** にエラーがあるかどうかを検査します。
2. **Workloads → Pods** ページに移動し、**openshift-nfd** プロジェクトで Pod のログを確認します。

3.3. NODE FEATURE DISCOVERY OPERATOR の使用

Node Feature Discovery (NFD) Operator は、**NodeFeatureDiscovery** CR を監視して Node-Feature-Discovery デーモンセットの実行に必要な全リソースをオーケストレーションします。**NodeFeatureDiscovery** CR に基づいて、Operator は任意の namespace にオペランド (NFD) コンポーネントを作成します。CR を編集して、他にあるオプションの中から、別の **namespace**、**image**、**imagePullPolicy**、および **nfd-worker-conf** を選択することができます。

クラスター管理者は、OpenShift Container Platform CLI または Web コンソールを使用して **NodeFeatureDiscovery** を作成できます。

3.3.1. CLI を使用した NodeFeatureDiscovery インスタンスの作成

クラスター管理者は、CLI を使用して **NodeFeatureDiscovery** CR インスタンスを作成できます。

前提条件

- OpenShift Container Platform クラスター
- OpenShift CLI (**oc**) をインストールすること。
- **cluster-admin** 権限を持つユーザーとしてログインすること。
- NFD Operator をインストールすること。

手順

1. 以下の **NodeFeatureDiscovery** カスタムリソース (CR) を作成し、YAML を NodeFeatureDiscovery **.yaml** ファイルに保存します。

```
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:v4.12
    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"]
```

NFD ワーカーをカスタマイズする方法は、nfd-worker の設定ファイルリファレンス を参照してください。

1. 以下のコマンドを実行し、**NodeFeatureDiscovery** CR インスタンスを作成します。

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

検証

- インスタンスが作成されたことを確認するには、以下を実行します。

```
$ oc get pods
```

出力例

```
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
```

正常にデプロイされると、**Running** ステータスが表示されます。

3.3.2. Web コンソールを使用した NodeFeatureDiscovery CR の作成

手順

1. **Operators → Installed Operators** ページに移動します。
2. **Node Feature Discovery** を見つけ、**Provided APIs** でボックスを表示します。
3. **Create instance** をクリックします。
4. **NodeFeatureDiscovery** CR の値を編集します。
5. **Create** をクリックします。

3.4. NODE FEATURE DISCOVERY OPERATOR の設定

3.4.1. コア

core セクションには、共通の設定が含まれており、これは特定の機能ソースに固有のものではありません。

core.sleepInterval
core.sleepInterval は、次に機能検出または再検出するまでの間隔を指定するので、ノードの再ラベル付けの間隔も指定します。正の値以外は、無限のスリープ状態を意味するので、再検出や再ラベル付けは行われません。

この値は、指定されている場合は、非推奨の **--sleep-interval** コマンドラインフラグで上書きされます。

使用例

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

デフォルト値は **60s** です。

core.sources
core.sources は、有効な機能ソースの一覧を指定します。特殊な値 **all** はすべての機能ソースを有効にします。

この値は、指定されている場合は非推奨の **--sources** コマンドラインフラグにより上書きされます。

デフォルト: **[all]**

使用例

```
core:
  sources:
    - system
    - custom
```

core.labelWhiteList
core.labelWhiteList は、正規表現を指定してラベル名に基づいて機能ラベルをフィルターします。一致しないラベルは公開されません。

正規表現は、ラベルのベース名 ('/' の後に名前の一部) だけを照合します。ラベルの接頭辞または namespace は省略されます。

この値は、指定されている場合は、非推奨の **--label-whitelist** コマンドラインフラグで上書きされます。

デフォルト: **null**

使用例

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

core.noPublish
core.noPublish を **true** に設定すると、**nfd-master** による全通信が無効になります。これは実質的にはドライランフラグです。**nfd-worker** は通常通り機能検出を実行しますが、ラベル付け要求は **nfd-master** に送信されます。

この値は、指定されている場合には、**--no-publish** コマンドラインフラグにより上書きされます。

例:

使用例

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

デフォルト値は **false** です。

core.klog
以下のオプションは、実行時にほとんどを動的に調整できるロガー設定を指定します。

ロガーオプションはコマンドラインフラグを使用して指定することもできますが、対応する設定ファイルオプションよりもこちらが優先されます。

core.klog.addDirHeader
true に設定すると、**core.klog.addDirHeader** がファイルディレクトリーをログメッセージのヘッダーに追加します。

デフォルト: **false**

ランタイム設定可能: yes

core.klog.alsologtostderr
標準エラーおよびファイルにロギングします。

デフォルト: **false**

ランタイム設定可能: yes

core.klog.logBacktraceAt
file:N の行にロギングが到達すると、スタックストレースを出力します。

デフォルト: **empty**

ランタイム設定可能: yes

core.klog.logDir
空でない場合は、このディレクトリーにログファイルを書き込みます。

デフォルト: **empty**

ランタイム設定可能: no

core.klog.logFile
空でない場合は、このログファイルを使用します。

デフォルト: **empty**

ランタイム設定可能: no

core.klog.logFileMaxSize
core.klog.logFileMaxSize は、ログファイルの最大サイズを定義します。単位はメガバイトです。値が **0** の場合には、最大ファイルサイズは無制限になります。

デフォルト: **1800**

ランタイム設定可能: no

core.klog.logtostderr
ファイルの代わりに標準エラーにログを記録します。

デフォルト: **true**

ランタイム設定可能: yes

core.klog.skipHeaders
core.klog.skipHeaders が **true** に設定されている場合には、ログメッセージでヘッダー接頭辞を使用しません。

デフォルト: **false**

ランタイム設定可能: yes

core.klog.skipLogHeaders
core.klog.skipLogHeaders が **true** に設定されている場合は、ログファイルを表示する時にヘッダーは使用されません。

デフォルト: **false**

ランタイム設定可能: no

core.klog.stderrthreshold
このしきい値以上のログは stderr になります。

デフォルト: **2**

ランタイム設定可能: yes

core.klog.v
core.klog.v はログレベルの詳細度の数値です。

デフォルト: **0**

ランタイム設定可能: yes

core.klog.vmodule
core.klog.vmodule は、ファイルでフィルターされたロギングの **pattern=N** 設定 (コンマ区切りの一覧) です。

デフォルト: **empty**

ランタイム設定可能: yes

3.4.2. ソース

sources セクションには、機能ソース固有の設定パラメーターが含まれます。

sources.cpu.cpuid.attributeBlacklist
このオプションに記述されている **cpuid** 機能は公開されません。

この値は、指定されている場合は **source.cpu.cpuid.attributeWhitelist** によって上書きされます。

デフォルト: **[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]**

使用例

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

sources.cpu.cpuid.attributeWhitelist
このオプションに記述されている **cpuid** 機能のみを公開します。

sources.cpu.cpuid.attributeWhitelist は **sources.cpu.cpuid.attributeBlacklist** よりも優先されます。

デフォルト: **empty**

使用例

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

sources.kernel.kconfigFile
sources.kernel.kconfigFile は、カーネル設定ファイルのパスです。空の場合には、NFD は一般的な標準場所で検索を実行します。

デフォルト: **empty**

使用例

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

sources.kernel.configOpts
sources.kernel.configOpts は、機能ラベルとして公開するカーネル設定オプションを表します。

デフォルト: **[NO_HZ、NO_HZ_IDLE、NO_HZ_FULL、PREEMPT]**

使用例

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

sources.pci.deviceClassWhitelist
sources.pci.deviceClassWhitelist は、ラベルを公開する PCI デバイスクラス ID の一覧です。メインクラスとしてのみ (例: **03**) か、完全なクラスサブクラスの組み合わせ (例: **0300**) として指定できます。前者は、すべてのサブクラスが許可されていることを意味します。ラベルの形式は、**deviceLabelFields** でさらに設定できます。

デフォルト: **["03", "0b40", "12"]**

使用例

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

sources.pci.deviceLabelFields
sources.pci.deviceLabelFields は、機能ラベルの名前を構築する時に使用する PCI ID フィールドのセットです。有効なフィールドは **class**、**vendor**、**device**、**subsystem_vendor** および **subsystem_device** です。

デフォルト: **[class, vendor]**

使用例

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

上記の設定例では、NFD は **feature.node.kubernetes.io/pci-<class-id>_<vendor-id>_<device-id>.present=true** などのラベルを公開します。

sources.usb.deviceClassWhitelist
sources.usb.deviceClassWhitelist は、機能ラベルを公開する USB デバイスクラス ID の一覧です。ラベルの形式は、**deviceLabelFields** でさらに設定できます。

デフォルト: **["0e", "ef", "fe", "ff"]**

使用例

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

sources.usb.deviceLabelFields
sources.usb.deviceLabelFields は、機能ラベルの名前を作成する USB ID フィールドのセットです。有効なフィールドは **class**、**vendor**、および **device** です。

デフォルト: **[class, vendor, device]**

使用例

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

上記の設定例では、NFD は **feature.node.kubernetes.io/usb-<class-id>_<vendor-id>.present=true** などのラベルを公開します。

sources.custom
sources.custom は、ユーザー固有のラベルを作成するためにカスタム機能ソースで処理するルールの一覧です。

デフォルト: **empty**

使用例

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

3.5. NFD トポロジーアップデータの使用

Node Feature Discovery (NFD) Topology Updater は、ワーカーノードに割り当てられたリソースを調べるデーモンです。これは、ゾーンごとに新規 Pod に割り当てることができるリソースに対応し、ゾーンを Non-Uniform Memory Access (NUMA) ノードにすることができます。NFD Topology Updater は、情報を nfd-master に伝達します。これにより、クラスター内のすべてのワーカーノードに対応する**NodeResourceTopology**カスタムリソース (CR) が作成されます。NFD Topology Updater のインスタンスが 1 台、クラスターの各ノードで実行されます。

NFD で Topology Updater ワーカーを有効にするには**Node Feature Discovery Operator の使用**のセクションで説明されているように、**Node Feature Discovery CR**で**topologyupdater**変数を**true**に設定します。

3.5.1. NodeResourceTopology CR

NFD Topology Updater を使用して実行すると、NFD は、次のようなノードリソースハードウェアトポロジーに対応するカスタムリソースインスタンスを作成します。

```
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 コマンドラインフラグ

使用可能なコマンドラインフラグを表示するには、 **nfd-topology-updater-help**コマンドを実行します。たとえば、podman コンテナーで、次のコマンドを実行します。

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

-ca-file
-ca-fileフラグは、 **-cert-file**フラグおよび `-key-file`フラグとともに、NFD トポロジーアップデータで相互 TLS 認証を制御する 3 つのフラグの 1 つです。このフラグは、nfd-master の信頼性検証に使用する TLS ルート証明書を指定します。

デフォルト: empty

重要

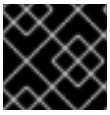
-ca-fileフラグは、**-cert-file** と **-key-file**フラグと一緒に指定する必要があります。

例

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

-cert-file
-cert-file フラグは、 **-ca-file** と **-key-file flags**とともに、NFD トポロジーアップデータで相互 TLS 認証を制御する 3 つのフラグの 1 つです。このフラグは、送信要求の認証時に提示する TLS 証明書を指定します。

デフォルト: empty



重要

-cert-file フラグは、 **-ca-file** と **-key-file**フラグと一緒に指定する必要があります。

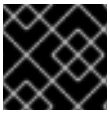
例

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

-h, -help
使用法を出力して終了します。

-key-file
-key-fileフラグは、 **-ca-file** と **-cert-file** フラグとともに、NFD Topology Updater で相互 TLS 認証を制御する 3 つのフラグの 1 つです。このフラグは、指定の証明書ファイルまたは**-cert-file**に対応する秘密鍵 (送信要求の認証に使用) を指定します。

デフォルト: empty



重要

-key-fileフラグは、 **-ca-file** と **-cert-file** フラグと一緒に指定する必要があります。

例

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

-kubelet-config-file
-kubelet-config-fileは、Kubelet の設定ファイルへのパスを指定します。

デフォルト: **/host-var/lib/kubelet/config.yaml**

例

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

-no-publish
-no-publishフラグは、nfd-master とのすべての通信を無効にし、nfd-topology-updater のドライランフラグにします。NFD Topology Updater は、リソースハードウェアトポロジー検出を正常に実行しますが、CR 要求は nfd-master に送信されません。

デフォルト: **false**

例

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

3.5.2.1. -oneshot

-oneshotフラグを使用すると、リソースハードウェアトポロジーの検出が 1 回行われた後も、NFD Topology Updater が終了します。

デフォルト: **false**

例

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

-podresources-socket
-podresources-socketフラグは、kubelet が gRPC サービスをエクスポートして使用中の CPU とデバイスを検出できるようにし、それらのメタデータを提供する Unix ソケットへのパスを指定します。

デフォルト: **/host-var/liblib/kubelet/pod-resources/kubelet.sock**

例

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

-server
-serverフラグは、接続する nfd-master エンドポイントのアドレスを指定します。

デフォルト: **localhost:8080**

例

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

-server-name-override
-server-name-overrideフラグは、nfd-master TLS 証明書から必要とされるコモンネーム (CN) を指定します。このフラグは、主に開発とデバッグを目的としています。

デフォルト: empty

例

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

-sleep-interval
-sleep-intervalフラグは、リソースハードウェアトポロジーの再検査とカスタムリソースの更新の間隔を指定します。正でない値は、スリープ間隔が無限であることを意味し、再検出は行われません。

デフォルト: **60s**。

例

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

-version
バージョンを出力して終了します。

-watch-namespace
-watch-namespaceフラグは namespace を指定して、指定された namespace で実行されている Pod に対してのみリソースハードウェアトポロジーの検査が行われるようにします。指定された namespace で実行されていない Pod は、リソースアカウンティングでは考慮されません。これは、テストとデバッグの目的で特に役立ちます。*値は、全 namespace に含まれるすべての Pod がアカウンティングプロセス中に考慮されることを意味します。

デフォルト: *

例

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

第4章 KERNEL MODULE MANAGEMENT OPERATOR

Kernel Module Management (KMM) Operator について、およびそれを使用して out-of-tree のカーネルモジュールとデバイスプラグインを OpenShift Container Platform クラスターにデプロイする方法について説明します。

4.1. KERNEL MODULE MANAGEMENT OPERATOR について

Kernel Module Management (KMM) Operator は、OpenShift Container Platform クラスター上の out-of-tree のカーネルモジュールとデバイスプラグインを管理、ビルド、署名、およびデプロイします。

KMM は、ツリー外のカーネルモジュールとそれに関連するデバイスプラグインを記述する新しい **Module** CRD を追加します。**モジュール** リソースを使用して、モジュールをロードする方法を設定し、カーネルバージョンの **ModuleLoader** イメージを定義し、特定のカーネルバージョンのモジュールをビルドして署名するための指示を含めることができます。

KMM は、任意のカーネルモジュールに対して一度に複数のカーネルバージョンに対応できるように設計されているため、ノードのシームレスなアップグレードとアプリケーションのダウンタイムの削減が可能になります。

4.2. KERNEL MODULE MANAGEMENT OPERATOR のインストール

クラスター管理者は、OpenShift CLI または Web コンソールを使用して Kernel Module Management (KMM) Operator をインストールできます。

KMM Operator は、OpenShift Container Platform 4.12 以降でサポートされています。バージョン 4.11 に KMM をインストールする場合、特に追加手順は必要ありません。KMM をバージョン 4.10 以前にインストールする方法の詳細は、「以前のバージョンの OpenShift Container Platform への Kernel Module Management Operator のインストール」セクションを参照してください。

4.2.1. Web コンソールを使用した Kernel Module Management Operator のインストール

クラスター管理者は、OpenShift Container Platform Web コンソールを使用して Kernel Module Management (KMM) Operator をインストールできます。

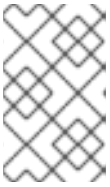
手順

1. OpenShift Container Platform Web コンソールにログインします。
2. Kernel Module Management Operator をインストールします。
    a. OpenShift Container Platform Web コンソールで、**Operators → OperatorHub** をクリックします。
    b. 使用可能な Operator のリストから **Kernel Module Management Operator** を選択し、**Install** をクリックします。
    c. **Install Operator** ページで、**Installation mode** を **A specific namespace on the cluster** として選択します。
    d. **Installed Namespace** リストから、**openshift-kmm** namespace を選択します。
    e. **Install** をクリックします。

検証

KMM Operator が正常にインストールされたことを確認するには、以下を実行します。

1. **Operators → Installed Operators** ページに移動します。
2. **Kernel Module Management Operator** が **openshift-kmm** プロジェクトにリストされ、**Status** が **InstallSucceeded** であることを確認します。



注記

インストール時に、Operator は **Failed** ステータスを表示する可能性があります。インストールが後に **InstallSucceeded** メッセージを出して正常に実行される場合は、**Failed** メッセージを無視できます。

トラブルシューティング

1. Operator のインストールに関する問題をトラブルシューティングするには、以下を実行します。
   a. **Operators → Installed Operators** ページに移動し、**Operator Subscriptions** および **Install Plans** タブで **Status** にエラーがあるかどうかを検査します。
   b. **Workloads → Pods** ページに移動し、**openshift-kmm** プロジェクトで Pod のログを確認します。

4.2.2. CLI を使用した Kernel Module Management Operator のインストール

クラスター管理者は、OpenShift CLI を使用して Kernel Module Management (KMM) Operator をインストールできます。

前提条件

- OpenShift Container Platform クラスターが実行中である。
- OpenShift CLI (**oc**) がインストールされている。
- **cluster-admin** 権限があるユーザーとして OpenShift CLI にログインしている。

手順

1. KMM を **openshift-kmm** namespace にインストールします。
   a. 次の **Namespace** CR を作成し、YAML ファイル (**kmm-namespace.yaml** など) を保存します。

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

   b. 次の **OperatorGroup** CR を作成し、YAML ファイル **(kmm-op-group.yaml** など) を保存します。

```
apiVersion: operators.coreos.com/v1
```

```
kind: OperatorGroup
metadata:
  name: kernel-module-management
  namespace: openshift-kmm
```

c. 次の **Subscription** CR を作成し、YAML ファイル (**kmm-sub.yaml** など) を保存します。

```
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. 以下のコマンドを実行して Subscription オブジェクトを作成します。

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

検証

- Operator のデプロイメントが正常に行われたことを確認するには、次のコマンドを実行します。

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

出力例

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

Operator は利用可能です。

4.2.3. 以前のバージョンの OpenShift Container Platform への Kernel Module Management Operator のインストール

KMM Operator は、OpenShift Container Platform 4.12 以降でサポートされています。バージョン 4.10 以前では、新しい **SecurityContextConstraint** オブジェクトを作成し、それを Operator の **ServiceAccount** にバインドする必要があります。クラスター管理者は、OpenShift CLI を使用して Kernel Module Management (KMM) Operator をインストールできます。

前提条件

- OpenShift Container Platform クラスターが実行中である。
- OpenShift CLI (**oc**) がインストールされている。
- **cluster-admin** 権限があるユーザーとして OpenShift CLI にログインしている。

手順

1. KMM を **openshift-kmm** namespace にインストールします。

   a. 次の **Namespace** CR を作成し、YAML ファイル (**kmm-namespace.yaml** ファイルなど) を保存します。

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

   b. 次の **SecurityContextConstraint** オブジェクトを作成し、YAML ファイル **(kmm-security-constraint.yaml** など) を保存します。

```
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. 次のコマンドを実行して、**SecurityContextConstraint** オブジェクトを Operator の **ServiceAccount** にバインドします。

```
$ 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. 次の **OperatorGroup** CR を作成し、YAML ファイル **(kmm-op-group.yaml** など) を保存します。

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

e. 次の **Subscription** CR を作成し、YAML ファイル (**kmm-sub.yaml** など) を保存します。

```
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. 以下のコマンドを実行して Subscription オブジェクトを作成します。

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

検証

- Operator のデプロイメントが正常に行われたことを確認するには、次のコマンドを実行します。

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

出力例

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

Operator は利用可能です。

4.3. カーネルモジュール管理(KMM)での署名の使用

Secure Boot 対応システムでは、すべてのカーネルモジュール(kmods)は、マシン所有者のキー(MOK) データベースに登録されている公開/秘密鍵のペアで署名する必要があります。ディストリビューションの一部として配布されるドライバーは、ディストリビューションの秘密鍵ですでに署名されているは

ずですが、カーネルモジュールの場合は、カーネルマッピングの **sign** セクションを使用してカーネルモジュールの署名をサポートします。

Secure Boot の使用に関する詳細は、公開鍵と秘密鍵のペアの生成を参照してください。

前提条件

- 正しい(DER)形式の公開秘密鍵のペア。
- MOK データベースに登録されている公開鍵を持つ 1 つ以上のセキュアブート対応ノード。
- 事前にビルドされたドライバーコンテナーイメージ、またはクラスター内のいずれかをビルドするために必要なソースコードおよび **Dockerfile** のいずれか。

4.4. SECUREBOOT の鍵の追加

KMM カーネルモジュール管理(KMM)を使用してカーネルモジュールに署名するには、証明書と秘密鍵が必要です。これらを作成する方法の詳細は、公開鍵 と秘密鍵のペアの生成 を参照してください。

公開鍵と秘密鍵のペアを抽出する方法は、秘密鍵を使用し たカーネルモジュールの署名 を 参照してください。手順 1 から 4 を使用して、キーをファイルに抽出します。

手順

1. 秘密鍵が含まれる証明書と **sb_cert.priv** ファイルを含む **sb_cert.cer** ファイルを作成します。

```
$ 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. 以下の方法のいずれかを使用してファイルを追加します。
   - ファイルを シークレット として直接追加します。

```
$ 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>
```

   - base64 エンコーディングでファイルを追加します。

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

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

3. エンコードされたテキストを YAML ファイルに追加します。

```
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) (1) (1) (2) **namespace**: **default** を有効な namespace に置き換えます。

4. YAML ファイルを適用します。

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

4.4.1. 鍵の確認

鍵を追加したら、それらをチェックして正しく設定されていることを確認する必要があります。

手順

1. 公開鍵のシークレットが正しく設定されていることを確認します。

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

これにより、シリアル番号、発行者、サブジェクトなどの証明書が表示されるはずです。

2. 秘密鍵のシークレットが正しく設定されていることを確認します。

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

これにより、-----BEGIN PRIVATE KEY----- 行および **-----END PRIVATE KEY-----** 行で囲まれたキーが表示されるはずです。

4.5. ビルド済みドライバーコンテナーの署名

ハードウェアベンダーが配布するか、別の場所でビルドしたイメージなど、ビルド済みのイメージがある場合には、この手順を使用します。

以下の YAML ファイルは、必要なキー名（秘密鍵のキー、公開鍵の証明書）を使用して、公開 **鍵** /秘密鍵のペアをシークレットとして追加します。**次に、クラスターは unsignedImage イメージをプルダウンし、これを開いて filesToSign に一覧表示されているカーネルモジュールに署名し、それらを再び追加し、作成されたイメージを containerImage としてプッシュします。**

次に、カーネルモジュール管理(KMM)は、署名された kmod をロードする DeamonSet をセレクターに一致するすべてのノードにデプロイする必要があります。ドライバーコンテナーは、公開鍵が

MOK データベースにあるノードで正常に実行され、セキュアブートが有効になっていないノード（署名を無視します）で正常に実行されるはずです。secure-boot が有効になっているものの、MOK データベースにそのキーがないかどうかは、ロードできません。

前提条件

- keySecret および certSecret シークレットが作成されている。

手順

1. YAML ファイルを適用します。

```
---
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: 読み込む kmod の名前。

4.6. MODULELOADER コンテナーイメージの構築および署名

ソースコードがあり、最初にイメージをビルドする必要がある場合は、この手順を使用します。

以下の YAML ファイルは、リポジトリーのソースコードを使用して新規コンテナーイメージをビルドします。生成されたイメージは一時的な名前でレジストリーに戻され、この一時イメージは sign セクションのパラメーターを使用して署名されます。

一時イメージ名は最終的なイメージ名に基づいており、< containerImage>:<tag>-<namespace>_<module name>_kmm_unsigned に設定されます。

たとえば、以下の YAML ファイルを使用して Kernel Module Management (KMM)が、署名されていない kmods のビルドを含む example.org/repository/minimal-driver:final-default_example-module_kmm_unsigned という名前のイメージをビルドし、レジストリーにプッシュします。次に、署名済み kmods を含む example.org/repository/minimal-driver:final という名前の 2 番目のイメージを作成します。これは、DaemonSet オブジェクトによって読み込まれ、kmods をクラスターノードにデプロイするこの 2 つ目のイメージです。

署名した後には、一時的なイメージをレジストリーから安全に削除できます。必要に応じて再構築されます。

前提条件

- keySecret および certSecret シークレットが作成されている。

手順

1. YAML ファイルを適用します。

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-module-dockerfile
  namespace: default (1)
data:
```

```
  Dockerfile: |
    ARG DTK_AUTO
    ARG KERNEL_VERSION
    FROM ${DTK_AUTO} as builder
    WORKDIR /build/
    RUN git clone -b main --single-branch https://github.com/rh-ecosystem-edge/kernel-
module-management.git
    WORKDIR kernel-module-management/ci/kmm-kmod/
    RUN make
    FROM registry.access.redhat.com/ubi8/ubi:latest
    ARG KERNEL_VERSION
    RUN yum -y install kmod && yum clean all
    RUN mkdir -p /opt/lib/modules/${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: default を有効な namespace に置き換えます。

(3)

serviceAccountName: デフォルトの serviceAccountName には、特権のあるモジュールを実行するために必要なパーミッションがありません。サービスアカウントの作成については、このセクションの関連情報のサービスアカウントの作成を参照してください。

4

imageRepoSecret: DaemonSet オブジェクトの imagePullSecrets として使用し、ビルドおよび署名機能のプルおよびプッシュを行います。

関連情報

サービスアカウント の作成については、サービスアカウントの作成 を参照してください。

4.7. デバッグとトラブルシューティング

ドライバーコンテナーの kmods が署名されていないか、間違ったキーで署名されている場合、コンテナーは PostStartHookError または CrashLoopBackOff ステータスになる可能性があります。コンテナーで oc describe コマンドを実行すると、このシナリオで以下のメッセージが表示されます。

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