Kubernetes

K8s

1.32.6, 1.33.2

Integration Guide

u.trust GP HSM Se-Series

utimaco®

Imprint

Copyright 2025	Utimaco IS GmbH Krefelder Straße 220 52070 Aachen Germany
Phone	AMERICAS +1-844-UTIMACO (+1 844-884-6226) EMEA +49 800-627-3081 APAC +81 800-919-1301
Internet	https://support.hsm.utimaco.com/
e-mail	support@utimaco.com
Document Version	1.0.0
Date	2025-11-11
Status	PUBLISHED
Document No.	IG-2025-0058

Table of Contents

1 Introduction 5
1.1 About This Guide 5
1.2 Target Audience 5
1.3 Purpose of the Integration 5
1.4 Abbreviations 6
1.5 Document Conventions 7
2 Product Overview 9
2.1 Overview of Kubernetes 9
2.2 Overview of Utimaco SecurityServer HSM 9
2.3 Joint Value Proposition 9
3 Integration Requirements and Prerequisites 11
3.1 Tested Versions 11
3.2 Hardware and Software Requirements 11
3.2.1 Hardware Requirements 11
3.2.2 Software Requirements 12
3.3 Prerequisites 12
4 Installation and Configuration 13
4.1 Setting Up Utimaco SecurityServer Software 13
4.2 Setting Up Kubernetes 15
5 Integration Steps 16
5.1 Configuration on Utimaco u.trust GP HSM Se-Series 16
5.2 Configuration on Kubernetes Control Plane Node 16
5.2.1 Kubernetes Control Plane Node Verification 16
5.2.2 Setup the Environment for KMS Plugin 17
5.2.3 Configure the KSM Plugin Configuration File 19
5.2.4 Load KMS Plugin Docker Image 21
5.2.5 KMS Plugin Deployment 21
5.2.6 Configure Encryption Provider 23
5.2.7 Configure kube-apiserver 23
6 Verification and Testing 26
6.1 Functional Testing 26

6.1.1 Secret Creation 26
6.1.2 Key Rotation 27
6.1.3 HSM Clustering 31
6.2 Logs and Validation Steps 33
7 Troubleshooting 34
7.1 Common Issues and How to Resolve Them 34
7.2 Log Locations and Interpretation 34
8 Contact and Support Information 36
9 Appendices 37
9.1 References 37
9.2 Command Summary 37

1 Introduction

This guide is a part of the information and support provided by Utimaco to facilitate secure database encryption practices. It explains the integration of Kubernetes K8s with Utimaco's Hardware Security Module (HSM) to protect sensitive data. This integration enables robust key management and enhanced data protection. All Utimaco SecurityServer product documentation is available from Utimaco's website at *https://utimaco.com/*.

The guide walks through the necessary steps for the integration.

1.1 About This Guide

This guide describes how to integrate Kubernetes with Utimaco u.trust GP HSM (Hardware Security Module) Se-Series to protect databases by enabling the encryption at rest of sensitive data.

1.2 Target Audience

This guide is intended for Utimaco Hardware Security Module (HSM) and Kubernetes administrators.

1.3 Purpose of the Integration

The purpose of integrating Utimaco SecurityServer HSM with Kubernetes is to provide a secured and centralized method for managing encryption keys used by containerized applications and services. Kubernetes supports encryption of sensitive data such as secrets, but relies on HSM to store and manage encryption keys securely.

The following are the primary objective of this integration:

- Enhance data security by encrypting the data at rest.
- Compliance and security requirements.

1.4 Abbreviations

Abbreviation	Meaning
HSM	Hardware Security Module
PKI	Public Key Infrastructure
PKCS	Public Key Cryptography Standards
PKCS#11	PKCS Part 11: The Cryptographic Token Interface Standard
DB	Database
MBK	Master Backup Key
CLI	Command Line Interface
K8s	Kubernetes
SO	Security Officer
AES	Advanced Encryption Standard
YAML	Yet Another Markup Language
API	Application Programming Interface
DEK	Data Encryption Key

Abbreviation	Meaning
KEK	Key Encryption Key

Table 1: Abbreviations

1.5 Document Conventions

The following conventions are used in this guide:

Convention	Use	Example
Bold	Items of the Graphical User Interface (GUI), e.g., menu options	Press **OK**
`Monospaced`	File names, folder and directory names, commands, file outputs, programming code samples.	`chsm-create`
Italic	References and important terms	See *Sample Chapter* in the *CryptoServer - Sample Manual*

Table 2: Document Conventions

We use special icons to highlight the most important notes and information.

Here you will find important safety information that should be followed.

Here you will find additional notes or supplementary information.

This message indicates the expected result after the successful execution of an instruction.

2 Product Overview

2.1 Overview of Kubernetes

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. For security, it can encrypt sensitive data such as secrets at rest. This functionality relies on an external Key Management Service (KMS) to handle the encryption and decryption. When integrated with the HSM, Kubernetes can leverage a centralized and secure service for these cryptographic operations This ensures enhanced security and helps meet regulatory compliance requirements for sensitive data within your cluster.

2.2 Overview of Utimaco SecurityServer HSM

SecurityServer is a hardware security module developed by Utimaco IS GmbH. It is a physically protected, specialized computer unit designed to perform sensitive cryptographic tasks and securely manage and store cryptographic keys and data. SecurityServer can be used as a universal, independent security component for heterogeneous computer systems.

2.3 Joint Value Proposition

The integration of Kubernetes with the HSM provides a complete solution for secure and compliant encryption key management in containerized environments. Kubernetes handles the deployment and management of applications, while HSM serves as a dedicated, hardened system for securely storing and managing the encryption keys.

This joint solution offers several key benefits:

- Secure Key Management: Encryption keys are safely stored and managed outside of the Kubernetes cluster, protecting them from an etcd database breach.
- Regulatory Compliance: It helps meet strict regulatory requirements by providing robust key access control, auditing, and rotation capabilities.
- Centralized Control: You gain centralized key management across all your Kubernetes clusters, simplifying security administration.
- Efficient and Scalable: The solution is designed to work efficiently for clusters of any size without compromising performance.

This integration empowers organizations to secure sensitive data with confidence, maintain operational agility, and simplify compliance.

3 Integration Requirements and Prerequisites

Ensure that the system environment you will be using meets the following hardware and software requirements.

This guide assumes that the user has already installed and configured the required software.

3.1 Tested Versions

The integration has been successfully tested with the Utimaco HSM and Kubernetes.

Operating System	Kubernetes version	Utimaco SecurityServer version	Utimaco HSM
Linux Server (Rocky Linux 9.6)	▪ v1.32.6 ▪ v1.33.2	▪ V6.1.1 ▪ V6.2.0	u.trust GP HSM Se-Series

Table 3: List of tested versions

3.2 Hardware and Software Requirements

3.2.1 Hardware Requirements

Hardware	Hardware Requirements
Utimaco LAN HSM	u.trust GP HSM Se-Series LAN with firmware SecurityServer 6.1.1 or higher
Utimaco PCI-e HSM	u.trust GP HSM Se-Series PCI-e with firmware SecurityServer 6.1.1 or higher

Table 4: List of hardware requirements

3.2.2 Software Requirements

Software	Software Requirements
HSM Interface	SecurityServer PKCS#11 Provider
HSM Utility	SecurityServer PKCS#11 Tool (p11tool2)
Kubernetes	v1.33.2 or later
k8s-kms-plugin-v2	V1.0

Table 5: List of software requirements

3.3 Prerequisites

Before you begin, please ensure that you have:

- Installed and set up the operating system listed in Tested Versions.
- Installed and set up the HSM listed in Tested Versions.
- Replaced the HSM default admin with a new admin user.
- Created and stored the MBK on each HSM. Refer to the SecurityServer documentation to set up the MBK.
- Set up and configured the SecurityServer. Refer to the SecurityServer documentation to set up the HSM.
- Set up and configured the PKCS#11 library according to your environment. Refer to the SecurityServer documentation for instructions on setting up and configuring the PKCS#11 library.
- Created the Security Officer (SO) user and Crypto user.
- Set up root access to the Kubernetes Cluster.
- Installed and set up the k8s-kms-plugin-v2 Docker image.

4 Installation and Configuration

The following section outlines the procedures required to configure both Utimaco SecurityServer Software and Kubernetes components for seamless integration.

4.1 Setting Up Utimaco SecurityServer Software

If you have not already done so, create and request an Utimaco Support Portal Account at https://support.hsm.utimaco.com/support. This will allow you to download the software components needed for this installation.

On Linux:

1. Copy the downloaded Utimaco SecurityServer software to the appropriate location on a Linux Server.
2. Create a *utimaco* folder under the */opt* directory and create 2 directories */opt/utimaco/bin* and */opt/utimaco/lib*.
3. Copy the pkcs11 library file `libcs_pkcs11_R3.so` from the Utimaco SecurityServer software to the */opt/utimaco/lib* directory and make the file executable.
4. Copy the *csadm* and *p11tool2* files from the Utimaco SecurityServer software to the */opt/utimaco/bin* directory and make both files executable.
5. Create the directory */etc/utimaco*. Locate the Utimaco PKCS#11 configuration file in your SecurityServer directory, *Software\Linux\Crypto_APIs\PKCS11_R3\sample*. Copy the Utimaco PKCS#11 configuration file *cs_pkcs11_R3.cfg* to */etc/utimaco* directory.
6. Set the environment variable '*CS_PKCS11_R3_CFG*' to map to the *cs_pkcs11_R3.cfg* file.

```
# export CS_PKCS11_R3_CFG=/etc/utimaco/cs_pkcs11_R3.cfg
```

On Windows:

On Windows, *cs_pkcs11_R3.cfg* will be automatically created and available in the *C:\ProgramData\UtimacoPKCS11_R3* folder as part of the SecurityServer software installation. Edit the `cs_pkcs11_R3.cfg` file and make the appropriate changes to the file. A sample `cs_pkcs11_R3.cfg` file is mentioned below.

```
library = C:\oracle\extapi\64\hsm\utimaco\6.1.1.0\cs_pkcs11_R3.dll
```

```
slot = 0
pin = Oracle123
[Global]
# For Unix:
Logpath = /tmp
# For Windows:
# Logpath = C:/ProgramData/Utimaco/PKCS11_R3
# Loglevel (0 = NONE; 1 = ERROR; 2 = WARNING; 3 = INFO; 4 = TRACE)
Logging = 1
# Prevents expiring session after inactivity of 15 minutes
KeepAlive = true
# Set the Device to connect with
#[CryptoServer]
# Device specifier
Device = <HSM_IP>
```

For detailed guidance on commands and their parameters, please refer to the Utimaco SecurityServer documentation. The device could be a u.trust GP HSM Se-Series, available in either PCIe or LAN form factors. Depending on the type, the device configuration line will follow one of these formats:

- LAN-based HSM: Device = 288@ipaddress
- PCIe-based HSM: Device = /dev/cs2.0

Make sure to select the appropriate format based on your specific hardware setup.

`library` specifies the path where the cs_pkcs11_R3.dll file is located.

`Slot` indicates the slot number associated with the created USER.

`Pin` represents the password assigned to the USER.

To simplify your testing process, it's recommended that you enable the PKCS#11 log file by adjusting the logging settings. Specifically:

- Set the `LogPath` to a writable directory (not a specific file).
- Set the `Logging` log level to 1 for basic logging. Increase it to 4 for more detailed output during testing.

This will generate a log file named `cs_pkcs11_R3.log` within the specified `LogPath` directory. Reviewing this log can help with troubleshooting if you encounter issues. Once testing is complete, it's advisable to reduce `Logging` log level to 1 or 2 to limit output to only critical or important messages.

4.2 Setting Up Kubernetes

Follow the instructions in the link to set up the Kubernetes cluster: *Creating a cluster with kubeadm*.

5 Integration Steps

This integration is implemented using the **Envelope Encryption scheme** to secure sensitive data at rest in *etcd*. A Data Encryption Key (DEK) is used to encrypt the actual data, such as secrets in *etcd,* and a Key Encryption Key (KEK) is used to encrypt the DEK. This KEK is stored and managed in a GP HSM. Kubernetes uses the **KMS v2 plugin interface** to enable encryption at rest. The custom **KMS plugin** facilitates secure communication between Kubernetes and the HSM for key management operations.

The following section outlines the procedures required to configure both Utimaco HSM and Kubernetes components for seamless integration.

5.1 Configuration on Utimaco u.trust GP HSM Se-Series

Create users SO (Security Officer) and USR (the Crypto user) and initialize a slot.

The slot must be initialized using the **p11tool2**.

First, create the **SO** using *p11tool2*. Then, using the *p11tool2* command, initialize the Slot you want to use and the slot user, as shown below.

- `# ./p11tool2 slot=<slot no.> Label=<token label> Login=ADMIN,ADMIN.key InitToken=<SO pin>`
- Initialize the SO user
- `# ./p11tool2 slot=<slot no.> LoginSO=<SO pin> InitPin=<Cryptouser pin>`

Make sure that the Utimaco GP HSM is accessible from the Kubernetes control plane.

The KMS plugin will generate the required key in the HSM.

5.2 Configuration on Kubernetes Control Plane Node

5.2.1 Kubernetes Control Plane Node Verification

Log in to the Kubernetes control plane node as the root user and verify that all Kubernetes pods are running properly. The following commands can be used for verifying.

```
# kubectl get nodes
```

```
# kubectl get pods -A
```

```
[root@master-node ~]#
[root@master-node ~]# kubelet --version
Kubernetes v1.33.2
[root@master-node ~]#
[root@master-node ~]# kubectl get nodes
NAME          STATUS   ROLES           AGE   VERSION
master-node   Ready    control-plane   41d   v1.33.2
[root@master-node ~]#
[root@master-node ~]# kubectl get pods -A
NAMESPACE     NAME                                       READY   STATUS    RESTARTS          AGE
kube-system   calico-kube-controllers-689744956f-vjmjv   1/1     Running   2520 (103m ago)   41d
kube-system   calico-node-pxll7                          1/1     Running   22 (13d ago)      41d
kube-system   coredns-674b8bbfcf-g25bw                   1/1     Running   22 (13d ago)      41d
kube-system   coredns-674b8bbfcf-t4wwp                   1/1     Running   22 (13d ago)      41d
kube-system   etcd-master-node                           1/1     Running   26 (6d6h ago)     41d
kube-system   kube-apiserver-master-node                 1/1     Running   0                 85m
kube-system   kube-controller-manager-master-node        1/1     Running   14 (86m ago)      6d6h
kube-system   kube-proxy-n95zs                           1/1     Running   0                 6d6h
kube-system   kube-scheduler-master-node                 1/1     Running   8 (86m ago)       6d6h
[root@master-node ~]#
```

Figure 1 : Kubernetes control plane verification

5.2.2 Setup the Environment for KMS Plugin

The script file `kms_plugin_env_setup.sh` can be used for setting up the necessary directories and file structures for KMS plugin.

1. Prepare a `kms_plugin_env_stup.sh` script file for setting up the environment for the KMS plugin in the K8s control plane environment with the code snippet below.

```
#!/bin/bash

# Run the script as root
set -e

SOCKET_DIR="/var/lib/kmsplugin"
SOCKET_FILE="$SOCKET_DIR/kmsplugin.sock"
KMS_CONFIG_DIR="/etc/kms/config"
KMS_LIB_DIR="/etc/kms/lib"
KMS_LOG_DIR="/etc/kms/log"
KMS_LOG_FILE="$KMS_LOG_DIR/KMSplugin.log"
PKCS11_LOG_FILE="$KMS_LOG_DIR/cs_pkcs11_R3.log"

#Check for SOCKET_DIR
if [ ! -d "$SOCKET_DIR" ]; then
    echo "Creating directory $SOCKET_DIR"
    mkdir -p "$SOCKET_DIR"
fi
```

```
#Remove old socket if exist
if [ -S $SOCKET_FILE ]; then
    echo "Removing old socket file $SOCKET_FILE"
    rm -f "$SOCKET_FILE"
fi

#Create a Unix domin socket file
echo "Creating a Unix socket file $SOCKET_FILE"
nc -lU $SOCKET_FILE &

#Including delay for socket file creation
sleep 1
SOCPID=$!

#Update the permission of the socket file
chmod 666 "$SOCKET_FILE"

echo "Socket created. PID of socat: $SOCPID"

#Check for KMS_CONFIG_DIR
if [ ! -d "$KMS_CONFIG_DIR" ]; then
    echo "Creating directory $KMS_CONFIG_DIR"
    mkdir -p "$KMS_CONFIG_DIR"
fi

#Check for KMS_LIB_DIR
if [ ! -d "$KMS_LIB_DIR" ]; then
    echo "Creating directory $KMS_LIB_DIR"
    mkdir -p "$KMS_LIB_DIR"
fi

#Check for KMS_LOG_DIR
if [ ! -d "$KMS_LOG_DIR" ]; then
    echo "Creating directory $KMS_LOG_DIR"
    mkdir -p "$KMS_LOG_DIR"
fi

#Create the log files
echo "Creating log files"
touch "$KMS_LOG_FILE"
touch "$PKCS11_LOG_FILE"
```

2. Provide execution permission to the script file `kms_plugin_env_stup.sh`.

chmod 755 will set the permission to read, write, and execute to the owner and read and execute to other users.

```
# chmod 755 kms_plugin_env_stup.sh
```

3. Execute the script file.

```
# ./kms_plugin_env_stup.sh
```

```
[root@master-node kubernetes]#
[root@master-node kubernetes]# ./kms_plugin_env_setup.sh
Removing old socket file /var/lib/kmsplugin/kmsplugin.sock
Creating a Unix socket file /var/lib/kmsplugin/kmsplugin.sock
Socket created. PID of socat: 2440102
Creating directory /etc/kms/config
Creating directory /etc/kms/lib
Creating directory /etc/kms/log
Creating log files
[root@master-node kubernetes]#
```

Figure 2 : Script execution

Once the script execution is complete, the following file structures will be created.

```
[root@master-node ~]#
[root@master-node ~]# ls -lh /var/lib/kmsplugin/kmsplugin.sock
srw-rw-rw- 1 root root 0 Oct 17 03:44 /var/lib/kmsplugin/kmsplugin.sock
[root@master-node ~]#
[root@master-node ~]# tree /etc/kms/
/etc/kms/
├── config
├── lib
└── log
    ├── cs_pkcs11_R3.log
    └── KMSplugin.log

3 directories, 2 files
[root@master-node ~]#
```

Figure 3 : Created file structures from the script

4. Copy the PKCS #11 API config file.

Copy the PKCS #11 API config file '`cs_pkcs11_R3.cfg`' to the path '`/etc/kms/config`'.

5. Copy the PKCS #11 shared library.

Copy the PKCS #11 shared library '`libcs_pkcs11_R3.so`' to the path '`/etc/kms/lib`'.

5.2.3 Configure the KSM Plugin Configuration File

The KMS configuration file '`/etc/kms/config/cs_pkcs11_R3.cfg`' needs to be updated.

Attribute	Remark
Devices	HSM ip; eg: 3001@127.0.0.1

Attribute	Remark
SlotId	Slot id number
UserPIN	Pin of Crypto user
Primary_key	Label of the Primary key
Secondary_key	Label of the Secondary key
KMS_Plugin_log_level	Set KMS plugin log level

Table 6: Attribute list for KMS plugin

The attribute *Devices* will already be present in the config file '`cs_pkcs11_R3.cfg`'. The HSM ip needs to be updated.

A sample of PKCS#11 config file changes for using the KMS plugin is mentioned below. Add the lines below in the config file '`cs_pkcs11_R3.cfg`'.

```
# Cryptographic user authentication used for KMS plugin
#Slot id number
SlotId=0
#Pin of Crypto user
UserPIN=12345678
#Label of primary key
Primary_key=K8s_UTA_HSM_Key1
#Label of secondary key. If not available provide value as <none>
Secondary_key=<none>
#KMS plugin logs can be enabled ((0 = NONE; 1 = ERROR; 2 = ERROR + INFO; 3 = ERROR
+ INFO + DEBUG))
KMS_Plugin_log_level=1
```

If PKCS#11 logging is enabled, then change the log path as below. The attribute value of '*Logpath*' for Unix needs to be updated to '*/tmp/k8s*'.

```
# For Unix:
```

```
Logpath = /tmp/k8s
```

5.2.4 Load KMS Plugin Docker Image

Load the KMS plugin Docker image from `'k8s-kms-plugin-hsm_v1.0.tar'`. The Docker image can be loaded using the following command:

```
# docker load -i k8s-kms-plugin-hsm_v1.0.tar
```

```
[root@master-node kubernetes]#
[root@master-node kubernetes]#
[root@master-node kubernetes]# docker load -i k8s-kms-plugin-hsm_v1.0.tar
Loaded image: k8s-kms-plugin-v2:v1.0
[root@master-node kubernetes]#
```

Figure 4 : Load KMS plugin docker image

5.2.5 KMS Plugin Deployment

Create a static pod manifest on the control plane for the KMS plugin. The KMS plugin will be deployed as a static pod.

The yaml file '`kms-plugin.yaml`' needs to be created in the path '`/etc/kubernetes/manifests`', so that the yaml file will be automatically managed by the *kubelet*.

The *kubelet* running on the control plane node monitors the directory '`/etc/kubernetes/manifests`'. If a yaml file in this directory is modified, then the *kubelet* will automatically update the corresponding pod.

Prepare the '`kms-plugin.yaml`' file using the below code snippet:

```
apiVersion: v1
kind: Pod
metadata:
  name: kms-plugin-v2
  namespace: kube-system
spec:
  hostNetwork: true
  containers:
    - name: kms-plugin
      image: k8s-kms-plugin-v2:v1.0
      imagePullPolicy: IfNotPresent
      command: ["/app/kms_server"]
      env:
        - name: CS_PKCS11_R3_CFG
          value: /app/cs_pkcs11_R3.cfg
        - name: PKCS_LIBRARY_PATH
```

```
          value: /app/lib/libcs_pkcs11_R3.so
        - name: LD_LIBRARY_PATH
          value: /app/lib/
      volumeMounts:
        - name: config-volume
          mountPath: /app/cs_pkcs11_R3.cfg
          subPath: cs_pkcs11_R3.cfg
        - name: socket-dir
          mountPath: /var/lib/kmsplugin
        - name: kms-plugin-log
          mountPath: /tmp/k8s/KMSplugin.log
          subPath: KMSplugin.log
        - name: kms-plugin-log
          mountPath: /tmp/k8s/cs_pkcs11_R3.log
          subPath: cs_pkcs11_R3.log
        - name: lib-volume
          mountPath: /app/lib/libcs_pkcs11_R3.so
          subPath: libcs_pkcs11_R3.so
  volumes:
    - name: config-volume
      hostPath:
        path: /etc/kms/config
        type: Directory
    - name: socket-dir
      hostPath:
        path:  /var/lib/kmsplugin
        type: DirectoryOrCreate
    - name: kms-plugin-log
      hostPath:
        path: /etc/kms/log/
        type: Directory
    - name: lib-volume
      hostPath:
        path: /etc/kms/lib/
        type: Directory
```

Verify the KMS plugin is running using the command:

```
# kubectl get pods -A
```

```
[root@master-node kubernetes]#
[root@master-node kubernetes]# kubectl get pods -A
NAMESPACE     NAME                                       READY   STATUS    RESTARTS          AGE
kube-system   calico-kube-controllers-689744956f-4nfvw   1/1     Running   1875 (14s ago)    28d
kube-system   calico-node-sdldn                          1/1     Running   6 (25h ago)       28d
kube-system   coredns-674b8bbfcf-hvz94                   1/1     Running   2 (25h ago)       3d
kube-system   coredns-674b8bbfcf-zp8pm                   1/1     Running   2 (25h ago)       3d
kube-system   etcd-master-node                           1/1     Running   2 (25h ago)       3d
kube-system   kms-plugin-v2-master-node                  1/1     Running   0                 30s
kube-system   kube-apiserver-master-node                 1/1     Running   1 (25h ago)       2d23h
kube-system   kube-controller-manager-master-node        1/1     Running   6 (25h ago)       3d
kube-system   kube-proxy-v9gzm                           1/1     Running   2 (25h ago)       3d
kube-system   kube-scheduler-master-node                 1/1     Running   6 (25h ago)       3d
[root@master-node kubernetes]#
```

Figure 5 : KMS plugin pod status

p11tool2 can be used for verifying the generated key in the HSM.

5.2.6 Configure Encryption Provider

Create an `'encryption-config.yaml'` file at `'/etc/kubernetes'`.

The `'encryption-config.yaml'` can be created using the below code snippet for KMS v2. Save the file.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
    providers:
      - kms:
          apiVersion: v2
          name: kms-hsm
          endpoint: unix:///var/lib/kmsplugin/kmsplugin.sock
          timeout: 3s
      - identity: {}
```

5.2.7 Configure kube-apiserver

Configure the kube-apiserver to use the KMS plugin for data at rest encryption. The changes need to be made in the api-server yaml file '`/etc/kubernetes/manifests/kube-apiserver.yaml`'.

1. Set the `--encryption-provider-config` flag on the kube-apiserver to point to the location of the encryption configuration file. Under the command section, add `encryption-provider-config=/etc/kubernetes/encryption-config.yaml` to enable the use of an external encryption provider.

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 172.31.1.76:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=172.31.1.76
    - --allow-privileged=true
    - --encryption-provider-config=/etc/kubernetes/encryption-config.yaml
    - --authorization-mode=Node,RBAC
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
```

Figure 6 : Highlighted the changes for encryption provider flag addition

2. In the volumeMounts section, add the following entries.

```
    volumeMounts:
    - mountPath: /etc/ssl/certs
      name: ca-certs
      readOnly: true
    - mountPath: /etc/pki/ca-trust
      name: etc-pki-ca-trust
      readOnly: true
    - mountPath: /etc/pki/tls/certs
      name: etc-pki-tls-certs
      readOnly: true
    - mountPath: /etc/kubernetes/pki
      name: k8s-certs
      readOnly: true
    - mountPath: /etc/kubernetes/encryption-config.yaml
      name: encryption-config
      readOnly: true
    - mountPath: /var/lib/kmsplugin
      name: socket-dir
  hostNetwork: true
```

Figure 7 : Highlighted the changes for volumeMounts section

3. In the volumes section, add the corresponding volume definitions.

```
  volumes:
  - hostPath:
      path: /etc/ssl/certs
      type: DirectoryOrCreate
    name: ca-certs
  - hostPath:
      path: /etc/pki/ca-trust
      type: DirectoryOrCreate
    name: etc-pki-ca-trust
  - hostPath:
      path: /etc/pki/tls/certs
      type: DirectoryOrCreate
    name: etc-pki-tls-certs
  - hostPath:
      path: /etc/kubernetes/pki
      type: DirectoryOrCreate
    name: k8s-certs
  - hostPath:
      path: /etc/kubernetes/encryption-config.yaml
      type: File
    name: encryption-config
  - hostPath:
      path: /var/lib/kmsplugin
      type: DirectoryOrCreate
    name: socket-dir
status: {}
```

Figure 8 : Highlighted the changes for volumes section

4. Save the *kube-apiserver* yaml file and restart the *kube-apiserver*.

6 Verification and Testing

In this chapter, we will verify that the integration between Utimaco u.trust GP HSM Se-Series and Kubernetes via the custom KMS plugin is functioning as expected. This includes validating the connectivity between the Kubernetes KMS plugin and the HSM, testing encryption and decryption workflows, and ensuring the Kubernetes control plane is successfully interacting with the plugin. By the end of this section, you should be able to confirm that the integration is correctly configured, secure, and fully operational.

6.1 Functional Testing

6.1.1 Secret Creation

Create a secret using `kubectl`. This secret will be stored in *etcd*. Only kube-apiserver will have the access to read or write data to *etcd*.

1. Create a secret.

   Create a simple secret using *kubectl*.

```
[root@master-node ~]#
[root@master-node ~]# kubectl create secret generic test-secret --from-literal=foo=bar
secret/test-secret created
[root@master-node ~]#
[root@master-node ~]# kubectl get secrets
NAME          TYPE     DATA   AGE
test-secret   Opaque   1      12s
[root@master-node ~]#
```

Figure 9 : Step for secret creation

2. Verify the encrypted secret in *etcd*. Use `` `etcdctl` `` to directly inspect the secret's value in the *etcd* database.

   The output should show that the data has been encrypted by the KMS provider. The output will be unreadable and begin with the `` `k8s:enc:kms:v2: kms-hsm:` `` header. This header is the definitive proof that the KMS plugin has successfully encrypted the secret data before it was stored in *etcd*. The encrypted payload includes the UUID of the associated key used by the KMS plugin.

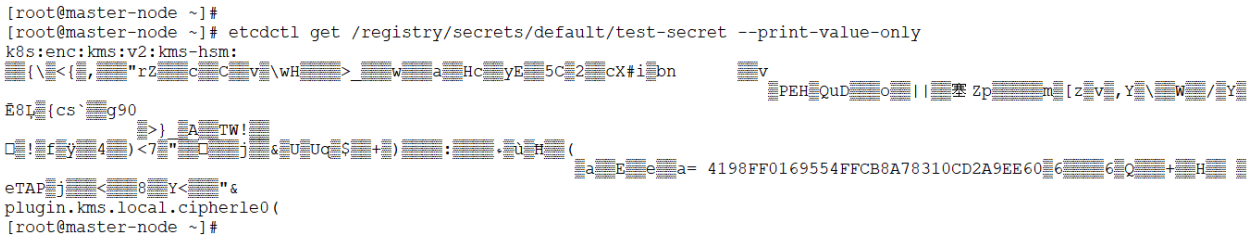

Figure 10 : Step for verifying the secret

3. Verify decryption.

   Retrieve the secret using ' `kubectl` '. This action prompts the kube-apiserver to check its cache for the decrypted data. If the secret is not found in the cache, the kube-apiserver will then initiate a decryption request to KMS plugin, which returns the plaintext data.

```
[root@master-node ~]#
[root@master-node ~]# kubectl get secret test-secret -o yaml
apiVersion: v1
data:
  foo: YmFy
kind: Secret
metadata:
  creationTimestamp: "2025-10-16T08:57:57Z"
  name: test-secret
  namespace: default
  resourceVersion: "3234501"
  uid: 23b0b780-dfee-42f1-b859-424a9493f863
type: Opaque
[root@master-node ~]#
[root@master-node ~]# echo "YmFy" | base64 --decode
bar[root@master-node ~]#
[root@master-node ~]#
```

Figure 11 : Step for verifying the decryption

   The output will show the secret's data in its original, unencrypted format (base64-encoded). The ` *bar*` field, which was encrypted in etcd, is now readable as `*YmFy*` (the Base64 for `*bar*`). This confirms that the entire decryption workflow is working as expected.

6.1.2 Key Rotation

This procedure outlines the steps to safely rotate the encryption key used by the KMS plugin. Key rotation is a critical security practice that ensures secrets are regularly re-encrypted with a new key.

1. Create a secret.

   a. Create a secret using *kubectl* command.

```
[root@master-node kubernetes]#
[root@master-node kubernetes]# kubectl create secret generic test-secret --from-literal=foo=bar
secret/test-secret created
[root@master-node kubernetes]#
[root@master-node kubernetes]# kubectl get secrets
NAME          TYPE     DATA   AGE
test-secret   Opaque   1      4s
[root@master-node kubernetes]#
[root@master-node kubernetes]# etcdctl get /registry/secrets/default/test-secret --print-value-only
k8s:enc:kms:v2:kms-hsm:
▒:▒D[▒▒▒▒▒▒aN
                ▒M▒▒`ae▒▒▒ _▒~)▒N2▒▒g3▒8▒▒6▒▒á!▒▒;▒{u▒▒b#▒▒▒q▒▒00~W▒▒▒V▒▒▒H▒▒>▒▒▒c▒▒1.▒/▒N▒▒▒▒▒{▒e▒▒({▒J▒▒▒It▒▒▒F▒▒-f▒▒▒k▒z▒▒#WB▒▒J▒▒▒▒J8▒k
▒▒He1▒▒T▒:▒_▒5M79=▒-▒87▒▒▒▒$▒{▒▒▒0f▒$%▒▒▒d▒▒B▒▒▒▒}▒N▒▒d▒,▒3▒
▒wLU▒▒Sudh'▒▒▒C▒0▒})▒g▒▒▒▒▒2▒▒T▒▒,▒▒&3▒▒ 38CCAB77D0664C1295D8C4F25DECF4D60▒▒`▒▒J▒2▒XZé{▒Q▒[j▒b}▒        ▒▒t▒▒l▒ŧ▒Ч"&
plugin.kms.local.cipherle0(
[root@master-node kubernetes]#
```

Figure 12 : Step for secret creation

The key UUID is available in the encrypted payload of the secret in the *etcd*.

Verify the KMS plugin log for identifying the key UUID. The KMS plugin pod logs can be accessed using the following command.

`kubectl logs kms-plugin-v2-master-node -n kube-system`

```
[root@master-node kubernetes]# kubectl logs kms-plugin-v2-master-node -n kube-system
[Main]:: KMS plugin version: v1.0
[Main]:: Value of Primary key label is: K8s_UTA_HSM_Key1
[Main]:: Value of slotID is: 0
[Main]:: Current log level: 3
[Main]:: Socket file exist /var/lib/kmsplugin/kmsplugin.sock
[Main]:: Old socket file removed
[Main]:: Socket creation completed
[Main]:: Token initialized.


PKCS#11 Library Version: 6.2
-> USER already exists on slot 0.
[Main]:: Opened session on slot 0.
-> User logged in.
[SearchKeyLabel]: -> Key search is done for keylabel.
[Main]:: Search result for keylabel (primary)chKey ->1
[Main]:: Search result for keylabel (primary)count ->1
[Main]:: Key is available
[Main]:: Key can be used for encryption
[GetKeyUUIDByLabel]:: key found
38CCAB77D0664C1295D8C4F25DECF4D6
KMS plugin listening on /var/lib/kmsplugin/kmsplugin.sock
[root@master-node kubernetes]#
```

Figure 13 : KMS plugin pod log

2. Update the KMS plugin config file.

    a. Set the '*Secondary_key*' attribute with the value of '*Primary_key*'.

    b. Set the '*Primary_key*' attribute with a new key label.

A sample config file changes are shown in the figure below, where '`cs_pkcs11_R3.cfg`' is the current config file and '`keyrotation/cs_pkcs11_R3.cfg`' is the modified config file for key rotation.

```
[admin@master-node config]$
[admin@master-node config]$ diff -Nurb cs_pkcs11_R3.cfg keyrotation/cs_pkcs11_R3.cfg
--- cs_pkcs11_R3.cfg    2025-11-06 07:22:30.369595795 -0800
+++ keyrotation/cs_pkcs11_R3.cfg        2025-11-06 07:22:46.475485421 -0800
@@ -133,8 +133,8 @@
 #Pin of Crypto user
 UserPIN=12345678
 #Label of primary key
-Primary_key=K8s_UTA_HSM_Key1
+Primary_key=K8s_UTA_HSM_Key2
 #Label of secondary key. If not available provide value as <none>
-Secondary_key=<none>
+Secondary_key=K8s_UTA_HSM_Key1
 #KMS plugin logs can be enabled ((0 = NONE; 1 = ERROR; 2 = ERROR + INFO; 3 = ERROR + INFO + DEBUG))
 KMS_Plugin_log_level=1
[admin@master-node config]$
```

Figure 14 : Sample config file 'cs_pkcs11_R3.cfg' changes for key rotation

Do not delete the old key from the HSM, as the old key will still be used to decrypt the old data.

3. Restart the KMS plugin pod.

```
[root@master-node kubernetes]#
[root@master-node kubernetes]# pwd
/etc/kubernetes
[root@master-node kubernetes]#
[root@master-node kubernetes]# mv manifests/kms-plugin.yaml .
[root@master-node kubernetes]# mv kms-plugin.yaml manifests/
[root@master-node kubernetes]#
```

Figure 15 : Sample step to restart the KMS plugin pod

4. Re-encrypt existing secrets with the new key.

    a. This step relies on the kube-apiserver's ability to decrypt secrets with the old key and then re-encrypt them with the new one. The following command retrieves all secrets and forces the kube-apiserver to replace them, triggering re-encryption with the new primary key.

The following command can be used for re-encrypting the existing secrets with the new key.

```
kubectl get secrets --all-namespaces -o json | kubectl replace --force -f -
```

```
[root@master-node kubernetes]#
[root@master-node kubernetes]# kubectl get secrets --all-namespaces -o json | kubectl replace --force -f -
secret "test-secret" deleted
secret/test-secret replaced
[root@master-node kubernetes]#
```

Figure 16 : Step for replacing existing secrets

This command re-encrypts all secrets, including system secrets. It is recommended to perform this in a controlled maintenance window.

5. Verify the secrets.

Verify the replaced encrypted secret in *etcd*. Use `etcdctl` to directly inspect the secret's value in the *etcd* database. The associated new *key UUID* will be embedded in the payload.

```
[root@master-node kubernetes]#
[root@master-node kubernetes]# kubectl get secrets --all-namespaces -o json | kubectl replace --force -f -
secret "test-secret" deleted
secret/test-secret replaced
[root@master-node kubernetes]#
[root@master-node kubernetes]# kubectl get secrets
NAME          TYPE     DATA   AGE
test-secret   Opaque   1      29s
[root@master-node kubernetes]#
[root@master-node kubernetes]# etcdctl get /registry/secrets/default/test-secret --print-value-only
k8s:enc:kms:v2:kms-hsm:
▒ @▒▒R▒▒▒▒▒k▒M▒(▒##D▒A▒▒15n▒▒▒`▒4o▒2▒ -:p Z&▒▒{z1Im□▒/▒▒f▒=+▒2ç▒a▒-8▒O▒4▒0p9WF▒jE▒<▒E  ▒)md▒)▒▒▒▒s▒z▒2▒▒4ê▒▒c▒%▒)▒▒?1/▒2
                                                                                                                                L▒▒□x▒▒b▒Jr▒▒
▒▒u▒▒V$&@▒=RsD▒▒K ▒w6T▒Z▒+B▒(▒R▒YY▒▒M▒Pv▒=1▒▒▒;▒e8▒▒▒q▒▒a8▒<▒▒▒qʊ▒▒M▒hY*▒▒▒hS!▒▒▒W▒▒z▒AT▒▒;t▒▒▒$n A74A5A2E92BC4B85BA13F501B7B7CEF60CwT
▒xY▒▒▒▒▒\4b▒d&z▒5▒▒ ?▒_Rj▒▒▒ ▒▒"&
plugin.kms.local.cipherle0(
[root@master-node kubernetes]#
```

Figure 17 : Verify the secrets

Check the KMS plugin log for identifying the new key UUID. The KMS plugin pod logs can be accessed using the following command.

```
kubectl logs kms-plugin-v2-master-node -n kube-system
```

```
[root@master-node kubernetes]# kubectl logs kms-plugin-v2-master-node -n kube-system
[Main]:: KMS plugin version: v1.0
[Main]:: Value of Primary key label is: K8s_UTA_HSM_Key2
[Main]:: Value of slotID is: 0
[Main]:: Current log level: 1
[Main]:: Socket file exist /var/lib/kmsplugin/kmsplugin.sock
[Main]:: Old socket file removed
[Main]:: Socket creation completed
[Main]:: Token initialized.


PKCS#11 Library Version: 6.2
-> USER already exists on slot 0.
[Main]:: Opened session on slot 0.
-> User logged in.
[SearchKeyLabel]: -> Key search is done for keylabel.
[Main]:: Search result for keylabel (primary)chKey ->0
[Main]:: Search result for keylabel (primary)count ->0
[Main]:: No key is available
[Main]:: Generate new Key
-> Generated AES 32 bit key.
[Main]:: Key generation is successful with label K8s_UTA_HSM_Key2
[GetKeyUUIDByLabel]:: key found
A74A5A2E92BC4B85BA13F501B7B7CEF6
KMS plugin listening on /var/lib/kmsplugin/kmsplugin.sock
[root@master-node kubernetes]#
```

Figure 18 : KMS plugin pod log

Verify the KMS plugin log and the secret for the new *key UUID* to confirm that the new key is used for re-encrypting the existing secret.

6.1.3 HSM Clustering

The HSM can be deployed in a clustered configuration, in which multiple HSM devices work together to provide high availability, load balancing, and fault tolerance for cryptographic operations.

The HSM clustering can be performed by following the steps below.

1. Confirm that all the HSMs in the cluster are reachable from the Kubernetes control plane.
2. Make sure that all the HSMs in the cluster are using the same MBKs.
3. Update the PKCS#11 config file '`/etc/kms/config/cs_pkcs11_R3.cfg`' for HSM clustering setup.
    a. Set *SynchronizationOnline* & *SynchronizationOffline* values accordingly.
    b. Set *FallbackInterval* accordingly.
    c. Comment *Device specifier* line.
    d. Enable HSM clustering.

A sample PKCS#11 config file for setting up HSM clustering with two HSMs is mentioned below.

```
# If not set, the default is: SynchronizationOnline = Enforced and SynchronizationOffline = None
# Options:
# SynchronizationOnline  = Enforced | Relaxed
# SynchronizationOffline = Backups | None
# Warning: Using 'SynchronizationOnline = Relaxed' in combination with
#          'SynchronizationOffline = None' may lead to unwanted behavior. If a device is not
#          available during a state-changing command, it is marked as out-of-sync only for other
#          state-changing commands. For read-only commands, it is still an eligible target.
#          To avoid issues, make sure to manually synchronize an out-of-sync device as soon as it
#          is brought back online.

# Example:
SynchronizationOnline  = Enforced
SynchronizationOffline = None
 :
# Configures load balancing mode ( == 0 ) or failover mode ( > 0 )
# In failover mode, n specifies the interval (in seconds) after which a reconnection attempt to the failed CryptoServer is
 started
FallbackInterval = 3600
 :
# Device specifier (here: internal PCI device)
# For Unix:
#Devices = /dev/cs2.0

# For Windows:
#Devices = PCI:0

# Device specifier (here: cHSM on local u.trust Anchor device - last number represents cHSM slot (1))
#Devices = /dev/cs2.0.1

# Device specifier (here: local simulator)
#Devices = 3001@127.0.0.1

# Device specifier (here: cluster of simulators)
#Devices = { 3001@127.0.0.1 3003@127.0.0.1 }

# Device specifier (here: cHSM on remote u.trust Anchor device - port = base port (4000) + cHSM slot (1))
#Devices = 4001@192.168.0.1

# Device specifier (here: remote device with IP address 192.168.0.1)
#Devices = 192.168.0.1

# Device specifier (here: cluster of remote devices - first as above, others using format <port>@<ip>)
#Devices = { 192.168.0.1 288@192.168.0.2 4001@192.168.0.3 }
 :
#[HSMCluster]
ClusterID = 0000
# Overwrite global SlotCount setting
SlotCount = 4
Devices = { 3001@172.31.1.61 3001@172.31.1.54 }
```

Figure 19 : Sample config file 'cs_pkcs11_R3.cfg' for HSM cluster

Make sure to restart the KMS plugin pod if changes are made in the '`cs_pkcs11_R3.cfg`' config file. The steps below can be performed for verifying the HSM clustering.

1. Create a secret.

   Create a secret using *kubectl* command.

2. Verify the encrypted secret in *etcd*.

   Use `` `etcdctl` `` command to directly inspect the secret's value in the *etcd* database.

3. Shutdown the first HSM mentioned in the config file.

4. Restart the KMS plugin pod.

5. Verify the KMS plugin pod status.

   The KMS plugin pod status can be verified using the command `kubectl get pods -A`. The KMS plugin pod should run without any issues, indicating that the generated key in the HSM is accessible to the KMS plugin pod.

6. Verify the encrypted secret in the *etcd*. The secret should be accessible.

6.2 Logs and Validation Steps

The KMS plugin logs, PKCS#11 API logs and KMS plugin pod logs can be used for analysis.

1. KMS plugin logging

   a. The KMS plugin provides configurable logging to assist with monitoring and troubleshooting. Logging can be enabled and controlled via the configuration file located at '`/etc/kms/config/cs_pkcs11_R3.cfg'`.

   b. Log level configuration: The logging verbosity is controlled by the `KMS_Plugin_log_level` attribute. Supported values are *0* to *3*.

   c. The KMS plugin writes its logs to the file '`/etc/kms/log/KMSplugin.log`'.

2. PKCS#11 API Logging

   a. The PKCS#11 library used by the KMS plugin also supports logging for the cryptographic operations performed. The logging is configured in the configuration file '`/etc/kms/config/cs_pkcs11_R3.cfg`'.

   b. Set the `Logging` attribute to enable PKCS#11 API logging in the config file. Ensure the `Logpath` is correctly set for Unix systems as '`/tmp/k8s`'.

   c. The PKCS#11 API logs are written to '`/etc/kms/log/cs_pkcs11_R3.log`'.

3. KMS plugin pod logs

   a. The KMS plugin pod logs can be accessed using the command below:

```
# kubectl logs kms-plugin-v2-master-node -n kube-system
```

7 Troubleshooting

7.1 Common Issues and How to Resolve Them

1. Ensure the Utimaco HSM is accessible from the Kubernetes control plane.
2. Ensure the KMS plugin pod is restarted every time a change is made to the config file `'/etc/kms/config/cs_pkcs11_R3.cfg'`.
3. Issue in loading the Docker image.

   The Docker image '*k8s-kms-plugin-v2*' loaded in the Kubernetes control plane may experience issues, which can occur if the Docker load is not functioning or if containers are being used.

   A solution is to explicitly import the image. Follow the steps below for explicitly loading the KMS plugin Docker image.

   i. Load the KMS plugin image to Kubernetes control plane node.

   ```
   ctr -n=k8s.io images import k8s-kms-plugin-hsm_v1.0.tar
   ```

   ii. Verify that the image is available.

   ```
   ctr -n=k8s.io images list | grep k8s-kms-plugin-v2
   ```

7.2 Log Locations and Interpretation

The KMS plugin logs, PKCS#11 API logs, and KMS plugin pod logs can be used for analysis.

1. KMS plugin logging
   a. The KMS plugin provides configurable logging to assist with monitoring and troubleshooting. Logging can be enabled and controlled via the configuration file located at '`/etc/kms/config/cs_pkcs11_R3.cfg'`.
   b. Log level configuration: The logging verbosity is controlled by the `KMS_Plugin_log_level` attribute. Supported values are *0* to *3*.
   c. The KMS plugin writes its logs to the file '`/etc/kms/log/KMSplugin.log`'.
2. PKCS#11 API Logging

a. The PKCS#11 library used by the KMS plugin also supports logging for the cryptographic operations performed. The logging is configured in the configuration file '`/etc/kms/config/cs_pkcs11_R3.cfg`'.

b. Set the `Logging` attribute to enable PKCS#11 API logging in the config file. Ensure the `Logpath` is correctly set for Unix systems as '`/tmp/k8s`'.

c. The PKCS#11 API logs are written to '`/etc/kms/log/cs_pkcs11_R3.log`'.

3. KMS plugin pod logs

a. The KMS plugin pod logs can be accessed using the command below.

```
# kubectl logs kms-plugin-v2-master-node -n kube-system
```

8 Contact and Support Information

You can reach us from Monday to Friday, 09.00 a.m. to 05.00 p.m., Central European Time (CET).

Utimaco IS GmbH
Krefelder Str. 220
52070 Aachen
Germany

RMA Query

If you need to send the device back to Utimaco IS GmbH, please open a new RMA case (Return Merchandise Authorization). We request that you use the following web address. RMA cases cannot be opened by email or phone.

https://support.hsm.utimaco.com/support/rma/new

Other Support Queries

- Mail (preferred contact method)
  support@utimaco.com
  Attach the diagnostic information to your email.
- Web portal
  https://support.hsm.utimaco.com/support/cases/new/
  The diagnostic information will be requested in our response if necessary.
- By phone
  AMERICAS +1-844-UTIMACO (+1 844-884-6226)
  EMEA +49 800-627-3081
  APAC +81 800-919-1301
  The diagnostic information will be requested in our response if necessary.

9 Appendices

9.1 References

This document serves as a comprehensive guide for integrating Utimaco u.trust GP HSM Se-Series with Kubernetes. For more information on other Utimaco products and offerings, please visit the official Utimaco Website.

The official document for setting up a Kubernetes Cluster will be in: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/

The official Kubernetes documentation covering setup, configuration, and management of Kubernetes setup will be in: https://kubernetes.io/docs/tasks/administer-cluster/kms-provider/

Title	Document Number
utrust_Anchor_LAN_V5_Operating_Manual	2021-0039
utrust_Anchor_PCIe_Operating_Manual	2020-0042

Table 7: References

9.2 Command Summary

Most of the CLI commands used are mentioned in the table below.

Command used	Purpose
# docker load -i k8s-kms-plugin-hsm_v1.0.tar	To load KMS plugin docker image.
# chmod 755 kms_plugin_env_setup.sh	To set all permission to owner.
# ./kms_plugin_env_setup.sh	To execute the environment setup script.

Command used	Purpose
# vi /etc/kubernetes/manifests/kms-plugin.yaml	To create/edit KMS plugin yaml file.
*# vi /etc/kubernetes/manifests/*kube-apiserver.yaml	To edit kube-apiserver.yaml file.
# vi /etc/kubernetes/encryption-config.yaml	To create/edit encryption-config.yaml file.
# kubectl create secret generic test-secret --from-literal=foo=bar	To create a secret.
# etcdctl get /registry/secrets/default/test-secret --print-value-only	To verify encryption in etcd.
# kubectl get pods -A	To list all the pods running in Kubernetes cluster.
# kubectl get secret test-secret -o yaml	To verify decryption.
# kubectl get secrets	To view the created secrets.
# kubectl get secrets --all-namespaces -o json | kubectl replace --force -f -	To re encrypt existing secrets.
# kubectl logs kms-plugin-v2-master-node -n kube-system	To view the logs of KMS plugin pod running in the Kubernetes cluster.

Table 8: CLI commands used