THALES

CTE for Kubernetes Administration

ADMINISTRATION GUIDE

Introduction to CipherTrust Transparent Encryption for Kubernetes

Kubernetes is an open-source container-orchestration system that aims to provide a "platform for automating deployment, scaling, and operations of container workloads".

CTE for Kubernetes is an implementation of the CipherTrust Transparent Encryption with native support for Kubernetes through the implementation of a CSI (Container Storage Interface) driver. Unlike traditional CTE, product installation and GuardPoint management is done through Kubernetes. This means that as the cluster scales up with more nodes, CTE for Kubernetes scales with it. CTE for Kubernetes is designed to protect Kubernetes Persistent Storage Claims that are backed by storage with filesystem semantics.

In order to support customers with diverse workloads, registration to CipherTrust Manager has been decentralized from the cluster nodes/ hosts operating system. Registration now happens through the use of Storage Classes which allows for a single cluster, and even a single node, to register different CTE for Kubernetes groups, each with a different set of policies and keys.

CTE Agent and CSI driver are deployed as container images. CTE devices are exposed as Persistent Storage volumes and Customer application containers do not need to be modified.

This document shows some examples on how a customer can protect their pod’s persistent storage data through the use of CTE for Kubernetes.

Prerequisites

The CTE for Kubernetes solution requires the following:

- Kubernetes v1.22 or subsequent versions Kubernetes IO releases
- Available Persistent Storage for protecting with CTE for Kubernetes
- CipherTrust Manager v2.8 and subsequent versions
- Kubernetes nodes with access to the Internet to fetch CTE for Kubernetes images from Docker Hub
- Public Docker Hub credentials for pulling the images

- Helm
- Working Kubernetes Cluster that can communicate with the cluster using `kubectl`

Minimum System Requirements

Pod Type	Memory	CPU
CTE CSI Controller	512Mb	1
CTE CSI Node Server	512Mb + 100Mb per GuardPoint	2
CTE CSI Staging pod	5Mb	0.1

Note

- Encryption CPU requirements on the node server are highly dependent on the workload that is accessing the CTE for Kubernetes encrypted volume. Additional CPU resources may be required depending on the expected number of IOPS for the application
- Applying hard memory limits on the CTE for Kubernetes node server pods can result in the pod being evicted. This will cause a service interruption to the CTE for Kubernetes volume. Hard memory limits are not recommended for the CTE node server.
- Staging pods do not consume any CPU resources after starting and very minimal memory resources after starting.

Audience

This document is intended for personnel responsible for maintaining your organization's security infrastructure. This includes security officers, key manager administrators, and network administrators.

All products manufactured and distributed by Thales Group are designed to be installed, operated, and maintained by personnel who have the knowledge, training, and qualifications required to safely perform the tasks assigned to them. The information, processes, and procedures contained in this document are intended for use by trained and qualified personnel only.

Thales expects that the users of this product are proficient with security concepts and knowledgeable in all aspects of Kubernetes cluster administration and management.

Users must be able to:

- Install the CTE for Kubernetes CSI driver
- Create RBAC (role-based access control) rules
- Add Persistent Storage Claims to a namespace within the Kubernetes cluster

Limitations

- Killing CTE CSI pods can leave a pod with protected volumes in an unusable state. User should terminate all pods using a CTE protected volume before attempting to stop the CTE for Kubernetes service pods.

Support for Managed Clouds

CTE for Kubernetes can be deployed in the following Cloud environments:

- Amazon Elastic Kubernetes Service (AWS)
- Azure Kubernetes Service (Azure)
- Google Kubernetes Engine (GKE)
- Red Hat OpenShift

Kubernetes Terminology

The following Kubernetes terms are important to understand when working with CTE for Kubernetes containers.

Container Storage Interface

Container Storage Interface (CSI) is a plugin for Kubernetes that allows the implementation of third-party persistent storage solutions. This provides Kubernetes users more options for storage. CTE for Kubernetes is implemented as a CSI plugin which allows Thales to provide encryption and data protection while using Kubernetes volume configuration semantics.

Persistent Volumes

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator, or dynamically provisioned using Storage Classes. It is a resource in the cluster. PVs are volume plugins like volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage.

Unlike other storage plugins, CTE PVs are virtual in nature and do not hold store actual data. They functions as an overlay on top of other Kubernetes PVs. It then encrypts the contents of that PV.

Persistent Volume Claims

A Persistent Volume Claim (PVC) is a request to bind a PV to a namespace. Once claimed, a PV can be attached to any Pod within the same namespace. A PV that has not been claimed is not used by any Pod.

Storage Class

Provides a method for administrators to describe the classes of storage they offer. They are the foundation for defining parameters for dynamic provisioning.

Namespaces

In Kubernetes, namespaces provide a mechanism for isolating groups of resources within a single cluster. Namespaces are a way to divide cluster resources between multiple users.

Note

- Namespaces **cannot** be nested inside one another. Each Kubernetes resource can only reside in one namespace.
- Avoid creating namespaces with the prefix `kube-`. It is reserved for Kubernetes system namespaces.
- Names of the resources need to be unique within a namespace, but not across namespaces.

Pods

A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. A pod consumes the nodes resources. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host. Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

CTE Staging Pod

A CTE staging pod is a special pod that mounts on a Persistent Volume available to a Kubernetes cluster, so that CTE can attach the protection layer on top of the PV. CTE for Kubernetes automatically generates a staging pod on the same node where a Pod with a CTE PVC volume is deployed.

Storage Group

Similar, conceptually, to a Client Group. A Kubernetes Storage Group is created in CipherTrust Manager and consists of Kubernetes clients, similar to how a Client group consists of CTE Clients.

Configuration Concepts

The following concepts are important to understand for success with CTE for Kubernetes.

For more information on Kubernetes, consult the Kubernetes documentation.

CipherTrust Manager CTE for Kubernetes Storage Groups

CTE for Kubernetes configuration is accomplished with CipherTrust Managers(CM) through **K8s Storage Groups**. Security administrators define which combination of Kubernetes Storage Classes and Namespaces are allowed to register to CM. Once a K8s Storage Group is defined, the user can create a **GuardPolicy** that can be used by the CTE-CSI volume to create, view, edit, and delete Kubernetes (K8s) storage groups on the K8s Storage Groups page of the CipherTrust Manager GUI.

Note

A single node in a Kubernetes cluster can register itself multiple times to different K8s Storage Groups based on different Storage Class and Namespace usage patterns. This allows for separation of security context based on the security needs of the Kubernetes namespace that the application pod will be running in.

- See Managing Kubernetes Storage Groups and Clients for more information about creating storage groups in CipherTrust Manager.

Using CipherTrust Manager with CTE for Kubernetes

On the CipherTrust Manager, to setup your Kubernetes environment:

1. Create a Kubernetes Client.
2. Create a Kubernetes storage group, which includes a client profile which contains a Guard Policy. The Guard Policy must contain:
   a. Supported access controls
   b. Resource sets
   c. Process sets
   d. Signature sets
   e. User sets with UIDs
3. Register the Kubernetes Client
   Registration is the process of configuring a Kubernetes (Kubernetes) client with CipherTrust Manager. This process creates SSL certificates for further communication between the CipherTrust Manager and the Kubernetes client.
   After registration, the Kubernetes client can communicate with the CipherTrust Manager. All of the Guard Policies applied to the Kubernetes storage group are automatically added to the Kubernetes client. The client configuration is then built for the Kubernetes client (exactly like a CTE client) and sent to the CipherTrust Manager.
   After successful registration, the Kubernetes client appears on the Kubernetes Clients page of the CipherTrust Manager GUI. The client status becomes Healthy.

CTE for Kubernetes CSI driver

Protecting Persistent volumes in a Kubernetes Cluster

Unlike traditional CSI drivers, the CTE for Kubernetes CSI driver (CTE-CSI) does not store persistent data. CTE-CSI relies on other Persistent Volumes(PV) in the cluster for actual data storage. The CTE-CSI driver provides a layer on top of those PVs that adds encryption and data protection. This encryption is transparent to the Pod that wants to use a CTE-CSI volume. Each CTE Persistent Volume Claim (PVC) must be backed by a filesystem-capable PV that is claimed by a PVC in the same namespace.

Each CTE PVC must be configured to use a GuardPolicy, configured in CM, that will provide the required access control and key rules.

When protecting a PV, you must:

1. Choose a PVC in the cluster to protect and to act as your data source.
2. Create a CTE-CSI Storage Class with a parameter to connect to CM.
3. Create a CTE-CSI PVC with a GuardPolicy and a data source for PVC parameters.
4. Add the CTE-CSI PVC to a Pod as a regular volume.

Supported Persistent volumes

CTE Kubernetes is expected to work well with most persistent volumes types with file system semantics. Raw volumes with no filesystem are not supported at this time.

For more information, see Types of Persistent Volumes

The following table lists the **unsupported** PVC volumes:

Storage Type	Supported	Full Name
configMap	No	ConfigMap
downwardAPI	No	Downward API
emptyDir	No	EmptyDir
fc	No	Fibre Channel (FC) storage
flocker	No	Flocker storage
gitRepo	No	GIT repository

Storage Type	Supported	Full Name
hostPath	No	HostPath volume
projected	No	Projected volume
quobyte	No	Quobyte volume
secret	No	Secrets
storageOS	No	StorageOS volume
subPath	No	Subpath volume

CTE service pods

CTE for Kubernetes is implemented as a Kubernetes CSI driver. A typical CTE-CSI driver deployment will contain one **cte-csi-node-xxxx** for every worker node in the cluster. These pods are the main workers for CTE for Kubernetes and perform tasks such as volume attachment, CTE agent registration and volume encryption within the cluster node. The cluster will also contain at least one **cte-csi-controller-xxxx** which is in charge of dynamic provisioning of CTE-CSI volumes. These two types of service pods run in the **kube-system** namespace and are able to serve volumes across different namespaces in the cluster.

Warning

It is important to *never* restart a node's **cte-csi-node-xxx** pod if this node still has CTE-CSI volumes attached to it from other application pods. Killing a busy CTE-CSI pod will result in application Pods losing their connection to the encryption service. This will render the volume on those pods inaccessible, and can result in data loss. When this happens, the application Pod must be restarted to restore proper functionality.

CTE for Kubernetes registration

CTE for Kubernetes dynamically registers the agent when a CTE-CSI volume attachment request is performed on a node. CTE will use the configuration parameters of the StorageClass, as well as the Namespace of the Pod, requesting the volume to attempt registration. Registration only succeeds if the Storage Class name and the Pod's Namespace matches that of the Storage Group defined in CipherTrust Manager. The CTE-CSI driver will automatically deregister itself for a given StorageClass/

Namespace combination once all Pods that were consuming a particular registration in the node have been evicted from the node. The amount of time that the CTE-CSI driver waits from last eviction to deregistration can be configured through the StorageClass **registration_period** parameter

Note

Registration and deregistration to CM is dynamic, based on pod usage within the nodes in the cluster. This can quickly exhaust registration token client capacity. Therefore, it is recommended that the user define a CipherTrust Manager registration token with a large client capacity.

Warning

Kubernetes will always retry and mount a Pod's volume in the event of a failure. If that failure is due to a registration failure (for example, non-matching StorageClass/ Namespace parameters in the K8s Storage Group), retry attempts by Kubernetes will quickly exhaust registration token client capacity.

CTE-CSI Storage Class

For CTE for K8s, as soon as you create a pod to protect a cte-csi volume, cte-csi is using a storage class. Once this storage class is created, it automatically creates a K8s client and registers it with CipherTrust Manager.

When first creating Storage Class objects, Administrators set the name and other parameters for a class in a yaml file. Note that objects cannot be updated once they are created.

Important standard Kubernetes specification definitions

The following list defines the parts of the yaml file:

- **metadata.name**: Name of the StorageClass. This name **MUST** match the StorageClass name specified in the CM K8s Storage Group.
- **provisioner**: Determines what volume plugin is used for provisioning Persistent Volumes. This field must be specified. For CTE for Kubernetes, it is always defined as **csi.cte.cpl.thalesgroup.com**

- **reclaimPolicy**: Determines what to do with the PersistentVolume once its claim has been deleted. CTE PVs do not hold any data. They act solely as configuration placeholders for real PVs. The policy **MUST** be set to **Delete** as they are not usable outside of the context of its PVC that created it.
- **allowVolumeExpansion**: CTE for Kubernetes does not store data so these volumes are not expandable. This does not prevent the source unprotected volume from being expandable. This **MUST** be set to **false**

CTE-CSI Storage Class parameters

- **key_manager_addr**: Domain name or IP address of the CipherTrust Manager. This is a required field.
- **k8_storage_group**: Name of the CipherTrust Manager K8s Storage Group. This is a required field.
- **registration_token_secret**: Kubernetes Secret with CM registration token. The CM registration token must be saved into the Kubernetes secret with as a base64 encoded string. This is a required field.
- **registration_period**: Time in minutes to wait before deregistering from the CipherTrust Manager once all volumes on a node have been unguarded. Parameter must be added as a string integer value. Default "10" minute. This field is optional.

Example

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: <CHANGEME to K8s Storage Class>
provisioner: csi.cte.cpl.thalesgroup.com
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: false
parameters:
  Domain name or IP address of the ${cm} (Required)
  key_manager_addr: <CHANGEME to your CM ADDR>

  Name of the ${cm} K8s Storage Group. (Required)
  k8_storage_group: <CHANGEME to your K8s Storage Group name>
```

```
  Kubernetes Secret with CM registration token (Required)
  registration_token_secret: <CHANGEME to K8s secret>

  Time in minutes to wait before unregistering from the ${cm}
  once all volumes have been unguarded. Parameter must be added as a s
tring
  integer value. Default "10" minute. (Optional)
  registration_period: "10"
```

CTE Persistent Volume Claim

Important standard Kubernetes spec definitions

The following list defines the parts of the yaml file:

- **metadata.name**: Name of the CTE-CSI PVC. Use this PVC name in your Pod definition instead of the unprotected source PVC.
- **metadata.annotations.csi.cte.cpl.thalesgroup.com/policy**: CTE for Kubernetes GuardPolicy name. This GuardPolicy is located on the CipherTrust Manager and should match a policy name available on the storage class for this PVC. This parameter is required.
- **metadata.annotations.csi.cte.cpl.thalesgroup.com/source_pvc**: Name of the unprotected source PVC that will be protected by this CTE-PVC. This parameter is required.
- **spec.storageClassName**: CTE-CSI StorageClass name. This parameter is required
- **spec.resources.requests.storage**: Kubernetes PV capacity request. This parameter is required by Kubernetes but ignored by CTE-CSI as capacity is dictated by the source PVC.

Example

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cte-claim
  annotations:
     CTE for Kubernetes GuardPolicy name. This GuardPolicy is located
on the
     CipherTrust Manager and should match a policy name available on t
he
     storage class for this PVC. (Required)
```

```
    csi.cte.cpl.thalesgroup.com/policy: <CHANGEME CTE for Kubernetes
policy>

    Name of the unprotected source PVC that will be protected by
this CTE-PVC.
    (Required)
    csi.cte.cpl.thalesgroup.com/source_pvc: <CHANGEME to a regular P
VC to protect>
spec:
  storageClassName: <CHANGEME to the CTE-CSI StorageClass>
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      This parameter is required by Kubernetes but ignored by CTE-CSI.
      storage: 1Ki
```

Container Image Fingerprint

Thales recommends that customers use container images from trusted repositories for their Kubernetes deployments to ensure that the containers have not been tampered with.

Verify that the CTE for Kubernetes Container Image Fingerprint matches the version and architecture that you are installing:

Release Date	Version	Tag	Architecture	Image Digest
2024-05-01	1.4.0	1.4.0.37	x86	sha256:dd6af74116e0b55da5b9b1d61b9fd2e3b1e12b27fb635a601c4c3374e38d60b8
2024-05-01	1.4.0	1.4.0.37	arm	sha256:e1754ae2be1ea3d3f911f0f8901f9291eb275fab97a638a4442330fdf912a149

Install CTE for Kubernetes

Installation Dependencies

When installing CTE for Kubernetes, CTE requires the following dependencies:

- helm

- kubectl

Getting CTE for Kubernetes deployment files

Install CTE for Kubernetes through the `yaml` files available in the cte-csi-deploy Git repository at:

```
git clone https://github.com/thalescpl-io/ciphertrust-transparent-encryption-kubernetes.git
```

How to Deploy pods/services in Kubernetes Cluster

The CTE for Kubernetes images are distributed through the Thales Docker Hub.

- All of the pods/services are deployed using `yaml` files. The `yaml` files are executed using the provided `deploy.sh` scripts.
- CTE for Kubernetes **only** supports homogeneous Kubernetes clusters, i.e. all nodes of the Kubernetes clusters must be x86_64 or all nodes must be arm64.

Options for Deploy Scripts

Option	Function	Description
-r	remove	Remove all the running pods, services and secrets.
-t	tag	Tag of the image on the server. **Default** is the latest.
-o	--operator	Deploy the CTE for Kubernetes Operator and CSI driver.
	--operator-ns=	The namespace in which to deploy the Operator.
	--cte-ns=	The namespace in which to deploy the CSI driver.
	--cri-sock=	Container Runtime Interface socket path.

For CTE for Kubernetes v1.3.0 and subsequent versions, the deployment script uses the repository image index as the image name, instead of the individual platform image names. The manifest (`cte_csi : <tag>`) points to an image.

Note

The **default** image name is: `cte_csi`. You do not need to specify the image name if you use the default name.

Deploy all of the pods by using the following command and needed arguments from the above table:

1. Change to the CTE for Kubernetes directory, type:

```
cd ciphertrust-transparent-encryption-kubernetes
```

2. Deploy a specific image:

```
./deploy.sh <repository image-name> -t <image_tag>
```

Examples:

```
./deploy.sh cte_csi-amd64 -t 1.3.0.15

./deploy.sh cte-csi-arm64 -t 1.3.0.15
```

Terminating Pods

To terminate all of the pods and delete all of the services and secrets:

```
cd ciphertrust-transparent-encryption-kubernetes
./deploy.sh --remove
```

Verify CTE for Kubernetes

Verify that CTE for Kubernetes is running, type:

```
kubectl get pods --namespace=kube-system -o wide | grep cte-csi
```

RESPONSE

```
NAME
READY   STATUS    RESTARTS        AGE     IP              Node
kube-system    cte-csi-controller-5db888d6cb-tn6lr
3/3     Running   0               6m59s   10.244.1.5      ubuntu20-02-
kubcluster-worker    <none>           <none>
kube-system    cte-csi-node-lz7t9
4/4     Running   0               6m59s   10.244.0.26     ubuntu20-02-
kubcluster-master    <none>           <none>
kube-system    cte-csi-node-pzvwb
4/4     Running   0               6m59s   10.244.2.2      ubuntu20-02-
kubcluster-worker2   <none>           <none>
kube-system    cte-csi-node-wmhhl
4/4     Running   0               6m59s   10.244.1.4      ubuntu20-02-
kubcluster-worker    <none>           <none>
```

Using External Certificates with CTE for Kubernetes

You can use External Certificates for communication between CTE for Kubernetes and CipherTrust Manager.

Note

Install the external certificate before registering CTE for Kubernetes with CipherTrust Manager.

Overview

CipherTrust Transparent Encryption can now use an external certificate, available at a user-defined path, to communicate with CipherTrust Manager.

Prerequisites

The external certificate must be:

- On the file system
- In PEM format

A key pair must already exist for the client:

- Must have Encryption type of either:
  - sha256WithRSAEncryption
  - ecdsa-with-SHA384
- Must be Encrypted with a pass phrase

Initial setup

1. Obtain your external CA certificate.
2. Create a certificate using the external CA certificate and key.

CipherTrust Manager Setup

To setup CipherTrust Manager to communicate through an external certificate:

1. Import the CA certificate into the CipherTrust Manager, click **CA** > **External** > **Add External CA**.
2. In the **Add External CA** dialog, copy and paste the `<ca_certificate_name>.pem` file content from the UI page and provide a user-friendly name.
   For more information, see Using an Externally Generated Server Certificate for an Interface
3. Add the CA certificate to the list of trusted sources for the web interface, click **Admin Settings** > **Interfaces** > **web** > **Edit** > **External Trusted CAs**.
4. Restart the web server, click **Admin Settings** > **Services** > **web** > **Restart**.
5. Create a registration token for the CTE agent. See Creating a Registration Token for more information.

CTE for Kubernetes Setup for Client Registration

1. Add the Registration token to the configuration file `cte-csi-regtoken.yaml` with the `registration_token` parameter.

```
apiVersion: v1
kind: Secret
metadata:
    name: cm-reg-token
type: Opaque
data:
 This is a base64 encoded registration token.  To generate:
 echo -n <CM REGISTRATION TOKEN STRING> | base64 -w 0
    cm-reg-token: <registration token in base64 format>
```

2. Create a Kubernetes secret(s) with client certificate details:

```
kubectl create secret generic <client-secret-name> --from-
file=<clientName>.crt=client_cert.pem --from-
file=<clientName>.key=client_key.pem --from-
file=<clientName>.passphrase=<passphrase>
```

where:

* `<client-secret-name>` = unique name for secret
* `<clientName>` = unique name for client
* `<passphrase>` = passphrase for client

Example

```
kubectl create secret generic <client-secret-name> --from-
file=client1.crt=client_cert.pem --from-
file=client1.key=client_key.pem --from-
file=client1.passphrase=passphrase
```

You can also use files from a **specific path** as a Partial or Absolute path:

```
kubectl create secret generic <client-secret-name> --from-
file=<clientName>.crt=./client1/client_cert.pem --from-
file=client1.key=./client1/client_key.pem --from-
file=client1.passphrase=./client1/passphrase
```

To create multiple client certificate details (certificate, key and passphrase) with the secret:

```
kubectl create secret generic <client-secret-name> --from-
file=<client1Name>.crt=<client1_cert.pem> --from-
file=<client1Name>.key=<client1_key.pem> --from-
file=<client1Name>.passphrase=<client1-passphrase>  --from-
file=<client2Name>.crt=<client2_cert.pem> --from-
file=<client2Name>.key=<client2_key.pem> --from-
file=<client2Name>.passphrase=<client2-passphrase>
```

where:

* `<client-secret-name>` = unique name for secret
* `<clientName>` = unique name for client
* `<passphrase>` = passphrase for client

Example

```
kubectl create secret generic client1-client2-secret --from-
file=client1.crt=client1_cert.pem --from-
file=client1.key=client1_key.pem --from-
file=client1.passphrase=client1-passphrase  --from-
file=client2.crt=client2_cert.pem --from-
file=client2.key=client2_key.pem --from-
file=client2.passphrase=client2-passphrase
```

If certificates are on **different paths**, provide the filename with the path (Partial or Absolute path)

```
kubectl create secret generic client1-client2-secret --from-
file=client1.crt=./client1/client_cert.pem --from-
file=client1.key=./client1/client_key.pem --from-
file=client1.passphrase=./client1/passphrase --from-
file=client2.crt=./client2/client_cert.pem --from-
file=client2.key=./client2/client_key.pem --from-
file=client2.passphrase=./client2/passphrase
```

Note

There must be an additional client certificate detail added to the secret `parameter external_ca_client_secret: <client-secret-name>`. This additional detail is required for each storage class deployment. This is because there is an additional requirement for `Signer Registration` for each storage class. This Signer Registration is an additional default registration).

3. Create the storage-class yaml file with a client secret name:
   While creating the storage-class configuration file `cte-storageclass.yaml` add the following parameter:

```
external_ca_client_secret: <client-secret-name>
```

   Also make sure that the registration token secret name from the `cte-csi-regtoken.yaml` is correct and contains the following parameter:

```
registration_token_secret: cm-reg-token
```

cte-storageclass.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: csi-test-sc
provisioner: csi.cte.cpl.thalesgroup.com
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
parameters:

Domain name or IP address of the CipherTrust Manager (Required)

    key_manager_addr: 192.168.70.1

Name of the CipherTrust Manager K8s Storage Group. (Required)
```

```
    k8_storage_group: test-group

Kubernetes Secret with CipherTrust Manager registration token (Re
quired)

    registration_token_secret: cm-reg-Token

Kubernetes secret with External CA signed client certificate (Opt
ional)

    external_ca_client_secret: <client-secret-name>

Description will be displayed in ${cm} (Optional)

    client_description: "K8s client"

Time in minutes to wait before unregistering from the ${cm}. Defa
ult is 10 minutes.
Once all volumes have been unguarded. Parameter must be added as
a string integer value. (Optional)

    registration_period: "10"
```

Note

- Each storage-class will have its own client secret with a client certificate, key and passphrase details.
- If one storage-class is being used over multiple worker nodes, then the client secret will have multiple client details (certificate, key and passphrase)

Deploy a CTE for Kubernetes Volume

- Preparing CipherTrust Manager
- Creating a CTE-CSI-Policy
- Prepare a file system persistent volume
- Deploying CTE Kubernetes Storage Classes
- Using Dynamic PVCs with CTE for Kubernetes
- Creating a CTE for Kubernetes Claim Against the PVC
- Create a CTE-CSI protected Persistent Storage Claim by deploying an application pod
- Using Taints and Tolerations
- Allow only Trusted Pods to mount to CTE for Kubernetes volumes

Preparing CipherTrust Manager

Note

Make sure that all of the names are written in lower case letters or else the yaml file will generate errors.

Managing Kubernetes Storage Groups and Clients

For information on how to create and manage Kubernetes (K8s) storage groups, K8s clients, apply GuardPolicies to storage groups, and protect K8s clients, see the CipherTrust Manager K8s documentation:

- Managing Kubernetes Storage Groups
- Managing Kubernetes Clients
- Protecting Kubernetes Clients

Create a CTE Policy for Kubernetes

For the generic instructions on how to create a CTE policy, see Policies

Specifically, when creating a policy for protecting Persistent Volumes in CTE for Kubernetes: CTE for Kubernetes

1. Create a policy with type : **CTE for Kubernetes**
2. For the CTE CSI policy name, use the name listed for the policy parameter: `csi.cte.cpl.thalesgroup.com/policy` in the `cte-csi-claim.yaml` file.
3. Make the GuardPolicy name the same name as the CSI policy.
4. Attach this GuardPolicy to the K8s Storage Group.

> **Note**
>
> CTE for Kubernetes only supports **User Sets** with a UID and GID. User names and Group names are **not** supported.

Prepare a file system persistent volume

Prerequisite

1. Create an NFS Server.
2. Export the path.
3. In the example below, the `yaml` file uses the NFS server `10.10.10.10` and the exported path is: `/my/nfs/volume/path`.

The following example explains how to create a persistent storage for CTE for Kubernetes to guard. Find more information on creating Persistent Volumes at:

- Persistent Volumes

Create a Persistent Storage (PV) YAML file description

> **Note**
>
> Replace the dummy entries for path and server with your own working NFS configuration.

nfs-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
    name: nfs-test-pv
spec:
    capacity:
      storage: 1Gi
    accessModes:
    -  ReadWriteMany
    storageClassName: nfs
    persistentVolumeReclaimPolicy: Retain
    mountOptions:
        - hard
        - nfsvers=3.0
    nfs:
        path: /my/nfs/volume/path
        server: 10.10.10.10
```

Create a Persistent Storage Claim (PVC) YAML file description

nfs-claim.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
    name: nfs-test-claim
spec:
  storageClassName: nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Note

The PV name is not referenced in the PVC. This particular claim will claim any PV that matches the requirements.

Apply Persistent Storage YAML files

Type:

```
kubectl apply -f nfs-pv.yaml
kubectl apply -f nfs-claim.yaml
```

Deploying CTE Kubernetes Storage Classes

For information on K8 storage classes, see Storage Classes for more information.

All of the Kubernetes clients that you want to attach to a storage group must have the same Kubernetes Namespace and Kubernetes structureless pods.

To deploy a storage class for CTE for Kubernetes:

1. Create and save a registration token in CM. See Tokens for more information.
   Select **Base64** format for the registration token, if using CipherTrust Manager v2.10 and subsequent versions.
2. Create a client group in CM. See Creating a Client Group for more information.

3. If using CipherTrust Manager v2.9 or previous versions, encode the token in base64 format, type:

```
echo -n <CM REGISTRATION TOKEN STRING> | base64 -w 0
```

4. Copy the base64 encoding to create a Kubernetes secret YAML file, **cte-csi-cmtoken.yaml**:

```
apiVersion: v1
kind: Secret
metadata:
  name: <CHANGE to name of the K8s secret. For example: cm-reg-
token>
type: Opaque
data:
  # This is a base64 encoded registration token.  To generate:
  # echo <CM REGISTRATION TOKEN STRING> | base64 -w 0
  registration_token: bWlEaUJlZ08xNkNsbndqZmc4a1dvcU1SUG9uaVpnNkV
tUjVYSGFLUVZVTHRhbGRrb0M5T1ZwTEpvTXp4UldmSQ==
```

5. The name of the K8s secret must be embedded in the `registration_token_secret` parameter in the storage class YAML file. Use the YAML file, **cte-storageclass.yaml** and fill in the appropriate values.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: <CHANGE to name of the Kubernetes Storage Class. For
example: csi-test-sc>
provisioner: csi.cte.cpl.thalesgroup.com
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
parameters:

  # Domain name or IP address of the CipherTrust Manager
```

```
(Required)
  key_manager_addr: <CHANGE_ME to your CM IP ADDR>

  # Name of the CipherTrust Manager K8s Storage Group. (Required)
  k8_storage_group: <CHANGE to the name of the Kubernetes Storage
Group. For example: test-group>

  # Kubernetes Secret with CM registration token (Required)
  registration_token_secret: <CHANGE to the K8s secret. For
example: cm-reg-token>

  # Time in minutes to wait before unregistering from the
CipherTrust Manager

# once all of the volumes have been unguarded. Parameter must be
added as a string
  # integer value. Default is 10 minutes. (Optional)
  registration_period: "10"
```

6. Record the storage class name, for further use. Deploy by typing:

```
kubectl apply -f cte-csi-cmtoken.yaml
kubectl apply -f cte-storageclass.yaml
```

Using Dynamic PVCs with CTE for Kubernetes

Static provisioning for Persistent Volumes (PV) requires the administrator to make assumptions, in order to create PVs that applications may need. As your Kubernetes environment expands, this can become a bottleneck.

Dynamic provisioning solves this issue. Instead of the Kubernetes administrator creating specific PVs, the administrator defines Storage Classes. Each Storage Class has a specific storage pool from which PVs can be provisioned automatically to meet an application's requirements.

Kubernetes provides a variety of internal provisioners. With dynamic provisioning, a developer can use a PVC to request a specific storage type and have a new PV provisioned automatically. The PVC must request a StorageClass that has already been created and configured on the target cluster, by an administrator, for dynamic provisioning to work.

CTE for Kubernetes now allows you to deploy HELM charts that use a StorageClass as input for creating volumes. Helm Charts help you define, install, and upgrade Kubernetes applications, thereby helping you manage Kubernetes clusters.

Specifying a Storage Class on a HELM chart means that the cluster will select volumes from the pool of volumes for that storage class. If no volumes exist, but the CTE-CSI driver has the capabilities for automatic provisioning, a new volume will be created and added to the cluster. Traditional CTE for Kubernetes PVCs must be specified with a source PVC in the definition, which makes it incompatible with this type of deployment. By using HELM charts, this will allow CTE for Kubernetes volumes to pass enough information to a source Storage Class so that dynamic provisioning in a CTE for Kubernetes PVC can dynamically provision and attach a data source PVC based on the PVC specifications.

Specifically, with dynamic PVCs now being supported, it changes the creation method for PVs. The CTE-K8s StorageClass is created with the new parameters and the CTE-K8s PVC is created without sourcePVC annotations, (policy annotation is optional). After the controller checks for the existence of policy and sourcePVC parameters, if it doesn't find sourcePVC parameters, it creates a new unprotected PVC (source PVC) using parameters from the CTE-K8s PVC and CTE-K8s StorageClass. CTE-K8s then binds the unprotected PVC to a PV if any qualifying PVs are available on the cluster. If not, it requests for a new PV to be provisioned. Then the CTE-K8s PVC provisions the PVC as normal. After that, the Application Pod can use the CTE-K8s PVC claim.

CTE-K8s Storage Class definition

To support dynamic PVCs, three new parameters must be added to the CTE-K8s Storage Class definition:

- source_storage_class
- default_policy
- allow_source_pvc_delete

Example

Following is an example of an entire Persistent Volume Claim set with the new parameters:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
name: csi-test-sc
provisioner: csi.cte.cpl.thalesgroup.com
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
parameters:
Domain name or IP address of the CipherTrust Manager (Required)
key_manager_addr: 192.168.70.1

Name of the CipherTrust Manager K8s Storage Group. (Required)
k8_storage_group: test-group

Kubernetes Secret with CM registration token (Required)
registration_token_secret: cm-reg-token

Kubernetes secret with External CA signed client certificate
(Optional)
kubectl create secret generic <secret-name> --from-file=<clientName>.c
rt=<client_cert.pem> --from-file=<clientName>.key=<client_key.pem> --f
rom-file=<clientName>.passphrase=<passphrase>
<clientName> i.e. client1 or client2 or client3 ...
Multiple client cert details can be added with same secret as well wi
th above command
external_ca_client_secret: <secret-name>

Small registration description to be displayed in the CipherTrust Man
ager (Optional)
client_description: "Describe your K8s client"

Time in minutes to wait before unregistering from the CipherTrust Man
```

```
ager
once all volumes have been unguarded. Parameter must be added as a st
ring
integer value. Default "10" minute. (Optional)
registration_period: "10"

When specified, this parameter automatically adds the
csi.cte.cpl.thalesgroup.com/source_pvc parameter to the CTE-K8s PVC b
ased
on the requested parameters (Optional)
source_storage_class: some_sc_name

When specified, this parameter automatically adds the
csi.cte.cpl.thalesgroup.com/policy parameter to the CTE-K8s PVC based
on
the requested parameters.  (Optional)(Required if
source_storage_class is set)
default_policy: <default_policy_name>

When specified and set to "true", this parameter automatically
deletes the dynamic sourcePVC,
and possibly, the actual data volume, depending on the provisioner dr
iver implementation.
If set to "false," you must manually delete the created source PVC. (O
ptional)
allow_source_pvc_delete: "false"
```

New Persistent Volume Claim Usage

The support for dynamic PVCs changes some of the rules for PVCs:

- The `source_pvc` and `policy` parameters are now optional if you use the new Storage Class parameters
- If `source_pvc` is specified, but the policy is not, then the PVC uses the Storage Class default_policy
- If a policy is specified, but the `source_pvc` is not specified, then the PVC uses the storage size when requesting a new volume

- Specifying both policy and `source_pvc` parameters will permit the cte-csi driver work as it did previously, meaning:
    - `source_storage_class` and `default_policy` parameters in the Storage Class are ignored
    - `storage parameter` in the Persistent Volume Claim is ignored

Example: Persistent Volume Claim

Using these new parameters, you could configure CTE for Kubernetes as follows:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: cte-claim1
annotations:
    csi.cte.cpl.thalesgroup.com/policy: policy_1
    csi.cte.cpl.thalesgroup.com/source_pvc: nfs-test-claim
spec:
storageClassName: csi-test-sc
accessModes:
    - ReadWriteMany
resources:
    requests:
    storage: 1Gi No longer ignored by the driver
```

CTE-CSI protected Persistent Storage Claim

Creating a CTE-CSI protected Persistent Storage Claim by deploying an application pod:

```
apiVersion: v1
kind: Pod
metadata:
name: cte-csi-demo
spec:
volumes:
```

```
    - name: test-vol
    persistentVolumeClaim:
        claimName:  cte-claim1
containers:
- name: ubuntu
    image: ubuntu
    volumeMounts:
    - mountPath: "/data"
        name: test-vol
    command:
    - "sleep"
    - "604800"
    imagePullPolicy: IfNotPresent
restartPolicy: Always
```

External Provisioner support with CTE-K8s for dynamically provision volume

If your driver is not capable of dynamically provisioning the persistent volume for a storage type, you can use an external provisioner for that specific storage type to create the volume dynamically and deploy with CTE for Kubernetes. Following are some external provisioners which provision data volumes dynamically for a storage class. You can use this storage class as the source StorageClass in cte-storageclass.yaml parameter (source_storage_class).

- NFS Subdirectory External Provisioner
- NFS CSI driver for Kubernetes
- AWS-EFS CSI dynamic provisioning
- Drivers list for different type of storage that may be used for Dynamic Volume Provisioning

After deploying a cte-csi-claim.yaml, the data persistent volume is created dynamically.

Creating a CTE for Kubernetes Claim Against the PVC

Note

For the next step, you must have your PV and PVC already deployed.

1. Create the following YAML file: **cte-csi-claim.yaml**

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: cte-claim
    annotations:
        The following must match your CTE Kubernetes Policy name.
        csi.cte.cpl.thalesgroup.com/policy: policy_1
        NFS source persistent volume claim
        csi.cte.cpl.thalesgroup.com/source_pvc: nfs-test-claim
spec:
    storageClassName: <CHANGE to the storageclass name that you d
eployed. For example: e.g. csi-test-sc>
    accessModes:
        - ReadWriteMany
resources:
    requests:
        storage: 1Ki
```

2. To deploy the `cte-csi-claim.yaml` script, type:

```
kubectl apply -f cte-claim.yaml
```

Create a CTE-CSI protected Persistent Storage Claim by deploying an Application/ Staging pod

To create a CTE-CSI protected Persistent Storage Claim by deploying an application pod:

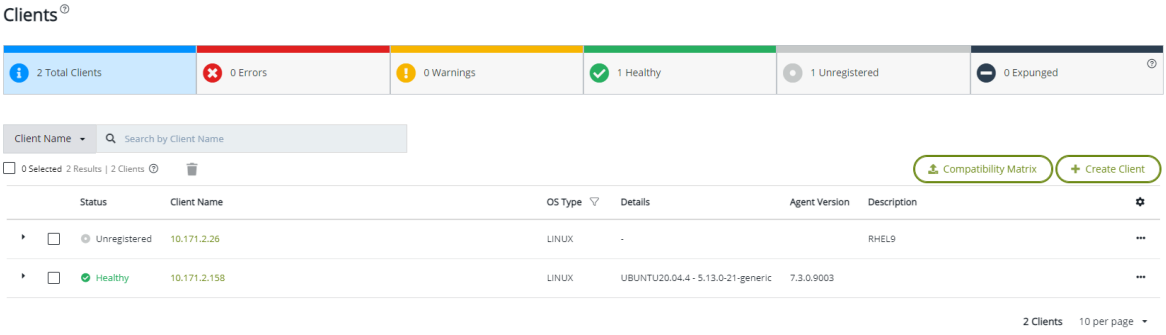
1. Create the yaml file to deploy a pod and protect the contents of the Persistent Volume.

```
apiVersion: v1
kind: Pod
metadata:
    name: cte-csi-demo
spec:
  volumes:
    - name: test-vol
      persistentVolumeClaim:
        claimName: cte-test-claim1
  containers:
  - name: ubuntu
    image: ubuntu
    volumeMounts:
      - mountPath: "/data"
        name: test-vol
    command:
     - "sleep"
     - "604800"
    imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

2. Deploy the pod, type:

```
kubectl apply -f cte-csi-demo.yaml
```

After the pod is successfully deployed and it running, you can see the client registered in K8s client.



3. To check the status of the pod, type:

```
root@ip-172-30-1-55:~# kubectl get all
```

```
NAME                                                    READY
STATUS    RESTARTS        AGE
pod/cte-csi-demo                                        1/1
Running   0               116m
pod/cte-csi-user-demo                                   1/1
Running   0               134m
pod/cte-staging-pod9jhtn                                1/1
Running   0               133m
pod/cte-staging-podt8jxs                                1/1
Running   0               116m

NAME                                    TYPE        CLUSTER-
IP         EXTERNAL-IP   PORT(S)    AGE
service/kubernetes                      ClusterIP
10.96.0.1          <none>        443/TCP    224d
```

After the pod is successfully deployed and running, you can see the clients registered in the Kubernetes client.

4. Login to the pod and check if the files are encrypted:

```
root@ip-172-30-1-55:# kubectl exec -it cte-csi-demo /bin/
bash
```

```
kubectl exec [POD] -- [COMMAND]
root@cte-csi-demo:/ cd /data/
```

```
root@cte-csi-demo:/data mkdir sub_dir
root@cte-csi-demo:/data cd sub_dir/
root@cte-csi-demo:/data/sub_dir echo "testfile" >> test
root@cte-csi-demo:/data/sub_dir
root@cte-csi-demo:/data/sub_dir cat test
testfile
```

5. Check the same file on the Server side:

```
root@aws-thales-dockerregistry:~ cd /nfs-share/sub_dir/
root@aws-thales-dockerregistry:/nfs-share/sub_dir ls -l
total 8
-rw-r--r-- 1 root root 4102 Dec 13 20:04 test
root@aws-thales-dockerregistry:/nfs-share/sub_dir cat test
EROV▒▒▒▒=▒bu▒▒4勵▒▒4▒▒▒1kCroot@aws-thales-dockerregistry:/nfs-sh
are/sub_dir#
```

Using Taints and Tolerations

> **Note**
>
> Using Taints and Tolerations is **optional**.

You can apply Taints and Tolerations to the CTE for Kubernetes application pod that you are running. You can also create your staging pod with toleration.

Taints allow a node to repel a set of pods. **Tolerations** are applied to pods. Tolerations allow the scheduler to schedule pods with matching taints as long as it's allowed by the other associated parameters.

Taints and Tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. Once one or more taints are applied to a node, the node should not accept any pods that do not tolerate the taints.

For more information, see Kubernetes Taints and Tolerations.

Using Taints and Tolerations with CTE for Kubernetes

CTE for Kubernetes now supports nodes with taints. You can specify a toleration in a pod_spec for the staging pod of a CTE for Kubernetes volume. To do this, CTE for Kubernetes added annotation support to a CTE for Kubernetes PVC. This annotation uses a base64 encoded string for input. Kubernetes requires the pod to pass the info to the cluster as a single base64 encoded string. The only way to create this is to manually encode the yaml file that contains the Taints and Tolerations. CTE for Kubernetes takes the yaml file and extracts any toleration definitions contained in it and appends them to the staging pod when that PVC is mounted anywhere in the cluster. Any values in the YAML file that are not related to toleration are ignored.

Note

To apply Taints and Tolerations to a pod, you **must** first shut down the pod. Additionally, support for toleration **must** be added at PVC creation time. Modifying an existing PVC with the added toleration is **not** supported because the CTE PVC was not created with the required immutable volume attributes.

Applying Taints and Tolerations

1. Create and add the taint to a node, type:

   ```
   kubectl taint nodes <nodeName> <keyName>=<valueName>:<taintEffect>
   ```

   Example:

   ```
   kubectl taint nodes node1 key1=value1:NoExecute
   ```

2. Create a tolerations yaml file. The tolerations must match the taint created in the previous step. The file should look similar to the following:

   ```
   cat <cte_tolerations>.yaml
   ```

   Response:

```
spec:
    tolerations:
    key: "key1"
    operator: "Equal"
    value: "value1"
    effect: "NoExecute"
```

> **Note**
>
> The taint effect **NoSchedule** does not work with Taints and Tolerations for CTE for Kubernetes.

3. Write the tolerations yaml file to a text file while generating the base64 encoding and appending the file with the string, type:

```
# cat <cte_tolerations>.yaml | base64 -w0
```

Response:

```
c3BlYzoKICB0b2xlcmF0aW9uczoKICAtIGtleTogImtleTEiCiAgICBvcGVyYXRvc
jogIkVxdWFsIgogICAgdmFsdWU6ICJ2YWx1ZTEiCiAgICBlZmZlY3Q6ICJOb0V4ZW
N1dGUiCg==
```

4. Once the base64 string is generated, the PVC is appended with the string. For example:

```
cat <cte_pvc_fileName>.yaml
```

Response:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: cte-csi-claim
    annotations:
```

```
        csi.cte.cpl.thalesgroup.com/policy: <policyName>
        csi.cte.cpl.thalesgroup.com/source_pvc: <pvcName>-claim
        csi.cte.cpl.thalesgroup.com/spec_append: c3BlYzoKICB0b2xl
cmF0aW9uczoKICAtIGtleTogImtleTEiCiAgICBvcGVyYXRvcjogIkVxdWFsIgogI
CAgdmFsdWU6ICJ2YWx1ZTEiCiAgICBlZmZlY3Q6ICJOb0V4ZWN1dGUiCg==
spec:
    storageClassName: <storageClassName
    accessModes:
    - ReadWriteMany
resources:
    requests:
        storage: 10Gi
```

For more information, see the Command line tool (kubectl).

Allow only Trusted Pods to mount to CTE for Kubernetes volumes

CTE for Kubernetes allows only trusted pods to access protected data volumes and attach to CTE for Kubernetes claims if you activate this optional feature. CTE for Kubernetes uses signature sets to validate the pod as trusted. Signatures are in the form of key-value pairs which contain the image name and the corresponding hash value. After mapping the image signature with the received signature set, CTE for Kubernetes will allow, or prevent, mounting of the the encrypted volume.

Overview of CTE for Kubernetes for Support for Trusted Pods

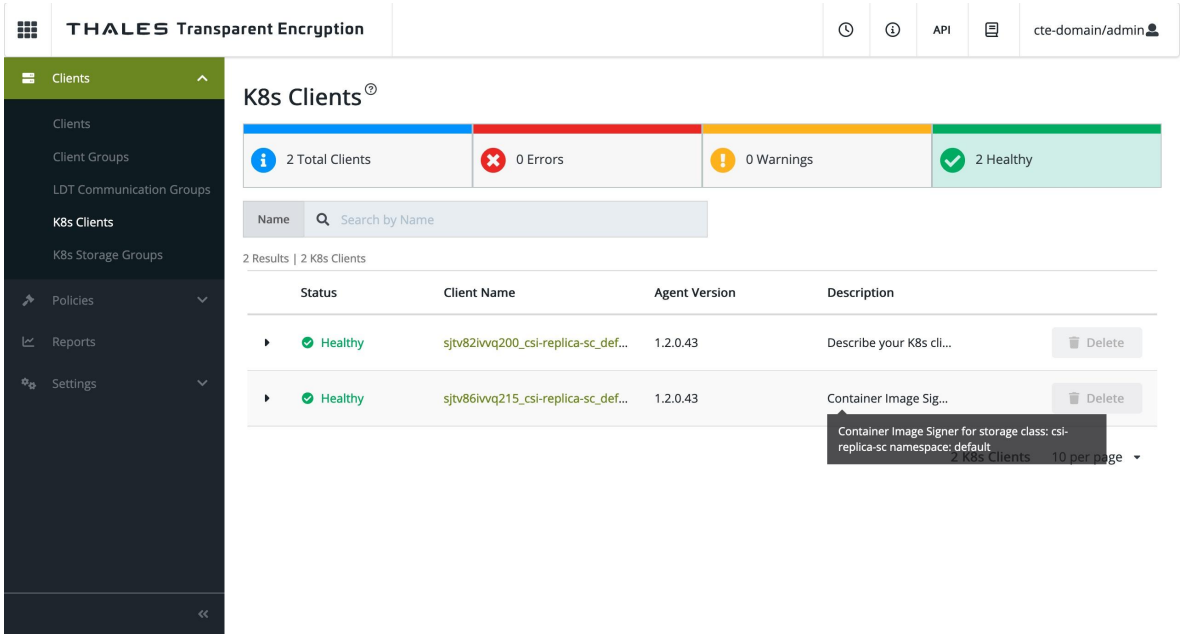
Support for Trusted Pods is disabled by default. At least one signature set, with a container image digest, must be attached to a security policy to enable this feature. When the request to mount/publish the volume is received:

1. CTE for Kubernetes determines whether Support for Trusted Pods is enabled or not by scanning for the presence of signature rules.

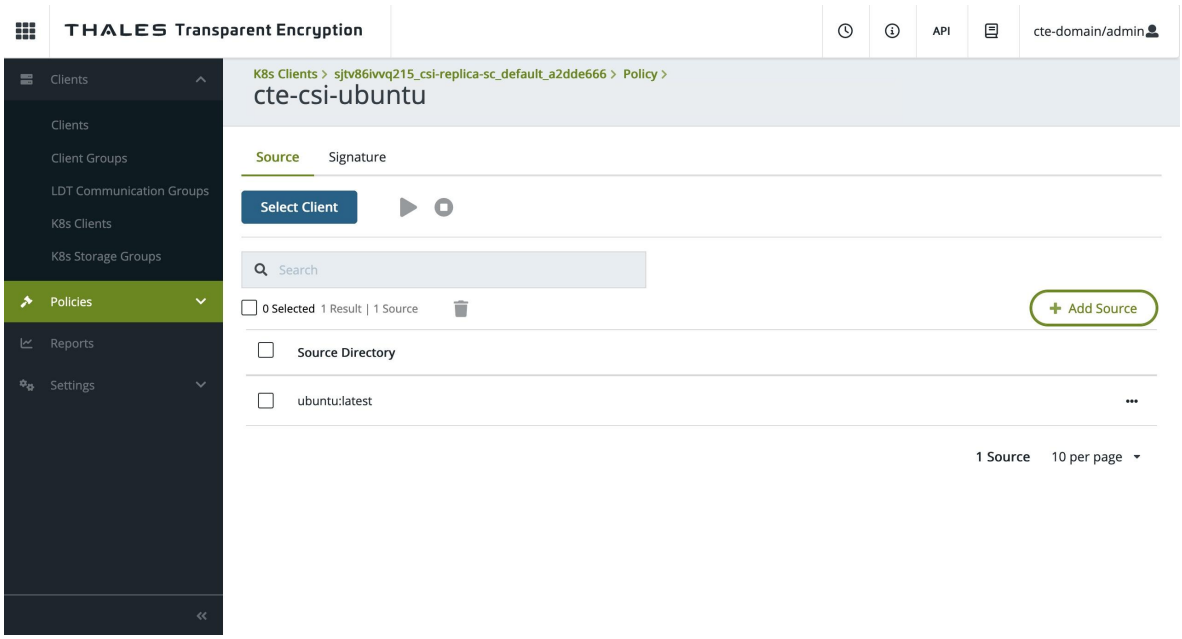
2. Support for Trusted Pods is skipped (default case) if the security policy does not include container signature sets.
3. CTE for Kubernetes fetches the digests for all containers in the running pod.
4. CTE for Kubernetes tries to match the digests of running pods to a container image signature set. Support for Trusted Pods is considered successful only if all digests within a running pod have an entry in same signature set.
5. CTE for Kubernetes checks the logs to determine if it succeeded. If no signature rules exist, the Support for Trusted Pods is disabled.
6. A new container named `cte-csi-signer` is added to a controller pod. It will monitor CTE for Kubernetes storage classes for the following operations: `creation, deletion` and `updating`. It manages automatic registration as a signer service to CipherTrust Manager.

Workflow for obtaining a Signature Set on CipherTrust Manager and adding it to a Policy

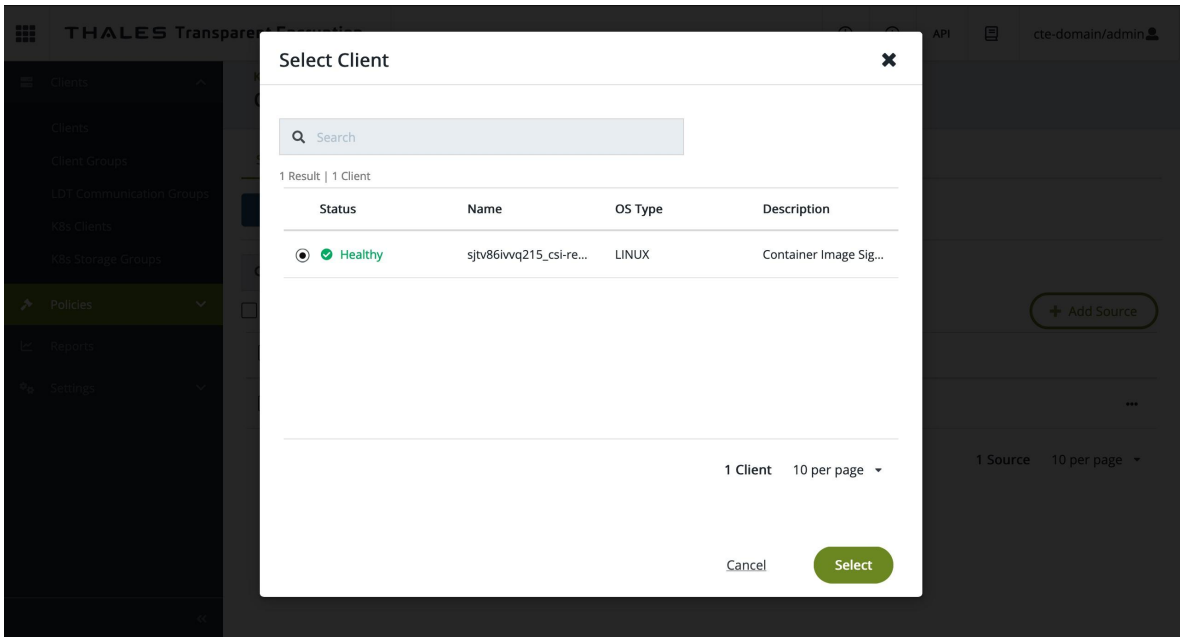
1. After signing the container image for a specific storage class and namespace, it should display in CipherTrust Manager in the Kubernetes client.



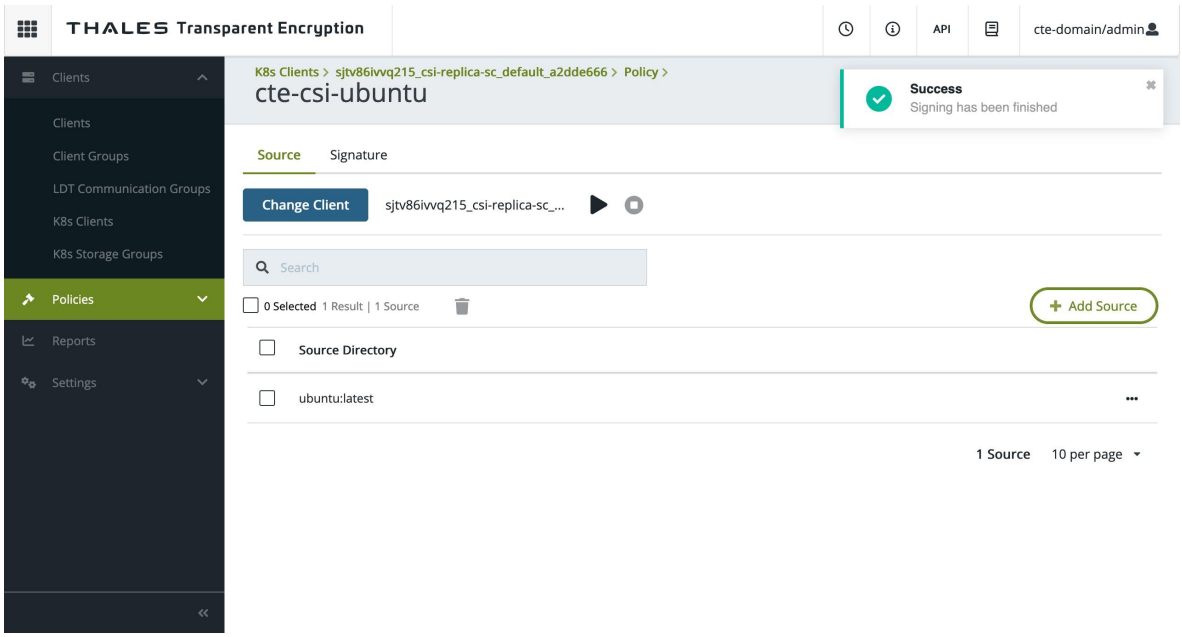
2. Add a container image source and select a client for signing in the Signature Set page.



3. Selecting a client for signing.



4. Displays a message after a successful signing operation.



Creating a Signature Set on CipherTrust Manager and adding it to the Policy

Create the signature set in CipherTrust Manager:

1. See Creating Signature Sets for Container Images. Follow those instructions for creating your Signature Set.
2. Add the following as a signature rule in the Kubernetes policy to enable Trusted Pod enforcement:
    - **Signature set, which contains all of the signatures for all of the containers to which you want to have access**

> **Note**
>
> Unlike normal CipherTrust Manager signature sets, CTE for Kubernetes signature sets **do not** attach to process sets.

Data Transformation for CTE for Kubernetes

Using Initial Data Transformation for CTE for Kubernetes

The Data Transformation utility is an application that encrypts data-in-place in a GuardPoint. To perform dataxform for CTE for Kubernetes, two additional steps are required:

1. An Data Transformation GuardPolicy, along with a CTE for Kubernetes production policy, must be added to the CTE for Kubernetes storage group on CipherTrust Manager.
2. You must add an Data Transformation policy name in the CTE for Kubernetes claim against the PVC. The name of the claim is `cte-csi-claim.yaml`.

For the standard Data Transformation steps, see Data Transformation for more information.

Prerequisites

- Verify that you have valid backup files of the data to be encrypted.

Note

You will need to stop **ALL** access and services to the data being encrypted during part of this procedure. Access will **NOT** be restored until the encryption process is complete. Make sure that you plan for this outage and that users know the data will be inaccessible for some time.

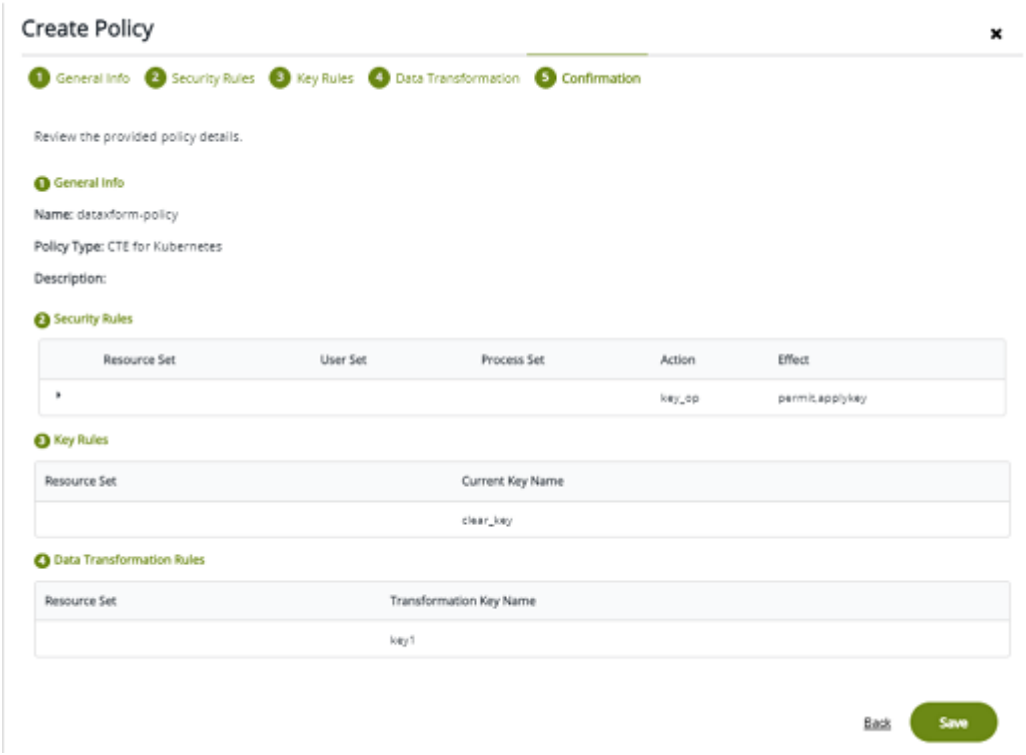
Create a Storage Group

Prior to the deployment of any `yaml` files for registration, you must create a CTE for Kubernetes Storage Group.

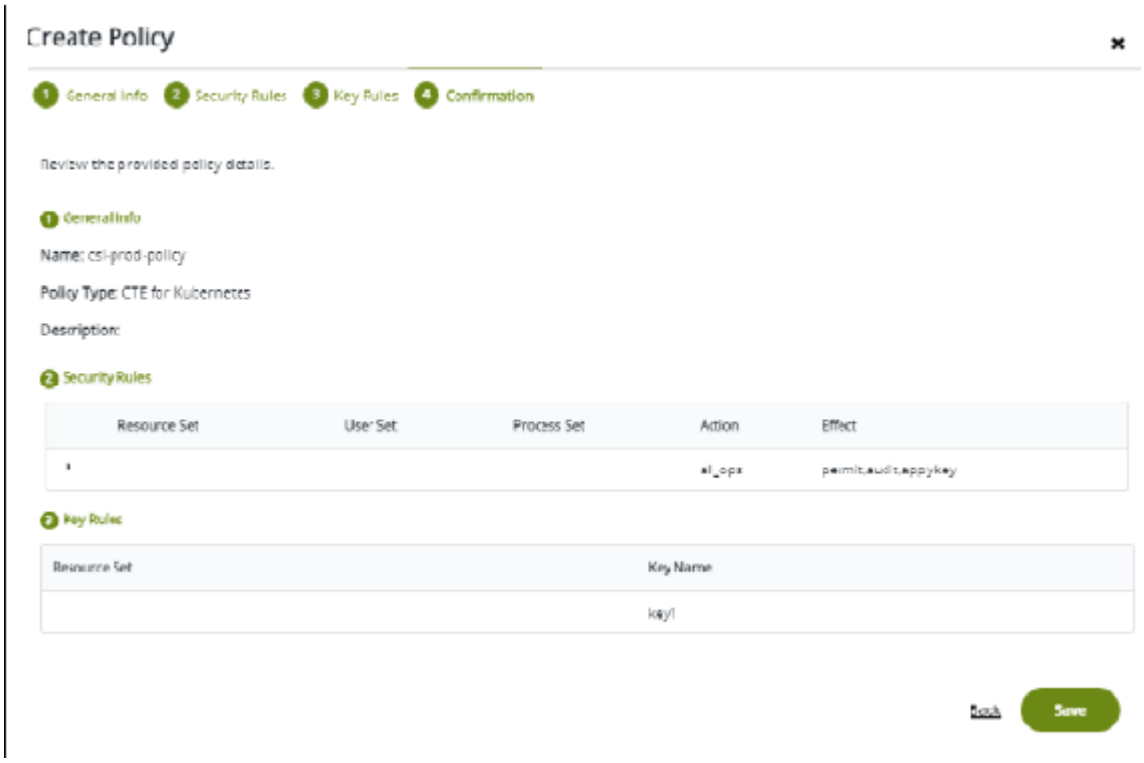
Create Policies

In CipherTrust Manager, you must create policies before you can transform the data. See Creating Policies for more information.

1. Create an **Data Transformation policy** with **Policy Type**: CTE for Kubernetes.
2. Toggle the **Data Transformation** button to **on**. This policy is only for encrypting the clear text data in persistent volumes where the policy is applied.



3. Create a **CTE for Kubernetes Production policy** with **Policy Type**: CTE for Kubernetes.
4. Make sure that the Data Transformation toggle is set to **off**.
5. For the key rule, you must use the **same key** as in the Data Transformation policy.



Apply GuardPolicies

- Add both policies, that you just created in the previous section, to the K8s Storage Group.

Adding an Annotation for Data Transformation to a claim

- Add an annotation for the Data Transformation policy to your `yaml` claim file. For example:

  cte-csi-claim.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: cte-claim
    annotations:
        csi.cte.cpl.thalesgroup.com/dataxform_policy: <dataxformP
olicyName>
        csi.cte.cpl.thalesgroup.com/policy: production-policy  //T
his must match your CTE CSI Policy name.
```

```
        csi.cte.cpl.thalesgroup.com/source_pvc: nfs-test-
claim  // nfs source persistent volume claim
spec:
    storageClassName: <CHANGE to the storageclass name that you d
eployed. For example: e.g. csi-test-sc>
    accessModes:
        - ReadWriteMany
resources:
    requests:
        storage: 1Ki
```

Note

- The Data Transformation will start as soon as the protected pod is deployed using the above cte-claim. Check that the logs are updated or described in the Application Pod.

Warning

- Do not delete the protected pod while Data Transformation is in progress.
- The `dataxform_auto_lock` file is created in the GuardPoint path. Do not edit/delete this file.

Applying Data Transformation Key Rotation to CTE for Kubernetes

Data encryption keys are the keys used to encrypt data in a GuardPoint. Key rotation is the process of changing the encryption key used to encrypt your GuardPoint data. Key Rotation is also called Rekeying. Changing GuardPoint encryption keys increases security. Rekeying with Data Transformation requires that you change the current key of the production policy on your GuardPoint to a new key.

Warning

All users and applications will be blocked from accessing the data until Data Transformation is finished executing. Make sure that you plan for this outage and that users know the data will be inaccessible for some time.

Performing a Key Rotation

Delete Pods and PVC

1. Delete all of the application pods which are using `cte-claim`:

```
kubectl delete -f <applicationPodName>.yaml
```

Note

Users can also use the CTE PVC (# `kubectl describe pvc <CTE-PVC>`) to find the list of all of the pods that are using a specific CTE PVC.

2. Delete the CTE PVC:

```
kubectl delete -f cte-csi-claim.yaml
```

Create New Polices and Keys

Create a new Data Transformation and a new production Policy, with required Key rules on CipherTrust Manager.

Alternatively, (recommended), clone the original policies, (Data Transformation and Production policy) that you used to encrypt the data.

- If creating new policies, make sure to set **Policy Type: CTE for Kubernetes**.
- Toggle the Data Transformation button to **on** while creating the Data Transformation policy.
- Toggle the Data Transformation button to **off** while creating the production policy.

Updating Key Rules

Update the key rules, for both policies, by editing the Key Rule, and the Data Transformation Rule, and changing the Current Key Name and Transformation Key Name to reflect the new key to use.

For Example:

If the GuardPoint data is encrypted with Key1, and you want to rotate the key from Key1 to Key2:

Data Transformation clone/new policy key rule update:

```
Key Rules (Current Key Name): key1
Data Transformation Rules(Transformation Key Name): key2
```

Production clone/new policy key rule update:

```
Key Rules(Key Name): key2
```

Applying the New/Cloned Policies

1. Add the newly created/cloned policies to CipherTrust Manager K8s Storage Groups. See Configuration Concepts for more information about creating CTE for Kubernetes storage groups in CipherTrust Manager.
2. Update the data transformation and production policy names in the CTE claim yaml file (`cte-csi-claim.yaml`) with the following annotations.

   ```
   csi.cte.cpl.thalesgroup.com/dataxform_policy:
   <NewDataxformPolicy>    // newly created dataxform policy
   csi.cte.cpl.thalesgroup.com/policy:
   <NewProductionPolicy>  // newly created production policy
   ```

   Example

   ```
   apiVersion: v1
   kind: PersistentVolumeClaim
   metadata:
       name: cte-claim
   ```

```
    annotations:
        csi.cte.cpl.thalesgroup.com/dataxform_policy: dataxform-
key1-to-key2-rotation-policy  // newly created dataxform standard
policy
        csi.cte.cpl.thalesgroup.com/policy: csi-prod-with-key2-
policy    // newly created production policy
        csi.cte.cpl.thalesgroup.com/source_pvc: nfs-test-
claim   // nfs source persistent volume claim spec:
    storageClassName: <CHANGEME to the storageclass name deployed
e.g. csi-test-sc>
    accessModes:
        - ReadWriteMany
    resources:
        requests:
            storage: 1Ki
```

3. Deploy the CTE claim file `cte-csi-claim.yaml`:

```
kubectl apply -f cte-csi-claim.yaml
```

4. Add the `dataxform_cleanup` annotation to the NFS source PV from `nfs-pv.yaml`. See Create a Persistent Storage (PV) YAML file description for more information.

```
# kubectl annotate pv <SOURCE_PV_NAME>
csi.cte.cpl.thalesgroup.com/dataxform_cleanup='require'
```

`**<SOURCE_PV_NAME>**`: PersistentVolume name from `nfs-pv.yaml` file

5. Deploy the Application Pod which uses the `cte-claim`.

```
# kubectl apply -f <applicationPodName>.yaml
```

Note

- When Data Transformation is running, the Guard Policy does not display as active on CipherTrust Manager, because it does not send any Guard Policy details to CipherTrust Manager. Only after the production policy is applied to the GuardPoint do the details populate CipherTrust Manager.
- The Data Transformation will start rekeying as soon as the protected pod is deployed. Check that the logs are updated or described in the Application Pod.

Warning

- Do not delete the protected pod while Data Transformation is in progress.
- The `dataxform_auto_lock` file is created in the GuardPoint path. Do not edit/delete this file.

Decryption using Data transformation with CTE for Kubernetes

GuardPoint data can be unencrypted using Data Transformation.

The steps are same as with the section Applying Data Transformation Rotation for CTE for Kubernetes, except for policy creation which requires specific key rules for decryption.

Create policies for decryption with required keys on CipherTrust Manager

1. Create new policies (Data Transformation standard policy and Production policy) with required key rules.
    - **Alternatively, clone the original policies that you used to encrypt the data.**
2. Update the key rules for both of the policies by editing the Key Rule, and the Data Transformation Rule, and changing the Current Key Name and Transformation Key Name to `clear_key`.
   For Example:
   If GuardPoint data is encrypted with Key2:
   Data Transformation clone/new policy key rule update:

   ```
   Key Rules (Current Key Name): Key2
   Data Transformation Rules(Transformation Key Name): Clear_key
   ```

   Production clone/new policy key rule update:

   ```
   Key Rules(Key Name): Clear_key
   ```

3. Save and apply the policies.

Upgrading CTE for Kubernetes

Introduction

Upgrading CTE for Kubernetes to the latest release is accomplished by invoking the `deploy.sh` script with the desired release tag. The CTE for Kubernetes node server cannot be automatically upgraded. Doing this will result in application failures for existing pods using CTE volumes on the nodes. You must use a manual procedure to stop the pods running on the nodes and delete the CTE for Kubernetes node server pods on that node.

Redeploy CTE for Kubernetes

To redeploy CTE for Kubernetes, type:

```
./deploy.sh -t <version#>.<build#>
```

For example:

```
./deploy.sh -t 1.0.0.1223
```

Drain and update a node

To upgrade the nodes:

1. Get a list of nodes currently running CTE for Kubernetes, type:

   ```
   kubectl get pods -n kube-system -o wide -l app=cte-csi-node --no-headers -o custom-columns=":spec.nodeName"
   ```

   Response Example:

   ```
   ub20-master
   ub20-work1
   ```

2. Drain a node from the list, type:

   ```
   kubectl drain --delete-emptydir-data --ignore-daemonsets <nodeName>
   ```

   Note

   The drain command pauses while it waits for the node to be drained.

   Response example:

```
node/ub20-work1 cordoned
node/ub20-work1 drained
```

3. Use a compound kubectl command to delete the CTE for Kubernetes node server on this node:

```
kubectl delete -n kube-system pod $(kubectl get pods -n kube-system --field-selector spec.nodeName=<nodeName> -l app=cte-csi-node --no-headers -o custom-columns=":metadata.name")
```

   - The `kubectl get pods` needs to be bounded to only output the CTE for Kubernetes-node label for this node
   - Use `--field-selector spec.nodeName=<nodeName>` to specify the CTE for Kubernetes node to drain.
   - Use `-l app=cte-csi-node` to select only the CTE for Kubernetes node server from that node.
   - Use `--no-headers -o custom-columns=":metadata.name"` to tell `kubectl` to only output the pod names.
   - Pass the output of that command to the `kubectl delete` pod which triggers a new CTE for Kubernetes node server pod to be created with the desired image.

4. Verify that the pod has started successfully, type:

```
kubectl get pods -n kube-system --field-selector spec.nodeName=<nodeName> -l app=cte-csi-node -o wide
```

Response example:

```
NAME                  READY   STATUS    RESTARTS   AGE
IP               NODE          NOMINATED NODE   READINESS GATES
cte-csi-node-kgwkj   4/4     Running   0          14m
192.168.77.163   ub20-work1   <none>           <none>
```

5. Return node to working status, type:

```
kubectl uncordon <nodeName>
```

Response example:

```
node/ub20-work1 uncordoned
```

Introduction to the Kubernetes Operator

Note

- The CTE for Kubernetes feature is supported with CTE for Kubernetes v1.2.0 and subsequent versions.
- CTE for Kubernetes is certified with Redhat and supported for Openshift Container Platform for CTE for Kubernetes v1.2.0 and subsequent versions.
- CTE-Kubernetes Operator version v1.3.2 has been certified with Redhat and published on the Redhat portal: RedHat Portal.
- For Kubernetes cluster, including managed Kubernetes clusters in the cloud, namely Google GKE cluster, Amazon EKS cluster and Microsoft's AKS cluster, the operator is also certified and available at: Kubernetes Catalog.
- CTE for Kubernetes Operator supports both x86_64 and arm64 deployment

Kubernetes has various default resources like Pod, Deployment, DaemonSet, etc. When you define a manifest for instantiating one of those resources, you must specify the **Kind** as Pod/Deployment/DaemonSet etc. Kubernetes provides a default set of controllers that understand the resource definitions and know how to manage their life cycle. For instance, the Deployment Controller manages the Create/Update/Delete of the Deployment resource.

Kubernetes architecture allows users to extend the API server in a way that users can create their own custom resource (CR) and write their own controller to manage the custom resource. An operator bundles a CR and the Controller that manages the CR. An operator can watch resources across the cluster and take actions when required. For more information on how to use operators to manage other applications, refer to the Kubernetes Operator pattern.

The CTE for Kubernetes Operator can deploy, monitor, upgrade and delete CTE for Kubernetes. When the CTE for Kubernetes Operator is deployed, its controller deploys the CTE for Kubernetes driver on the OpenShift cluster. The manifests required to deploy the CTE-K8s driver are bundled with the operator.

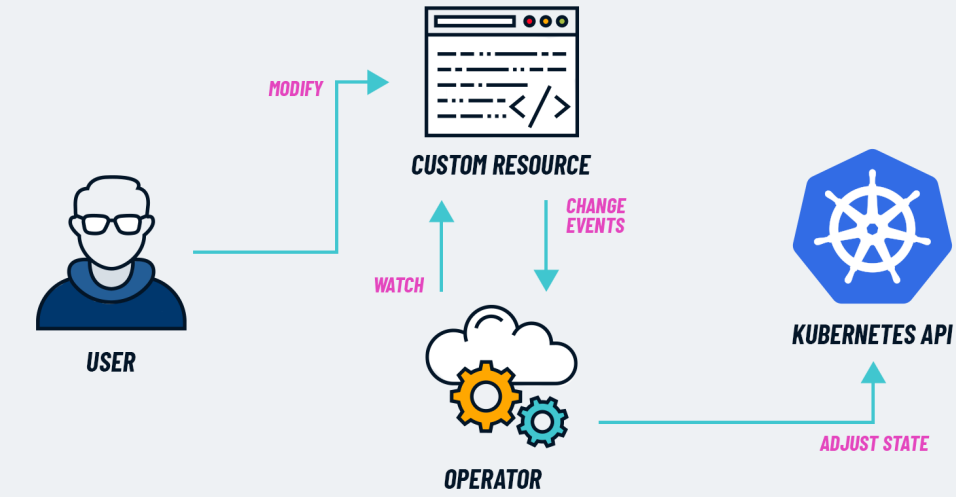


Image source: https://www.cncf.io/blog/2022/06/15/kubernetes-operators-what-are-they-some-examples/

CTE for Kubernetes Operator

The CTE for Kubernetes Operator is an Openshift operator that Thales created for CTE for Kubernetes. This operator contains a CR, an API for managing a CR, and a custom controller that manages this resource. When you install the CTE for Kubernetes Operator on an OpenShift cluster, the operator registers the new CR and the controller with the Kubernetes API server. Whenever the API server receives a request to create a resource, where Kind=`CTEK8sOperator`, it passes on the request CTE for Kubernetes Operator's Custom Controller. The controller contains all of the logic needed to complete tasks before, during or after the deployment of the CTE for Kubernetes driver.

You can deploy CTE for Kubernetes using a Custom Resource Definition (CRD). The following displays a sample CRD used to create an instance of the CTE for Kubernetes:

CTE-K8S-Operator-crd.yaml

```
apiVersion: cte-k8s-operator.csi.cte.cpl.thalesgroup.com/v1
kind: CteK8sOperator
```

```
    metadata:
        labels:
            app.kubernetes.io/name: ctek8soperator
            app.kubernetes.io/instance: ctek8soperator
            app.kubernetes.io/part-of: cte-k8s-operator
            app.kubernetes.io/managed-by: kustomize
            app.kubernetes.io/created-by: cte-k8s-operator
        name: ctek8soperator
    spec:
        replicas: 1
        image: "docker.io/thalesciphertrust/ciphertrust-transparent-
encryption-kubernetes"
        version: "1.2.0-latest"
        imagePullPolicy: Always
        logLevel: 5
        apiburst: 300
        apiqps: 200
        imagePullSecrets:
            - name: cte-csi-secret
        registrationCleanupInterval: 10
        pauseimage: "k8s.gcr.io/pause:latest"
```

Applying a CRD

To apply the CRD, type:

```
    kubectl apply -f `<path to\cte-k8s-operator.yaml>`
```

Installing the CTE for Kubernetes Operator | CLI Method

Warning

- Do not uninstall CTE for Kubernetes and the operator from the CLI, if they were installed using the GUI.
- Similarly, do not uninstall CTE for Kubernetes and the operator from the GUI, if it was installed using the CLI.

Note

You can install CTE for Kubernetes and the Operator **only** in the Openshift-operators namespace. If you want to install in other namespaces, use the CLI option.

The CTE for Kubernetes Operator can be installed using one of two methods:

- **CLI**
- **Cluster Console GUI**

CLI Method Prerequisites

1. Install the Operator Lifecycle Manager (OLM) on the cluster. Refer to Installing OLM for instructions on how to install OLM on the cluster.

   Note

   The latest version of Openshift is installed by default.

2. Install `oc` (Openshift CLI command for Openshift cluster) on the cluster.

> **Note**
>
> The user installing the Operator must have Cluster Admin permissions.

3. Download the Deploy Scripts.

4. Execute the `deploy.sh` script from the deploy directory, type:

```
./deploy.sh --operator --operator-ns=<namespace-in-which-to-deploy-the-operator> --cte-ns=<namespace-in-which-to-deploy-cte-4-k8s>
```

   If either of the namespace options is not specified, the script sets `kube-system` as the default namespace for deployment.

Note

Ensure that the namespace passed to the deployment script exists before initiating deployment. This prevents the script from prompting for creation of namespace during deployment. For example, if the deployment script is invoked as:

```
./deploy.sh --operator --operator-ns=my-ns1 cte-ns=my-ns2
```

where both namespaces `my-ns1` and `my-ns2`, do not exist, the script would prompt with the following:

```
./deploy.sh --operator --operator-ns=my-ns1 cte-ns=my-ns2
```

Response

```
Starting the cte-csi containers
NAMESPACE my-ns1 not found!!!!!!
Namespace my-ns1 not found. Do you want to create it now [N/y]
```

Once the namespace information is available, the script proceeds to create the objects required and installs the operator. After installing the operator, the script deploys `ctek8soperator` CRD. This deploys CTE for Kubernetes on the cluster in the namespace specified.

Installing CTE for Kubernetes

To install CTE for Kubernetes (when CTE for Kubernetes Operator is already installed):

1. Navigate to the deploy directory that was downloaded for operator install.
2. Ensure the file `ctek8soperator-crd.yaml` has correct values.

3. Type:

```
# oc apply -f <path to>/ctek8soperator-crd.yaml
```

Updating CTE for Kubernetes

To update CTE for Kubernetes:

1. Stop any application that is using CTE for Kubernetes volumes.
2. Navigate to the deploy directory that was downloaded for operator install.
3. Edit the `ctek8soperator-crd.yaml` file.
4. Update the `version` field in the spec section to the latest version of CTE for Kubernetes.
5. Apply the change with the command:

```
# oc apply -f <path to>/ctek8soperator-crd.yaml
```

Deleting CTE for Kubernetes

To delete CTE for Kubernetes:

1. Stop any application that is using CTE for Kubernetes volumes.
2. Navigate to the deploy directory that was downloaded for operator install.
3. Type:

```
# oc delete -f <path to>/ctek8soperator-crd.yaml
```

Uninstalling CTE for Kubernetes Operator

To uninstall the CTE for Kubernetes Operator:

1. Stop any application that is using CTE for Kubernetes volumes.

2. Navigate to the deploy directory that was downloaded for operator install, type:

```
# ./deploy.sh --operator --operator-ns=<namespace-in-which-ctek8soperator-is-deployed> --cte-ns=<namespace-in-which-ctek8s-is-deployed> --remove
```

This removes both CTE for Kubernetes and the operator.

Using the Cluster Console Web GUI

The CTE for Kubernetes Operator is certified with Red Hat for the Openshift platform. It is integrated with OperatorHub. The operator can be discovered on the OperatorHub page.

1. Open a browser and navigate to the **Operators > OperatorHub** link, in the left navigation panel on the console GUI. Type `CTE` in the search field under **All Items** to find the CTE for Kubernetes Operator.
2. Click on the tile to access the install option.
3. Ensure that all prerequisites are met before installing the operator.
4. Click **Install** to install the operator. Do not change the default values on the install page.

Deploying the CTE for Kubernetes Operator after installation

Once the operator is installed, the Kubernetes API servers becomes aware of the Kubernetes customer resource (CR). The installation process registers the:

- CR
- API for managing the CR
- Controller that handles the requests, for the CR, from the API Server

To instantiate the CR:

1. Click **View Operator** on the page displayed immediately after the operator is installed.

2. Alternatively:

    a. Expand “Operators” section and click **Installed Operators** page from the left hand bar on the page.

    b. Click on the Name of the operator installed.

    c. Click **Create Instance** link on the page displayed.

    d. Click **Create** to deploy CTE for Kubernetes.

Upgrading CTE for Kubernetes

1. Check the Docker site for a new version of CTE for Kubernetes:
2. Stop any application that is using CTE for Kubernetes volumes.
3. Expand “Operators” section and click **Installed Operators** page from the left hand bar on the page.
4. Click on the Name of the operator installed.
5. On the next page, select the **YAML** tab. This displays the manifest of CTE for Kubernetes instances installed on the cluster.
6. Update the version parameter in the CRD spec with the updated CTE-K8s version.
7. Click **Save**.
   The upgrade process terminates all of the CTE for Kubernetes pods running the previous version, while activating new pods running the new version. Once the `cte-csi-node-XXXX` pods are running on all of the nodes of the cluster, the application that uses the CTE for Kubernetes volume can be re-deployed.
8. If any of the other parameters in `CTE-K8s-Operator.yaml`, like logLevel, apiburst, apqps, etc. need to be updated, follow the same process as above to update the relevant parameter.

Uninstalling CTE for Kubernetes Operator

To uninstall CTE for Kubernetes Operator:

1. Stop any application that is using CTE for Kubernetes volumes.
2. Expand “Operators” section and click **Installed Operators** page from the left hand bar on the page.
3. Click on the Name of the operator installed.
4. On the next page, select the **Actions** tab.
5. Click **Uninstall Operator** in the list of actions displayed.
6. Click **Uninstall** on the confirmation pop-up to uninstall CTE for Kubernetes.

Upgrading CTE for Kubernetes Operator

The operator is designed to upgrade to a newer version automatically as soon as a new version is published by Thales. The Operator Lifecycle Manager, or OLM, constantly polls for updates and upgrades the operator whenever an update is available.

Troubleshooting

This section contains the following troubleshooting topics:

- Backing up Databases after Encryption
- FSGroupID is not working

Inspecting Deployment

Deployment of CTE for Kubernetes typically uses a single controller pod and at least one worker pod per Kubernetes node. Inspect the deployment using `kubectl get` commands to access basic output.

Example

Get the CTE for Kubernetes pod names that were deployed on the Kubernetes cluster, type:

```
NAMESPACE="kube-system"
kubectl get pods -n ${NAMESPACE} -o wide | grep cte-csi
```

Response

```
    cte-csi-controller-0      2/2     Running   0      25h
192.168.77.182   ub20-work1    <none>   <none>
    cte-csi-node-28srz        4/4     Running   0      25h
192.168.77.181   ub20-work1    <none>   <none>
    cte-csi-node-cjzzn        4/4     Running   0      25h
192.168.76.68    ub20-master   <none>   <none>
```

Getting CTE for Kubernetes Agent version

The version of CTE for Kubernetes that you are running displays in the log files for each pod:

```
    NAMESPACE="kube-system"
    for podName in `kubectl get pods -n ${NAMESPACE} | grep cte-csi-
[node,controller] | cut -d " " -f1`; do
        echo -n "${podName}:  "
        kubectl logs -n ${NAMESPACE} ${podName} cte-csi | grep "Versi
```

```
on:"
    done
```

Response

```
        cte-csi-controller-7d7d8bd46b-wbffd:  I0308 17:25:13.608929 1
cte.go:139] Version: 1.2.0.123
        cte-csi-node-8qx5q:  I0308 17:25:11.643673 2218968 cte.go:
139] Version: 1.2.0.123
        cte-csi-node-h8q8q:  I0308 17:25:13.465795 2205645 cte.go:
139] Version: 1.2.0.123
```

Inspecting Events

CTE for Kubernetes relies on Kubernetes event infrastructure for diagnostics of problems with a CTE for Kubernetes volume. Any errors in attaching a volume to a pod will display in the event logs. Look at all of the events in a Kubernetes namespace by typing:

```
    NAMESPACE="kube-system"
    kubectl get event -n ${NAMESPACE}
```

To target events of a specific pod, type:

```
    PODNAME="my_application_pod"
    NAMESPACE="kube-system"
    kubectl get event -n ${NAMESPACE} --field-selector involvedObject.
name=${PODNAME}
```

Inspecting CTE for Kubernetes logs

Examining logs for CTE for Kubernetes could offer supplementary insights that might not be conveyed through Kubernetes Events.

Deployment of CTE for Kubernetes is split between two different types of pods:

- **Controller Server**: Manages the dynamic provisioning of CTE for Kubernetes persistent volumes. Denoted by a `cte-csi-controller-X` pod name.

- **Node server**: Manages attaching CTE for Kubernetes volumes to pods. Denoted by a `cte-csi-node-XXXXX` pod name.

Inspecting the Controller Server

Inspect the logs for the controller server to debug CTE for Kubernetes persistent volume provisioning issues:

```
NAMESPACE="kube-system"
kubectl logs -n ${NAMESPACE} cte-csi-controller-0 cte-csi
```

Inspecting the Node Server

The logs for the CTE for Kubernetes pods are distributed across two containers:

- The `cte-csi container` contains logs relevant to CTE for Kubernetes activity, encompassing details like volume mounting and registration
- The `cte-agent-logs` for the CTE encryption engine agent can be found in the container.

Inspecting the Node Server requires first identifying which Kubernetes node the application pod is scheduled on. Once that node has been identified, find the CTE for Kubernetes node server running on that node.

After you obtain the pod name, use the two relevant `kubectl logs` commands to view the logs, type:

```
NAMESPACE="kube-system"
kubectl logs -n ${NAMESPACE} cte-csi-node-XXXXX cte-csi
kubectl logs -n ${NAMESPACE} cte-csi-node-XXXXX cte-agent-logs
```

Problems with Registration

CTE for Kubernetes automatically registers to CipherTrust Manager based on demand for volumes on a node. Failure to register is typically due to the registration token being either invalid or the token has no more client capacity. These types of errors are reported back through Kubernetes so analyzing the log files of the troubled pod reveals the registration failure message seen by the agent.

Troubleshooting Trusted Pods failures

The following are two examples of trusted pod failures:

Example 1

In `cte-csi-node` logs, the following error indicates that a running pod digest was not found in any signature set attached to a security policy.

```
    E0310 06:41:16.614410 1687839 server.go:106] GRPC error: rpc err
or: code = Internal desc = Pod did not pass attestation checks: rpc e
rror: code = Internal desc = Found no signature set for container ubu
ntu with digest 2adf22367284330af9f832ffefb717c78239f6251d9d0f58de50b8
6229ed1427
```

Example 2

In `cte-csi-node` logs, the following error indicates that running pod digests cannot be matched to the same signature set. There were two containers (ubuntu and ubuntu2) with digests included in different signature sets. Partial matches are displayed to help with troubleshooting.

```
    E0407 09:07:33.393609 1492044 server.go:106] GRPC error: rpc err
or: code = Internal desc = Pod did not pass attestation checks: rpc e
rror: code = Internal desc = Pod attestation failed! Unable to match a
ll pod digests to same signature set. Partial matches:
    Signature set policy-sigset2: [ubuntu:
8ae9bafbb64f63a50caab98fd3a5e37b3eb837a3e0780b78e5218e63193961f9]
    Signature set policy-sigset3: [ubuntu2:69665d02cb32192e52e07644d76
bc6f25abeb5410edc1c7a81a10ba3f0efb90a]
```

Backing up Databases after Encryption

After encrypting a database, CipherTrust Transparent Encryption cannot make a backup of the database. Both scheduled and manual backup fail. The problem was the user's policy. A policy used in this scenario must follow a few rules.

With a CBC_CS1 key, a guarded file is modified to have a 4096 byte header holding key information. When an **Apply Key** effect is specified, the CipherTrust Transparent Encryption code adjusts the length and file offset for this header. Without an **Apply Key** effect, the size and access of the offset include the CBC_CS1 header.

Thales recommends that you modify the first rule of your policy. Remove the action entry for `f_rd_att` from the first rule and add a new rule before it:

```
    **action**: f_rd_att

    **effect**: Permit, Apply Key
```

Policy processing starts with the first rule and continues until a matching rule is found. The effect for the matching rule is then applied.

For the `f_rd_att` action, this results in the Secfs code including the CBC_CS1 key header and adjusts the file size value. Without the Apply Key effect, the file size includes the CBC_CS1 header size and the file appears as 4096 bytes larger than its real size.

FSGroupID is not working with NFS shared storage volume

The `fsgroup` ID Security Context option allows an administrator to change the permissions of volumes before a pod starts. Some users have found that when adding the `fsgroup` ID to the SecurityContext section in the pod yaml file, it does not work as expected in the NFS storage environment.

The reason is that the `fsgroup` ID Security Context option is not supported with NFS volumes. It is only supported with local storage. This is a limitation from Kubernetes and not an issue with the CTE-U fuse driver.

- See Configure a Security Context for a Pod or Container in the Kubernetes documentation for more information.

Support Contacts

If you encounter a problem while installing, registering, or operating the product, please refer to the documentation before contacting support. If you cannot resolve the issue, contact your supplier or Thales Customer Support.

Thales Customer Support operates 24 hours a day, 7 days a week. Your level of access to this service is governed by the support plan arrangements made between Thales and your organization. Please consult this support plan for further information about your entitlements, including the hours when telephone support is available to you.

Customer Support Portal

The Customer Support Portal, at Thales Customer Support, is where you can find solutions for most common problems. The Customer Support Portal is a comprehensive, fully searchable database of support resources, including software and firmware downloads, release notes listing known problems and workarounds, a knowledge base, FAQs, product documentation, technical notes, and more. You can also use the portal to create and manage support cases.

Tip

You require an account to access the Customer Support Portal. To create a new account, go to the portal and click on the REGISTER link.

Telephone Support

If you have an urgent problem, or cannot access the Customer Support Portal, you can contact Thales Customer Support by telephone at +1 410-931-7520. Additional local telephone support numbers are listed on the support portal.

Email Support

You can also contact technical support by email at technical.support@Thales.com.