



ENTRUST

Kubernetes and Entrust KeyControl Secrets Vault

Integration Guide

2025-04-24

Table of Contents

1. Introduction	1
1.1. Integration architecture	1
1.2. Product configurations	1
1.3. Requirements	2
2. Procedures	3
2.1. Install docker	3
2.2. Install k3s	4
2.3. Deploy the KeyControl vault cluster	7
2.4. Create a secrets vault to be used with Kubernetes	7
2.5. Create a secret in the secrets vault	8
2.6. Set up and configure the integration environment	11
2.7. Test the integration	23
3. Troubleshooting	31
3.1. Look at the logs	31
3.2. Look at the init container	31
4. Additional resources and related products	32
4.1. nShield Connect	32
4.2. nShield as a Service	32
4.3. KeyControl	32
4.4. KeyControl as a Service	32
4.5. nShield Container Option Pack	32
4.6. Entrust products	32
4.7. nShield product documentation	32

Chapter 1. Introduction

This guide describes how to integrate a Kubernetes cluster with KeyControl Secrets Vault.

Kubernetes is an open-source system that automates the deployment, management, and scaling of containerized applications. It makes it easy for developers to quickly build, launch, and scale container-based web applications in a public cloud environment.

This integration allows pulling secrets from a secrets vault in KeyControl Vault and mount them as either environment variables or as volume mounts in containers. It focuses on the way one can pull secrets into Kubernetes pods or containers using a KeyControl Secrets Vault. For other details on the vault, please refer to the Entrust KeyControl Vault (KCV) documentation.

1.1. Integration architecture

Kubernetes cluster

In this integration, a Kubernetes K3s cluster is deployed on a Red Hat Linux VM. Container images are used from a third-party cloud registry.

Container images

Two container images are created for the purpose of this integration to demonstrate how secrets can be pulled into a container from KeyControl Vault.

Two more images are deployed to support the integration. These images come from the [PASM Vault Kubernetes Agent v1.0](https://github.com/EntrustCorporation/PASM-Vault-Kubernetes-Agent/releases). They are available at <https://github.com/EntrustCorporation/PASM-Vault-Kubernetes-Agent/releases>.

Docker Registry

An external Docker registry is required. This is where the container images from the PASM Kubernetes agents will be stored and referenced by the Kubernetes containers when they are created.

1.2. Product configurations

Entrust has successfully tested the integration KeyControl Secrets Vault with Kubernetes in the following configurations:

Product	Version
Base OS	Red Hat Enterprise Linux release 9.4 (Plow)
Kubernetes (K3s)	1.30.4
KeyControl Vault	10.3.1
PASM Vault Kubernetes Agent	1.0

1.3. Requirements

1.3.1. Before starting the integration process

Familiarize yourself with:

- The documentation for the Entrust KeyControl Vault.
- The documentation and setup process for a Kubernetes cluster.

Chapter 2. Procedures

2.1. Install docker

On the RedHat Linux Server, install docker.

1. Make sure system is up to date.

```
% sudo yum update
```

2. Remove Older Docker Versions.

If you have any older versions of Docker installed, it's essential to remove them along with their associated dependencies. Use the following command to uninstall older Docker packages:

```
% sudo yum remove docker \  
docker-client \  
docker-client-latest \  
docker-common \  
docker-latest \  
docker-latest-logrotate \  
docker-logrotate \  
docker-engine
```

3. Install Required Dependencies.

To prepare your system for Docker installation, install the necessary dependencies:

```
% sudo yum install -y yum-utils device-mapper-persistent-data lvm2
```

4. Add Docker Repository.

Next, you need to add the official Docker repository to your system:

```
% sudo yum-config-manager --add-repo https://download.docker.com/Linux/centos/docker-ce.repo
```

5. Install Docker.

With the repository added, you're now ready to install Docker:

```
% sudo yum install docker-ce
```

6. Start and Enable Docker.

After the installation is complete, start the Docker service and enable it to start on boot:

```
% sudo systemctl start docker
% sudo systemctl enable docker
```

7. Verify Docker Installation.

Confirm that Docker is installed and running by checking its version:

```
% sudo docker --version
```

8. Test Docker with a Hello World Container.

Test your Docker installation by running a "Hello World" container:

```
% sudo docker run hello-world
```

9. Managing Docker as a Non-root User.

To avoid using the `sudo` command with every Docker command, you can add your user to the `docker` group:

```
% sudo usermod -aG docker <your_userid>
```

10. Reboot the system for the changes to take effect.

2.2. Install k3s

We will use k3s environment to deploy our Kubernetes cluster to demonstrate this integration. It is up to the user to select and use the best Kubernetes environment of choice for a Kubernetes cluster. You can find information on how to deploy a k3s Kubernetes cluster here: <https://k3s.io/>.

On the RedHat Linux Server, install the Kubernetes Cluster.

1. Install the latest k3s release.

To install the latest k3s stable release do the following:

```
% curl -sL https://get.k3s.io | sh -
```

```
[INFO] Finding release for channel stable
[INFO] Using v1.30.4+k3s1 as release
[INFO] Downloading hash https://github.com/k3s-io/k3s/releases/download/v1.30.4+k3s1/sha256sum-amd64.txt
[INFO] Downloading binary https://github.com/k3s-io/k3s/releases/download/v1.30.4+k3s1/k3s
[INFO] Verifying binary download
[INFO] Installing k3s to /usr/local/bin/k3s
[INFO] Finding available k3s-selinux versions
Updating Subscription Management repositories.
Rancher K3s Common (stable)
3.3 kB/s | 1.3 kB    00:00
Dependencies resolved.
```

```
=====
```

Package	Architecture	Size	Version
Repository			

```
=====
```

Installing:

k3s-selinux	noarch	22 k	1.5-1.e19
rancher-k3s-common-stable			

Transaction Summary

```
Install 1 Package
```

```
Total download size: 22 k
Installed size: 96 k
Downloading Packages:
k3s-selinux-1.5-1.e19.noarch.rpm
68 kB/s | 22 kB    00:00
```

```
-----
```

```
Total
68 kB/s | 22 kB    00:00
Rancher K3s Common (stable)
35 kB/s | 2.4 kB    00:00
Importing GPG key 0xE257814A:
  Userid   : "Rancher (CI) <ci@rancher.com>"
  Fingerprint: C8CF F216 4551 26E9 B9C9 18BE 925E A29A E257 814A
  From     : https://rpm.rancher.io/public.key
```

```
Key imported successfully
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing      :
1/1
  Running scriptlet: k3s-selinux-1.5-1.e19.noarch
1/1
  Installing     : k3s-selinux-1.5-1.e19.noarch
1/1
  Running scriptlet: k3s-selinux-1.5-1.e19.noarch
1/1

  Verifying      : k3s-selinux-1.5-1.e19.noarch
1/1
Installed products updated.
```

```
Installed:
  k3s-selinux-1.5-1.e19.noarch
```

```
Complete!
```

```
[INFO] Creating /usr/local/bin/kubectl symlink to k3s
[INFO] Creating /usr/local/bin/crictl symlink to k3s
[INFO] Skipping /usr/local/bin/ctr symlink to k3s, command exists in PATH at /usr/bin/ctr
```

```
[INFO] Creating killall script /usr/local/bin/k3s-killall.sh
[INFO] Creating uninstall script /usr/local/bin/k3s-uninstall.sh
[INFO] env: Creating environment file /etc/systemd/system/k3s.service.env
[INFO] systemd: Creating service file /etc/systemd/system/k3s.service
[INFO] systemd: Enabling k3s unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s.service → /etc/systemd/system/k3s.service.
[INFO] systemd: Starting k3s
```

2. Set up **KUBECONFIG** for the user:

```
% export KUBECONFIG=~/.kube/config
% mkdir ~/.kube 2> /dev/null
% sudo /usr/local/bin/k3s kubectl config view --raw > "$KUBECONFIG"
% chmod 600 "$KUBECONFIG"
```

You may want to put this in your profile so when you login into the server it gets set:

```
export KUBECONFIG=~/.kube/config
```

3. Check the cluster.

```
% kubectl get nodes

NAME                 STATUS    ROLES                  AGE     VERSION
redhat-9-kcv-secrets Ready    control-plane,master   9m29s   v1.30.4+k3s1
```

4. Test the server connection.

Run the following command and notice the **server** attribute:

```
% kubectl config view

apiVersion: v1
clusters:
- cluster:
  certificate-authority-data: DATA+OMITTED
  server: https://127.0.0.1:6443
  name: default
contexts:
- context:
  cluster: default
  user: default
  name: default
current-context: default
kind: Config
preferences: {}
users:
- name: default
  user:
    client-certificate-data: DATA+OMITTED
    client-key-data: DATA+OMITTED
```

The **server*** attribute is set to:

```
server: https://127.0.0.1:6443
```

The KeyControl node will need access to that URL. Since it is using the localhost IP address, you can just replace that with the server IP address. Now open a browser and attempt to connect to that url. In our case:

```
https://1X.19X.14X.XXX:6443
```

Attempt the connection from another server in the same subnet as the KeyControl nodes. If you can't connect, the port may be blocked by the firewall.

Open port 6443 on the firewall in the server:

```
% sudo firewall-cmd --zone=public --add-port=6443/tcp --permanent
success
% sudo firewall-cmd --reload
success
```

Test the connection again.

Some VPN blocks access to port 6443. Make sure you test the connect from a server that will not use a VPN.

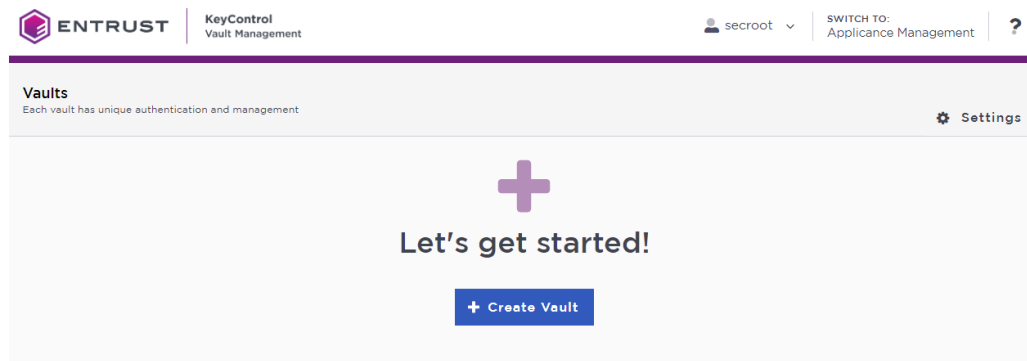
2.3. Deploy the KeyControl vault cluster

For this integration, KeyControl Vault is deployed as a two-node cluster.

Follow the installation and setup instructions in [KeyControl Vault Installation and Upgrade Guide](#).

2.4. Create a secrets vault to be used with Kubernetes

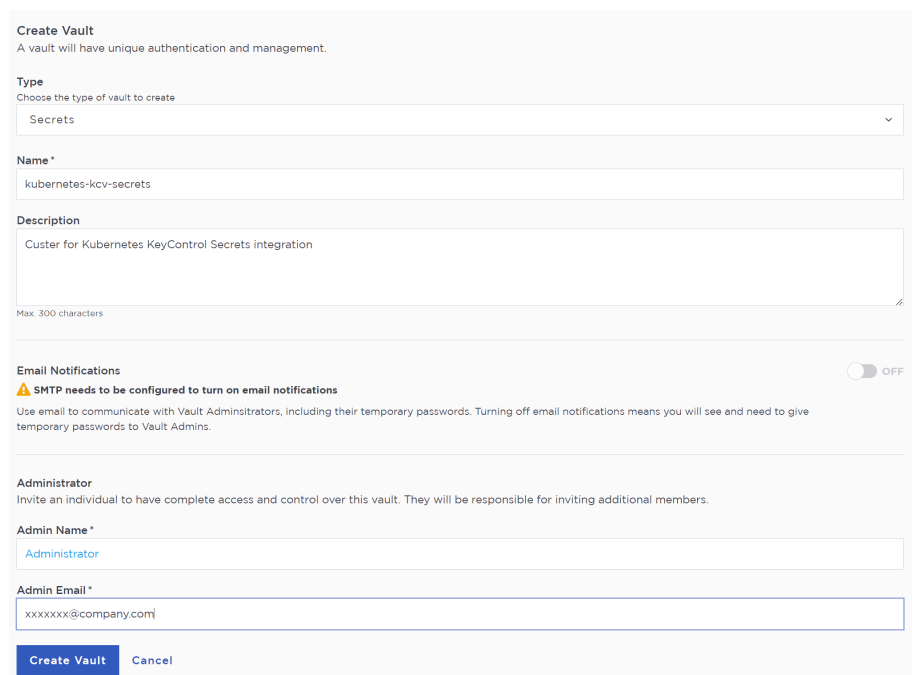
1. Sign in to the KeyControl Vault Manager.
2. In the home page, select **Create Vault**.



The **Create Vault** dialog appears.

3. In the **Type** drop-down box, select **Secrets**. Enter the required information.
4. Select **Create Vault**.

For example:



5. When you receive an email with a URL and sign-in credentials to the KeyControl vault, bookmark the URL and save the credentials.

You can also copy the sign-in credentials when the vault details are displayed.

6. Sign in to the URL provided.
7. Change the initial password when prompted.

2.5. Create a secret in the secrets vault

After you sign in to the secrets vault, create a box that will contain the secret.

1. Select **Manage** in the Secrets Vault **Home** tab, then select **Manage Boxes**.
2. In the **Manage Boxes** Tab, select **Add a Box Now**.
3. In the **Create a Box** Window, enter the **Name** and **Description**.







In this integration guide and the configuration file examples it contains, the box will be named **box1**.

4. Select **Continue**.

The screenshot shows a 'Create a Box' dialog box with a close button (X) in the top right corner. At the top, it says 'Create a Box' followed by a lock icon and 'box1'. Below this are three tabs: '1: About' (selected), '2: Checkout Details', and '3: Rotation Details'. The 'Name' field is labeled 'Name' with an information icon and an asterisk, and contains the text 'box1'. The 'Description' field is labeled 'Description' with a character count of '1990 Characters' and contains the text 'Box to be used in the secrets integration with kubernetes.'. Below the description field is the 'Secret Versions' section, which includes the text 'Maximum number of a secret's versions to keep before they are deleted.' and a text input field containing '10'. There is also a 'Secret Expiration' section with a checkbox that is unchecked, the text 'Set a default expiration duration for a secret. If not checked, the secrets in the box will not expire by default.', and a text input field followed by a dropdown menu currently set to 'Days'. At the bottom right of the dialog are two buttons: 'Cancel' and 'Continue'.

5. In **Checkout Details**, select **Continue**.
6. In **Rotation Details**, select **Create**. The box gets created.
7. Select the new box.
8. In the **Secrets** Pane, select **Add a Secret Now**.
9. From the **Choose a type of secret to create** list, select **Text**.

Choose a type of secret to create ✕

-  **ESXi Host**
Manage the password for an ESXi host
-  **File**
Upload a file
-  **Key-Value Pair**
Store Key-value pairs
-  **Password**
Generate and store a password
-  **Text**
Plain text based secret
-  **SSH Key**
Upload and manage SSH Key

10. In the **Create Secret: Text** window, enter the following:
 - a. **Name:** Enter the name of the secret. This will be used later in configuration files in this integration.
 - b. **Description:** Enter a brief description.
 - c. **Secret Data:** Enter the value of the secret. This is the value the Kubernetes containers will try to retrieve during the integration.

Create Secret: Text ✕

Name ⓘ *

Description 1991 Characters

Secret Data *
Add a secret consisting of plain text. 1951 Characters

[Show Advanced Options](#) ▼

[Cancel](#) [Create](#)

11. Select **Create**.

2.6. Set up and configure the integration environment

From this point on, all setup and configuration will be done from the Linux server that you configured earlier.

2.6.1. Set up KUBECONFIG

This should give you access to the Kubernetes cluster.

```
% export KUBECONFIG=~/.kube/config
```

Check that you can see the Kubernetes cluster nodes:

```
% kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
redhat-9-kcv-secrets	Ready	control-plane,master	2d2h	v1.30.4+k3s1

2.6.2. Set up namespaces

The integration will use two namespaces in Kubernetes.

1. Set up the mutating webhook namespace.
 - a. Create a **yaml** file containing the following code:

```
apiVersion: v1
kind: Namespace
metadata:
  name: mutatingwebhook
```

- b. Name the file **mutatingwebhooknamespace.yaml**
- c. Create the namespace:

```
% kubectl create -f mutatingwebhooknamespace.yaml
```

2. Set up the test namespace.
 - a. Create a **yaml** file containing the following code:

```
apiVersion: v1
kind: Namespace
metadata:
  name: testnamespace
```

- b. Name the file `testnamespace.yaml`.
- c. Create the namespace:

```
% kubectl create -f testnamespace.yaml
```

2.6.3. Register Docker container images with the Docker registry

Register some of the Docker container images to a Docker registry so they can be used inside the Kubernetes cluster.

1. Set up `DOCKER_CONFIG`

Export `DOCKER_CONFIG` to the directory where your docker configuration will be stored:

```
% export DOCKER_CONFIG=~/.docker
```

2. Log in to the registry:

```
% docker login -u YOURUSERID <registry-url>
```

3. Deploy the init container image.

The `init` container image needs to be deployed to the Docker registry.

Download the init container image:

```
% wget https://github.com/EntrustCorporation/PASM-Vault-Kubernetes-Agent/releases/download/v1.0/init-container.tar
```

4. Load the provided image `init-container.tar` into the Docker registry:

```
% docker load --input init-container.tar
```

5. Check the image:

```
% docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost/init-container	latest	1c476f57b72c	10 months ago	186 MB

6. Tag the image:

```
% docker tag localhost/init-container:latest <registry-url>/init-container
```

7. Push the image into the Docker registry that is used within Kubernetes:

```
% docker push <registry-url>/init-container:latest
```

8. Deploy the mutating webhook image.

The webhook code is in the form of image and needs to be deployed as container. It needs to be deployed to the Docker registry.

Download the webhook container image:

```
% wget https://github.com/EntrustCorporation/PASM-Vault-Kubernetes-Agent/releases/download/v1.0/mutating-webhook.tar
```

9. Load the provided image `mutating-webhook.tar` into the Docker registry:

```
% docker load --input mutating-webhook.tar
```

10. Check the image:

```
% docker images
REPOSITORY          TAG          IMAGE ID      CREATED      SIZE
localhost/mutating-webhook  latest      805efa734095  10 months ago  222 MB
```

11. Tag the image:

```
% docker tag localhost/mutating-webhook:latest <registry-url>/mutating-webhook
```

12. Push the image into the Docker registry that is used within Kubernetes:

```
% docker push <registry-url>/mutating-webhook:latest
```

2.6.4. Get the secrets vault authentication token for the vault admin user

1. Sign in to the secrets vault as admin using the KeyControl Vault Rest API and get the vault authentication token.

```
https://<VAULT-IP>/vault/1.0/Login/<VAULT-UUID>
```

This URL is visible on vault management UI of the KeyControl. Also note the **VAULT-UUID** from the URL. This will be required in further steps.

Here is an example:

```
https://xx.xxx.xxx.xxx/vault/1.0/Login/d86e3c22-6563-45b4-bfb9-45ba6c911ec8
```

The request body should be in the form:

```
{
  "username": "admin",
  "password": "password"
}
```

The **username** should be one of the administrators of the vault. The examples uses **admin**, but this should be the actual admin user name.

- 2. Get the token using the following curl command:

```
% curl -k -X POST https://xx.xxx.xxx.xxx/vault/1.0/Login/d86e3c22-6563-45b4-bfb9-45ba6c911ec8/ -d '{
"username": "admin", "password": "xxxxxxx" }'
```

The response body for the command above should be similar to this:

```
{
  "is_user": false,
  "box_admin": false,
  "appliance_id": "bc1a085c-70c9-476c-b4f6-abff0c73cce4",
  "is_secondary_approver": false,
  "access_token":
  "ZDg2ZTNjMjItNjU2My00NWl0LWJmYjktNDViYTZjOTExZWm4.eyJkYXRhIjoU0ZSUFVBRUFkYkktGNXNnWlBRSGJQNEQ3T2xTR2NXUHM2aVdqdzVjN21jRythdXZ5NUhIQjZYQ0tPRG9iK0FGcjRsSjNDZm1oZGZzbVc4aUt5aEtENmgydnLRQUBQ1FBQUBFYkxndE9LcE1mTDIyK1JzQXQ4ZVlYanRyQ3QrL2JBWU00FNYZGx4YVd1Qj16N3RvYk8xRDhPOVZ1bFFWdkZKenNIU3F6OXQ5VHd1TVhtMHNUMk11bjB0c1RSakdDOUsva1VQdXRMMUxDT2dXK00xd0ZSM1BUZjZTbGZoZ3BUc3IwVjJCcEVPL3FRME1LaHVucUNiRUNjVldJNkVIbDQrWER0WUNiYzVVCYT15NWppR0NVSHRFa2trcXdxjQjBiTj10SVBoSItCcnU5ZjVqV3I0ZUFvNW5ZQ1U4d3VKZmRqYkMvTWJ3R3RsNgpxSzRERnLwRVFzQUk5bGdRM3NvcM0wKznlQjBk0pWQ00wVTJZakZsWVRNM1pDMWLaVEF4TFRRd1ppVXRZV114TnKxaE5tTXpNek5rT1RVNU9UWT0iLCJzcGVjIjoxLjZlIjZiMjVhMzdkLWJlMDEtNDNmNS1hZjE3LWE2YzZmZmQ1NTk5NiJ9",
  "expires_at": "2024-09-03T19:08:52.175191Z",
  "admin": true,
  "user": "admin"
}
```

- 3. Copy the token received in the response body in the field **access_token**.

This is required when executing APIs described in further steps and is referred to as **Vault Authentication Token**. The **access_token** field will be used in the **curl** headers from now on for the rest of the configuration calls. For example:

```
curl -H "X-Vault-Auth:
ZDg2ZTNjMjItNjU2My00NWl0LWJmYjktNDViYTZjOTExZWm4.eyJkYXRhIjoU0ZSUFVBRUFkYkktGNXNnWlBRSGJQNEQ3T2xTR2NXUHM2aVdqdzVjN21jRythdXZ5NUhIQjZYQ0tPRG9iK0FGcjRsSjNDZm1oZGZzbVc4aUt5aEtENmgydnLRQUBQ1FBQUBFYkxndE9LcE1mTDIyK1JzQXQ4ZVlYanRyQ3QrL2JBWU00FNYZGx4YVd1Qj16N3RvYk8xRDhPOVZ1bFFWdkZKenNIU3F6OXQ5VHd1TVhtMHNUMk11bjB0c1RSakdDOUsva1VQdXRMMUxDT2dXK00xd0ZSM1BUZjZTbGZoZ3BUc3IwVjJCcEVPL3FRME1LaHVucUNiRUNjVldJNkVIbDQrWER0WUNiYzVVCYT15NWppR0NVSHRFa2trcXdxjQjBiTj10SVBoSItCcnU5ZjVqV3I0ZUFvNW5ZQ1U4d3VKZmRqYkMvTWJ3R3RsNgpxSzRERnLwRVFzQUk5bGdRM3NvcM0wKznlQjBk0pWQ00wVTJZakZsWVRNM1pDMWLaVEF4TFRRd1ppVXRZV114TnKxaE5tTXpNek5rT1RVNU9UWT0iLCJzcGVjIjoxLjZlIjZiMjVhMzdkLWJlMDEtNDNmNS1hZjE3LWE2YzZmZmQ1NTk5NiJ9"
```

```
XQ4ZV1YanRyQ3QrL2JBWFU00FNYZGx4YVd1Qj16N3RvYk8xRDhPOVZ1bFFWdkZKenNIU3F60XQ5VHd1TVhtMHNUMk11bjB0c1RSakdDOUsv
a1VQdXRMUxDT2dXK00xd0ZSM1BUZjZTbGZoZ3BUc3IwVjJCcEVPL3FRME1LaHVucUNiRUNjV1dJnkVIbDQrWER0WUNiYzVcYTL5NWPqR0N
VSHRFa2trcXdjQjBiTj10SVBoSittCcnU5ZjVqV3I0ZUFvNW5ZQ1U4d3VKZmRqYkMvTWJ3R3RsNgpxSzRERnlwRVFzQUk5bGdRM3NvcM0wKz
N1QjBkb0pWQ00wVTJZakZsWVRNM1pDMWLaVEF4TFRRd1pqVXRZV114TnkxaE5tTXpNek5rT1LRVNU9UWT0iLCJzcGVjIjoxLXJpZCI6IjIzIm
WVhMzdkLWJlMDEtNDNmNS1hZjE3LWE2YzZmZ2Q1NTk5NiJ9" ...
```

2.6.5. Configure the Kubernetes cluster with the KeyControl secrets vault

To configure a Kubernetes cluster in the KeyControl vault, the following API needs to be executed.

```
https://<VAULT-IP>/vault/1.0/SetK8sConfiguration
```

The headers must have the vault authentication token, **X-Vault-Auth***: **<VAULT-AUTHENTICATION-TOKEN>**.

The request body should be in the form:

```
{
  // URL of the Kubernetes API server (or Loadbalancer if one is setup in front of
  // control plane) along with port. This URL must be accessible by PASM vault.
  // If Cluster is behind a service mesh, a proper ingress should be configured
  // so that API server is accessible.
  "k8s_url": "https://<KC-ACCESSIBLE-IP>/<FQDN of K8s API server>:6443",

  // Base64 encoded string of the certificate presented by API server during SSL handshake
  "k8s_ca_string": "<BASE64-ENCODED-CA-STRING>",

  // Indicates whether this configuration should be enabled or not. Valid values
  // are 'enabled' or 'disabled'. If the value is 'disabled', then "k8s_url" and
  // "k8s_ca_string" won't be validated and won't be saved in the configuration.
  "k8s_status": "enabled",
}
```

The certificate string can be obtained from the **kube config** file (`~/.kube/config`):

```
% cat ~/.kube/config

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tck1JSUJlRENDQVlyZ0F3SUJBZ0lCQURBS0JnZ3Foa2pPUFFFRREfQWpNU0V3SHdZRFZRUUREQmhyT
TNNdGMvYnkKZG1WeUxXTmhrREUzTWpVME5EZzJOekF3SGhjTk1qUXdPVEEwTVRFeE56VXdXaGNOTXpRd09UQXlNVEV4TnpVdWpXakFqTVNFd0h3WU
RWUVFEREJock0zTXRjM1Z5ZG1WeUxXTmhrREUzTWpVME5EZzJOekF3V1RBVEJnY3Foa2pPClBR5UJC2Z2dxaGtqT1BRTUJCd05DQUFUQzFGNE1uQTM
zb3RpcERCT2V2M11tV1BpRmc4QUtVW2U3MkhFRXNudFUKbE5mSStLMmVoa1h0MVZkREhrZ1ZUQUZ5Q0VIRDhIbmtYU2F5UUhWUnZONThvME13UURB
T0JnTlZiUThCQWY4RQpCQU1DQXFRd0R3WURWUjBUQVFIL0JBVXdBd0VCL3pBZEJnTlZiUFRFRmdRV2U2bERZK2ZiL2xvck5Md1BpWGW22CjJlK1V4d
W93Q2dZSUtvWk16ajBFQXdJRFNRQXdsZ0loQUxrU1Z4M2I1e1l13YzFur1BUNzVMcTJpMGZ2UEhMRWskbm5qRk1VOFp3d2VoQWlFQXh1K2NjZlQ1VC
9pbmVMUW1KaERlcm1HVjRRTDBUbmVKVTRqcmVDaHpvCVU9Ci0tLS0tRU5EIEF1RjRk1DQVRF1S0tLS0k
    server: https://127.0.0.1:6443
    name: default
contexts:
- context:
    cluster: default
```

```

user: default
name: default
current-context: default
kind: Config
preferences: {}
users:
- name: default
  user:
    client-certificate-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tCk1JSUJrRENDQVRlZ0F3SUJBZ0lJUUUM5QWRSc1Z4dWN3Q2dZSUtvWk16ajBFQXdJd016RWhNQjhHQ
TFVRUF3d1kKYXp0ekxXTnNhV1Z1ZEMxa1lVQXh0ekxTkRRNE5qY3dNQjRFRFRJME1Ea3d0REV4TVRjMU1Gb1hEVEkxTURrdwp0REV4TVRjMU1Gb3
dNREYVYUJjV0R0ExVUVDaE1PYzNsemRHVnRPbTFoYzNSbGNuTXhgVEFUQmd0VkJBTvRESE41CmMzUmxiVHBoWkcxGjQlPnQk1HQnlxR1NNND1BZ0V
HQ0Nxr1NNND1Bd0VIQTBJQUJCdW54c1R2cC9XNDdTVkEKWU1kdmEwckhaQ09kb0dLSUZHN0hKQU1KSzdIN1FSV28xeG0ycHvpN3pwSUNmQUc2N2I1
bUJ1NVJzZFRFNTB0Ugpp4ZGxzaTRDa1NEQkdNQTRHQTFVZER3RUIvd1FFQXdJRm9EQVRCZ05WSFNVRUREQUtCZ2dyQmdFRkJKRY0RBakFmCk1JnTlZlU
01FR0RBV2dCUVlVQVRyQ1BKYkRpejJacTJTay80MmXFESjB1VEFLQmdncWhrak9QUVFEQWd0SEFEQkUKQW1CVUQyWHBMM3k0NW8rbV1GZXAvSzQ5cD
dVVW41RS85LyttrkRERVDQaXRFQU1nRD1PVDREdFFrMfdQMhkyMQpiMCTaaajdIcmF3WDBaeDh1MTNPVjJKdUdhbU9Cio0tLS0tRU5EIEIENFU1RJRk1
DQVRFLS0tLS0KLS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tCk1JSUJkakNDQVlyZ0F3SUJBZ0lCQURBS0JnZ3Foa2pPUFFRREFqQWpNU0V3SHdZ
RFZRUUREQmhyTTNndFkyeHAKWlclMExXTmhrREUzTWpVME5EzZJ0ekF3SGhjTk1qUXdPVEEwTVRFeE56VXdXaGN0TXpRd09UQU1NVEV4TnpVdwpXa
kFqTVNFd0h3WURWUUFEREJoek0zTXRZMnhwWlclMExXTmhrREUzTWpVME5EzZJ0ekF3V1RBVEJnY3Foa2pPClBRUJJC2ZdaGtqT1BRTUJJCd05DQU
FUVlFicGRhZ0hnU2REVDfjcwVSNy8zZTN5b21RL1ZJVXRwNnFiVmQxVG0KbFhneXB6NmLuMxBEMjd0Rk1Zczhx0E0xEx6YzdGMDNVRnN0Q1GMzU
4aC9vME13UURBT0JnTlZlUThCQWY4RQpCQU1DQXFRd0R3WURWUjBUQVF1L0JlVjBvXDBd0VCL3pBZEJnTlZlUURFRmdRVUdLUU2d2p5V3c0cz1tYXRr
cFARck5hZ31kTg0t3Q2dZSUtvWk16ajBFQXdJRFJ3QXdSQU1nWU1HQUI0TzRiQjhhBR3lhR6xNQ1pjdGNXSG9DZXJZemoKcWU1U09PQmtYVXdsSUNtV
DBPREF0RUVIeIdmccTQ4aFdiT0F1dmU5anppcGgwTXR3UmoyS2tmZmQKLS0tLS1FTkQgQ0VSVe1GSUNBVEUtLS0tLQo=
    client-key-data:
LS0tLS1CRUdJTiBFQyBQdUlkWQVRFIEtFWS0tLS0tCk1IY0NBUEUvSUtWdHlza1UvSEx3Y3VHS2RpwWzMOw5u0C9NRVh6ZEJFVVRMNW1QbVpxT31vQ
W9HQ0Nxr1NNNDkKQXdF5G9VUURRZ0FFRzZmR3RPK245Ymp0S1VCZ3gyOXJtc2RrSTUyZ11vZ1V1c2NrcXdrncNmdEJGYWpYR2JhbQo2THZPa2dKOE
FicnR2bVlHN2xHeDFNUG5RukhGMl5TGdBT0KLS0tLS1FTkQgRUMGUJFJJkFURSBURVktLS0tLQo=

```

Look at the **certificate-authority-data** attribute. It contains the base64 encoded certificate string needed for the configuration.

Now look at the **server** attribute. This is the field that we are going to use to get to the needed **k8s_url**.

This attribute content is `https://127.0.0.1:6443`.

Replace the 127.0.0.1 in the URL with the IP address of the server. This URL needs to be accessible from the KeyControl server. Make sure port 6443 is not blocked by the firewall. If using a VPN, make sure you can access the URL from a server on the same subnet as the KeyControl node.

The URL in our case is `https://1X.19X.14X.XXX:6443`.

You should get the following response:

```

{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "Unauthorized",
  "reason": "Unauthorized",
  "code": 401
}

```

The final request body:

```
{
  "k8s_url": "https://1X.19X.14X.XXX:6443",
  "k8s_ca_string": "<certificate-string>",
  "k8s_status": "enabled"
}
```

Save the request body in a file named `SetK8sConfiguration.body` and execute the following command:

```
% curl -H "X-Vault-Auth:
ZDg2Z2TnJmJiInjU2My00NWl0LWJmYjktNDViYTZjOTExZWm4.eyJkYXRhIjoieU0ZSWFVBRUFkbktGNXNnWlBRSRGJQNEQ3T2xTR2NXUHM2aVdqdzVJ
N21jRythdXZ5NUhIQjZyQ0tPRG9iK0FGcjRsSjNDZm1oZGZZbVc4aUt5aEtENmgydnlRQUFBQ1FBQUFBYkxndE9LcE1mTDIyK1JzQXQ4ZVlYanRyQ
3QrL2JBWfU0FNYZGx4YVd1Qj16N3RvYk8xRDhPOVZ1bFFWdkZKenNIU3F6OXQ5VHd1TVhtMHNUMk11bjB0c1RSakdD0Usva1VQdXRMMUxDT2dXK0
0xd0ZSM1BUZjZTbGZoZ3BUc3IwVjJCcEVPL3FRME1LaHVucUNiRUNjVldJNkVIbDQrWER0WUNiYzVCYTl5NWppqR0NVSHRFa2trcXdjQjBiTj1l0SVB
oSitCcnU5ZjVqV3I0ZUFvNW5ZQ1U4d3VKZmRqYkMvTWJ3R3RsNgpxSzRERnLwRVFzQUk5bGdRM3NvcM0wKzN1QjBKb0pWQ00wVTJZakZsWVRNM1pD
MW1aVEF4TFRRd1pqVXRZV114TnkxaE5tTXpNek5rTlRVNU9UWT0iLCJzcGVjIjoxLCJpZCI6IjJiMwVhMzdkLWJlMDEtNDNmNS1hZjE3LWE2YzZm
2Q1NTk5NiJ9" -k -X POST https://xx.xxx.xxx.xxx/vault/1.0/SetK8sConfiguration/ --data @./SetK8sConfiguration.body
```

The response should be:

```
{
  "Status": "Success",
  "Message": "Kubernetes configuration updated successfully"
}
```

2.6.6. Set the namespace

Before proceeding with the rest of the configuration, set the namespace in Kubernetes to the `mutatingwebhook` namespace created earlier.

```
% kubectl config set-context --current --namespace=mutatingwebhook
```

2.6.7. Create the registry secrets inside the namespace

The credentials for the external Docker registry access need to be created so they can be mentioned in the pod deployment specification.

1. Create the secret in the namespace:

```
% kubectl create secret generic regcred --from-file=.dockerconfigjson=$HOME/.docker/config.json
--type=kubernetes.io/dockerconfigjson
```

2. Confirm that the secret has been created:

```
% kubectl get secret regcred
```

2.6.8. Deploy the webhook container

Use the following `yaml` specification to deploy the webhook container in Kubernetes:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mutatingwebhook
  namespace: mutatingwebhook
  labels:
    app: mutatingwebhook
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mutatingwebhook
  template:
    metadata:
      labels:
        app: mutatingwebhook
    spec:
      containers:
        - name: mutatingwebhook
          image: 1.2.3.4:5000/mutating-webhook:latest
          ports:
            - containerPort: 5000
          env:
            # Specify these variables either as a mutatingwebhook container environment variables or
            # add appropriate annotations in the pod specification. Refer documentation further to get
            # the details about annotations. The annotations in pod specifications, if specified, take
            # precedence over these environment variables.
            - name: ENTRUST_VAULT_IPS
              value: <comma-separated-list-of-vault-ips>
            - name: ENTRUST_VAULT_UUID
              value: <uuid-of-vault-from-which-secrets-to-be-fetched>
            - name: ENTRUST_VAULT_INIT_CONTAINER_URL
              value: <complete-url-of-init-container-image-from-image-registry>
```

If image registry requires authentication, then make sure to add the credentials as secret in the `yaml` specification by adding the `imagePullSecrets` tag in the `containers` section.

Save the `yaml` in a file called `mutatingwebhook.yaml`.

Example mutatingwebhook file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mutatingwebhook
  namespace: mutatingwebhook
  labels:
    app: mutatingwebhook
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mutatingwebhook
  template:
```

```

metadata:
  labels:
    app: mutatingwebhook
spec:
  imagePullSecrets:
    - name: regcred # external registry secret created earlier
  containers:
    - name: mutatingwebhook
      image: >-
        <registry-url>/mutating-webhook
      ports:
        - containerPort: 5000
      env:
        # Specify these variables either as a mutatingwebhook container environment variables or
        # add appropriate annotations in the pod specification. Refer documentation further to get
        # the details about annotations. The annotations in pod specifications, if specified, take
        # precedence over these environment variables.
        - name: ENTRUST_VAULT_IPS
          value: 1x.19x.14x.20x,1x.19x.14x.21x
        - name: ENTRUST_VAULT_UUID
          value: d86e3c22-6563-45b4-bfb9-45ba6c911ec8
        - name: ENTRUST_VAULT_INIT_CONTAINER_URL
          value: <registry-url>/init-container

```

Pay attention to these sections and adjust according to your environment.

```

imagePullSecrets:
  - name: regcred # external registry secret created earlier

image: >-
  <registry-url>/mutating-webhook

- name: ENTRUST_VAULT_IPS
  value: 1x.19x.14x.20x,1x.19x.14x.21x
- name: ENTRUST_VAULT_UUID
  value: d86e3c22-6563-45b4-bfb9-45ba6c911ec8
- name: ENTRUST_VAULT_INIT_CONTAINER_URL
  value: <registry-url>/init-container

```

Deploy the file:

```
% kubectl create -f mutatingwebhook.yaml
```

Check that the mutating webhook deployment is running:

```

% kubectl get pods
NAME                                READY  STATUS   RESTARTS  AGE
mutatingwebhook-7666fc54d7-jxs2s  1/1    Running  0          19s

% kubectl describe pod mutatingwebhook-7666fc54d7-jxs2s

Name:          mutatingwebhook-7666fc54d7-jxs2s
Namespace:    mutatingwebhook
Priority:      0
Service Account: default
Node:         redhat-9-kcv-secrets/1X.19X.14X.XXX
Start Time:   Wed, 04 Sep 2024 09:42:51 -0400
Labels:       app=mutatingwebhook
              pod-template-hash=7666fc54d7

```

```

Annotations:      <none>
Status:           Running
IP:              10.42.0.7
IPs:
  IP:            10.42.0.7
Controlled By:   ReplicaSet/mutatingwebhook-7666fc54d7
Containers:
  mutatingwebhook:
    Container ID:  containerd://49b07ac85063041fe772194dece7bb416965d13ebec2c74418d67bbd3a474a1a
    Image:         <registry-url>/mutating-webhook
    Image ID:     <registry-url>/mutating-
webhook@sha256:d5379f9b116f725c1e34cda7f10cba2ef7ed369681a768fb985d098cd24f1b1d
    Port:         5000/TCP
    Host Port:    0/TCP
    State:        Running
      Started:    Wed, 04 Sep 2024 09:43:06 -0400
    Ready:        True
    Restart Count: 0
    Environment:
      ENTRUST_VAULT_IPS:          1X.19X.14X.XXX,1X.19X.14X.XXX
      ENTRUST_VAULT_UUID:        d86e3c22-6563-45b4-bfb9-45ba6c911ec8
      ENTRUST_VAULT_INIT_CONTAINER_URL: <registry-url>/init-container
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-trrn7 (ro)
Conditions:
  Type                               Status
  PodReadyToStartContainers          True
  Initialized                         True
  Ready                               True
  ContainersReady                    True
  PodScheduled                       True
Volumes:
  kube-api-access-trrn7:
    Type:                               Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:             3607
    ConfigMapName:                      kube-root-ca.crt
    ConfigMapOptional:                  <nil>
    DownwardAPI:                        true
    QoS Class:                          BestEffort
    Node-Selectors:                     <none>
    Tolerations:                        node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                                         node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason      Age   From          Message
  ----    -
  Normal  Scheduled  67s   default-scheduler  Successfully assigned mutatingwebhook/mutatingwebhook-7666fc54d7-
jxs2s to redhat-9-kcv-secrets
  Normal  Pulling    67s   kubelet        Pulling image "<registry-url>/mutating-webhook"
  Normal  Pulled     52s   kubelet        Successfully pulled image "<registry-url>/mutating-webhook" in
14.716s (14.716s including waiting). Image size: 86935850 bytes.
  Normal  Created    52s   kubelet        Created container mutatingwebhook
  Normal  Started    52s   kubelet        Started container mutatingwebhook

```

2.6.9. Deploy the entrust-pasm service

1. Create a `yaml` file named `entrustpasmervice.yaml` that contains the following code:

```

apiVersion: v1
kind: Service
metadata:
  name: entrust-pasm          # DO NOT CHANGE THIS

```

```

namespace: mutatingwebhook # DO NOT CHANGE THIS
spec:
  selector:
    app: mutatingwebhook
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000

```

2. Run the following command to create the service:

```
% kubectl create -f entustpasmervice.yaml
```

3. Check the service:

```
% kubectl get service
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
entrust-pasm	ClusterIP	10.43.26.173	<none>	5000/TCP	9s

2.6.10. Obtain the webhook server certificate

After the container is deployed, obtain the webhook server certificate. This certificate will be needed to configure the webhook in Kubernetes. Use the following command:

```
% kubectl exec -it -n mutatingwebhook $(kubectl get pods --no-headers -o custom-columns=":metadata.name" -n mutatingwebhook) -- wget -q -O- localhost:8080/ca.pem
```

The output should be something like this:

```

LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tCk1JSUZTEkNDQXpZP0F3SUJBZ01VRURiWFlYazZHVUVJM21rUGZQbnp4MVRMRjdVd0RRRUwLb1pJa
HZjTkJFRRUwQ1FBd1FERXhNqzhHQTFVRFU3d29TSgxyVY25WemRDQkxawGxYyI1MGNT0XNJRUS5yY25ScFptbGpZWZJ3sSUVGMQpkR2h2Y21sMGVURU
xNQWtHQTFVRUJ0TUNWVk13SGhjTk1qUXdPREkzTVRjeU1ERTNXaGN0TXpRd09ESTFNVGN5Ck1ERTNXakJBTVRVd0x3WURWUVFERENoSWVWUWU1kWE4
wSUV0bGVVbnZib1J5YjJ3Z1EyVn1kR2xtYVd0aGRHVWcKUVhWMGFHOXlhWF11TVFzd0NRWURWUVFHRXdkVlV6Q0NBaU13RFFZSkvtWkLodmNOQVFF
QkJRQRnZ0LQURDQWpBZ29DZDZkQkFMTVMQk5KVVVzYU1nSH1rR3hHOwcvdFBtZXdfTi9DODVLSFJjMmREOHZyZndiNkZ5dFg3UG1BcNRhSVlIm
mJ40UdTaIdXVf1NXX1PL0g5WkY5L3BjUmJuYkdaNFBMTzBGdG1vM2FicCszRVJjTENWREZCamszQLcKTW9h0WQ1VVBkU2kzYnFBdGFrT2xFZhdOQ2
ZvSWh1MDZNOGF2TDCrVWtZV1FQWgppMzBDODZCUTRRWE01dkFKQwpwaFg1NmhCY3N1RjJDe0w2cGZYUj1JbFVjVTgwZFBK1B1t3ZubWdNbGpYakp
YUnpHREJOUG9VS25HNmJ5N0tpCjv2TtkNBaXYxRmJPQXU1ZCZWdRUHZEempJRmV0MzBZUDRkamlydk5Yc1pNZnRiamfZzFVpWGHIMGFYK1BYN3cK
YTNFRUpZL290TjR4VVLLaTFGRlhteDRRWTZaSTVheWdnTzNHSTZEWDJUCENQUzFqcnpXL2xETzhtd1QzNEhFSApKsno1T05abkRGME40YXJjWwFKa
2dUVTd2Q1RaL31taWFsc21DNk1JQM1LUTk15Nm1KY1loczFUyWRQUVpmaELnCMxqefhPQ2dDTXdLV1poNjdPdEtWTkNmc83ZzBsBhHyeW1YY0xY2
5aRy81azRDbm1mZHVtQWl4a1hRRVpXaTcKenRjL0ZIZjJmVGNrVknNsm5ND1JS1NpSTdVREQwQkFHTkVnTE5hVnFBQ0pRbTn1QmVTS24xZ0c0ODN
HMTRQawp5MkYyMEpEe1BreG5ZV1BoVkvP29scXZ3ZVpV1FNWFV5U1UyQTZ5RTQwZfZsSjBIZGJoTjdzOUJZQWJSUjB4Ck9xalcvTW0yL2trUE1r
Z3IyR1NsK2JhcVlZS1hKUKtsNFViemZLQUxNRXNIEcS3L2L1Mk0hBZ01CQUFHa1BUQTcKtUF3R0ExVWRFd0VCL3dRQ01BQXdLd1E1V1IwUk1JDUxJb
01nW1c1MGNuVnpkQzF3WVh0dExtMTfKR0YwYvc1bgpkMLZpYUc5dmF5NpkBU13RFFZSkvtWkLodmNOQVFFTEJRQRnZ0LQCU1aVfPpNmhSS29HVE
s0eHB3b1hqY3p1CnLoSzZCbURsejg0akd2ek85Q083dkZJb1FRSgkV1K1MrC31Bb116N3kwN1pnUDZJYi91ZEx5by9aTFB0UDArALAKSnlLoSUNO3L
XV1dhNk1jYNEE3VExBT1BpSWEvQmtaZHBaR3NMdENrbe5uckf1MFBbBY0QWpUZUZzZWF2aXREeApBTW1YSU1TYk1TZG55bEhFVTUrMWREdzRMRTBt
N01UR2J1U2E3TnZeeLVdWhCNkxDV1hxTWLUQTQ3MCtpdXF4Cm10dkZ3WVU5RVgv3ZLd1N1bWfZZjRKR24rL3pBMLNpU0d5V0R1NW03RctjZzoT
XBKenRXWw91Y1NR0wXQQTUKWmZxcDhBYkdnc1J3a1IyTjF1VzZFK1BJNWEzb11LU2NMKUJJeG5GTTBxNjJuv3ppdHB3S1RQN2VsNt1KSWFEQwpYaE
c3UWxvBDFCSUpUHR6M2Nra1dCTzhuN1JhOHntVwxTMVNSNkdUWDZ5RGErV1YzVzVranhmZ0LrNXJCRnLyClRXeEVPShLJQkdQc1d2UnAvNF1yTHR
z0W10ekpPUVJqeVpLUVZLaUhmaGI1T1JTSE9MT1Y3MERESHd5K2hIc0wKemTKRkc3Mm9ocDJYEDkweHdoeUVFdlhWYU5mRfpTekdKVXpWSFhSDRW
dG1WaxFBbkFYqkLhYdHvIA10XBRVwphOEtuZWNVZkk3VExlSk9ya09He1ZLeU05YUFHaS83MM92WfUwZnZOWXZsc1LENU9FajVKME5mQ1hzWXdlc
DE5CnVyY1FlRUJjRFRNMBE1epkIxk1LUYUsv0TJHQ1IwaU1Je650N2NWU311UEJ1emRZSk4tQjdNZehhZXF5M1FERZQKcUpjSm58NmRhUFZ1dzFqMi

```

```
9sZmgKLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo=
```

This is a base64 encoded certificate. With this value, configure webhook in Kubernetes.

2.6.11. Configure the webhook in Kubernetes

In sample YAML spec replace the **caBundle** value with the value obtained in the previous section and configure the webhook with the **kubectl apply** command.

- 1. Create a **webhook.yaml** with the following code:

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  name: "entrust-pasm.mutatingwebhook.com"
webhooks:
  - name: "entrust-pasm.mutatingwebhook.com"
    objectSelector:
      matchLabels:
        entrust.vault.inject.secret: enabled
    rules:
      - apiGroups: ["*"]
        apiVersions: ["*"]
        operations: ["CREATE"]
        resources: ["deployments", "jobs", "pods", "statefulsets"]
    clientConfig:
      service:
        namespace: "mutatingwebhook"
        name: "entrust-pasm"
        path: "/mutate"
        port: 5000
        # Replace value below with the value obtained from command
      caBundle:
LS0tLS1CRUdJTTiBDRVJUSUZJQ0FURSB0tLS0tCk1JSUZTekNDQXpPZ0F3SUJBZ01VRUFiWF1YazZHVUVJM2l2rUGZGbnp4MVRMRjdVd0RRWUp
Lb1pJaHZjTkFRRUwKQlFBd1FERXhNqzHqTFVRUF3d29TSgXVY25WemRDQkxaWGxEYjI1MGNtOXNJRUs5sY25ScFtbGpZWF3sSUVGMQpkR2
h2Y21sMgVURUxNQWtHQTFVRUJoTUNWVk13SGhjTk1qUXdPREkzTVRjeU1ERTNXaGN0TXpRd09ESTFNvGN5Ck1ERTNXakJBTvRFd0x3WURWU
VFERENoSWWUUnlKWE4wSUV0bGVVtNzib1J5YjJ3Z1EyVn1kR2xtYVd0aGRHVWcKUVhWMGFHOXlhWFI1TVFzd0NRWURWUUVFHRXdkVlV6Q0NB
aU13RFFZSkvtWk1odmNOQVFFkJRURnZ0lQURDQWpBZ29DZ2dJQkFMTlVMQk5KVVVzYU1nSHlrR3hHOWcvdFBtZXdFTI9DODVlSFJjMmR
EOHZyZndiNkZ5dFg3UG1BCnRhSV1iMmJ4OUdTaldXVFlNNXlPL0g5WkY5L3BjUmJuYkdaNFBMTzBGdG1vM2FicCszRVJjTENWREZCamszQL
cKTW9hOWQ1VVBkU2kzYnFBdGFRT2xZFdHd0Q2ZvSWhlMDZNOGF2TDcrVWtZV1FQWgppMzBDODZCUTRRWE01dkFKQWpwaFg1NmhCY3N1RjJdC
0w2cGZYUj1JbFVjVtGwZFBKN1BiT3ZubWdNbGpYakpYUnpHREJOU6V5S25HNmJ5N0tpCjV2TkNBaXYxRmJPQUxU1ZCZWRUHZEdmpJrmVo
MzBZUDRkamLydk5Yc1pNZnRiamFzZFPWghIMGFYK1BYN3cKYTNFRUpZL290TjR4VVlLaTFGRlhteDRRWTZaSTVheWdnTzNHSTZEWDJUCEN
QUzFqcnpXL2eTzhtd1QzNEhFSApKSno1T05abkRGME40YXJJWnFka2dUVTd2Q1RaL3ltaWFsc21DNk1JQM1LUTk15Nm1KY11ocZFUyWRQUV
pmaElnCmxqeFhPQ2dDTXdlV1poNjdpdEtWTkNmeC83ZzBsHhyeW1YY0xoY25aRy81azRDbmImZHVtQWl4a1hRRVpXaTcKenRjL0ZIZjJMv
GNrVknSms5ND1JSLNpSTdVREQwQkFHTkVnTE5hVnFBQ0pRbTlQmVTS24xZ0c0ODNHMTRQawp5MkYyMEpEe1BReG5ZVlBoVkpK29scXZ3
ZFVPVlFNWFV5UyUyQTZ5RTQwZFsSjBIZGJoTjdzOUJZQWJSUjB4Ck9xa1cvT0yL2trUE1rZ3IyRlNsK2JhcVlZS1hKuktsNFViemZlQUx
NRXNIeCs3L2MLk0hBZ01CQUFHAlBUQTcKTUF3R0ExVWRfD0VCL3dRQ01BQXdLd11EVL1WUk1DUJdJb0lnWlc1MGNUvNpkQzF3WVhOdExtMT
FkR0YwVvc1bgpKMLZpYUc5dmF5NXpkBU13RFFZSkvtWk1odmNOQVFFTEJRURnZ0lCQU1aVfPnMhSS29HVEs0eHB3b1hqY3p1Cn1oSzZCb
URsejg0akd2ek85Q083dkZJb1FRSkVlK1MrC31Bb116N3kwN1pnUDZJY191ZEX5by9aTFB0UDArAlAKSnLoSUNoN31XVldhNk1JyNEE3VExB
T1BpSWEvQmtaZHBaR3NMdENrbE5uck1MFBBbXY0QWpUzUZzWf2aXREApBTLWYSU1TYk1TZG555EhFVTUrMWREdzRMRTBt01UR2J1U2E
3TnZe1VZdWhCNkxDV1hxTWLUQTQ3MctpdXF4Cm10dkZ3WVU5RVgvV3ZLd1N1bWfZzjRKR24rL3pBMLNpU0d5V0R1NW03RCtjazZoTXBKen
RXW091Y1LNRQWxQQTUKWmZxcDhBYKdnc1J3a1IyTjF1VzZFK1BJNWEzb11LU2NKMUJJeG5GTTBxNjJuv3ppdHB3S1RQn2VsNTLkSWFEQwpYa
Ec3UWvxbDFCSUpUHR6M2Nra1dCTzhuN1JhOHntVWxTMVNSNkdUW0Z5RGIrV1YzVzVranhmZ01rNXJCRnLyCLRXeEVPShlJQkdQc1d2UnAv
NF1yTHRz0W10ekpPUVJqeVpLUVZ1aUhmaGI1T1JTSE9MT1Y3MERESHd5K2hIc0wKemtKRkc3Mm9ocDJYeDkweHdoeUVFdlhWYU5mRfPtekd
KVXpWfSfHSDRwdG1WaXFbBkFYQkhLYxdhVLA10XBRVwph0EtuZWNVZkk3VE1Sk9ya09HeLZLeU05YUfHaS83MW92FWUnZ0WXZscTLENU
9FajVKME5mQ1hWXd1cDE5CnVyyLFpRUJJRFNMbE1pek1xK11YUsv0TJHJ1IwaU1JeG50N2NWU311UEJ1emRZSk5t4QjdnZEhZXF5M1FER
zQKcUpjSm5BNmRHUFZ1dzFqMi9sZmgKLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo=
      admissionReviewVersions: ["v1", "v1beta1"]
```

```
sideEffects: None
timeoutSeconds: 5
```

2. Run the following command to apply the changes:

```
% kubectl apply -f webhook.yaml

mutatingwebhookconfiguration.admissionregistration.k8s.io/entrust-pasm.mutatingwebhook.com created
```

2.7. Test the integration

Now that the Kubernetes cluster and KeyControl Vault are properly configured and set up let's deploy a couple of pods that will attempt to use the `ocsecret` created earlier inside the pod containers.

Important points:

- The deployment of pod (in which the secrets need to be fetched) must happen by the service accounts mentioned in the policy created above.
- The service accounts added in policy must have **system:auth-delegator** ClusterRole at the Kubernetes side.

2.7.1. Set the namespace

We created two namespaces earlier. Currently the integration should be set to the `mutatingwebhook` namespace. Change the namespace so it points to the test namespace.

```
> kubectl config set-context --current --namespace=testnamespace
```

2.7.2. Create the registry secrets inside the namespace

The credentials for the external Docker registry access need to be created so they can be mentioned in the pod deployment specification.

1. Create the secret in the namespace:

```
% kubectl create secret generic regcred --from-file=.dockerconfigjson=$HOME/.docker/config.json
--type=kubernetes.io/dockerconfigjson
```

2. Confirm that the secret has been created:

```
% kubectl get secret regcred
```

2.7.3. Create a policy in Secrets Vault for Kubernetes service accounts

Create a policy in the Secrets Vault that will enable Kubernetes service accounts to read secrets from the vault. To create such policy, the following API needs to be executed.

URL: <https://<VAULT-IP>/vault/1.0/CreatePolicy>

The headers must have the vault authentication token.

X-Vault-Auth: <VAULT-AUTHENTICATION-TOKEN>

The request body should be in the form:

```
{
  "name": "vault_user_policy",          // Name of the policy to be created
  "role": "Vault User Role",          // Role of user. DO NOT CHANGE THIS
  "principals": [
    {
      "k8s_user": {
        "k8s_namespace": "default",    // Namespace in which the service account resides
        "k8s_service_account": "default" // Name of the service account
      }
    }
  ],
  "resources": [
    {
      "box_id": "box1", // Name of the box in which the secret(s) that need access reside. Can be '*' to
                        // indicate all boxes
      "secret_id": [    // List of secrets to which access needs to be granted. Can be '*' to indicate all
                        // secrets.
        "secret1",
        "secret2"
      ]
    }
  ]
}
```

You must match the namespace `k8s_namespace` to the same namespace the service account resides in and where you will be deploying the pods.

1. Save the request body for the environment to a file named `createpolicy.body`:

```
{
  "name": "kubernetes_vault_user_policy",
  "role": "Vault User Role",
  "principals": [
    {
      "k8s_user": {
        "k8s_namespace": "testnamespace",

```

```
    "k8s_service_account": "default"
  }
}
],
"resources": [
  {
    "box_id": "box1",
    "secret_id": [
      "*"
    ]
  }
]
}
```

In this example, the policy will allow the **default** user in the **testnamespace** to have access to any secret in **box1** in the KeyControl Secrets vault.

2. Check and adjust the following sections according to your environment:

```
"k8s_user": {
  "k8s_namespace": "testnamespace",
  "k8s_service_account": "default"

  "box_id": "box1",
  "secret_id": [
    "*"
  ]
}
```

3. Create the policy:

```
curl -H "X-Vault-Auth: <VAULT-IDENTIFICATION-TOKEN>" -k -X POST
https://1X.19X.14X.XXX/vault/1.0/CreatePolicy/ --data @./createpolicy.body

{"policy_id": "kubernetes_vault_user_policy-82756a"}
```

2.7.4. Create the clusterrolebinding

Create the clusterrolebinding to allow the default user access to the secrets:

```
% kubectl create clusterrolebinding authdelegator --clusterrole=system:auth-delegator
--serviceaccount=testnamespace:default
```

Test the access by running the following command:

```
% kubectl auth can-i create tokenreviews --as=system:serviceaccount:testnamespace:default
```

The output should be **yes** if set up correctly.

2.7.5. Deploy the pod with secrets

Secrets can be added to the pod either as volume mounts or as environment variables.

2.7.5.1. Add secrets as volume mounts to the pod

The sample pod specification along with labels and annotations required for successfully pulling secrets as volumes mounts:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  namespace: testnamespace
  labels:
    app: test
    entrust.vault.inject.secret: enabled      # Must have this label
  annotations:
    entrust.vault.ips: <comma-separated-list-of-vault-ips>
    entrust.vault.uuid: <uuid-of-vault-from-which-secrets-to-be-fetched>
    entrust.vault.init.container.url: <complete-url-of-init-container-image-from-image-registry>
    entrust.vault.secret.file.k8s-box.ca-cert: output/cert.pem # box and secret name from vault and value
    # denoting location of secret on the container. The location
spec:
  serviceAccountName: k8suser      # Service account name configured in PASM Vault Policy
  containers:
  - name: ubuntu
    image: 10.254.154.247:5000/ubuntu:latest
    command: ['cat', '/output/ans.txt']
    imagePullPolicy: IfNotPresent
```

The pod specification must have the following annotations:

- **entrust.vault.inject.secret: enabled:** This label indicates that it needs secret.
- **entrust.vault.ips:** This value is a comma-separated list of IPs of vaults which are in the cluster.

This annotation, if specified, will override the **ENTRUST_VAULT_IPS** environment variable from the mutating webhook configuration.

- **entrust.vault.uuid:** This value denotes the UUID of the vault from which we need to fetch the secrets.

This annotation, if specified, will override the **ENTRUST_VAULT_UUID** environment variable from the mutating webhook configuration.

- **entrust.vault.init.container.url:** This value denotes the complete URL of the init container image pushed into image registry.

This annotation, if specified, will override the **ENTRUST_VAULT_INIT_CONTAINER_URL** environment variable from the mutating webhook configuration.

- Annotations in the form of `entrust.vault.secret.file.{box-name}.{secret-name}`.

The `box-name` and `secret-name` placeholders within the annotation should denote the name of the box and the secret name within that box that needs to be pulled.

For example, if the secret was named `db-secret` and the box was `k8s-box`, then the name of the annotation would be `entrust.vault.secret.file.k8s-box.db-secret`. The value of the annotation should be the path to the file where the secret needs to be stored within the container. The path should be relative to the `/` directory, meaning that if the secret needs to be present in `/output/cert.pem`, then the value of the annotation should be `output/cert.pem`

- In the `spec` section, the `serviceAccountName` value should be the name of the service account that was added in the policy in the Secrets vault.

To add the secrets as volume mounts to the pod:

1. Create a file named `pod1.yaml` with the yaml for the environment. We are using the box and secret we created earlier in the Secrets vault. (`box1` and `ocsecret`). We also had to provide the credentials for the Docker registry.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  namespace: testnamespace
  labels:
    app: test
    entrust.vault.inject.secret: enabled          # Must have this label
  annotations:
    entrust.vault.ips: 1x.19x.14x.20x,1x.19x.14x.21x
    entrust.vault.uuid: d86e3c22-6563-45b4-bfb9-45ba6c911ec8
    entrust.vault.init.container.url: <registry-url>/init-container
    entrust.vault.secret.file.box1.ocsecret: output/ocsecret.txt
spec:
  imagePullSecrets:
    - name: regcred          # external registry secret created earlier
  serviceAccountName: default # Service account name configured in PASM Vault Policy
  containers:
    - name: ubuntu
      image: ubuntu
      command: ["sh", "-c"]
      args:
        - echo "Getting secret from KeyControl Secrets Vault";
          cat /output/ocsecret.txt;
          echo;
          echo "DONE" && sleep 3600
      imagePullPolicy: IfNotPresent
```

2. Check and adjust the following sections according to your environment:

```
annotations:
```

```
entrust.vault.ips: 1x.19x.14x.20x,1x.19x.14x.21x
entrust.vault.uuid: d86e3c22-6563-45b4-bfb9-45ba6c911ec8
entrust.vault.init.container.url: <registry-url>/init-container
entrust.vault.secret.file.box1.ocsecret: output/ocsecret.txt

imagePullSecrets:
- name: regcred                # external registry secret created earlier

command: ["sh", "-c"]
args:
- echo "Getting secret from KeyControl Secrets Vault";
  cat /output/ocsecret.txt;
  echo;
  echo "DONE" && sleep 3600
```

3. Deploy the pod:

```
% kubectl create -f pod1.yaml
```

4. Check the pod to verify that it is capable of pulling the secret from a KeyControl Secrets Vault:

```
% kubectl logs pod/pod1

Defaulted container "ubuntu" out of: ubuntu, secret-v (init)
Getting secret from KeyControl Secrets Vault
This is the secret coming from KCV to Kubernetes.
DONE
```

2.7.5.2. Pull a secret as an environment variable into the pod

Kubernetes supports initiating environment variables for a container either directly or from Kubernetes secrets. For security purposes, the Secrets Vault takes the later approach. Also, if secrets are injected using environment variables, a sidecar container will be added which will delete the Kubernetes secrets after the successful injection of KeyControl Secret Vault secrets as environment variables in the main application container. Since we need to create and delete Kubernetes secrets for this, the service account must also have the required permissions to create and delete the Kubernetes secrets.

1. Grant the service account with required permissions: (replace the **namespace** and **serviceaccount** values appropriately)

```
% kubectl create rolebinding secretrole --namespace testnamespace --clusterrole=edit
--serviceaccount=testnamespace:default
```

2. To check if proper permissions are set up for the service account, use the following commands:

```
% kubectl auth can-i create secrets -n testnamespace --as=system:serviceaccount:testnamespace:default
% kubectl auth can-i delete secrets -n testnamespace --as=system:serviceaccount:testnamespace:default
```

The output of both the above commands should be **yes**.

3. Save the following sample pod specification yaml in a file called `pod2.yaml`. It shows how to deploy a pod with secrets as environment variables.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod2
  namespace: testnamespace
  labels:
    app: test2
    entrust.vault.inject.secret: enabled      # Must have this label
  annotations:
    entrust.vault.ips: 10.19x.14x.20x.,10.19x.14x.21x
    entrust.vault.uuid: d86e3c22-6563-45b4-bfb9-45ba6c911ec8
    entrust.vault.init.container.url: <registry-url>/init-container
    entrust.vault.secret.env.box1.ocsecret: OCSECRET
spec:
  imagePullSecrets:
    - name: regcred          # external registry secret created earlier
  serviceAccountName: default # Service account name configured in PASM Vault Policy
  containers:
    - name: ubuntu
      image: ubuntu
      command: ["sh", "-c"]
      args:
        - echo "Getting secret from KeyControl Secrets Vault";
          printenv OCSECRET;
          echo "DONE" && sleep 3600
      imagePullPolicy: IfNotPresent
```

4. To pull the secret as an environment variable, add an annotation of the form `entrust.vault.secret.env.{box-name}.{secret-name}`. The value of the annotation should indicate the name of the environment variable in which the secret data is expected to be present.
5. Check and adjust the following sections according to your environment. These annotations are as documented in the previous section.

```
annotations:
  entrust.vault.ips: 10.19x.14x.20x.,10.19x.14x.21x
  entrust.vault.uuid: d86e3c22-6563-45b4-bfb9-45ba6c911ec8
  entrust.vault.init.container.url: <registry-url>/init-container
  entrust.vault.secret.env.box1.ocsecret: OCSECRET

imagePullSecrets:
  - name: regcred          # external registry secret created earlier

command: ["sh", "-c"]
args:
  - echo "Getting secret from KeyControl Secrets Vault";
    printenv OCSECRET;
    echo "DONE" && sleep 3600
```

6. Create and test the pod:

```
% kubectl create -f pod2.yaml
```

7. Check the pod output to verify that it is capable of pulling the secret from the KeyControl Secrets vault:

```
% kubectl logs pod/pod2
```

```
Defaulted container "ubuntu" out of: ubuntu, pasm-sidecar, secret-v (init)  
Getting secret from KeyControl Secrets Vault  
This is the secret coming from KCV to Kubernetes.  
DONE
```

Chapter 3. Troubleshooting

Use the following commands troubleshoot pods during deployment.

3.1. Look at the logs

You can use `kubectl logs`:

```
% kubectl logs pod/podname
```

For example:

```
% kubectl logs pod/pod1
```

You can also use `kubectl describe`:

```
% kubectl describe pod podname
```

For example:

```
% kubectl describe pod pod1
```

3.2. Look at the init container

In the guide, the `init` container is deployed at each pod that gets the secret.

To look at the logs for the `init` container:

```
% kubectl logs pods/pod1 -c secret-v --namespace=testnamespace
```

Chapter 4. Additional resources and related products

4.1. nShield Connect

4.2. nShield as a Service

4.3. KeyControl

4.4. KeyControl as a Service

4.5. nShield Container Option Pack

4.6. Entrust products

4.7. nShield product documentation