**OCI Registry in the Context of Oracle Kubernetes Engine (OKE) and Oracle Cloud Infrastructure (OCI)**

The **OCI Registry** (Oracle Cloud Infrastructure Registry) is a fully managed, private Docker container registry service provided by Oracle Cloud Infrastructure. It allows you to store, manage, and deploy container images securely within the Oracle Cloud ecosystem. When combined with **Oracle Kubernetes Engine (OKE)**, it provides a seamless way to manage containerized applications in a Kubernetes environment.

Below is a detailed explanation of how the OCI Registry works, its features, and its integration with OKE and Oracle Cloud.

**1. What is OCI Registry?**

The OCI Registry is a private container image registry that supports the **Open Container Initiative (OCI)** standards. It is used to store and manage Docker container images and other OCI-compliant artifacts. Key features include:

* **Private Repositories:** Store container images securely in private repositories.
* **High Availability:** Built on OCI's robust infrastructure, ensuring high availability and durability.
* **Integration with OCI Services:** Seamlessly integrates with OCI services like OKE, Oracle Functions, and CI/CD pipelines.
* **Security:** Supports IAM policies, encryption, and vulnerability scanning for container images.

**2. Key Use Cases**

* **Pushing and Storing Container Images:** Developers can push Docker images to the OCI Registry after building them locally or through CI/CD pipelines.
* **Pulling Images for Deployment:** OKE clusters can pull images from the OCI Registry to deploy containerized applications.
* **Versioning and Management:** Manage multiple versions of container images using tags.
* **Security Scanning:** Scan images for vulnerabilities before deploying them to production.

**3. Pushing, Storing, and Managing Container Images in OCI Registry**

**Pushing Images to OCI Registry**

To push a Docker image to the OCI Registry:

1. **Authenticate with OCI Registry:**  
   Use the OCI CLI or Docker CLI to authenticate with the OCI Registry.

bash

Copy

docker login <region-key>.ocir.io

Replace <region-key> with the appropriate region code (e.g., iad for US East).

1. **Tag the Docker Image:**  
   Tag the local Docker image with the OCI Registry path.

bash

Copy

docker tag <local-image-name> <region-key>.ocir.io/<tenancy-namespace>/<repo-name>:<tag>

* + <tenancy-namespace>: Your OCI tenancy namespace.
  + <repo-name>: Name of the repository in the OCI Registry.
  + <tag>: Tag for the image (e.g., v1, latest).

1. **Push the Image:**  
   Push the tagged image to the OCI Registry.

bash

Copy

docker push <region-key>.ocir.io/<tenancy-namespace>/<repo-name>:<tag>

**Storing Images**

* Images are stored in private repositories within the OCI Registry.
* You can organize images by repository and tag.
* OCI Registry supports lifecycle policies to automatically delete old or unused images.

**Managing Images**

* Use the OCI Console, CLI, or SDKs to manage repositories and images.
* Perform actions like listing images, deleting images, or setting up access policies.

**4. Integration with Oracle Kubernetes Engine (OKE)**

**Pulling Images from OCI Registry in OKE**

To deploy applications in OKE using images stored in the OCI Registry:

1. **Create a Kubernetes Secret:**  
   Create a Kubernetes secret to store the credentials for accessing the OCI Registry.

bash

Copy

kubectl create secret docker-registry <secret-name> \

--docker-server=<region-key>.ocir.io \

--docker-username='<tenancy-namespace>/<username>' \

--docker-password='<auth-token>' \

--docker-email='<email>'

* + <auth-token>: OCI Auth Token generated for the user.
  + <username>: OCI IAM username.

1. **Reference the Secret in a Pod Definition:**  
   Use the secret in your Kubernetes deployment YAML to pull the image.

yaml

Copy

apiVersion: v1

kind: Pod

metadata:

name: my-app

spec:

containers:

- name: my-app-container

image: <region-key>.ocir.io/<tenancy-namespace>/<repo-name>:<tag>

imagePullSecrets:

- name: <secret-name>

1. **Deploy the Application:**  
   Apply the deployment to the OKE cluster.

bash

Copy

kubectl apply -f deployment.yaml

**5. Security and Access Control**

* **IAM Policies:** Control access to the OCI Registry using OCI Identity and Access Management (IAM) policies.
* **Vulnerability Scanning:** OCI Registry integrates with OCI Vulnerability Scanning Service to scan images for vulnerabilities.
* **Encryption:** Images are encrypted at rest using OCI's encryption mechanisms.

**6. Benefits of Using OCI Registry with OKE**

* **Seamless Integration:** OCI Registry is natively integrated with OKE, making it easy to deploy containerized applications.
* **Scalability:** OCI Registry scales automatically to handle large numbers of images and high traffic.
* **Cost-Effective:** Pay only for the storage and bandwidth you use.
* **Enterprise-Grade Security:** Leverage OCI's security features to protect your container images.

**7. Example Workflow**

1. **Develop and Build:** Develop your application and build a Docker image.
2. **Push to OCI Registry:** Push the Docker image to the OCI Registry.
3. **Deploy to OKE:** Create a Kubernetes deployment in OKE that pulls the image from the OCI Registry.
4. **Manage and Scale:** Use OKE to manage and scale your application.

**8. Tools and Interfaces**

* **OCI Console:** Web-based interface for managing repositories and images.
* **OCI CLI:** Command-line interface for interacting with the OCI Registry.
* **SDKs:** Use OCI SDKs (Python, Java, etc.) to automate tasks.
* **Terraform:** Manage OCI Registry resources using Terraform.

**9. Pricing**

OCI Registry pricing is based on:

* **Storage:** Cost per GB of stored images.
* **Data Transfer:** Cost for pulling images to OKE or other services.
* **Requests:** Cost for API requests (e.g., push, pull, list).

**10. Conclusion**

The OCI Registry is a powerful tool for managing container images in Oracle Cloud. When combined with Oracle Kubernetes Engine (OKE), it provides a robust platform for deploying and managing containerized applications at scale. By leveraging OCI's security, scalability, and integration capabilities, you can build and deploy applications with confidence in a cloud-native environment.

**Lab Steps**

**Step 1: Set Up OCI CLI and Docker**

1. **Configure OCI CLI:**  
   Run the following command to configure the OCI CLI:

bash

Copy

oci setup config

Follow the prompts to enter your OCI credentials (tenancy OCID, user OCID, region, etc.).

1. **Generate an Auth Token:**
   * Go to the OCI Console > Identity > Users > Your User > Auth Tokens.
   * Generate an auth token and save it securely.
2. **Login to OCI Registry with Docker:**  
   Authenticate Docker with the OCI Registry:

ruser1@cloudshell:.oci (ap-sydney-1)$ oci os ns get

{

"data": "idq2n1jub2jz"

}

Ruser1@cloudshell:.oci (ap-sydney-1)$

ruser1@cloudshell:.oci (ap-sydney-1)$ docker login syd.ocir.io

Username: [idq2n1jub2jz/ruser1@techlanders.com](mailto:idq2n1jub2jz/ruser1@techlanders.com) ( tenancy namespace/username)

Password: (Your auth token)

WARNING! Your password will be stored unencrypted in /home/ruser1/.docker/config.json.

Configure a credential helper to remove this warning. See

https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded

**Step 2: Build and Push a Docker Image to OCI Registry**

1. **Create a Simple Dockerfile:**  
   Create a directory for your Docker project:

bash

Copy

mkdir my-app

cd my-app

Create a Dockerfile:

Dockerfile

Copy

FROM nginx:alpine

COPY index.html /usr/share/nginx/html/index.html

Create an index.html file:

html

Copy

<!DOCTYPE html>

<html>

<body>

<h1>Hello from OCI Registry and OKE!</h1>

</body>

</html>

Run HTML

1. **Build the Docker Image:**  
   Build the Docker image:

bash

Copy

docker build -t my-app:1.0 .

1. **Tag the Docker Image:**  
   Tag the image for the OCI Registry:

bash

Copy

docker tag my-app:1.0 <region-key>.ocir.io/<tenancy-namespace>/my-app:1.0

* region key :

"Region": "ap-sydney-1", "Key": "SYD" }, { "Region": "us-ashburn-1", "Key": "IAD"

Example:

bash

Copy

docker tag my-app:1.0 syd.ocir.io/idq2n1jub2jz/my-app:1.0

1. **Push the Docker Image:**  
   Push the image to the OCI Registry:

bash

Copy

docker push <region-key>.ocir.io/<tenancy-namespace>/my-app:1.0

Example:

bash

Copy

docker push syd.ocir.io/idq2n1jub2jz/my-app:1.0

* Above will fail cz thrs no repo and no permissions
* Create the repository in the OCI Console:
  1. Go to the OCI Console > Developer Services > Container Registry.
  2. Click **Create Repository**.
  3. Enter the repository name (e.g., my-app) and click **Create**.

**3. Permissions :**

* Ensure your user has the necessary permissions to push images to the OCI Registry.

Noe : we don’t require belo cz our user is already in administrators group which is having access to manage all resources in tenancy already ..

* The required policy is:

plaintext

Copy

Allow group <group-name> to manage repos in tenancy

Example:

plaintext

Copy

Allow group Developers to manage repos in tenancy

1. docker push syd.ocir.io/idq2n1jub2jz/my-app:1.0
2. **Verify in OCI Console:**
   * Go to the OCI Console > Developer Services > Container Registry.
   * Verify that the my-app repository and image are listed.

**Step 3: Create an OKE Cluster**

1. **Create an OKE Cluster:**
   * Go to the OCI Console > Developer Services > Kubernetes Clusters (OKE).
   * Click **Create Cluster** and follow the wizard to create a Quick Start cluster.
   * Choose the default options for networking, node pool, and shape.
2. **Download Kubeconfig:**
   * Once the cluster is created, click **Access Cluster**.
   * Follow the instructions to download the kubeconfig file and set up kubectl.
3. **Verify Cluster Access:**  
   Run the following command to verify access to the OKE cluster:

bash

Copy

kubectl get nodes

**Step 4: Deploy the Application on OKE**

1. **Create a Kubernetes Secret for OCI Registry:**  
   Create a secret to store the OCI Registry credentials:

bash

Copy

kubectl create secret docker-registry ocir-secret \

--docker-server=<region-key>.ocir.io \

--docker-username='<tenancy-namespace>/<username>' \

--docker-password='<auth-token>' \

Example:

bash

Copy

kubectl create secret docker-registry ocir-secret \

--docker-server=syd.ocir.io \

--docker-username='idq2n1jub2jz/ruser1@techlanders.com' \

--docker-password='j]Z(t7+i5)j2eCROU(1V' \

1. **Create a Kubernetes Deployment:**  
   Create a deployment.yaml file:

yaml

Copy

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app

spec:

replicas: 2

selector:

matchLabels:

app: my-app

template:

metadata:

labels:

app: my-app

spec:

containers:

- name: my-app-container

image: <region-key>.ocir.io/<tenancy-namespace>/my-app:1.0

imagePullSecrets:

- name: ocir-secret

1. **Deploy the Application:**  
   Apply the deployment:

bash

Copy

kubectl apply -f deployment.yaml

1. **Verify the Deployment:**  
   Check the status of the deployment:

bash

Copy

kubectl get pods

1. **Expose the Application:**  
   Create a NodePort/LoadBalancer service to expose the application:

yaml

Copy

apiVersion: v1

kind: Service

metadata:

name: my-app-service

spec:

type: NodePort

ports:

- port: 80

targetPort: 80

selector:

app: my-app

Apply the service:

bash

Copy

kubectl apply -f service.yaml

1. **Access the Application:**  
   Get the external IP of thethe nodeport/loadbalancer:

bash

Copy

kubectl get svc

Open the IP in a web browser to see the "Hello from OCI Registry and OKE!" message.

**Lab Cleanup**

1. Delete the OKE cluster from the OCI Console.
2. Delete the container image from the OCI Registry.
3. Remove local Docker images and Kubernetes resources.

**Conclusion**

This lab demonstrated how to:

* Push a Docker image to the OCI Registry.
* Create an OKE cluster.
* Deploy an application on OKE using the image stored in the OCI Registry.

By following these steps, you can leverage the power of OCI Registry and OKE to build, deploy, and manage containerized applications in Oracle Cloud.

Securing Oracle Kubernetes Engine (OKE) clusters is crucial to ensure the safety and integrity of your containerized applications. Below are some best practices and tools you can use to enhance the security of your OKE clusters:

**1. Trivy: Scanning for Vulnerabilities in Container Images**

* **What is Trivy?**  
  Trivy is a comprehensive vulnerability scanner for container images, file systems, and Git repositories. It can detect vulnerabilities in OS packages (e.g., Alpine, Ubuntu, etc.) and application dependencies (e.g., npm, pip, etc.).
* **Best Practices:**
  + **Integrate Trivy into CI/CD Pipelines:** Automate vulnerability scanning during the build process to catch issues early.
  + **Scan Images Before Deployment:** Ensure that all container images are scanned before being deployed to the OKE cluster.
  + **Regularly Update Base Images:** Use Trivy to identify outdated base images and update them to the latest versions with fewer vulnerabilities.
  + **Monitor for New Vulnerabilities:** Continuously monitor your running containers for new vulnerabilities that may emerge after deployment.
* **Example Command:**

bash

Copy

trivy image <your-container-image>

**2. Kube-Bench: Benchmarking Cluster Security Against Kubernetes Best Practices**

* **What is Kube-Bench?**  
  Kube-Bench is a tool that checks whether your Kubernetes cluster is configured according to the security best practices outlined in the CIS (Center for Internet Security) Kubernetes Benchmark.
* **Best Practices:**
  + **Run Kube-Bench Regularly:** Schedule periodic scans to ensure your cluster remains compliant with security benchmarks.
  + **Address Identified Issues:** Act on the recommendations provided by Kube-Bench to harden your cluster’s security.
  + **Automate Compliance Checks:** Integrate Kube-Bench into your CI/CD pipeline or use it as part of your cluster’s monitoring and alerting system.
  + **Use Role-Based Access Control (RBAC):** Ensure that RBAC is properly configured to limit access to cluster resources.
* **Example Command:**

bash

Copy

kube-bench run

**3. Hadolint: Dockerfile Linting for Secure and Efficient Builds**

* **What is Hadolint?**  
  Hadolint is a linter for Dockerfiles that helps you write secure and efficient Docker images by checking for common mistakes and best practices.
* **Best Practices:**
  + **Lint Dockerfiles During Development:** Use Hadolint to catch issues in your Dockerfiles before building images.
  + **Minimize Image Layers:** Follow best practices for reducing the number of layers in your Docker images to minimize the attack surface.
  + **Use Multi-Stage Builds:** Leverage multi-stage builds to reduce the size of your final image and exclude unnecessary tools and dependencies.
  + **Avoid Running as Root:** Ensure that your containers run as a non-root user whenever possible to reduce the risk of privilege escalation.
* **Example Command:**

bash

Copy

hadolint Dockerfile

**Additional Security Best Practices for OKE Clusters:**

* **Enable Network Policies:** Use Kubernetes Network Policies to control traffic between pods and restrict unnecessary communication.
* **Secrets Management:** Use Kubernetes Secrets to manage sensitive information like passwords, tokens, and keys. Avoid hardcoding secrets in your Dockerfiles or application code.
* **Regularly Update Kubernetes:** Keep your OKE cluster up to date with the latest Kubernetes version to benefit from security patches and new features.
* **Monitor and Audit:** Use tools like Falco or OCI Monitoring to detect and respond to suspicious activities in your cluster.

**Lab: Implementing Security Best Practices in OKE**

**Lab Objectives**

1. Set up an OKE cluster.
2. Deploy a sample application.
3. Use **Trivy** to scan container images for vulnerabilities.
4. Use **Kube-Bench** to benchmark the OKE cluster against Kubernetes security best practices.
5. Use **Hadolint** to lint Dockerfiles for secure and efficient builds.

**Step 3: Use Trivy to Scan Container Images**

1. docker run --rm -v /var/run/docker.sock:/var/run/docker.sock \

-v $HOME/Library/Caches:/root/.cache/ aquasec/trivy:0.18.3 python:3.4-alpine

1. **Scan the Container Image**:
   * Run Trivy to scan the image for vulnerabilities:
   * Test some present images for vulnerabilities

* Create an image from below and test it :

docker run --rm -v /var/run/docker.sock:/var/run/docker.sock \

-v $HOME/Library/Caches:/root/.cache/ aquasec/trivy:0.18.3 raman-app-image

* + Review the output for vulnerabilities and fix them if necessary.

**Step 4: Use Kube-Bench to Benchmark Cluster Security**

1. **Install Kube-Bench**:

kube-bench :

https://www.cisecurity.org/cis-benchmarks

installation :

https://github.com/aquasecurity/kube-bench/blob/main/docs/installation.md

fromdocker :

docker run --rm -v `pwd`:/host docker.io/aquasec/kube-bench:latest install

ln -s /root/kube-bench /usr/bin/kube-bench

203 clear

204 ls

205 kube-bench

1. **Run Kube-Bench**:
   * Execute Kube-Bench to check the OKE cluster against CIS benchmarks:

bash

Copy

kube-bench run

* + Review the results and address any failures or warnings.

**Step 5: Use Hadolint to Lint Dockerfiles**

1. docker run --rm -i ghcr.io/hadolint/hadolint < Dockerfile
2. test some dockerfiles and match the rules
3. **Lint the Dockerfile**:
   * Run Hadolint to check the Dockerfile for best practices:

bash

Copy

hadolint Dockerfile

* + Review the output and fix any issues (e.g., using latest tags, running as root, etc.).

**Step 6: Clean Up**

1. **Delete the OKE Cluster**:
   * Go to the OCI Console, navigate to **Developer Services > Kubernetes Clusters (OKE)**, and delete the cluster.
2. **Remove Local Resources**:
   * Delete the sample application and Docker images:

bash

Copy

docker rmi your-oci-username/sample-app:latest

rm -rf sample-app

**Lab Summary**

* You set up an OKE cluster and deployed a sample application.
* You used **Trivy** to scan container images for vulnerabilities.
* You used **Kube-Bench** to benchmark the OKE cluster against Kubernetes security best practices.
* You used **Hadolint** to lint Dockerfiles for secure and efficient builds.

This lab demonstrates how to integrate security tools into your OKE workflow to ensure a secure and compliant Kubernetes environment.

**OCI Vaults Integration with OKE (Oracle Kubernetes Engine)**

Managing secrets and credentials securely is a critical aspect of deploying and running applications on **Oracle Kubernetes Engine (OKE)**. **OCI Vaults** provides a secure way to store and manage sensitive data such as passwords, API keys, database credentials, and certificates.

**1. What is OCI Vault?**

**OCI Vault** is a managed service in Oracle Cloud Infrastructure (OCI) that provides:

* **Secrets Management** (storing API keys, passwords, certificates, etc.)
* **Encryption Keys** (to encrypt data at rest and in transit)
* **Access Control** (using OCI Identity and Access Management (IAM))

When using OKE, applications often need credentials for databases, APIs, and third-party services. Storing these credentials **inside container images or Kubernetes configurations is unsafe**. Instead, we use **OCI Vault** to securely store and retrieve them when needed.

**2. How OCI Vaults Work with OKE?**

OCI Vaults **securely store secrets**, and OKE pods can access them dynamically without hardcoding sensitive information.

**Steps to integrate OCI Vaults with OKE:**

1. **Create a Vault** → Store sensitive data (secrets, API keys).
2. **Assign Policies** → Grant OKE permissions to access the Vault.
3. **Retrieve Secrets in OKE** → Use Kubernetes Secrets or direct API calls.

**3. Detailed Steps to Use OCI Vault in OKE**

**Step 1: Create an OCI Vault**

1. Go to the **OCI Console** → Navigate to **Vaults**.
2. Click **Create Vault** → Choose **Type: Virtual Private Vault**.
3. Click **Create Key** → Choose **Key Type: AES** (for encryption).
4. Click **Create Secret** → Enter the **secret value** (e.g., database password).

**Step 2: Define IAM Policies for OKE to Access OCI Vault**

By default, OKE **does not** have permission to access OCI Vaults. You must grant access using IAM policies.

* Go to **Identity & Security → Policies**.
* Create a new policy:

pgsql

CopyEdit

allow dynamic-group oke-pods to read secret-bundles in compartment <compartment-name>

* + **Dynamic Group**: A group of OKE pods that need access.
  + **Secret-bundles**: Allows reading secrets from OCI Vault.
* To allow OKE to **list** secrets:

pgsql

CopyEdit

allow dynamic-group oke-pods to inspect vaults in compartment <compartment-name>

allow dynamic-group oke-pods to read vaults in compartment <compartment-name>

**Step 3: Expose OCI Vault Secrets to OKE**

OCI Vault secrets can be accessed in OKE using two methods:

1. **Using Kubernetes Secrets (Recommended)**
2. **Direct API Calls to OCI Vault (For advanced use cases)**

**Method 1: OCI Vault + Kubernetes Secrets**

1. **Create a Kubernetes Secret** using OCI Vault secret:

yaml

CopyEdit

apiVersion: v1

kind: Secret

metadata:

name: my-secret

type: Opaque

data:

password: BASE64\_ENCODED\_SECRET\_VALUE

* + Replace BASE64\_ENCODED\_SECRET\_VALUE with the **Base64-encoded** secret.

1. **Mount the Secret in a Pod**:

yaml

CopyEdit

apiVersion: v1

kind: Pod

metadata:

name: my-app

spec:

containers:

- name: my-container

image: my-app:latest

env:

- name: DB\_PASSWORD

valueFrom:

secretKeyRef:

name: my-secret

key: password

**Method 2: Direct API Calls to OCI Vault**

For dynamic secret retrieval inside the application, the pod can call OCI Vault APIs.

1. **Enable IAM authentication in the Pod** using a **Service Account**.
2. **Use OCI SDK or CLI** inside the application to fetch secrets.
3. **Example API call to fetch a secret:**

bash

CopyEdit

oci secrets secret-bundle get --secret-id <SECRET\_OCID>

**4. Best Practices for Using OCI Vault in OKE**

* **Never store secrets in ConfigMaps or environment variables in plain text.**
* **Use Kubernetes Secrets** to inject OCI Vault secrets into pods.
* **Use IAM Policies** to control access to secrets securely.
* **Rotate Secrets Regularly** using OCI Vault’s built-in rotation features.
* **Use Least Privilege Access** (grant only necessary permissions to pods).

**5. Summary**

| **Feature** | **Benefit** |
| --- | --- |
| **OCI Vault** | Securely stores secrets (passwords, API keys, certificates). |
| **IAM Policies** | Ensures only authorized OKE pods can access secrets. |
| **Kubernetes Secrets** | Integrates OCI Vault with Kubernetes applications. |
| **Direct API Calls** | Allows dynamic retrieval of secrets inside applications. |

By integrating OCI Vault with OKE, you ensure that sensitive data is securely managed, reducing security risks while maintaining seamless application deployment.

**Lab Overview**

**Objectives:**

1. **Create an OCI Vault** and store a secret (e.g., database password).
2. **Configure IAM policies** to allow OKE to access the vault.
3. **Deploy a Kubernetes Secret** referencing OCI Vault.
4. **Use the secret inside an OKE Pod**.
5. **(Optional) Retrieve the secret dynamically using OCI CLI or SDK.**

**📌 Prerequisites**

✔️ **OCI Account** with permissions to create Vaults, Policies, and OKE resources  
✔️ **OKE Cluster** (with worker nodes running)  
✔️ **OCI CLI installed** on your local machine  
✔️ **kubectl configured** to connect to the OKE cluster

**🛠️ Step 1: Create an OCI Vault and Store a Secret**

1. **Go to the OCI Console** → Navigate to **Identity & Security → Vault**.
2. Click **Create Vault**
3. Click **Create Master Encryption Key** → Select **AES 256-bit**.
4. Click **Create Secret**, enter:
   * **Name:** db-password
   * **Value:** SuperSecret123
   * **Encryption Key:** Select the key you created earlier.
5. **Copy the OCID** of the secret for later use.

**🔑 Step 2: Configure IAM Policies for OKE to Access Vault**

1. Navigate to **Identity & Security → Policies**.
2. Click **Create Policy**, name it **OKE-Vault-Access**.
3. Add the following policy (**replace <compartment-name> with your compartment**):

bash

CopyEdit

allow dynamic-group oke-pods to read secret-bundles in compartment <compartment-name>

allow dynamic-group oke-pods to inspect vaults in compartment <compartment-name>

allow dynamic-group oke-pods to read vaults in compartment <compartment-name>

OR

* If ur applying policy in root compartment ; then :

allow dynamic-group oke-pods to read secret-bundles in tenancy

allow dynamic-group oke-pods to inspect vaults in tenancy

allow dynamic-group oke-pods to read vaults in tenancy

allow dynamic-group oke-pods to use secret-family in tenancy

1. **Create a Dynamic Group** for OKE pods: oke-pods
   * Navigate to **Identity & Security → Dynamic Groups**.
   * Create a group with **matching rule**:

bash

CopyEdit

ALL {instance.compartment.id = '<compartment-OCID>'}

* + This allows **OKE worker nodes** to authenticate and retrieve secrets.

**📥 Step 3: Fetch Secret from OCI Vault and Create Kubernetes Secret**

1. Run the following **OCI CLI command** to get the secret value:

oci secrets secret-bundle get --secret-id <secretID>

oci secrets secret-bundle get --secret-id <secretID> --query 'data."secret-bundle-content"."content"' --raw-output | base64 --decode

**Automating OCI Vault to Kubernetes Secret Sync**

Instead of manually pulling and encoding the secret, we can automate the process :

**step 2: Deploy Secrets Store CSI Driver & OCI Provider**

* 1. **Install Secrets Store CSI Driver**

helm repo add oci-provider https://oracle.github.io/oci-secrets-store-csi-driver-provider/charts

helm install oci-provider oci-provider/oci-secrets-store-csi-driver-provider --namespace kube-system

**3. Verify Installation**

bash

Copy

kubectl get pods -n kube-system -l "app.kubernetes.io/name in (oci-secrets-store-csi-driver-provider, secrets-store-csi-driver)"

**Step 4: Create SecretProviderClass**

**secret-provider.yaml**

yaml

Copy

apiVersion: secrets-store.csi.x-k8s.io/v1

kind: SecretProviderClass

metadata:

name: oci-vault-demo

spec:

provider: oci

parameters:

vaultId: "ocid1.vault.oc1..<your-vault-ocid>" # Replace with your vault OCID

authType: "instance" # Uses Instance Principal

secrets: |

- name: "demo-db-password" # Name of your secret in OCI Vault

stage: "CURRENT" # or "LATEST", "PENDING", etc.

**Apply Configuration**

bash

Copy

kubectl apply -f secret-provider.yaml

**Step 5: Deploy a Pod to Access Secrets**

**pod-demo.yaml**

yaml

Copy

apiVersion: v1

kind: Pod

metadata:

name: demo-app

spec:

containers:

- name: demo-app

image: nginx:latest

volumeMounts:

- name: secrets-store

mountPath: "/mnt/secrets-store"

readOnly: true

volumes:

- name: secrets-store

csi:

driver: secrets-store.csi.k8s.io

readOnly: true

volumeAttributes:

secretProviderClass: "oci-vault-demo"

**Deploy the Pod**

bash

Copy

kubectl apply -f pod-demo.yaml

**NOTE : if u get error ; than check for the issue :**

**kubectl logs -n kube-system -l app.kubernetes.io/name=oci-secrets-store-csi-driver-provider**

* **If network connectivity is an issue , open all ports on security list on workernode subnet …ingress**
* **Than restart the csi driver :**

**kubectl delete pod -n kube-system -l app.kubernetes.io/name=oci-secrets-store-csi-driver-provider**

**Step 6: Verify Secrets in the Pod**

**Check Mounted Secrets**

bash

Copy

kubectl exec -it demo-app -- cat /mnt/secrets-store/demo-db-password

# Output: my-super-secret-password

**Optional: Sync to Kubernetes Secret**

Add to SecretProviderClass:

yaml

Copy

secretObjects: # Optional: Sync to Kubernetes Secret

- secretName: "db-password"

type: Opaque

data:

- objectName: "demo-db-password"

key: "password"

**Troubleshooting Common Issues in Oracle Kubernetes Engine (OKE)**

When working with Oracle Kubernetes Engine (OKE), you may encounter a variety of issues related to networking, storage, and access. Here’s how you can identify and fix common problems, along with some tools and commands to efficiently debug them.

**1. Networking Issues**

Networking issues are among the most common problems in OKE clusters, especially when communication between pods, services, or external systems fails.

**Common Networking Issues:**

* **Pods unable to communicate with each other**: This can be due to misconfigured Network Policies or issues with CNI (Container Network Interface) plugins.
* **Service not accessible**: A service may not be reachable due to improper port exposure or misconfigured load balancers.
* **DNS resolution failure**: When DNS within the cluster fails, pods may not be able to reach each other via service names.

**Troubleshooting Steps:**

1. **Check Network Policies**: Verify if any network policies are blocking traffic.

bash

CopyEdit

kubectl get networkpolicy

1. **Verify Pod Network Configuration**: Ensure that the CNI plugin is correctly installed and configured.

bash

CopyEdit

kubectl get pods -n kube-system -l k8s-app=kube-dns

kubectl logs <kube-dns-pod-name> -n kube-system

1. **Inspect Service and Endpoints**: Ensure services are correctly configured and endpoints are reachable.

bash

CopyEdit

kubectl get svc

kubectl get endpoints

1. **Check DNS Resolution**: Test DNS resolution inside a pod.

bash

CopyEdit

kubectl run -i --tty --rm debug --image=busybox --restart=Never -- nslookup <service-name>

**Useful Tools:**

* **kubectl describe**: Gives detailed information on networking resources.
* **kubectl logs**: To check logs for networking-related issues in CNI pods.

**2. Storage Issues**

Storage problems often arise when persistent volumes (PVs) are not properly provisioned or accessed by pods.

**Common Storage Issues:**

* **PVC binding failure**: Persistent Volume Claims (PVCs) might fail to bind to available Persistent Volumes (PVs) if there is no matching storage class or available PV.
* **Pod can't access volume**: Permissions or mount issues may cause pods to be unable to access volumes.
* **Read-write conflict**: Multiple pods trying to mount the same volume in incompatible modes (e.g., read-only vs. read-write).

**Troubleshooting Steps:**

1. **Check PVC Status**: Ensure PVCs are correctly bound to PVs.

bash

CopyEdit

kubectl get pvc

1. **Inspect PVs**: Ensure that the PVs are available and have the correct access modes and storage class.

bash

CopyEdit

kubectl get pv

1. **Examine Pod Volume Mounts**: Check if volumes are properly mounted to the pod and accessible.

bash

CopyEdit

kubectl describe pod <pod-name>

1. **Check for Events**: Sometimes, storage issues generate events that will indicate what went wrong.

bash

CopyEdit

kubectl get events --sort-by='.metadata.creationTimestamp'

**Useful Tools:**

* **kubectl describe**: Provides detailed info on the PVC, PV, and pod.
* **OCI Console/CLI**: For inspecting the status of underlying cloud storage resources.

**3. Access Issues**

Access issues often occur due to misconfigured RBAC, authentication, or authorization settings.

**Common Access Issues:**

* **Permission Denied**: Users or service accounts might not have the right permissions to access resources.
* **Token Expiration**: If using service accounts, tokens might expire.
* **Cluster Role/RoleBinding Misconfiguration**: The cluster may be misconfigured in terms of roles and bindings, preventing access to certain namespaces or resources.

**Troubleshooting Steps:**

1. **Verify RBAC Configurations**: Ensure that the user or service account has the correct roles and bindings.

bash

CopyEdit

kubectl get roles,rolebindings --all-namespaces

kubectl get clusterroles,clusterrolebindings

1. **Check Service Account Permissions**: Ensure that the service account used by the pods has the required access.

bash

CopyEdit

kubectl describe sa <service-account-name> -n <namespace>

1. **Verify kubeconfig**: Ensure the kubeconfig file is set up correctly with the right context and credentials.

bash

CopyEdit

kubectl config view

kubectl config current-context

1. **Check Token Expiry**: Ensure that service account tokens or user tokens are not expired.

**Useful Tools:**

* **kubectl auth can-i**: Check if a user or service account has permission to perform an action.

bash

CopyEdit

kubectl auth can-i create pods --as=<user> --namespace=<namespace>

**4. General Debugging Commands and Tools**

* **kubectl logs**: Access pod logs for error messages related to networking, storage, and access issues.

bash

CopyEdit

kubectl logs <pod-name>

* **kubectl describe**: Get detailed information on resources like pods, services, deployments, PVCs, etc.

bash

CopyEdit

kubectl describe <resource-type> <resource-name>

* **kubectl get events**: View the cluster’s events for any anomalies or errors.

bash

CopyEdit

kubectl get events

* **kubectl exec**: Execute commands inside a pod to troubleshoot networking or access issues.

bash

CopyEdit

kubectl exec -it <pod-name> -- /bin/bash

By systematically using these tools and methods, you can efficiently troubleshoot and resolve common issues in OKE, ensuring smooth operations for your Kubernetes workloads.