**Objective: Bolster security measures for applications on EKS.**

**Tasks:**

1. Store secrets in AWS Secrets Manager.

2. Access secrets in Kubernetes applications.

3. Implement Role-Based Access Control.

**Documentation:**

- Kubernetes security principles.

- AWS Secrets Manager overview.

- RBAC in Kubernetes.

**Task: 1**

**1. Store secrets in AWS Secrets Manager**

AWS Secrets Manager helps you to protect access to your applications, services, and IT resources without the upfront investment and on-going maintenance costs of operating your own infrastructure. Here's how to store a secret:

**AWS CLI:**

First, make sure you have the AWS CLI installed and configured.

| # Store a secret aws secretsmanager create-secret --name MySecretName --description "My secret description" --secret-string "MySecretValue" |
| --- |

**Task: 2**

**2. Access secrets in Kubernetes applications**

Now that you've stored the secret, you can access it in your Kubernetes applications using different ways. One common method is using a controller like external-secrets, which syncs AWS Secrets Manager secrets and creates Kubernetes secrets.

**Steps:**

Given your provided values for username and password, let's create a Kubernetes secret for them.

1. Encode the values in Base64:

For the username (secret-user):

| echo -n 'secret-user' | base64 |
| --- |

This should output: c2VjcmV0LXVzZXI=

For the password (secret-pass):

| echo -n 'secret-pass' | base64 |
| --- |

This should output: c2VjcmV0LXBhc3M=

1. **Create the Kubernetes Secret:**

**Using the Base64 encoded values, we'll craft a Secret manifest:**

| **apiVersion: v1 kind: Secret metadata:  name: mysecret type: Opaque data:  username: c2VjcmV0LXVzZXI=  password: c2VjcmV0LXBhc3M=** |
| --- |

**Save this to a file, for instance mysecret.yaml.**

| **kubectl apply -f mysecret.yaml** |
| --- |

1. **Use the secret in a Pod:**

| **apiVersion: v1 kind: Pod metadata:  name: mypod spec:  containers:  - name: mycontainer  image: nginx  env:  - name: MY\_USERNAME  valueFrom:  secretKeyRef:  name: mysecret  key: username  - name: MY\_PASSWORD  valueFrom:  secretKeyRef:  name: mysecret  key: password** |
| --- |

| **kubectl apply -f mypod.yaml** |
| --- |

**1. Check that the secret exists in Kubernetes:**

Using kubectl, you can check if the Kubernetes secret (mysecret in our example) has been created:

| kubectl get secrets mysecret -o=jsonpath='{.data.mySecretKey}' |
| --- |

This command will return the Base64 encoded value of the secret. You can decode it using:

| **kubectl** get secrets mysecret -o=jsonpath='{.**data**.mySecretKey}' | base64 --decode |
| --- |

**First, get a shell inside your Pod:**

**Then, print the environment variable:**

| kubectl exec -it mypod -- /bin/sh |
| --- |

| echo $MY\_USERNAME echo $MY\_PASSWORD |
| --- |

**Task: 3**

**3. Implement Role-Based Access Control (RBAC) in Kubernetes**

Kubernetes RBAC allows you to control who can do what on your cluster. The main components are **Role**, **ClusterRole**, **RoleBinding**, and **ClusterRoleBinding**.

**Example:**

Suppose you want to give a specific user read-only access to pods in the "default" namespace:

1. **Create a Role:**

| apiVersion: rbac.authorization.k8s.io/v1 kind: Role metadata:  namespace: default  name: pod-reader rules: - apiGroups: [""]  resources: ["pods"]  verbs: ["get", "list"] |
| --- |

Apply it:

| kubectl apply -f pod-reader-role.yaml |
| --- |

1. **Create a RoleBinding to bind the Role to a specific user:**

| apiVersion: rbac.authorization.k8s.io/v1 kind: RoleBinding metadata:  name: read-pods  namespace: default subjects: - kind: User  name: "ali"  apiGroup: rbac.authorization.k8s.io roleRef:  kind: Role  name: pod-reader  apiGroup: rbac.authorization.k8s.io |
| --- |

Apply it:

| kubectl apply -f pod-reader-rolebinding.yaml |
| --- |

Now, user **"ali"** can read pods in the "default" namespace.

**To check the permissions of the jack user:**

| kubectl auth can-i **get** pod --**as** ali |
| --- |

**Documentation:**

* **Kubernetes Security Principles:** Kubernetes offers a multi-layered security system, starting from the cluster level to the pod level. Key features include Network Policies, Pod Security Policies, RBAC, and Node authentication/authorization.
* **AWS Secrets Manager Overview:** AWS Secrets Manager allows you to securely store, retrieve, and manage sensitive information like API keys, passwords, and database credentials. Rotation, auditing, and access control are some of its features.
* **RBAC in Kubernetes:** RBAC is a method of regulating access to computer or network resources based on the roles of individual users within an enterprise. In Kubernetes, it lets administrators control who can access the Kubernetes API and what permissions they have.