**Lab 1: Review Kubernetes Security Using CIS Benchmarks**

**Lab Title:**

Review Kubernetes Security Using CIS Benchmarks

**Lab Objective:**

* Assess your Kubernetes cluster security against the **CIS Kubernetes Benchmark**.
* Identify misconfigurations and potential security vulnerabilities in cluster components (control plane, worker nodes, kubelet, etcd, kube-proxy, kube-apiserver).
* Understand the remediation recommendations for security hardening.

**Prerequisites:**

1. Kubernetes cluster (3 nodes recommended: 1 master, 2 workers)
2. kubectl configured and pointing to the cluster
3. Linux terminal access to deploy jobs
4. curl or wget installed for fetching resources

**Step 0: Verify Cluster Access**

Before starting, ensure you can access the cluster:

kubectl get nodes -o wide

**Expected Output:**

* List of all nodes with status Ready.
* Example:

| **NAME** | **STATUS** | **ROLES** | **AGE** | **VERSION** |
| --- | --- | --- | --- | --- |
| master-1 | Ready | control-plane | 10d | v1.28.0 |
| worker-1 | Ready | <none> | 10d | v1.28.0 |
| worker-2 | Ready | <none> | 10d | v1.28.0 |

✅ If any node is NotReady, fix cluster connectivity before proceeding.

**Step 1: Deploy kube-bench**

kube-bench is a tool that checks Kubernetes clusters against **CIS Benchmarks**.

1. Deploy as a **Job** in your cluster:

kubectl apply -f https://raw.githubusercontent.com/aquasecurity/kube-bench/main/job.yaml

1. Verify the job was created:

kubectl get jobs

**Expected Output:**

NAME COMPLETIONS DURATION AGE

kube-bench 0/1 10s 5s

**Step 2: Run the kube-bench Job**

1. Get the pod name created by the job:

kubectl get pods -l job-name=kube-bench

Example Output:

NAME READY STATUS RESTARTS AGE

kube-bench-xyz123 0/1 Completed 0 10s

1. View the logs to see the benchmark results:

kubectl logs kube-bench-xyz123

**Step 3: Analyze kube-bench Output**

* The log output is divided into sections for different components: **Master Node**, **Worker Node**, and **Policies**.
* Key sections include:
  + **Etcd** – data store security
  + **Kubelet** – node agent security
  + **Kube-API Server** – access control and TLS configurations
  + **Kube-Proxy** – network security
  + **Controller Manager** & **Scheduler** – component access control

**Sample Output Section:**

[INFO] 1.1.1 Ensure that the --anonymous-auth argument is set to false (Master Node)

[FAIL] Anonymous access to API server is enabled

[INFO] 1.1.2 Ensure that the --kubelet-https argument is set to true (Master Node)

[PASS] Kubelet HTTPS is enabled

**Legend:**

* [PASS] – configuration is compliant
* [FAIL] – configuration is non-compliant, action required
* [WARN] – configuration may not be fully secure

**Step 4: Identify Critical Findings**

1. Focus on [FAIL] or [WARN] items with **high severity**, e.g.:
   * Anonymous access enabled on API server
   * etcd not using TLS
   * HostPath volumes used without restrictions
   * Default service accounts not restricted
2. Document findings in a table for remediation planning:

| **Component** | **CIS Rule** | **Status** | **Notes / Recommendation** |
| --- | --- | --- | --- |
| API Server | --anonymous-auth=false | FAIL | Disable anonymous access |
| Etcd | TLS enabled | FAIL | Configure TLS for etcd |
| Kubelet | Authentication and Authorization enabled | WARN | Restrict access using RBAC |

**Step 5: (Optional) Run kube-bench Locally**

For more detailed analysis, run kube-bench locally on nodes:

curl -L https://github.com/aquasecurity/kube-bench/releases/download/v0.7.0/kube-bench\_0.7.0\_linux\_amd64.tar.gz -o kube-bench.tar.gz

tar -xvf kube-bench.tar.gz

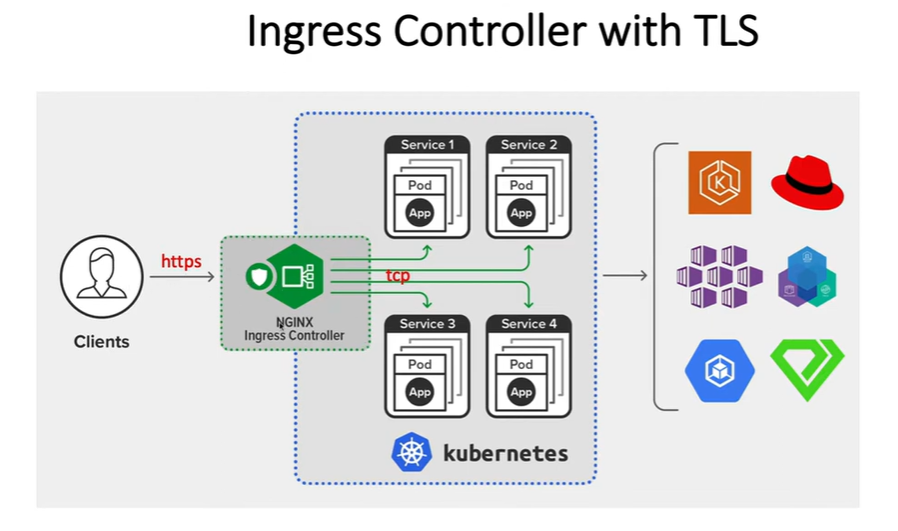
sudo ./kube-bench

* This allows you to test the cluster without deploying a Job.
* Useful for iterative remediation testing.

**Key Takeaways**

* CIS benchmarks are **the standard for Kubernetes security**.
* kube-bench automates compliance checks.
* Lab focuses on **detection first**, remediation comes in later labs.

**Lab 2 : Configure Ingress with TLS for Multiple Services**



**Lab Title:**

Secure Multi-Service Ingress Setup with TLS

**Lab Objective:**

* Deploy an Nginx Ingress controller using Helm.
* Deploy multiple applications with their services.
* Configure Ingress rules for multiple paths.
* Add TLS certificates to secure traffic.
* Test HTTPS access via Ingress.

**Prerequisites:**

* Kubernetes cluster running with kubectl access.
* Latest aws-cli installed if using EKS.
* Helm installed (v3+).
* Linux terminal access with root/sudo.

**Step 0: Verify Cluster Access**

alias k=kubectl

k get nodes -o wide

**Expected:** All nodes should show STATUS as Ready.

**Step 1: Install Helm (if not already installed)**

curl -fsSL -o get\_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3

chmod 700 get\_helm.sh

./get\_helm.sh

helm version

**Step 2: Create Ingress Namespace**

k create ns ingress

k get ns

**Expected Output:** Namespace ingress created.

**Step 3: Add and Update Nginx Helm Repository**

alias “helm= sudo helm”

helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

helm repo update

helm list

helm repo list

**Step 4: Install Nginx Ingress Controller**

helm install my-release ingress-nginx/ingress-nginx \

--namespace ingress \

--set controller.replicaCount=2 \

--set controller.nodeSelector."beta\.kubernetes\.io/os"=linux \

--set defaultBackend.nodeSelector."beta\.kubernetes\.io/os"=linux

Check pods:

k get pods -n ingress

**Expected:** Two controller pods running (Running 1/1).

**Step 5: Deploy Sample Applications and Services**

Create ingdeploy.yaml:

# Deployment and Service 1

apiVersion: apps/v1

kind: Deployment

metadata:

name: test-app

labels:

app: test-app

namespace: test-app

spec:

replicas: 3

selector:

matchLabels:

app: test-app

template:

metadata:

labels:

app: test-app

spec:

containers:

- name: test-app

image: nginx:latest

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: raman-service

namespace: test-app

spec:

type: NodePort

ports:

- port: 80

selector:

app: test-app

# Deployment and Service 2

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: test-app2

labels:

app: test-app2

namespace: test-app

spec:

replicas: 3

selector:

matchLabels:

app: test-app2

template:

metadata:

labels:

app: test-app2

spec:

containers:

- name: test-app2

image: httpd

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: raman-service2

namespace: test-app

spec:

type: NodePort

ports:

- port: 80

selector:

app: test-app2

Apply the deployments and services:

k create -f ingdeploy.yaml -n ingress

k get pods -n ingress

k get svc -n ingress

**Step 6: Create TLS Secret**

1. Generate self-signed certificate:

openssl req -x509 -nodes -days 365 -newkey rsa:2048 \

-keyout tls.key -out tls.crt \

-subj "/CN=example.com/O=example"

1. Create TLS secret in the Ingress namespace:

k create secret tls raman-tls \

--cert=tls.crt \

--key=tls.key \

-n test-app

k get secret -n test-app

**Step 7: Create Ingress Resource with TLS**

Create ingress.yaml:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: test-app-ingress

namespace: test-app

annotations:

nginx.ingress.kubernetes.io/ssl-redirect: "true"

nginx.ingress.kubernetes.io/use-regex: "true"

nginx.ingress.kubernetes.io/rewrite-target: /$2

spec:

ingressClassName: nginx

tls:

- secretName: raman-tls

rules:

- http:

paths:

- path: /raman(/|$)(.\*)

pathType: ImplementationSpecific

backend:

service:

name: raman-service

port:

number: 80

- path: /raman2(/|$)(.\*)

pathType: ImplementationSpecific

backend:

service:

name: raman-service2

port:

number: 80

Apply the ingress:

k apply -f ingress.yaml -n ingress

k get ingress -n ingress

**Test HTTPS Access locally and on browser :**

k get nodes -o wide

285 k get svc -A

286 curl http://172.31.47.130:31074/raman

287 curl https://172.31.47.130:30342/raman

288 curl - https://172.31.47.130:30342/raman

289 curl -k <https://172.31.47.130:30342/raman>

curl -kv <https://172.31.47.130:30342/raman>

On browser :

<https://18.218.88.186:30342/raman2>

https://18.218.88.186:30342/raman

**Test HTTPS Access ( if having an external ip of lb )**

1. Map example.com to your cluster’s external IP in /etc/hosts:

sudo nano /etc/hosts

# Add line: <ALB-EXTERNAL-IP> example.com

1. Access services via HTTPS:

curl -vk https://example.com/raman

curl -vk https://example.com/raman2

**Expected:** Both services respond over HTTPS.

**Lab Outcome**

* We can deploy an **Ingress controller via Helm**.
* Deploy multiple applications/services in a namespace.
* Configure **Ingress paths** for multiple services.
* Add **TLS certificates** for secure HTTPS access.
* Extend ingress by adding **new applications and paths**.

**Lab 3: Protect Node Metadata and Endpoints**

**Lab Title:**

🔐 *Securing Node Metadata and Endpoints in Kubernetes*

**Lab Objective**

By the end of this lab, trainees will be able to:

* Understand how the cloud instance metadata API (e.g., AWS EC2 169.254.169.254) can leak sensitive credentials.
* Simulate metadata access from inside a Kubernetes Pod.
* Restrict metadata endpoint access using **NetworkPolicies**.
* Harden node configuration to protect **Kubelet, etcd**, and other component endpoints.
* Validate the protection using real tests.

**Prerequisites**

* Kubernetes cluster with kubectl access (EKS preferred but works on any cluster).
* Namespace security-lab created for isolation:
* k create ns security-lab
* Basic familiarity with NetworkPolicies.
* Node access (for kubelet and metadata inspection).

**Step 0: Verify Cluster Connectivity**

alias k=kubectl

k get nodes -o wide

✅ **Expected Output:**  
All nodes show STATUS=Ready.

**Step 1: Understand the Metadata Endpoint**

Every cloud provider exposes an **Instance Metadata Service (IMDS)** that provides credentials, tokens, and configuration data to workloads.

For example:

* **AWS**: http://169.254.169.254/latest/meta-data/
* **GCP**: http://metadata.google.internal/computeMetadata/v1/
* **Azure**: http://169.254.169.254/metadata/instance

If a pod inside Kubernetes can access this endpoint, it could **steal cloud credentials** and gain access to cloud APIs.

**Step 2: Simulate a Metadata Attack**

We’ll simulate how an attacker might exploit this from a pod.

Create a test pod that allows shell access:

cat <<EOF | k apply -n security-lab -f -

apiVersion: v1

kind: Pod

metadata:

name: metadata-tester

spec:

containers:

- name: curl

image: curlimages/curl

command: ["sleep", "3600"]

EOF

Wait for it to run:

k get pods -n security-lab

Once it’s running, **exec into it**:

k exec -it metadata-tester -n security-lab -- sh

Inside the pod, run:

curl -v http://169.254.169.254/latest/meta-data/

✅ **Expected Output (if allowed):**  
You should see metadata keys like ami-id, instance-id, etc.

IF ABOVE DOESN’T WORK

* **Access Metadata Using IMDSv2 (Modern AWS)**

AWS now enforces **IMDSv2**, which requires a session token for security.

Inside your pod, run:

# Get a metadata token

TOKEN=$(curl -s -X PUT "http://169.254.169.254/latest/api/token" \

-H "X-aws-ec2-metadata-token-ttl-seconds: 60")

# Use token to access metadata

curl -H "X-aws-ec2-metadata-token: $TOKEN" \

http://169.254.169.254/latest/meta-data/ami-id

✅ **Expected Output:**

ami-07ab25c1234ef5678

You can also fetch other values:

curl -H "X-aws-ec2-metadata-token: $TOKEN" \

http://169.254.169.254/latest/meta-data/instance-id

💡 This proves that IMDSv2 adds a **protective layer**, but if Pods can still connect to this IP, they can still exploit it.

Exit the pod:

exit

⚠️ **This demonstrates a real security risk** — a compromised pod can access your node’s metadata!

Exit the container:

exit

**Step 3: Block Metadata Access Using NetworkPolicy**

Now let’s create a **restrictive NetworkPolicy** that prevents pods from reaching the metadata endpoint.

Create a YAML file:

cat <<EOF > block-metadata.yaml

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: deny-metadata-access

namespace: security-lab

spec:

podSelector: {}

policyTypes:

- Egress

egress:

- to:

- ipBlock:

cidr: 0.0.0.0/0

except:

- 169.254.169.254/32

ports:

- protocol: TCP

port: 80

EOF

Apply the policy:

k apply -f block-metadata.yaml

Check:

k get networkpolicy -n security-lab

✅ **Expected:**  
Policy deny-metadata-access should be listed.

**Step 4: Validate the Protection**

Re-run the same test inside the pod:

k exec -it metadata-tester -n security-lab – sh

# Use token to access metadata

curl -H "X-aws-ec2-metadata-token: $TOKEN" \

http://169.254.169.254/latest/meta-data/ami-id

✅ **Expected Result:**  
The request should now *fail* with:

curl: (7) Failed to connect to 169.254.169.254 port 80: Connection timed out

This confirms metadata access is successfully blocked!

Exit:

exit

**Step 5: Additional Node Endpoint Protections**

**5.1 Protect Kubelet API**

Each node runs a **Kubelet API** (default on port 10250), which exposes sensitive information and control endpoints.

To verify access:

curl -k https://<NODE-IP>:10250/metrics

If this returns metrics without authentication, it’s a misconfiguration.

**Mitigation:**

* Ensure the Kubelet is configured with:
* --anonymous-auth=false
* --read-only-port=0
* Validate via:
* ps -ef | grep kubelet | egrep "anonymous|read-only-port"

**5.2 Protect etcd**

**etcd** stores all cluster secrets. Ensure it:

* Uses client/server TLS (--cert-file, --key-file, --client-cert-auth).
* Listens only on localhost or private subnet.

Check via:

sudo netstat -plnt | grep etcd

✅ **Expected:**  
etcd should be bound to 127.0.0.1:2379 and not exposed publicly.

**Step 6: Clean Up**

When done testing:

k delete ns security-lab

**Step 7: Verification and Discussion**

✅ Confirm:

* Metadata access from Pods is blocked.
* Kubelet and etcd are not exposed.
* You understand where and how metadata endpoints can leak credentials.

**Lab Outcome**

By completing this lab, you have learned:

* How metadata endpoints can be exploited by compromised Pods.
* How to use NetworkPolicies to prevent metadata leaks.
* How to check and harden node-level APIs (Kubelet, etcd).
* How to enforce cloud-level metadata protections (IMDSv2 / iptables).

**Lab 4: Verify Platform Binaries Before Deployment**

**Lab Title:**

🔒 *Validating Kubernetes Component Binaries and Container Images*

**Lab Objective**

By the end of this lab, trainees will be able to:

* Verify the integrity of Kubernetes binaries (kube-apiserver, kubelet, kubectl, kube-proxy) using **checksums**
* Understand the risks of running unverified binaries or images.
* Validate container images before deployment using **SHA256 digests**.
* Identify tampered binaries or images before deployment in production clusters.

**Prerequisites**

* Kubernetes cluster with kubectl access.
* Linux workstation or control plane node with internet access.
* curl, sha256sum, gpg, tar installed.
* docker or podman installed for image verification.

**Step 0: Verify Cluster Access**

alias k=kubectl

k get nodes -o wide

✅ **Expected Output:**  
All nodes show STATUS=Ready.

**Step 1: Download Official Kubernetes Binaries**

1. Go to the official Kubernetes release page:

<https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.28.md#server-binaries>

mkdir ~/k8s-binaries && cd ~/k8s-binaries

1. Download the binaries:

wget <https://dl.k8s.io/v1.28.15/kubernetes-server-linux-amd64.tar.gz>

tar -xvzf kubernetes-server-linux-amd64.tar.gz

**Step 2: Verify SHA512 Checksums**

1. Verify binary:

sha512sum

✅ **Expected Output:**

sha512sum kubernetes-server-linux-amd64.tar.gz

Verify the checksum with the one mentioned on the release page

⚠️ If it mismatches , the binary may be tampered — **do not use it in production**.

**Step 4: Verify Container Images : eg : api-server image**

Container images for control plane components :

Run below to confirm if ur using right image

POD\_NAME=$(sudo kubectl get pods -n kube-system -l component=kube-apiserver -o jsonpath='{.items[0].metadata.name}')

RUNNING\_DIGEST=$(sudo kubectl get pod $POD\_NAME -n kube-system -o jsonpath='{.status.containerStatuses[0].imageID}')

echo "Running kube-apiserver digest: $RUNNING\_DIGEST"

OR

sudo docker inspect registry.k8s.io/kube-apiserver:v1.33.5

sudo docker pull registry.k8s.io/kube-apiserver:v1.33.5

OFFICIAL\_DIGEST=$(sudo docker inspect registry.k8s.io/kube-apiserver:v1.33.5 --format='{{index .RepoDigests 0}}')

echo "Official kube-apiserver digest: $OFFICIAL\_DIGEST"

sudo docker images –digests

* + Check if official digest is matching running digest ort not

**Step 5: Verify Container Images : eg : scheduler image**

**🧩Check for kube-scheduler Pod**

POD=$(sudo kubectl get pod -n kube-system -l component=kube-scheduler -o jsonpath='{.items[0].metadata.name}')

**🧩 Get Running Digest**

RUN\_DIGEST=$(sudo kubectl get pod $POD -n kube-system -o jsonpath='{.status.containerStatuses[0].imageID}')

echo "Running Digest: $RUN\_DIGEST"

**🧩 Pull Official Image**

sudo docker pull registry.k8s.io/kube-scheduler:v1.33.5

**🧩 Get Official Digest**

OFF\_DIGEST=$(sudo docker inspect registry.k8s.io/kube-scheduler:v1.33.5 --format='{{index .RepoDigests 0}}')

echo "Official Digest: $OFF\_DIGEST"

**🧩 Compare Both**

**🧩 (Optional) List All Image Digests**

sudo docker images --digests

**Step 6: Post-Lab Discussion**

* Running unverified binaries/images can introduce **supply chain attacks**.
* SHA256 checksums prevent accidental or malicious tampering.
* Always enforce **image digest verification** in CI/CD pipelines.

**Lab 5: Apply Role-Based Access Control (RBAC)**

**Lab Title:**

🔐 *Implementing Least-Privilege Access with RBAC in Kubernetes*

**Lab Objective**

By the end of this lab, trainees will be able to:

* Understand and differentiate **Role vs ClusterRole** and **RoleBinding vs ClusterRoleBinding**.
* Apply **least-privilege access** for users, service accounts, and pods.
* Restrict access to critical Kubernetes resources (pods, secrets, configmaps).
* Validate permissions using kubectl auth can-i.
* Harden default service accounts.

**Prerequisites**

* Kubernetes cluster with kubectl access.
* Cluster-admin access for initial setup.
* Namespace rbac-lab for isolation:
* kubectl create ns rbac-lab

**Step 0: Verify Cluster Access**

alias k=kubectl

k get nodes -o wide

k get ns

✅ **Expected Output:**  
All nodes show STATUS=Ready, and rbac-lab namespace exists.

**Step 1: Understand RBAC Concepts**

* **Role:** Grants permissions within a **specific namespace**.
* **ClusterRole:** Grants permissions **cluster-wide** or for cluster-scoped resources.
* **RoleBinding:** Assigns a Role to a user, group, or service account within a namespace.
* **ClusterRoleBinding:** Assigns a ClusterRole to a user, group, or service account cluster-wide.

**Step 2: Create a Limited Role**

Create a Role that allows **read-only access to Pods** in the rbac-lab namespace:

cat <<EOF > pod-reader-role.yaml

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: rbac-lab

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list", "watch"]

EOF

Apply the Role:

kubectl apply -f pod-reader-role.yaml

kubectl get role -n rbac-lab

✅ **Expected Output:** Role pod-reader is created.

**Step 3: Create a Service Account**

Create a service account that will use the Role:

kubectl create sa test-reader -n rbac-lab

kubectl get sa -n rbac-lab

**Step 4: Bind Role to Service Account**

Create a RoleBinding:

cat <<EOF > pod-reader-binding.yaml

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: pod-reader-binding

namespace: rbac-lab

subjects:

- kind: ServiceAccount

name: test-reader

namespace: rbac-lab

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

EOF

Apply the binding:

kubectl apply -f pod-reader-binding.yaml

kubectl get rolebinding -n rbac-lab

**Step 5: Test Permissions Using kubectl auth can-i**

1. Test **allowed action**:

kubectl auth can-i get pods --as=system:serviceaccount:rbac-lab:test-reader -n rbac-lab

✅ **Expected Output:** yes

1. Test **disallowed action**:

kubectl auth can-i delete pods --as=system:serviceaccount:rbac-lab:test-reader -n rbac-lab

✅ **Expected Output:** no

**Step 6: Create a ClusterRole and ClusterRoleBinding (Optional / Advanced)**

Create a ClusterRole that allows reading **all nodes**:

cat <<EOF > node-reader-clusterrole.yaml

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: node-reader

rules:

- apiGroups: [""]

resources: ["nodes"]

verbs: ["get", "list", "watch"]

EOF

kubectl apply -f node-reader-clusterrole.yaml

kubectl get clusterrole node-reader

Bind it to a service account cluster-wide:

cat <<EOF > node-reader-binding.yaml

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: node-reader-binding

subjects:

- kind: ServiceAccount

name: test-reader

namespace: rbac-lab

roleRef:

kind: ClusterRole

name: node-reader

apiGroup: rbac.authorization.k8s.io

EOF

kubectl apply -f node-reader-binding.yaml

Test:

kubectl auth can-i get nodes --as=system:serviceaccount:rbac-lab:test-reader

✅ **Expected Output:** yes

⚠️ Best practice: Only assign **explicit Roles/ClusterRoles** to service accounts.

**Step 8: Clean Up (Optional)**

kubectl delete ns rbac-lab

**Lab Outcome**

By completing this lab, trainees can:

* Understand RBAC hierarchy and differences between Role, ClusterRole, RoleBinding, ClusterRoleBinding.
* Apply **least-privilege access** for service accounts and users.
* Test and validate RBAC policies using kubectl auth can-i.
* Harden default service accounts to reduce attack surface.