# 🧪 Lab: Apply Security Policies in AI Infrastructure

**Goal:** harden a GPU-enabled Kubernetes environment used for AI by applying layered controls: **RBAC**, **pod security**, **network segmentation**, **secrets & TLS**, **resource governance for GPUs**, **admission policies**, and **basic telemetry + alerting**. You'll validate each control with quick tests.

**Prereqs**

- A Kubernetes cluster (v1.25+), `kubectl`, and cluster-admin access
- At least 1 GPU node with NVIDIA drivers + **NVIDIA device plugin** (for GPU tests)
- A CNI that supports **NetworkPolicy** (e.g., Calico/Cilium)
- Optional: Gatekeeper (OPA) and Prometheus/Grafana stack

## 0) Create secure namespaces & labels (Pod Security Standards)

```
kubectl create namespace team-a
kubectl create namespace team-b
```

Apply **restricted** Pod Security Standards to both (enforce + warn):

```
kubectl label ns team-a \
  pod-security.kubernetes.io/enforce=restricted \
  pod-security.kubernetes.io/warn=restricted \
  --overwrite
kubectl label ns team-b \
  pod-security.kubernetes.io/enforce=restricted \
  pod-security.kubernetes.io/warn=restricted \
  --overwrite
```

**Why:** The “restricted” profile blocks privileged containers, root users, host mounts, etc. This is your foundation against container breakout.

---

## 1) RBAC: least-privilege roles per team

Create a **Role** for data scientists in `team-a` that lets them manage common workload objects but **not** cluster-wide objects or secrets:

```
kubectl -n team-a apply -f - <<'YAML'
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: ds-contributor
rules:
- apiGroups: [""]               # core
  resources: ["pods","pods/log","services","configmaps"]
  verbs: ["get","list","watch","create","update","delete"]
- apiGroups: ["batch"]          # jobs/cronjobs
  resources: ["jobs","cronjobs"]
  verbs: ["get","list","watch","create","update","delete"]
- apiGroups: ["apps"]           # deployments
  resources: ["deployments","replicasets"]
  verbs: ["get","list","watch","create","update","delete"]
YAML
```

Bind it to a **service account** your users or CI will use:

```
kubectl -n team-a create serviceaccount ds-runner
kubectl -n team-a apply -f - <<'YAML'
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-ds-runner
subjects:
```

```
- kind: ServiceAccount
  name: ds-runner
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: ds-contributor
YAML
```

**Why:** Principle of least privilege: team members can deploy and operate their workloads within their namespace—but cannot touch cluster-scoped resources or secrets.

## 2) Resource governance: GPU quotas & sane defaults

Limit how many GPUs `team-a` can consume, and set default CPU/RAM limits:

```
kubectl -n team-a apply -f - <<'YAML'
apiVersion: v1
kind: ResourceQuota
metadata:
  name: rq-gpu-and-objects
spec:
  hard:
    # at most 2 GPUs in this namespace
    nvidia.com/gpu: "2"
    pods: "50"
    services: "20"
    configmaps: "50"
---
apiVersion: v1
kind: LimitRange
metadata:
  name: defaults
spec:
  limits:
  - type: Container
    default:
      cpu: "1"
      memory: "2Gi"
    defaultRequest:
      cpu: "250m"
      memory: "512Mi"
YAML
```

**Why:** Prevents a single team from monopolizing GPUs and enforces predictable resource requests/limits for scheduling and cost control.

---

## 3) NetworkPolicies: default-deny, allow only what’s needed

Start with **default deny** for ingress/egress in `team-a`:

```
kubectl -n team-a apply -f - <<'YAML'
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes: ["Ingress","Egress"]
YAML
```

Allow **intra-namespace** traffic and DNS egress:

```
kubectl -n team-a apply -f - <<'YAML'
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-same-namespace-and-dns
spec:
  podSelector: {}
  policyTypes: ["Ingress","Egress"]
  ingress:
  - from:
    - podSelector: {}  # any pod in same namespace
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: kube-system
      podSelector:
        matchLabels:
          k8s-app: kube-dns
    ports:
    - protocol: UDP
      port: 53
YAML
```

(Optionally) allow egress to your object storage / model registry by CIDR/FQDN (Calico supports FQDN policies; otherwise use CIDR).

**Why:** Segments workloads by namespace, blocks lateral movement, and preserves only necessary egress (DNS + storage).

---

## 4) Secrets & TLS: create a TLS Secret and mount it

Generate a self-signed cert (dev/demo):

```
openssl req -x509 -nodes -newkey rsa:2048 -days 365 \
  -keyout key.pem -out cert.pem -subj "/CN=triton.team-a.svc"
kubectl -n team-a create secret tls triton-tls --key key.pem --cert cert.pem
```

**Why:** Encrypt service traffic (TLS in transit). In production, use cert-manager + an internal CA.

---

## 5) Secure workload spec: non-root, read-only FS, seccomp, drop caps

Deploy a sample GPU job (CUDA vector add) **safely**:

```
kubectl -n team-a apply -f - <<'YAML'
apiVersion: batch/v1
kind: Job
metadata:
  name: cuda-secure-job
spec:
  backoffLimit: 0
  template:
    spec:
      serviceAccountName: ds-runner
      restartPolicy: Never
      containers:
      - name: cuda
        image: nvcr.io/nvidia/k8s/cuda-sample:vectoradd-cuda11.7
        command: ["bash","-lc","/vectorAdd"]
        resources:
          limits:
            nvidia.com/gpu: 1
        securityContext:
          allowPrivilegeEscalation: false
          readOnlyRootFilesystem: true
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop: ["ALL"]
          seccompProfile:
            type: RuntimeDefault
YAML
```

**Why:** This enforces non-root, drops Linux capabilities, enables seccomp, and uses a read-only root FS—key controls the **restricted** profile expects.

## 6) Admission control (optional but powerful): require non-root & approved registries

If you run **Gatekeeper (OPA)**, install a simple constraint that rejects pods not running as non-root and anything not from approved registries (e.g., `nvcr.io`, your private registry).

**ConstraintTemplate:**

```
kubectl apply -f - <<'YAML'
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8sapprovedimages
spec:
  crd:
    spec:
      names:
        kind: K8sApprovedImages
      validation:
        openAPIV3Schema:
          properties:
            repositories:
              type: array
              items:
                type: string
  targets:
  - target: admission.k8s.gatekeeper.sh
    rego: |
      package k8sapprovedimages
      violation[{"msg": msg}] {
        input.review.kind.kind == "Pod"
        some i
        img := input.review.object.spec.containers[i].image
        not startswith(img, approved[_])
        msg := sprintf("image %v is not in an approved registry", [img])
      }
      approved := {r | r := input.parameters.repositories[_]}
YAML
```

**Constraint (allow only nvcr.io and your-registry.example.com):**

```
kubectl apply -f - <<'YAML'
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sApprovedImages
```

```
metadata:
  name: only-approved-registries
spec:
  parameters:
    repositories:
    - "nvcr.io/"
    - "your-registry.example.com/"
YAML
```

**Why:** Admission policies prevent risky images/configs from ever landing in the cluster. (You can add a second constraint to enforce `runAsNonRoot: true` as well.)

---

## 7) Observability for security: basic GPU telemetry + alert

If you already have Prometheus, deploy **DCGM Exporter** (telemetry for GPUs). Then add a simple alert rule (pseudo-CRD) to detect a **sustained 100% GPU util** (possible crypto-mining/anomaly) or **high temperature**.

**Example PrometheusRule (kube-prometheus-stack):**

```
kubectl -n monitoring apply -f - <<'YAML'
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: gpu-security-alerts
spec:
  groups:
  - name: gpu-anomalies
    rules:
    - alert: GpuUtilizationAnomaly
      expr: avg_over_time(DCGM_FI_DEV_GPU_UTIL[5m]) > 95
      for: 10m
      labels: { severity: warning }
      annotations:
        summary: "High sustained GPU utilization"
        description: "Possible runaway job or unauthorized workload."
    - alert: GpuOverTemp
```

```
      expr: max_over_time(DCGM_FI_DEV_GPU_TEMP[5m]) > 85
      for: 5m
      labels: { severity: critical }
      annotations:
        summary: "GPU temperature critical"
        description: "Check cooling, throttling, or abusive workload."
YAML
```

**Why:** Security is also detection. Telemetry + alerts surface abuse or misconfigurations quickly.

---

## 8) Validation tests (prove each control works)

1. **RBAC:**
   Try to list secrets with the `ds-runner` SA (should fail):

   ```
   kubectl -n team-a auth can-i list secrets --as=system:serviceaccount:team
   ```

   Expect: `no`.

2. **GPU quota:**
   Submit 3 jobs each requesting 1 GPU; the 3rd should **remain Pending** due to `nvidia.com/gpu: 2` quota.

3. **Pod Security (restricted):**
   Attempt a privileged pod (should be **denied**):

   ```
   kubectl -n team-a apply -f - <<'YAML'
   apiVersion: v1
   kind: Pod
   metadata: { name: naughty }
   spec:
     containers:
     - name: c
       image: busybox
   ```

```
    command: ["sh","-c","sleep 3600"]
    securityContext:
      privileged: true
```

Yaml

Expect: admission error referencing restricted policy.

4. **NetworkPolicy:**
   From a `team-b` pod, try to curl a `team-a` service (should **time out**). Within `team-a`, pods should communicate normally; DNS should work.

5. **Admission (if Gatekeeper enabled):**
   Try to deploy a pod using `docker.io/library/ubuntu`. Expect rejection: “image … not in an approved registry”.

6. **TLS:**
   Mount `triton-tls` in a service that terminates TLS (e.g., an NGINX sidecar) and confirm HTTPS endpoint. (In production, use cert-manager and a proper ingress with TLS.)

7. **Telemetry alerts:**
   Temporarily run a high-utilization job and verify **GpuUtilizationAnomaly** fires in Alertmanager/Grafana (then silence/resolve).

---

## 9) (Optional) Extra hardening

- **Image scanning:** integrate Trivy in CI; block images with critical CVEs via Gatekeeper policy.
- **Secrets management:** use Sealed Secrets or an external KMS (AWS KMS/HashiCorp Vault) for key custody.
- **Node isolation:** taint GPU nodes and tolerate only GPU workloads; add **nodeRestriction** and disable SSH where possible.

- **mTLS service mesh:** use Linkerd/Istio to enforce mTLS for east-west traffic automatically.

---

## 10) Cleanup

```
kubectl delete ns team-a team-b
# If you installed Gatekeeper constraints or Prom rules, delete them from thei
```

---

### ✅ What you achieved

- Enforced **restricted** pod security across namespaces
- Implemented **RBAC** with a least-privilege role and service account
- Capped GPU usage via **ResourceQuota** and set default **LimitRange**
- Segmented traffic with **NetworkPolicies** (default-deny + DNS + allow-list)
- Secured workloads: **non-root**, **read-only**, **seccomp**, **no caps**
- Added **admission control** to block non-approved registries (and, optionally, unsafe specs)
- Wired **GPU telemetry alerts** to spot misuse or overheating