**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.

**Lab 6: Secure ServiceAccounts in Kubernetes**

**🎯 Objective**

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

* Understand how **ServiceAccounts** work in Kubernetes.
* Prevent automatic token mounting in Pods (automountServiceAccountToken: false).
* Create and bind minimal-permission Roles to ServiceAccounts.
* Validate access control and demonstrate the effect of disabling token mounts.
* Comprehend security implications of default ServiceAccounts.

**⚙️ 1. Background Concepts**

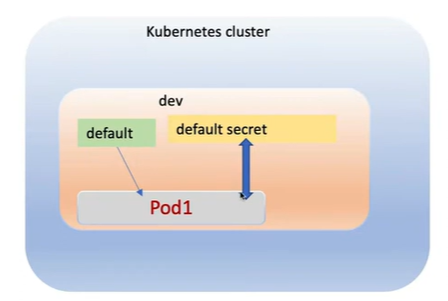
**🔹 What is a ServiceAccount?**

In Kubernetes, **ServiceAccounts (SAs)** are used by Pods or controllers (like Deployments, DaemonSets, Jobs) to authenticate to the Kubernetes API.

Each Pod runs as a particular ServiceAccount.  
If none is specified, it automatically uses the **default ServiceAccount** in its namespace.

By default:

* Each namespace has a default ServiceAccount created automatically.
* When a Pod starts, a **token secret** is automatically mounted at /run/secrets/kubernetes.io/serviceaccount/token.
* This token gives API access to that namespace — **potential security risk** if an attacker compromises the Pod.



**⚠️ Security Concern:**  
If you compromise a Pod, you automatically gain access to the cluster API using its token unless you disable automounting or restrict permissions.

**🧩 2. Lab Setup**

**✅ Prerequisites**

| **Requirement** | **Description** |
| --- | --- |
| Kubernetes Cluster | Working cluster with kubectl access |
| Namespace | We will use dev namespace |
| Access | You have cluster-admin rights |
| CLI | kubectl and optionally curl or wget inside Pod |

**🧱 3. Create Namespace**

We’ll isolate our experiment in a custom namespace called dev.

kubectl create namespace dev

Verify:

kubectl get ns

Expected output:

NAME STATUS AGE

default Active 10d

dev Active 5s

**🧰 4. Create a Custom ServiceAccount**

We will now create a ServiceAccount that **does not automatically mount its API token** to any Pod.

**🔧 Step 1: Create YAML file svc-account.yaml**

apiVersion: v1

kind: ServiceAccount

metadata:

name: test-svc

namespace: dev

automountServiceAccountToken: false

**🔧 Step 2: Apply it**

kubectl apply -f svc-account.yaml

**🔧 Step 3: Verify**

kubectl get sa -n dev

kubectl describe sa test-svc -n dev

Expected:

Name: test-svc

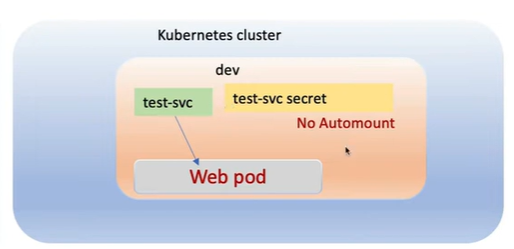
Namespace: dev

Mountable secrets: <none>

Tokens: <none>

automountServiceAccountToken: false

✅ This means **no token secret** will be automatically created or mounted.



**🧱 5. Create a Role with Limited Permissions**

We’ll now create a Role with minimal privileges for the dev namespace.

**🔧 Step 1: Create Role**

kubectl create role test-role -n dev \

--verb=get,list,watch,create \

--resource=pods,deployments,services,configmaps

This command:

* Creates a Role named test-role in namespace dev
* Grants only the verbs: get, list, watch, and create
* On limited resource types: pods, deployments, services, configmaps

**🔧 Step 2: Verify Role**

kubectl describe role test-role -n dev

Expected:

Rules:

Resources Verbs

--------- -----

pods, deployments, services, configmaps get, list, watch, create

**🪢 6. Create a RoleBinding**

Now we bind the test-svc ServiceAccount to the test-role.

**🔧 Step 1: Create RoleBinding**

kubectl create rolebinding cluster-test-binding -n dev \

--role=test-role \

--serviceaccount=dev:test-svc

**🔧 Step 2: Verify**

kubectl describe rolebinding cluster-test-binding -n dev

Expected:

RoleRef:

Kind: Role

Name: test-role

Subjects:

Kind: ServiceAccount

Name: test-svc

Namespace: dev

✅ The ServiceAccount now has minimal, namespace-scoped permissions.

**🚀 7. Create a Pod Using the Custom ServiceAccount**

**🔧 Step 1: Pod Manifest**

Save as web-pod.yaml:

apiVersion: v1

kind: Pod

metadata:

name: web

namespace: dev

spec:

containers:

- name: web

image: nginx

command: [ "sleep", "3600" ]

serviceAccountName: test-svc

automountServiceAccountToken: false

**🔧 Step 2: Deploy**

kubectl apply -f web-pod.yaml

**🔧 Step 3: Verify**

kubectl get pod -n dev

When it’s running:

kubectl describe pod web -n dev

Look for:

Service Account: test-svc

AutomountServiceAccountToken: false

✅ Confirmed — the Pod is running without an auto-mounted token.

**🔍 8. Validate Token Mounting Behavior**

**🔧 Step 1: Inspect Pod filesystem**

kubectl exec -it web -n dev -- ls /run/secrets/kubernetes.io/serviceaccount/

Expected output:

ls: cannot access '/run/secrets/kubernetes.io/serviceaccount/': No such file or directory

✅ Confirmed — the Pod does **not** have a token mounted (good security posture).

**🧠 9. Contrast with Default Behavior**

**🔧 Step 1: Run a Pod with default ServiceAccount**

kubectl run default-test -n dev --image=nginx --restart=Never -- sleep 3600

**🔧 Step 2: Check mounted secrets**

kubectl exec -it default-test -n dev -- ls /run/secrets/kubernetes.io/serviceaccount/

Expected output:

ca.crt

namespace

token

✅ Default SA always mounts a token unless explicitly disabled.

**🧪 10. Demonstration: Attempt Unauthorized Access**

**🔧 Step 1: Try using serviceaccount token (default behavior)**

kubectl exec -it default-test -n dev -- sh

Inside Pod:

TOKEN=$(cat /run/secrets/kubernetes.io/serviceaccount/token)

curl -k https://kubernetes.default/api/v1/namespaces/kube-system/secrets \

-H "Authorization: Bearer $TOKEN"

Expected:

* You may get **403 Forbidden** or a **partial list** depending on default permissions.
* The token gives **namespace-scoped access** — hence still a potential attack vector.

**🔧 Step 2: Try same with secured Pod**

kubectl exec -it web -n dev -- sh

Then run:

curl -k https://kubernetes.default/api/v1/namespaces/kube-system/secrets

Expected:

curl: (6) Could not resolve host or (7) Failed to connect

Because there’s **no token** available → the API server denies access.

✅ Confirmed — disabling automount effectively prevents unauthorized API calls from within Pods.

**🧰 11. Override at Pod Level (optional)**

You can override the ServiceAccount’s default automount setting at **Pod level**.

Even if ServiceAccount has automountServiceAccountToken: false,  
the Pod can explicitly re-enable it.

Example:

spec:

serviceAccountName: test-svc

automountServiceAccountToken: true

This **forces** token mount for that Pod — useful for debugging but should be avoided in production.

**🔒 12. Security Best Practices Summary**

| **Best Practice** | **Description** |
| --- | --- |
| Disable default ServiceAccount | Prevent accidental use of default SA in namespaces |
| Disable token automount | Use automountServiceAccountToken: false at SA and Pod level |
| Use RBAC least privilege | Create minimal Role/RoleBinding for each SA |
| Avoid broad verbs | Restrict verbs to get, list, watch only |
| Use namespaces wisely | Isolate apps into different namespaces |
| Regular audit | Use kubectl get rolebinding --all-namespaces to review bindings |
| Rotate tokens | Regularly delete/recreate ServiceAccounts for token refresh |

**🧾 13. Cleanup**

After completing the lab:

kubectl delete ns dev

**📚 14. References**

* 🔗 [Kubernetes Official Docs – Configure Service Accounts](https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/)
* 🔗 Kubernetes API Access and Tokens
* 🔗 CIS Kubernetes Benchmark Controls – Section 5.1.x

**🧭 15. Summary of What You Learned**

| **Concept** | **Key Takeaway** |
| --- | --- |
| ServiceAccount | Used by Pods for API authentication |
| Default behavior | Token mounted automatically — insecure |
| Custom SA | automountServiceAccountToken: false prevents token mount |
| Role/RoleBinding | Limit scope and verbs for least privilege |
| Verification | Check token absence in Pod filesystem |
| Security validation | Attempting API calls fails if token is absent |

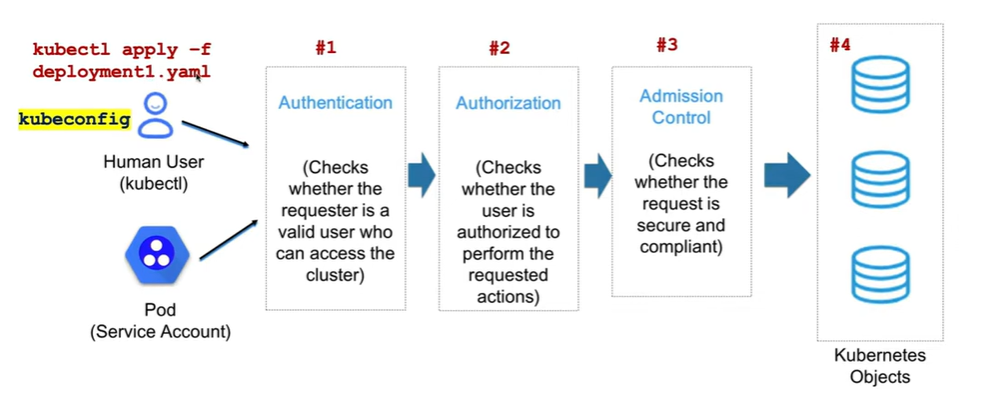
**Lab 7: Restrict Access to Kubernetes API**

**Objective**

1. Secure access to kube-apiserver.
2. Restrict anonymous requests.
3. Validate access control using kubectl auth can-i.

**Prerequisites**

* A working Kubernetes cluster (kubeadm, K3s, or similar).
* kubectl installed and configured.
* Access to the control plane node(s).

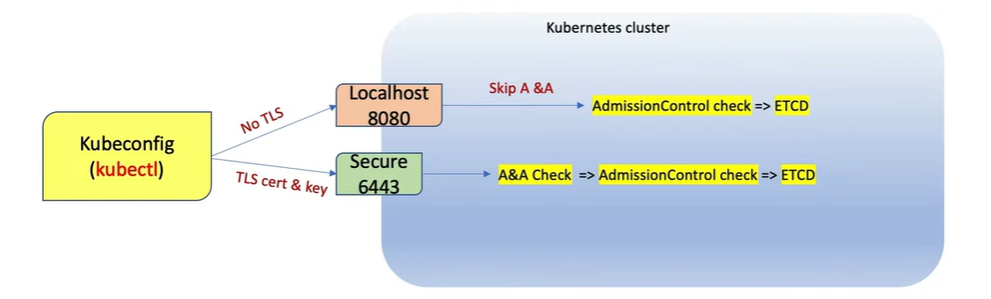
****

**Step 1: Understanding kube-apiserver Ports**

The kube-apiserver exposes two ports:

| **Port** | **Description** | **Default** |
| --- | --- | --- |
| 8080 | Insecure HTTP | localhost only, not TLS |
| 6443 | Secure HTTPS | TLS-enabled, default for cluster access |

**Goal:** disable the insecure port, ensure secure port is enabled, and configure TLS.



**1.1 Check current kube-apiserver configuration**

If using kubeadm:

ps -ef | grep kube-apiserver

You may see something like:

/usr/bin/kube-apiserver --advertise-address=192.168.1.10 --allow-privileged=true --secure-port=6443 --insecure-port=8080 ...

Take note of --insecure-port and --secure-port.

**1.2 Disable Insecure API Port**

Edit the kube-apiserver manifest:

sudo vi /etc/kubernetes/manifests/kube-apiserver.yaml

Find:

- --insecure-port=8080

Change to:

- --insecure-port=0

0 disables the insecure port entirely.

**1.3**

**Configure TLS Certificates**

Check or set TLS certificates:

- --tls-cert-file=/etc/kubernetes/pki/apiserver.crt

- --tls-private-key-file=/etc/kubernetes/pki/apiserver.key

* These ensure HTTPS access on the secure port (6443 by default).
* If using kubeadm, these are generated automatically at cluster init (/etc/kubernetes/pki).

**1.4 Verify Secure API Port**

After changes, kubelet will automatically restart kube-apiserver (as it’s a static Pod).

Verify:

netstat -tulnp | grep 6443

curl -k https://127.0.0.1:6443/version

* Should return the Kubernetes API version.
* Insecure port (8080) should no longer be listening.

**Step 2: Control Anonymous Access**

**2.1 Check if anonymous access is allowed**

By default, kube-apiserver may allow anonymous requests.

ps -ef | grep kube-apiserver | grep anonymous-auth

* If --anonymous-auth=true, anonymous requests are allowed.
* **Goal:** disable anonymous requests.

**2.2 Disable Anonymous Access**

In /etc/kubernetes/manifests/kube-apiserver.yaml, find:

- --anonymous-auth=true

Change to:

- --anonymous-auth=false

Kube-apiserver will restart automatically after saving the manifest.

**Step 3: Verify Access Restrictions**

**3.1 Verify Anonymous Access**

kubectl auth can-i --list --as=system:anonymous -n default

* Should show **very limited or no permissions**, depending on your ClusterRoleBinding.

**3.2 Verify Your Own Access**

kubectl auth can-i --list

* Should show full access according to your ServiceAccount / RBAC roles.

**3.3 Verify API Access via curl**

# As anonymous

curl -k https://127.0.0.1:6443/api/v1/namespaces/default/pods

# Should get 401 Unauthorized

# As authenticated user

# Corrected command

sudo curl -k https://127.0.0.1:6443/api/v1/namespaces/default/pods \

--cert /etc/kubernetes/pki/apiserver-kubelet-client.crt \

--key /etc/kubernetes/pki/apiserver-kubelet-client.key

# Should succeed

**Step 4: Best Practices**

1. **Always use the secure port (6443) for API access.**
2. **Disable insecure port (8080)** unless strictly required for local debugging.
3. **Disable anonymous-auth** unless you intentionally want limited anonymous access.
4. **Use RBAC for controlled API access**, including for anonymous or service accounts.
5. **Audit API access** regularly:
6. kubectl get events --all-namespaces
7. kubectl logs kube-apiserver -n kube-system

**Step 5: Lab Notes**

* All kube-apiserver options are configured in /etc/kubernetes/manifests/kube-apiserver.yaml for kubeadm.
* Changes take effect automatically because kubelet monitors static Pods.
* You can always check current flags:
* ps -ef | grep kube-apiserver
* Anonymous access is **disabled by default** in hardened clusters; explicitly create roles if required.

**References**

1. [API server ports and IPs](https://kubernetes.io/docs/concepts/security/controlling-access/#api-server-ports-and-ips)
2. [Anonymous requests](https://kubernetes.io/docs/reference/access-authn-authz/authentication/#anonymous-requests)
3. [Access the cluster without kubectl proxy](https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/#without-kubectl-proxy)
4. [Kubelet authentication/authorization](https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-authentication-authorization/#kubelet-authentication)

**LAB 8: Minimize Host OS Footprint (Reduce Attack Surface)**

**🎯 Objective**

This lab teaches you how to:

* Reduce exposure of the host operating system to pods and users.
* Limit Kubernetes pod privileges.
* Harden SSH access.
* Control kernel modules and running services.
* Configure a host firewall (UFW).
* Identify and eliminate unnecessary software and open ports.

**⚙️ Lab Setup**

| **Component** | **Description** |
| --- | --- |
| OS | Ubuntu 20.04+ (control plane or node) |
| Cluster | Kubernetes (kubeadm or minikube setup) |
| Privileges | Root or sudo access |
| Tools | kubectl, curl, netstat, systemctl, ufw |

**🔹 1. Host Namespace Exposure Control (Kubernetes Side)**

**1.1 Create an *unsafe* pod using host namespaces**

Let’s intentionally create a pod that uses **host network, PID, and IPC namespaces**.

apiVersion: v1

kind: Pod

metadata:

name: ubuntu-ps-demo

spec:

hostNetwork: true

hostPID: true

containers:

- name: ubuntu-container

image: ubuntu:22.04

command: ["sleep", "3600"]

**Apply the manifest:**

kubectl apply -f host-namespace-demo.yaml

**Verify:**

k exec -it ubuntu-ps-demo -- ps -ef | head

You’ll notice you can see **host processes**, not just container ones.

**1.2 Create a *secure* version : by default its false .**

**🔹 2. Privilege Restriction in Containers**

**2.1 Example of *insecure* container (privileged mode)**

apiVersion: v1

kind: Pod

metadata:

name: ubuntu-privileged

spec:

containers:

- name: ubuntu

image: ubuntu:22.04

command: ["sleep", "3600"]

securityContext:

privileged: true

**Apply & verify:**

kubectl apply -f privileged-pod.yaml

kubectl exec -it ubuntu-privileged -- ls -l /dev

ls -l /dev

You might be able to access physical devices list — extremely risky.

**2.2 Secure alternative (non-privileged + restricted runAs)**

apiVersion: v1

kind: Pod

metadata:

name: unpreviliged

spec:

securityContext:

runAsUser: 1000

runAsGroup: 1000

containers:

- name: ubuntu

image: ubuntu:22.04

command: ["sleep", "3600"]

securityContext:

privileged: false

**Apply:**

kubectl apply -f secure-pod.yaml

**Verify user context inside pod:**

kubectl exec -it unpreviliged -- ls -l /dev

kubectl exec -it unpreviliged -- id

✅ **Result:** Pod runs as non-root, non-privileged user.

**🔹 3. Limit Node Access (OS User Hardening)**

**3.1 Delete obsolete users and groups**

sudo userdel user1

sudo groupdel group1

**3.2 Disable (suspend) existing users**

sudo usermod -s /usr/sbin/nologin user2

**3.3 Create a new limited user**

sudo useradd -d /opt/sam -s /bin/bash -G admin -u 2328 sam

**Explanation:**

* -d sets home directory.
* -s sets login shell.
* -G assigns to admin group.
* -u assigns static UID.

Verify:

id sam

grep sam /etc/passwd

✅ **Result:** Controlled, minimal user access.

**🔹 4. Remove Obsolete/Unnecessary Software**

**4.1 List all installed packages**

apt list --installed

**4.2 Identify active services**

systemctl list-units --type=service

**4.3 Disable and remove unnecessary services**

Example with squid:

sudo systemctl stop squid

sudo systemctl disable squid

sudo apt remove squid -y

✅ **Result:** Reduced attack surface by minimizing unnecessary processes.

**🔹 5. SSH Hardening**

**5.1 Generate SSH key pair for secure login**

ssh-keygen -t rsa -b 4096

**5.2 Add public key to authorized keys**

cat ~/.ssh/id\_rsa.pub >> /home/ubuntu/.ssh/authorized\_keys

**5.3 Harden SSH configuration**

Edit /etc/ssh/sshd\_config:

PermitRootLogin no

PasswordAuthentication no

Restart SSH:

sudo systemctl restart sshd

✅ **Result:** Password and root-based logins disabled.  
Only SSH keys work — much more secure.

**🔹 8. Identify & Fix Open Ports / Running Services**

**8.1 Check open ports**

sudo netstat -tulnp

**8.2 Check mapping of services to ports**

cat /etc/services | grep -i ssh

**8.3 Example: Find & remove nginx service**

systemctl list-units --all | grep nginx

sudo systemctl stop nginx

sudo rm /lib/systemd/system/nginx.service

sudo apt remove nginx -y

**8.4 Verify no service is listening on critical ports**

sudo netstat -atnlp | grep -w -i listen

sudo netstat -an | grep 22 | grep -w -i listen

✅ **Result:** Attack surface verified to be minimal — only essential services run.

**🧩 10. Validation Checklist**

| **Area** | **Command** | **Expected Result** |
| --- | --- | --- |
| Namespace sharing | kubectl get pods -o wide | hostPID=false, hostNetwork=false |
| Privilege level | kubectl exec secure-pod -- id | non-root UID |
| SSH | sshd\_config | No root, no password |
| Open ports | netstat -tulnp | Only 22, 6443, etc. |
| Unused software | apt list --installed | Cleaned |

**🔒 11. Security Rationale**

Each step contributes to **reducing the host’s attack surface**:

| **Step** | **Threat Mitigated** |
| --- | --- |
| Disable host namespaces | Prevent container-to-host escape |
| Disable privileged mode | Stop root-level container exploits |
| User pruning | Prevent dormant accounts exploitation |
| SSH hardening | Stop brute-force or root login |
| Remove unused packages | Reduce vulnerability exposure |

**🧾 References**

1. CIS Kubernetes Benchmark v1.9: Section 4 — Minimize host OS footprint