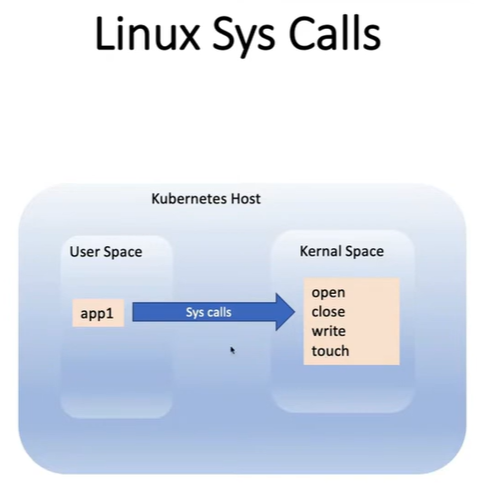
**Lab 9: AppArmor Security Profiles in Kubernetes**

**Lab Objectives**

1. Understand AppArmor as a **Linux kernel security module** for granular program access control.
2. Create and load a **custom AppArmor profile** in a Kubernetes node.
3. Apply the profile to a **pod**.
4. Verify profile enforcement and monitor status.

**Prerequisites**

* Linux nodes (Ubuntu/Debian recommended) running Kubernetes worker nodes.
* Root access to nodes.
* kubectl configured to interact with the cluster.
* Optional: SSH/scp access between nodes for profile distribution.

****

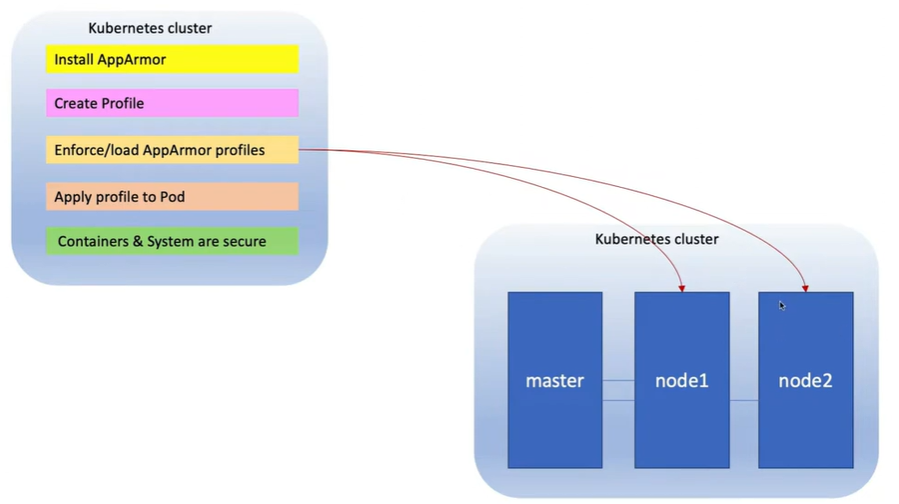
**Module 1: Understand AppArmor**

* **AppArmor**: Kernel Security Module that restricts program capabilities (file read/write, network, capabilities).
* Profiles define **allowed or denied operations**.
* Two modes:
  1. **Complain Mode**: Logs violations but does not block.
  2. **Enforce Mode**: Actively blocks operations violating profile rules.

**Key file locations**:

| **Component** | **Path** |
| --- | --- |
| Profiles directory | /etc/apparmor.d/ |
| Status | /sys/kernel/security/apparmor/profiles |
| Module parameters | /sys/module/apparmor/parameters/enabled |

**Module 2: Install AppArmor Utilities**

****

# Install AppArmor and utilities

sudo apt-get update

sudo apt-get install -y apparmor apparmor-utils

# Verify module loaded

sudo systemctl status apparmor

cat /sys/module/apparmor/parameters/enabled

* **aa-status**: Lists loaded profiles, complain/enforce status, processes confined.

aa-status

**Module 3: Create a Custom AppArmor Profile**

**Step 3.1: Profile Design**

Objective: Deny **all file writes** from a pod container.

**Profile file**: /etc/apparmor.d/deny-write

sudo vi /etc/apparmor.d/deny-write

**Content:**

#include <tunables/global>

profile k8s-apparmor-example-deny-write flags=(attach\_disconnected) {

#include <abstractions/base>

# Allow basic file access

file,

# Deny all writes to files/directories

deny /\*\* w,

}

**Explanation:**

* #include <tunables/global>: Provides global variables and system paths.
* flags=(attach\_disconnected): Ensures the profile can attach to processes not yet connected.
* file,: Allows read access to normal files.
* deny /\*\* w,: Denies write access to all files recursively.

**Step 3.2: Distribute Profile to Nodes**

If you have multiple nodes:

# Repeat for all nodes

Copy to the default AppArmor profile directory to all nodes:

sudo vi /etc/apparmor.d/deny-write

**Step 3.3: Load Profile on all nodes**

sudo apparmor\_parser -r /etc/apparmor.d/deny-write

* -r ensures profile is **reloaded** if it already exists.
* To verify:

cat /sys/kernel/security/apparmor/profiles

* Should list k8s-apparmor-example-deny-write.

**Module 4: Apply Profile to a Kubernetes Pod**

**Pod YAML example**:

apiVersion: v1

kind: Pod

metadata:

name: hello-apparmor

spec:

securityContext:

appArmorProfile:

type: Localhost

localhostProfile: k8s-apparmor-example-deny-write

containers:

- name: hello

image: busybox:1.28

command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]

**Explanation**:

* appArmorProfile:

type: Localhost

Annotation pointing to AppArmor profile.

* localhost/ prefix: Uses **local profile** from the node.
* Container will **inherit enforced AppArmor restrictions**.

Apply the pod:

kubectl apply -f hello-apparmor.yaml

**Step 4.1: Verify Enforcement**

Enter the pod:

kubectl exec -it hello-apparmor -- sh

Try writing a file:

touch /tmp/testfile

# Should fail with "Permission denied"

**Step 4.2: Switch Between Modes**

* **Complain mode**: Logs violations but does not block

sudo aa-complain /etc/apparmor.d/deny-write

* **Enforce mode**: Blocks violations

sudo aa-enforce /etc/apparmor.d/deny-write

* **Unload profile**:

sudo apparmor\_parser -R /etc/apparmor.d/deny-write

**Module 5: Useful AppArmor Commands**

| **Command** | **Purpose** |
| --- | --- |
| systemctl status apparmor | Check AppArmor service status |
| cat /sys/module/apparmor/parameters/enabled | Check kernel module enabled |
| cat /sys/kernel/security/apparmor/profiles | List loaded profiles |
| aa-status | Display loaded profiles and processes |
| apparmor\_parser /etc/apparmor.d/<profile> | Load a profile |
| aa-genprof /path/to/program | Generate a new profile interactively |
| aa-complain /etc/apparmor.d/<profile> | Switch to complain mode |
| aa-enforce /etc/apparmor.d/<profile> | Switch to enforce mode |
| apparmor\_parser -R /etc/apparmor.d/<profile> | Remove/unload profile |

**Module 6: Lab Verification Checklist**

1. AppArmor service running on all nodes.
2. Profile k8s-apparmor-example-deny-write loaded in enforce mode.
3. Pod hello-apparmor created successfully with annotation.
4. Attempt to write inside the container fails.
5. Switching to complain mode allows writes but logs violation in /var/log/syslog.
6. Profile can be unloaded and reloaded without restarting the pod.

**References**

* Kubernetes Tutorial: [AppArmor Security Profiles](https://kubernetes.io/docs/tutorials/clusters/apparmor/)
* Official AppArmor Wiki: [GitLab Documentation](https://gitlab.com/apparmor/apparmor/-/wikis/Documentation)
* Linux Syscalls: [man7.org](https://man7.org/linux/man-pages/man2/syscalls.2.html)

✅ **Learning Outcomes**

1. Understand AppArmor as a **kernel-level security module**.
2. Create custom profiles to **deny/allow operations** in containers.
3. Apply profiles to pods using annotations.
4. Manage profiles dynamically: **load, enforce, complain, unload**.
5. Test and verify restrictions in Kubernetes pods.

**Lab 11: Seccomp Profiles in Kubernetes**

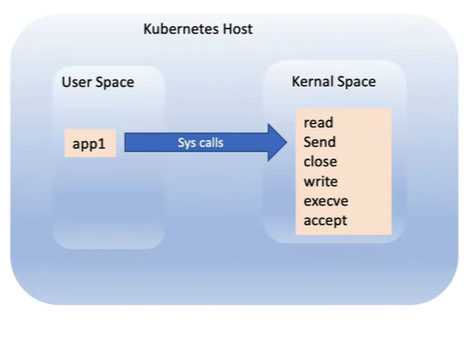
**Lab Objectives**

1. Understand Seccomp and its purpose in **restricting system calls**.
2. Implement **RuntimeDefault and custom Seccomp profiles** in pods.
3. Observe blocked syscalls for multiple examples using strace and dmesg.
4. Experiment with **profiles allowing some syscalls and denying others**.

**Module 1: Seccomp Concept Recap**

**1.1 What is Seccomp?**

* **Seccomp** (Secure Computing Mode) is a **Linux kernel security module**.
* It **filters syscalls** from user-space to kernel, restricting what a process can do.
* Helps implement **least privilege at the syscall level** for containers.



| **Feature** | **AppArmor** | **Seccomp** |
| --- | --- | --- |
| **Purpose** | Restricts what **files, directories, networks, and capabilities** a process can access | Restricts which **system calls** (syscalls) a process can make |
| **Level of control** | File system & resource-level | Kernel syscall-level |
| **Analogy** | “You can’t enter this folder or open that file.” | “You can’t even *ask* the kernel to do that action.” |

**1.3 Seccomp Profile Types**

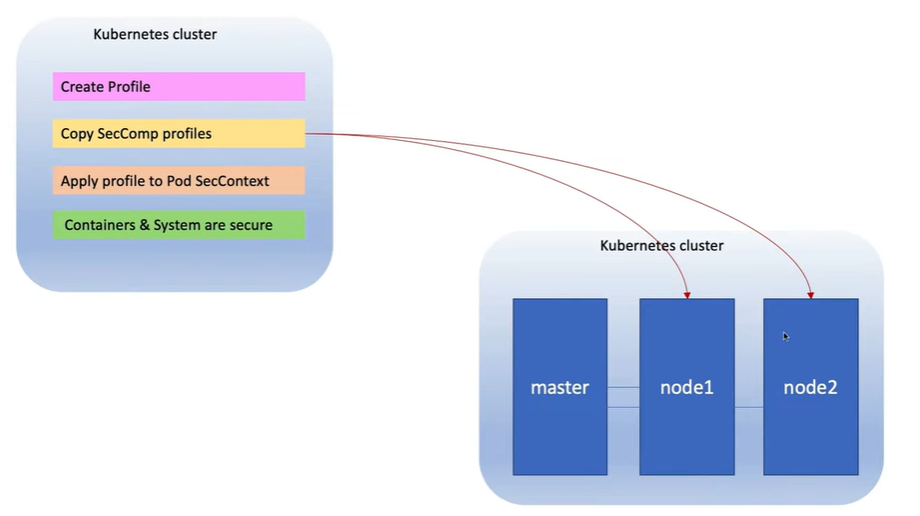
| **Profile Type** | **Purpose** |
| --- | --- |
| Default | RuntimeDefault — basic runtime protection |
| Audit | Logs violations (audit.json) |
| Violation | Blocks restricted syscalls (violation.json) |
| Custom | Fine-grained profile for application-specific needs |

* Default directory for kubelet-local profiles:

/var/lib/kubelet/seccomp/profiles/

**1.4 Seccomp Actions**

| **Action** | **Effect** |
| --- | --- |
| SCMP\_ACT\_ALLOW | Allow syscall |
| SCMP\_ACT\_ERRNO | Block syscall, return errno |
| SCMP\_ACT\_LOG | Log violation |
| defaultAction | Default action for unspecified syscalls |

****

**Module 2: Verify Kernel Support**

grep -i seccomp /boot/config-$(uname -r)

# Expect CONFIG\_SECCOMP=y and CONFIG\_SECCOMP\_FILTER=y

**Module 3: RuntimeDefault Pod Example**

apiVersion: v1

kind: Pod

metadata:

name: runtime-default-pod

spec:

securityContext:

seccompProfile:

type: RuntimeDefault

containers:

- name: test-container

image: busybox

command: ["sh", "-c", "sleep 1h"]

securityContext:

allowPrivilegeEscalation: false

* **Observation:** Most normal syscalls succeed; dangerous syscalls may be blocked.

**Module 4: Custom Seccomp Profiles with Multiple Syscalls**

**sudo mkdir -p /var/lib/kubelet/seccomp/profiles**

**4.1 Profile 1: Deny Dangerous Syscalls, Allow Basics**

File: /var/lib/kubelet/seccomp/profiles/deny-dangerous.json

ramanvm@raman-worker:~$ sudo cat /var/lib/kubelet/seccomp/profiles/deny-dangerous.json

{

"defaultAction": "SCMP\_ACT\_ALLOW",

"syscalls": [

{ "names": ["read", "write", "exit", "sigreturn", "close", "execve", "unlink", "open", "openat"], "action": "SCMP\_ACT\_ALLOW" },

{ "names": ["mkdir", "rmdir"], "action": "SCMP\_ACT\_ERRNO" }

]

}

RELOAD SECCOMP PROFILE :

sudo systemctl restart kubelet

**4.2 Apply Profile in Pod**

ramanvm@raman-master:~$ cat test.yaml

apiVersion: v1

kind: Pod

metadata:

name: seccomp-demo

spec:

securityContext:

seccompProfile:

type: Localhost

localhostProfile: profiles/deny-dangerous.json

containers:

- name: tester

image: busybox

command: ["sh", "-c", "sleep 1h"]

Apply pod:

kubectl apply -f deny-dangerous-pod.yaml

**4.3 Test Multiple Syscalls**

kubectl exec -it deny-dangerous-pod -- sh

| **Syscall** | **Command** | **Expected Result** |
| --- | --- | --- |
| mkdir | mkdir /tmp/test | Blocked (Operation not permitted) |
| rmdir | rmdir /tmp/test | Blocked |
| read/write | echo "hi" > /tmp/file; cat /tmp/file | Allowed |
| close | Implicit via file operations | Allowed |

* Go to tail -f /var/log/syslog on nodes to see logs:

**Lab 12: Using Appropriate Pod Security Standards in Kubernetes**

**Lab Objectives**

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

1. Understand **Pod Security Standards (PSS)** and why they matter.
2. Implement **Pod Security Admission (PSA)** using standards: Privileged, Baseline, Restricted.
3. Apply **namespace-level enforcement**.
4. Configure **pod-level overrides and annotations**.
5. Verify **pod security compliance**.
6. Observe **denied pods** due to policy violations.

**Prerequisites**

* Kubernetes cluster (v1.25+)
* kubectl configured
* Cluster-admin privileges to apply namespace policies
* Optional: Kubernetes Dashboard for visualization

**Module 1: Pod Security Standards Overview**

Kubernetes defines three **Pod Security Standards (PSS)** to enforce security best practices:

| **Standard** | **Description** |
| --- | --- |
| Privileged | Allows all capabilities, security contexts; unsafe for production. |
| Baseline | Enforces minimal restrictions; prevents known privilege escalations, avoids root containers, allows common workloads. |
| Restricted | Most secure; restricts host access, disallows privileged containers, disallows host networking/mounts. |

**Key security attributes controlled**:

* runAsNonRoot
* runAsUser
* allowPrivilegeEscalation
* capabilities
* hostNetwork, hostPID, hostIPC
* readOnlyRootFilesystem
* Seccomp and AppArmor profiles
* Volume types (hostPath restricted)

**Module 2: Enable Pod Security Admission**

Kubernetes 1.25+ replaces Pod Security Policies with **Pod Security Admission (PSA)**.

**2.1 Set PSA at Namespace Level**

kubectl create namespace secure-namespace

kubectl label --overwrite ns secure-namespace pod-security.kubernetes.io/enforce=restricted

kubectl label --overwrite ns secure-namespace pod-security.kubernetes.io/enforce-version=latest

**Explanation**:

* pod-security.kubernetes.io/enforce → sets enforcement level (privileged, baseline, restricted).
* pod-security.kubernetes.io/enforce-version → specifies version (latest recommended).

**2.2 Check Namespace Labels**

kubectl get ns secure-namespace --show-labels

**Module 3: Pod Examples for Each Standard**

**3.1 Privileged Pod (Unsafe)**

apiVersion: v1

kind: Pod

metadata:

name: privileged-pod

namespace: secure-namespace

spec:

containers:

- name: busybox

image: busybox

securityContext:

privileged: true

runAsUser: 0

allowPrivilegeEscalation: true

command: ["sh", "-c", "sleep 1h"]

* Expected Outcome: **Denied** by PSA if namespace is baseline or restricted.

**3.2 Baseline Pod**

apiVersion: v1

kind: Pod

metadata:

name: baseline-pod

namespace: secure-namespace

spec:

containers:

- name: busybox

image: busybox

securityContext:

runAsNonRoot: true

allowPrivilegeEscalation: false

readOnlyRootFilesystem: true

command: ["sh", "-c", "sleep 1h"]

* Should be allowed in **baseline** and **restricted** modes.
* Prevents root execution and privilege escalation.

**3.3 Restricted Pod**

apiVersion: v1

kind: Pod

metadata:

name: restricted-pod

namespace: secure-namespace

spec:

containers:

- name: busybox

image: busybox

securityContext:

runAsNonRoot: true

allowPrivilegeEscalation: false

readOnlyRootFilesystem: true

capabilities:

drop:

- ALL

seccompProfile:

type: RuntimeDefault

command: ["sh", "-c", "sleep 1h"]

* Meets **restricted standard** requirements.
* Dropping all capabilities, using RuntimeDefault Seccomp, read-only root filesystem.
* PSA will **allow this pod**.

**Lab 13: Managing Kubernetes Secrets**

**Lab Objectives**

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

1. Understand the **different types of Kubernetes Secrets**.
2. Create **Opaque, Docker, TLS, SSH, and Basic Auth Secrets**.
3. Inject secrets into pods using **Volumes** and **Environment Variables**.
4. Decode and inspect secret data.
5. Follow **best practices** for secret management in Kubernetes.

**Prerequisites**

* Kubernetes cluster (v1.20+ recommended)
* kubectl configured with cluster-admin privileges
* Access to TLS certificates, Docker registry credentials, or SSH keys if testing those types
* Optional: base64 installed on your client machine

**Module 1: Overview of Kubernetes Secrets**

Kubernetes Secrets are **objects used to store sensitive data** like passwords, tokens, or keys.

**Advantages of Secrets:**

* Avoid hardcoding sensitive information in Pod manifests.
* Can be **mounted as volumes** or **exposed as environment variables**.
* Works with **RBAC** to control access.

**1.1 Types of Secrets**

| **Type** | **Description** | **Example Use** |
| --- | --- | --- |
| **Opaque / Generic** | User-defined key/value pairs | Application credentials |
| **Service Account Token** | Automatically created for pods | Default service account authentication |
| **Docker Config / Registry** | Contains ~/.docker/config.json for image pull | Pull private images |
| **Basic Authentication** | Username/password pairs | HTTP Basic Auth |
| **SSH Authentication** | Private keys for SSH access | Git repositories or remote servers |
| **TLS** | Public/private certificates | HTTPS/TLS connections |
| **Bootstrap Tokens** | Tokens used to join nodes | Kubernetes cluster bootstrapping |

**Module 2: Creating Secrets**

**2.1 Opaque (Generic) Secret**

**Command-line creation:**

kubectl create secret generic mysecret \

--from-literal=username=devuser \

--from-literal=password='S!B\*d$zDsb='

**Verify creation:**

kubectl get secrets

kubectl describe secret mysecret

**Decode secret values:**

kubectl get secret mysecret -o jsonpath="{.data.password}" | base64 --decode

**2.2 Docker Registry Secret**

**Command-line creation:**

kubectl create secret docker-registry my-docker-secret \

--docker-server=DOCKER\_REGISTRY\_SERVER \

--docker-username=DOCKER\_USER \

--docker-password=DOCKER\_PASSWORD \

--docker-email=DOCKER\_EMAIL

**Use case:** Pull images from private registries.

**2.3 TLS Secret**

**Command-line creation:**

kubectl create secret tls tls-secret \

--cert=path/to/tls.cert \

--key=path/to/tls.key

* Contains a **public certificate** and a **private key** for TLS-enabled applications.

**2.4 SSH Authentication Secret**

**Command-line creation:**

kubectl create secret generic ssh-secret \

--from-file=ssh-privatekey=~/.ssh/id\_rsa \

--from-file=ssh-publickey=~/.ssh/id\_rsa.pub

* Can be used for **Git access** or remote login.

**2.5 Basic Authentication Secret**

kubectl create secret generic basic-auth \

--from-literal=username=myuser \

--from-literal=password='mypassword'

**Module 3: Using Secrets in Pods**

Secrets can be **mounted into pods via volumes** or **injected as environment variables**.

**3.1 Mount Secret as Volume**

apiVersion: v1

kind: Pod

metadata:

name: secret-volume-pod

spec:

containers:

- name: app

image: redis

volumeMounts:

- name: secret-volume

mountPath: "/etc/secret-volume"

readOnly: true

volumes:

- name: secret-volume

secret:

secretName: mysecret

**Steps:**

1. Apply pod:

kubectl apply -f secret-volume-pod.yaml

1. Enter pod:

kubectl exec -it secret-volume-pod -- sh

1. Check mounted secrets:

ls /etc/secret-volume

cat /etc/secret-volume/username

cat /etc/secret-volume/password

**3.2 Inject Secret as Environment Variables**

apiVersion: v1

kind: Pod

metadata:

name: secret-env-pod

spec:

containers:

- name: mycontainer

image: redis

envFrom:

- secretRef:

name: mysecret

env:

- name: SECRET\_USERNAME

valueFrom:

secretKeyRef:

name: mysecret

key: username

- name: SECRET\_PASSWORD

valueFrom:

secretKeyRef:

name: mysecret

key: password

restartPolicy: Never

**Steps:**

1. Apply pod:

kubectl apply -f secret-env-pod.yaml

1. Verify environment variables:

kubectl exec -it secret-env-pod -- sh

env | grep SECRET

**Module 4: Best Practices for Managing Secrets**

1. **Use least privilege:** Only give pods access to secrets they need.
2. **Enable encryption at rest:** Configure Kubernetes **EncryptionConfig** for secrets.
3. **Avoid exposing secrets in logs or manifests**.
4. **Rotate secrets regularly** using Kubernetes or external secret managers.
5. **Combine with RBAC:** Restrict secret access to only authorized service accounts.
6. **Use external secret management** for production (e.g., HashiCorp Vault, AWS Secrets Manager, Azure Key Vault).
7. **Read-only mounts:** Always mount secrets read-only in containers.

**Module 5: Verification and Cleanup**

**5.1 Verify Secrets in Cluster**

kubectl get secrets

kubectl describe secret <secret-name>

**5.2 Access Secret Data Inside Pod**

kubectl exec -it <pod-name> -- sh

cat /etc/secret-volume/password

echo $SECRET\_PASSWORD

**5.3 Delete Secrets**

kubectl delete secret mysecret

**Module 6: Optional Advanced Lab Exercises**

1. **Secret Rotation:** Update secret values and verify pods can use the new secret.
2. **TLS Secret:** Mount in an Nginx pod and serve HTTPS content.
3. **Docker Registry Secret:** Pull private images into pods.
4. **SSH Secret:** Mount SSH keys to clone a private Git repository inside the pod.

**References**

* [Kubernetes Secrets](https://kubernetes.io/docs/concepts/configuration/secret/)
* [Managing Secrets Using Kubectl](https://kubernetes.io/docs/tasks/configmap-secret/managing-secret-using-kubectl/)
* [Encrypting Secret Data at Rest](https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/)

**✅ Lab Outcome**

* We can **create, manage, and use Kubernetes secrets** in multiple ways.
* Observe secrets **mounted as files or exposed as environment variables**.
* Understand **best practices for secret security**.
* Can perform **advanced exercises with TLS, SSH, Docker secrets**.

**Lab 14: Using Container Runtime Sandboxes in Multi-Tenant Environments**

**Objective**

By the end of this lab, you will:

* Understand why **sandbox runtimes** like gVisor and Kata Containers are needed.
* Learn how **gVisor** isolates containers using a **user-space kernel (Sentry + Gofer)**.
* Learn how **Kata Containers** provide isolation using lightweight **virtual machines**.
* Configure Kubernetes to use **multiple runtimes via RuntimeClass**.
* Run and test Pods using different runtime sandboxes.

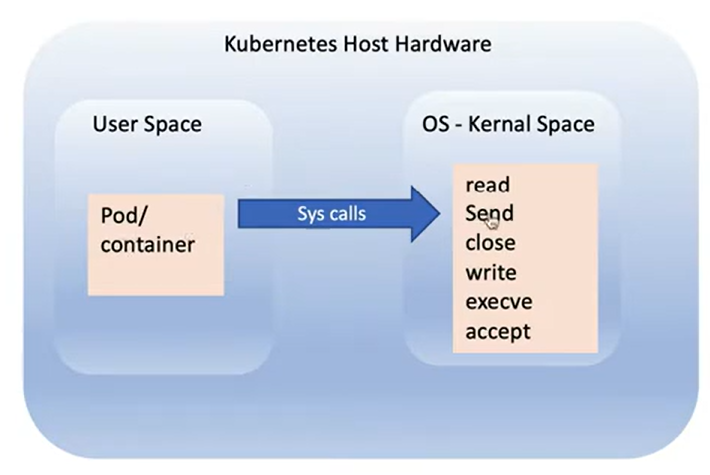
**1️⃣ Conceptual Overview**

**What is Runtime Sandboxing?**

Runtime sandboxing adds an **extra isolation layer** between the container and host kernel.  
In **multi-tenant Kubernetes clusters**, multiple applications share the same kernel.  
If one container escapes to the host, **it can compromise others** — sandbox runtimes reduce this risk.

**Traditional Container Path**

Container → Kernel → Host resources



All containers share the **same kernel** and rely on:

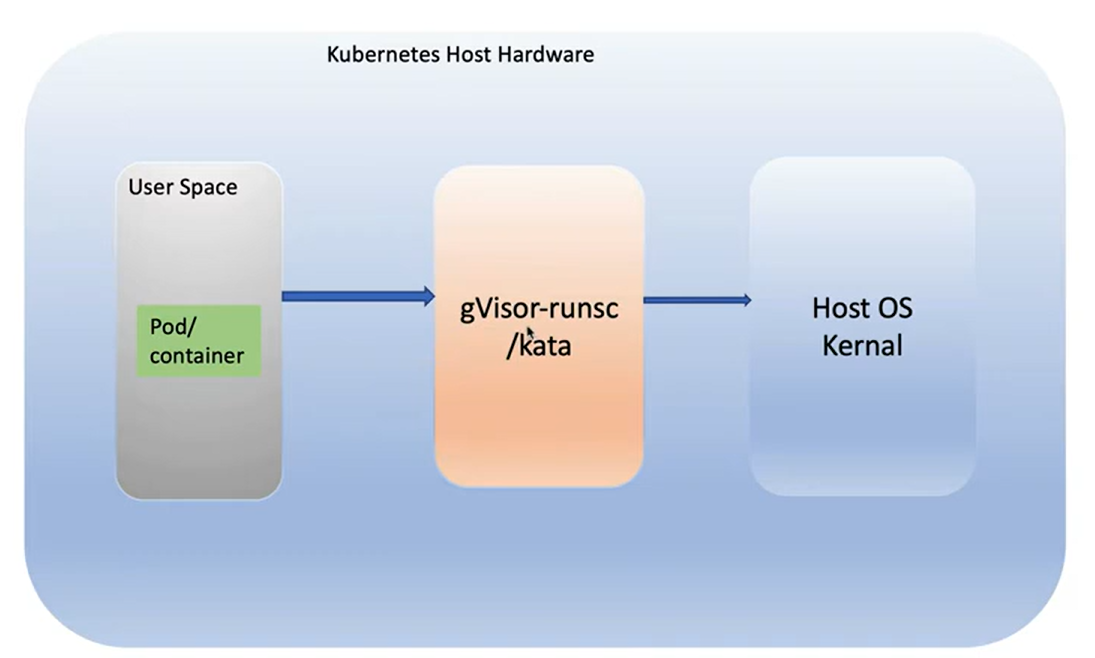
* **Seccomp**: limits syscalls
* **AppArmor / SELinux**: restrict file and resource access

But with thousands of containers, manual syscall filtering isn’t enough.

**Sandboxed Container Path**

Container → Sandbox Runtime → Kernel → Host resources

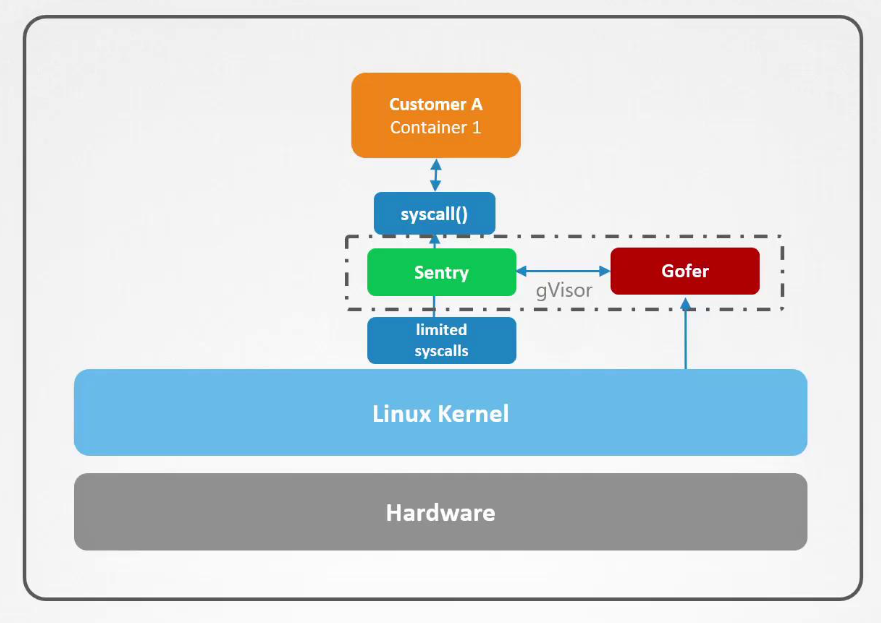
The sandbox runtime acts as an **intermediate layer** (like a “mini-kernel” or “mini-VM”) that traps system calls and controls access.



**2️⃣ Understanding the Two Main Sandbox Runtimes**

**🧠 A. gVisor**

* Developed by Google
* Sits **between container and Linux kernel**
* Implements a large portion of Linux syscalls in **user space**
* Prevents direct kernel access



**Components:**

| **Component** | **Description** |
| --- | --- |
| **Sentry** | User-space kernel ; handles most system calls |
| **Gofer** | File system proxy that interacts with the host kernel for I/O  For file system operations that require real disk access (open, read, write, stat) |
| **runsc** | gVisor runtime binary compatible with OCI (Containerd / Docker) |

**Analogy:**  
Each container has its own **mini kernel (Sentry)** intercepting syscalls before they reach the host kernel.

**🧱 B. Kata Containers**

* Lightweight Virtual Machines for containers.
* Uses **hardware virtualization (KVM/QEMU)**.
* Each container runs inside a **micro VM** with its **own kernel**.
* Offers stronger isolation (similar to a virtual machine).
* Useful for **multi-tenant environments** or untrusted workloads.

**Key difference:**  
Unlike gVisor, Kata doesn’t emulate syscalls — it **runs the container in a lightweight VM**.

**3️⃣ Environment Setup**

This lab assumes:

* Ubuntu 20.04+ node(s)
* containerd runtime (Kubernetes 1.24+)
* Cluster-admin privileges

**3.1 Install gVisor**

# Add gVisor repository

curl -fsSL https://gvisor.dev/archive.key | sudo gpg --dearmor -o /usr/share/keyrings/gvisor-archive-keyring.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/gvisor-archive-keyring.gpg] \

https://storage.googleapis.com/gvisor/releases release main" | sudo tee /etc/apt/sources.list.d/gvisor.list

# Install runsc

sudo apt-get update && sudo apt-get install -y runsc

**3.2 Configure containerd for multiple runtimes**

sudo vi /etc/containerd/config.toml

Add or modify:

version = 2

[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]

runtime\_type = "io.containerd.runc.v2"

[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runsc]

runtime\_type = "io.containerd.runsc.v1"

Restart containerd:

sudo systemctl restart containerd

**3.3 Install crictl for runtime debugging**

wget https://github.com/kubernetes-sigs/cri-tools/releases/download/v1.28.0/crictl-v1.28.0-linux-amd64.tar.gz

tar xf crictl-v1.28.0-linux-amd64.tar.gz

sudo mv crictl /usr/local/bin

cat <<EOF | sudo tee /etc/crictl.yaml

runtime-endpoint: unix:///run/containerd/containerd.sock

EOF

**4️⃣ Test gVisor Runtime (runsc)**

**4.1 Create RuntimeClass for gVisor**

apiVersion: node.k8s.io/v1

kind: RuntimeClass

metadata:

name: gvisor

handler: runsc

Apply it:

kubectl apply -f runtimeclass-gvisor.yaml

**4.2 Deploy a Pod using gVisor**

apiVersion: v1

kind: Pod

metadata:

name: gvisor-pod

spec:

runtimeClassName: gvisor

containers:

- name: web

image: nginx

command: ["sh", "-c", "echo 'Hello from gVisor!' && sleep 3600"]

kubectl apply -f gvisor-pod.yaml

Verify:

kubectl get pods -o wide

kubectl describe pod gvisor-pod

Confirm it’s using the gVisor runtime via logs:

kubectl exec gvisor-pod -- dmesg

**5️⃣ Test Kata Containers Runtime**

🧩 **Note:** Kata requires hardware virtualization (KVM). It won’t work on most managed clouds unless nested virtualization is enabled (GCP supports this).

**5.1 Install Kata Containers (if supported)**

sudo apt-get update

sudo apt-get install -y kata-runtime

**5.2 Configure containerd for Kata**

Edit /etc/containerd/config.toml:

[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.kata]

runtime\_type = "io.containerd.kata.v2"

Restart containerd:

sudo systemctl restart containerd

**5.3 Create RuntimeClass for Kata**

apiVersion: node.k8s.io/v1

kind: RuntimeClass

metadata:

name: kata

handler: kata

Apply it:

kubectl apply -f runtimeclass-kata.yaml

**5.4 Run a Pod using Kata runtime**

apiVersion: v1

kind: Pod

metadata:

name: kata-pod

spec:

runtimeClassName: kata

containers:

- name: secure

image: alpine

command: ["sh", "-c", "echo 'Hello from Kata VM!' && sleep 3600"]

kubectl apply -f kata-pod.yaml

**6️⃣ Verifying Isolation**

**6.1 Check Runtime in Use**

kubectl get pod kata-pod -o=jsonpath='{.spec.runtimeClassName}'

For gVisor:

kubectl exec gvisor-pod -- dmesg

You’ll notice fewer syscalls visible, because **Sentry intercepts them**.

**6.2 Compare Process View from Host**

On host:

ps -ef | grep nginx

For:

* **runc container** → normal process (visible host PID namespace)
* **gVisor container** → wrapped by runsc-sandbox
* **Kata container** → runs inside a lightweight VM (QEMU process visible)

**7️⃣ Cleanup**

kubectl delete pod kata-pod gvisor-pod

kubectl delete runtimeclass kata gvisor

**8️⃣ Visualization: Runtime Sandbox Architecture**

+---------------------------------------------------+

| Kubernetes |

|---------------------------------------------------|

| Pod Spec (runtimeClassName) |

+---------------------------------------------------+

| |

v v

+-------------+ +-------------+

| runC | | runsc |

| (default) | | (gVisor) |

+-------------+ +-------------+

| Shared Kernel | | Sentry + Gofer |

| (Host access) | | Syscall Proxy |

+-------------+ +-------------+

| |

v v

Host Kernel Host Kernel (mediated)

And for Kata:

+---------------------------------------------------+

| Kubernetes Pod (runtimeClassName: kata) |

+---------------------------------------------------+

|

v

+-------------------+

| Kata Runtime |

| (QEMU + KVM) |

+-------------------+

| Guest VM Kernel |

| (Own kernel) |

+-------------------+

|

v

Host Kernel

**9️⃣ When to Use Which Runtime**

| **Runtime** | **Isolation** | **Performance** | **Use Case** |
| --- | --- | --- | --- |
| **runc (default)** | Low | Fast | Trusted workloads |
| **gVisor (runsc)** | Medium | Moderate | Untrusted code, multi-tenant SaaS |
| **Kata Containers** | High | Slightly slower | Highly untrusted or compliance workloads |

**🔟 Key Takeaways**

* **gVisor** → Implements its own kernel (Sentry) to trap syscalls.
* **Kata Containers** → Launches a full micro-VM per container.
* **RuntimeClass** → Mechanism to choose runtime at Pod level.
* **containerd** → Supports multiple runtime handlers in the same cluster.
* Sandboxing is essential in **multi-tenant** or **zero-trust** clusters.

**References**

* 🔗 [gVisor Docs](https://gvisor.dev/docs/user_guide/install/)
* 🔗 Kata Containers Docs
* 🔗 [Kubernetes RuntimeClass](https://kubernetes.io/docs/concepts/containers/runtime-class/)
* 🔗 [CKS gVisor Example](https://sbulav.github.io/certifications/cks-gvisor/)
* 🔗 [Kubernetes Enhancement Proposal #585](https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/585-runtime-class)

✅ **Lab Outcome:**

* You now understand the **purpose and inner workings of sandbox runtimes**.
* You can **install, configure, and verify** gVisor and Kata runtimes in Kubernetes.
* You can **differentiate levels of container isolation** across runtimes.
* You’ve learned how to use **RuntimeClass** to dynamically select a runtime for Pods.