**LAB 15: Static Analysis & Vulnerability Scanning in Kubernetes**

**Objective**

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

1. Understand and implement **static analysis** for Kubernetes YAML manifests.
2. Identify security risks in pod specifications.
3. Scan container images for known vulnerabilities using Trivy.
4. Generate actionable insights for securing your Kubernetes workloads.

**Pre-requisites**

* Linux workstation or VM (Ubuntu/Debian preferred)
* Access to a **Kubernetes cluster** (minikube, kind, or cloud cluster)
* **kubectl** installed and configured
* Basic understanding of Kubernetes resources: Pods, Deployments, YAML manifests

**1. Static Analysis of Kubernetes Manifests**

**1.1 Why Static Analysis?**

Static analysis helps you detect security misconfigurations **before** workloads are deployed. Common risks include:

* Running containers as root.
* Privileged pods.
* Host namespace access.
* Using :latest image tags.

**1.2 Best Practices**

* Avoid hostNetwork: true, hostIPC: true, hostPID: true unless absolutely necessary.
* Avoid securityContext.privileged: true.
* Avoid running containers as root (runAsUser: 0); prefer nobody or a non-privileged user.
* Avoid :latest image tags; pin a specific version (e.g., busybox:1.33.1).

**1.3 Tool: Kubesec**

Kubesec is a **static YAML analysis tool** that evaluates Kubernetes manifests and assigns a **risk score**.

**Installation (Linux)**:

# Download latest release

*wget* [*https://github.com/controlplaneio/kubesec/releases/download/v2.11.4/kubesec\_linux\_amd64.tar.gz*](https://github.com/controlplaneio/kubesec/releases/download/v2.11.4/kubesec_linux_amd64.tar.gz)*tar -xvf kubesec\_linux\_amd64.tar.gz*

./kubesec scan test29.yaml

**1.4 Lab: Static YAML Analysis**

**Step 1: Create a sample pod manifest** (pod.yaml):

apiVersion: v1

kind: Pod

metadata:

name: vulnerable-pod

spec:

containers:

- name: nginx

image: nginx:latest

securityContext:

privileged: true

runAsUser: 0

hostNetwork: true

**Step 2: Run Kubesec scan locally**:

# Scan a single YAML file

./kubesec scan test29.yaml

**Step 5: Analyze the output**

* Risk scores range from **0 (highest risk)** to **10 (lowest risk)**.
* Kubesec flags:
  + hostNetwork: true → HIGH RISK
  + privileged: true → HIGH RISK
  + runAsUser: 0 → HIGH RISK
  + :latest image tag → MEDIUM RISK

**1.5 Remediation Example**

**Updated pod.yaml**:

apiVersion: v1

kind: Pod

metadata:

name: secure-pod

spec:

containers:

- name: nginx

image: nginx:1.23.0

securityContext:

runAsUser: 1000

runAsNonRoot: true

**2. Image Vulnerability Scanning**

**2.1 Why Scan Images?**

Even if your pod YAML is secure, the container image itself may contain **known vulnerabilities** (CVEs).  
Tools like **Trivy** help detect these issues in images before deployment.

**2.2 Tool: Trivy**

**Trivy** (by AquaSec) scans container images, file systems, and git repositories for vulnerabilities.

**Installation (Linux)**:

# Download Trivy

wget https://github.com/aquasecurity/trivy/releases/download/v0.18.3/trivy\_0.18.3\_Linux-64bit.deb

sudo dpkg -i trivy\_0.18.3\_Linux-64bit.deb

**2.3 Lab: Scan Container Images**

**Step 1: Scan an image for all vulnerabilities**

trivy image python:3.4-alpine

**Output includes**:

* Severity levels: **LOW, MEDIUM, HIGH, CRITICAL**
* Vulnerability ID (e.g., CVE-2023-12345)
* Package name, installed version, fixed version

**Step 2: Filter by severity**

trivy image --severity HIGH,CRITICAL python:3.4-alpine

**Step 3: Scan all images running in the cluster**

# List all images in all pods

kubectl get pods -A -o jsonpath="{..image}" | tr -s '[[:space:]]' '\n' | sort -u

# Example: pipe output to Trivy scan

kubectl get pods -A -o jsonpath="{..image}" | tr -s '[[:space:]]' '\n' | sort -u | xargs -I {} trivy image --severity HIGH,CRITICAL {}

**2.4 Remediation Example**

* Upgrade vulnerable images to a fixed version.
* Remove unnecessary packages from images.
* Switch to minimal base images (e.g., alpine) where possible.

**3. Recommended Lab Workflow**

1. **Static Analysis First**
   * Run Kubesec on YAML files.
   * Fix high-risk configurations (privileged, root, hostNetwork, :latest).
2. **Image Vulnerability Scan**
   * Scan all container images using Trivy.
   * Filter high/critical vulnerabilities.
   * Upgrade images as required.
3. **Deployment**
   * Only deploy manifests and images that have passed both checks.

**References**

* Kubernetes Security Blog: [11 ways not to get hacked](https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/#7-statically-analyse-yaml)
* Kubesec: <https://kubesec.io/>
* Trivy: https://aquasec.com/trivy

**Bonus Tips**

* Automate Kubesec scans using CI/CD pipelines (GitHub Actions, GitLab CI).
* Use Trivy as a **pre-deploy gate** in pipelines.
* Regularly update image versions to reduce CVE exposure.
* Combine with runtime security tools like **Kubernetes Pod Security Admission (PSA)** or **OPA/Gatekeeper** for ongoing enforcement.

**Lab 16: Minimize base image footprint & Supply Chain Security with OPA Gatekeeper**

**Part 1: Minimize Base Image Footprint**

**1.1 Why Minimize Base Images?**

Reducing the image footprint minimizes:

* **Attack surface:** Fewer binaries and tools for attackers.
* **Vulnerabilities:** Less code = fewer potential CVEs.
* **Image size:** Faster builds and deployments.

**Base Image Types Comparison**

| **Type** | **Description** | **Example** | **Size** |
| --- | --- | --- | --- |
| **Full OS** | Standard images with package managers, shell, tools | ubuntu:20.04 | ~70–80 MB |
| **Slim/Minimal** | Stripped-down images (no unnecessary tools) | python:3.9-slim | ~40 MB |
| **Distroless** | Contain only runtime binaries; no package managers or shells | gcr.io/distroless/python3 | ~20 MB |

**1.2 Techniques for Minimizing Image Size**

**✅ Use Slim/Minimal Base Images**

Instead of:

FROM ubuntu:20.04

use:

FROM python:3.9-slim

**✅ Use Distroless Images**

**What are Distroless Images?**

* Developed by Google.
* Contain **only** application binaries and their **runtime dependencies**.
* No shell, no package manager, no bash utilities.

**Benefits:**

* Extremely small.
* Immutable and harder to tamper with.
* Ideal for production workloads.

**Distroless Reference:**  
<https://github.com/GoogleContainerTools/distroless>

**Part 2: Secure Your Supply Chain with OPA Gatekeeper**

**2.1 Why Supply Chain Security?**

Your supply chain includes:

* **Code** → App source.
* **Images** → Base and app container images.
* **Registries** → Where images are pulled from.

To prevent **supply chain attacks**, you should:

* **Whitelist trusted registries**.
* **Disallow unknown/untrusted image sources**.

**2.2 Tool: OPA Gatekeeper**

**OPA (Open Policy Agent)** with **Gatekeeper** allows **policy-as-code** for Kubernetes Admission Controller.

It enforces custom rules **before** resources are created in the cluster.

**2.3 Lab: Install OPA Gatekeeper**

**Step 1: Deploy Gatekeeper via YAML**

kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.15/deploy/gatekeeper.yaml

All pods (e.g., gatekeeper-controller-manager) should be Running.

**2.4 Create a ConstraintTemplate to Whitelist Registries**

**File:** k8sallowedrepos-template.yaml

apiVersion: templates.gatekeeper.sh/v1beta1

kind: ConstraintTemplate

metadata:

name: k8sallowedregistries

spec:

crd:

spec:

names:

kind: K8sAllowedRegistries

validation:

# Parameters that can be passed in a Constraint

openAPIV3Schema:

properties:

allowedRegistries:

type: array

items:

type: string

targets:

- target: admission.k8s.gatekeeper.sh

rego: |

package k8sallowedregistries

violation[{"msg": msg}] {

input.review.kind.kind == "Pod"

container := input.review.object.spec.containers[\_]

registry := get\_registry(container.image)

not registry\_allowed(registry, input.parameters.allowedRegistries)

msg := sprintf("Image registry '%s' is not in the allowed list: %v", [registry, input.parameters.allowedRegistries])

}

# Support initContainers too

violation[{"msg": msg}] {

input.review.kind.kind == "Pod"

container := input.review.object.spec.initContainers[\_]

registry := get\_registry(container.image)

not registry\_allowed(registry, input.parameters.allowedRegistries)

msg := sprintf("InitContainer registry '%s' is not in the allowed list: %v", [registry, input.parameters.allowedRegistries])

}

# Helper function: extract registry part (e.g. docker.io from docker.io/library/nginx)

get\_registry(image) = registry {

parts := split(image, "/")

count(parts) > 1

registry := parts[0]

}

# Function: check if the registry is allowed

registry\_allowed(registry, allowed) {

registry == allowed[\_]

}

NOTE : This Rego script checks every container and initContainer in a Pod, extracts the registry part of its image (like docker.io or gcr.io), and compares it against a predefined list of allowed registries.

Apply it:

kubectl apply -f k8sallowedrepos-template.yaml

**2.5 Create a Constraint Resource**

**File:** k8sallowedrepos-constraint.yaml

apiVersion: constraints.gatekeeper.sh/v1beta1

kind: K8sAllowedRegistries

metadata:

name: allowed-registries

spec:

match:

kinds:

- apiGroups: [""]

kinds: ["Pod"]

parameters:

allowedRegistries:

- mycompany.azurecr.io

- docker.io

- gcr.io

Apply it:

kubectl apply -f k8sallowedrepos-constraint.yaml

**2.6 Test Policy Enforcement**

**✅ Test 1: Allowed Registry (docker.io)**

**File:** valid-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: dh-busybox

spec:

restartPolicy: Never

containers:

- name: busybox

image: docker.io/library/busybox

command: ["sh", "-c", "sleep 3600"]

kubectl apply -f valid-pod.yaml

**Expected:** Pod creation succeeds.

**❌ Test 2: Disallowed Registry**

**File:** invalid-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: gh-busybox

spec:

restartPolicy: Never

containers:

- name: busybox

image: ghcr.io/library/busybox

command: ["sh", "-c", "sleep 3600"]

kubectl apply -f invalid-pod.yaml

**Expected Output:**

Error from server (Forbidden): admission webhook "validation.gatekeeper.sh" denied the request:

container <busybox> has an invalid image repo <ghcr.io/library/busybox>, allowed repos are ["docker.io"]

**2.7 Validate Gatekeeper Enforcement**

List Gatekeeper constraints and their status:

kubectl get k8sallowedrepos.constraints.gatekeeper.sh

kubectl describe constraint k8sallowedrepos/whitelist-dockerhub

**3. Summary & Takeaways**

| **Area** | **Tool** | **Goal** | **Benefit** |
| --- | --- | --- | --- |
| **Static Analysis** | Kubesec | YAML risk scoring | Secure configurations |
| **Image Vulnerability Scan** | Trivy | CVE detection | Identify risky images |
| **Image Minimization** | Distroless + Multi-stage | Lean secure images | Reduced attack surface |
| **Supply Chain Security** | OPA Gatekeeper | Policy enforcement | Prevent untrusted images |

**4. References**

* Distroless Images → <https://github.com/GoogleContainerTools/distroless>
* Trivy Scanner → <https://github.com/aquasecurity/trivy>
* OPA Gatekeeper → https://open-policy-agent.github.io/gatekeeper/website/docs/

**LAB 17: Kubernetes Audit Logging and API/Kubelet Debugging**

**🎯 Lab Objective**

By the end of this lab, you will:

* Understand **Kubernetes Audit Logs** and how they work.
* Configure a **custom Audit Policy**.
* Enable audit logging on the **API Server**.
* Verify audit log generation and stages.
* Understand key **log sources** for cluster-level debugging.
* Diagnose **API Server** and **Kubelet** failures using system logs and container logs.

**🧩 Lab Prerequisites**

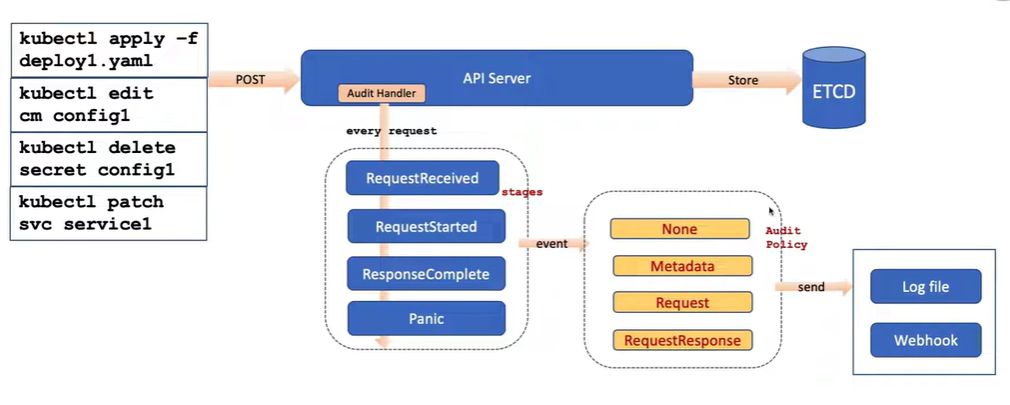
| **Component** | **Requirement** |
| --- | --- |
| Kubernetes Cluster | v1.20+ (works up to v1.31) |
| Access Level | Root or sudo privileges on control plane node |
| Tools Installed | kubectl, vi/nano, docker (if Docker runtime), journalctl |
| Cluster Type | Kubeadm or self-managed cluster (not GKE/EKS) |

**🧠 Section 1 — Understanding Kubernetes Audit Logs**

Kubernetes **Audit Logging** provides visibility into **who did what and when** in your cluster.

It captures every API call to the **Kubernetes API Server** and records:

* Request metadata (user, namespace, verb, resource, etc.)
* Request/response content (optional)
* Stages (lifecycle) of each request



**📋 Audit Event Stages**

| **Stage** | **Description** |
| --- | --- |
| RequestReceived | When the API server first receives the request |
| ResponseStarted | When response headers are sent (used for long-running requests like watch) |
| ResponseComplete | When the response body has been fully written |
| Panic | When a panic occurred while processing the request |

The defined audit levels are:

* None - don't log events that match this rule.

🧠 Use Case:

To **exclude noisy operations** (e.g., health probes, frequent list/watch calls).

Reduces disk usage.

* Metadata - log events with metadata (requesting user, timestamp, resource, verb, etc.) but not request or response body.

🧠 Use Case:

Ideal for **general observability**, compliance (who/what/when), without sensitive data.

* Request - log events with request metadata and body but not response body. This does not apply for non-resource requests.

**🧠 Use Case:**

Used in **audit compliance** or **debugging**.

Helps see what data was *submitted* (e.g., new configuration or object manifest).

* RequestResponse - log events with request metadata, request body and response body. This does not apply for non-resource requests.

🧠 Use Case:

Full **forensic tracing**, **incident response**, and **security analysis**.Use cautiously ,increases log volume and may capture sensitive information.

**🧱 Audit Log Architecture**

[User/API Client]

|

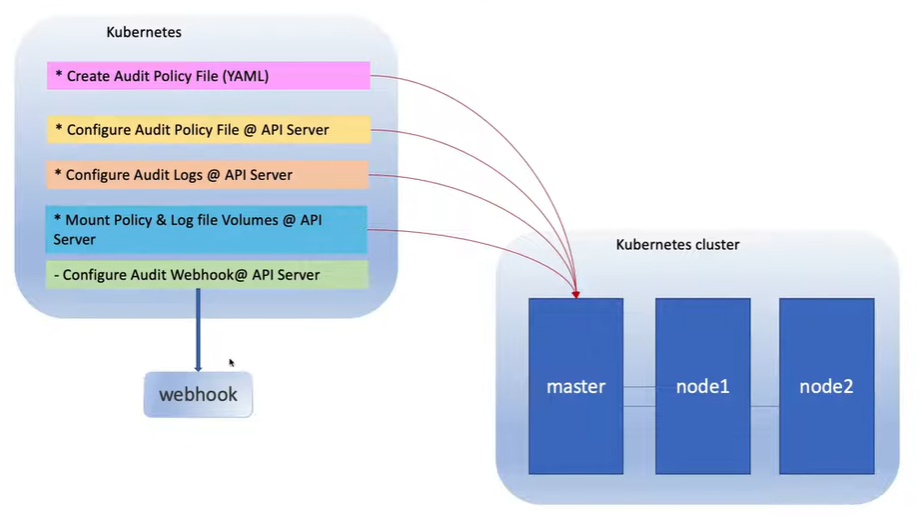
v

[API Server] ---> [Audit Policy] ---> [Audit Backend]

|

+--> [File Backend: /var/log/k8s-audit.log]

+--> [Webhook Backend: send to Splunk]



**⚙️ Section 2 — Creating the Audit Policy File**

The **Audit Policy** defines *what to record* and *how much detail* to log.

Create the file on the **control plane node**:

sudo mkdir -p /etc/kubernetes/audit

sudo vi /etc/kubernetes/audit/policy.yaml

Paste the following **policy configuration**:

apiVersion: audit.k8s.io/v1

kind: Policy

omitStages:

- "RequestReceived"

rules:

# High-detail log for deleting the webapp pod in prod

- namespaces: ["prod-namespace"]

verbs: ["delete"]

resources:

- group: ""

resources: ["pods"]

resourceNames: ["webapp-pod"]

level: RequestResponse

# Log all write operations (create/update/patch/delete) to Deployments

- verbs: ["create", "update", "patch", "delete"]

resources:

- group: "apps"

resources: ["deployments"]

level: Metadata

# Ignore read-only operations (to reduce noise)

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

level: None

# Ignore system namespaces

- namespaces: ["kube-system", "kube-public", "default"]

level: None

# Default catch-all for anything else (minimal)

- level: Metadata

**🧩 Section 3 — Enable Audit Logging in API Server**

Edit the **API Server manifest** (for kubeadm clusters):

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

Add the following flags under the command: section:

- --audit-policy-file=/etc/kubernetes/audit/policy.yaml

- --audit-log-path=/etc/kubernetes/audit/k8s-audit.log

- --audit-log-maxage=30

- --audit-log-maxbackup=10

- --audit-log-maxsize=100

💡 These parameters define:

* **audit-log-path** → where audit logs are stored.
* **audit-policy-file** → path to policy file.
* **maxage**, **maxbackup**, **maxsize** → log rotation policy.

Save and exit.

* Update the volume and volume mount section as well.

volumeMounts:

- mountPath: /etc/kubernetes/audit

name: audit

readOnly: false

volumes:

# Existing volumes...

- name: audit

hostPath:

path: /etc/kubernetes/audit

type: DirectoryOrCreate

**🌀 Verify the API Server Restart**

Since the kube-apiserver runs as a **static pod**, it will automatically restart when you save the file.

sudo crictl ps | grep kube-apiserver

# or

docker ps | grep kube-apiserver

Check if the container restarted:

docker logs kube-apiserver-<hostname> | tail -n 20

**📂 Section 4 — Validate Audit Log Generation**

Perform a test operation that matches your audit rule:

kubectl delete pod webapp-pod -n prod-namespace

Now inspect the log file:

sudo tail -f /etc/kubernetes/audit/k8s-audit.log

NOTE: MAKE SURE IN ABOVE THERE IS SMALL K IN KUBERNETES

NOTE : If changed policy file , than reload by vi into kube api servel maifect and exit without saving

✅ You have successfully enabled and verified Kubernetes Audit Logging.

TESTING :

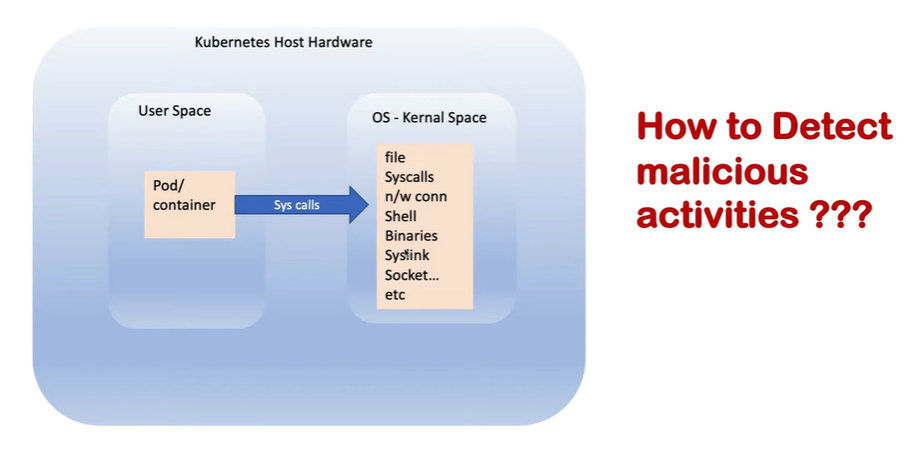
k create deploy ramandeploy -n raman --replicas 2 --image nginx

sudo tail -f /etc/kubernetes/audit/k8s-audit.log | grep -i raman

**LAB 17: Kubernetes Security Runtime Monitoring**

**1. Falco Concepts and Architecture**

Before diving into labs, you must understand **how Falco works**.

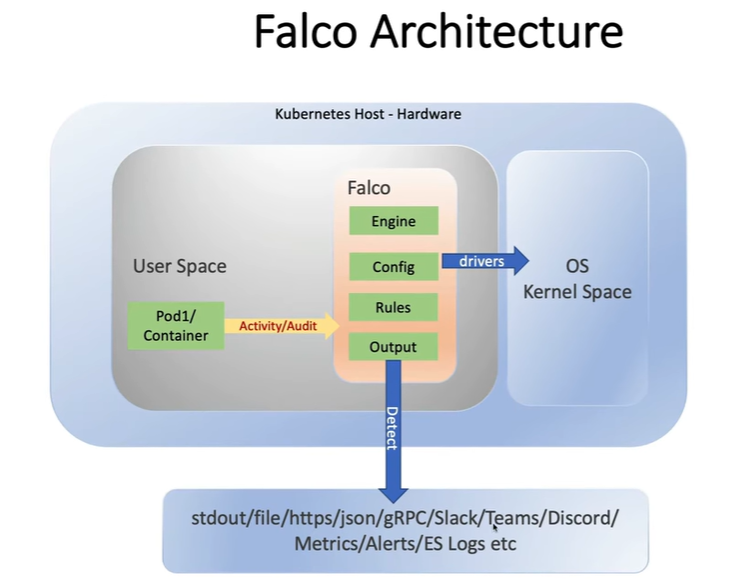


**1.1 Event Sources**

Falco monitors **events** generated by the system. Key concepts:

| **Component** | **Description** | **Example Events** |
| --- | --- | --- |
| **System Calls** | Falco captures Linux kernel syscalls to understand what processes are doing. | File opens, writes, execve, network connects |
| **Kubernetes Events** | Events from the Kubernetes API about pods, containers, and RBAC changes. | Pod creation, container deletion, role modification |
| **Container Runtime Events** | Falco can integrate with container runtimes (Docker, containerd). | Container start, stop, mounts, image pulls |

**Key Idea:** Falco doesn’t just look at logs—it observes **real-time runtime behaviors** using system calls and Kubernetes events.

****

**1.2 Falco Rules**

**Falco rules** define **what is considered suspicious or malicious**.  
Each rule has several components:

| **Component** | **Description** |
| --- | --- |
| **rule** | Unique name of the rule. |
| **desc** | Description of what the rule detects. |
| **condition** | Logical expression that triggers the alert. |
| **output** | The message printed or sent when the rule triggers. |
| **priority** | Severity level (INFO, WARNING, CRITICAL). |
| **tags** | Optional, used to categorize rules (e.g., kubernetes, file, network). |

**Example Rule (Prebuilt):**

- rule: Write below etc

desc: Detect write access below /etc

condition: evt.type = open and fd.name startswith /etc and evt.arg.flags contains O\_WRONLY

output: "Write below /etc (user=%user.name command=%proc.cmdline file=%fd.name)"

priority: WARNING

**1.3 Important Components of Falco**

| **Component** | **Role** |
| --- | --- |
| **falco-driver-loader** | Kernel module or eBPF probe loader to capture syscalls. |
| **falco** | Main engine that evaluates syscalls/events against rules. |
| **Rules YAML files** | Define what events are suspicious. Examples: falco\_rules.yaml |
| **Outputs** | Where alerts are sent (stdout, syslog, files, SIEM, Slack). |
| **Falco Sidekick** (Optional) | Helps integrate Falco alerts with external systems like Slack or Elasticsearch. |

**2. Prebuilt Rules Examples**

Here are **common prebuilt rules** you can demonstrate in your lab:

**2.1 File System Rules**

* **Write below /etc** – detects modifications to sensitive system files.
* **Write to /var/run/docker.sock** – containers trying to manipulate Docker.
* **Open sensitive files for writing** – /etc/shadow, /etc/passwd.

**2.2 Process & Command Rules**

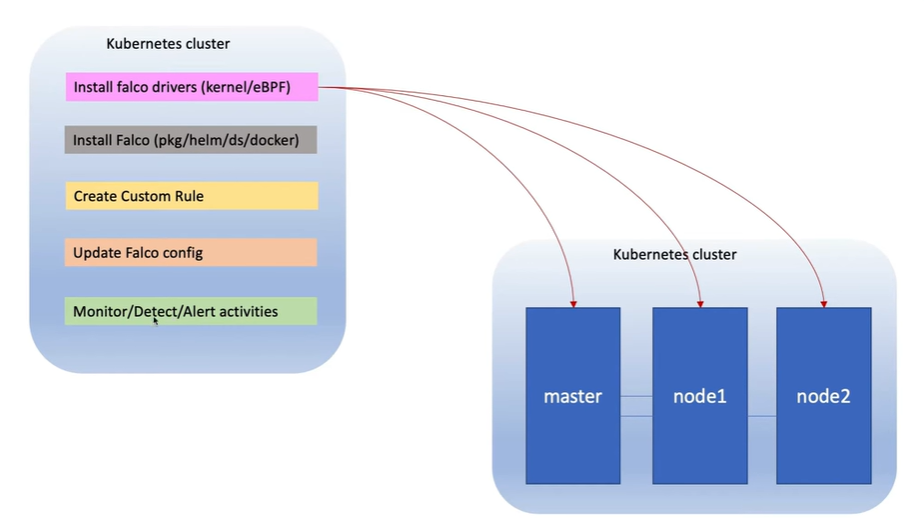
* **Shell in container** – any shell (bash/sh) spawned inside a container.
* **Unexpected process execution** – processes that shouldn’t run in certain workloads.
* **Privilege escalation** – sudo or setuid binary execution.

**2.3 Network Rules**

* **Outbound connections from container** – detect unexpected external communication.
* **Network port scanning** – multiple connection attempts to different IPs/ports.
* **Accessing sensitive ports** – SSH, database ports from unusual pods.

**2.4 Kubernetes Specific Rules**

* **Kubernetes API access** – detecting unauthorized API calls.
* **Privileged container creation** – detecting pods running with privileged: true.
* **Service account token usage** – detecting unexpected pod token mounts.



<https://falco.org/docs/setup/packages/#install-with-apt>

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

helm repo add falcosecurity https://falcosecurity.github.io/charts

helm repo update

helm install falco falcosecurity/falco -n falco --create-namespace

**. Lab Exercises**

**3.1 Lab 1: Verify Falco Deployment**

**Objective:** Ensure Falco is running and monitoring events.  
**Steps:**

1. Check Falco pods:

kubectl get pods -n falco

1. Check Falco logs:

kubectl logs -n falco <falco-pod-name>

1. Trigger a test alert:

kubectl exec -it <any-pod> -- bash

touch /etc/testfile

**Expected Result:** Falco should generate a warning: Write below /etc.

**3.2 Lab 2: Observe Prebuilt Alerts**

**Objective:** Understand how default rules work.

1. **Shell in container**

kubectl exec -it <any-pod> -- bash

* Falco should alert: Shell spawned in container.

1. **Modify sensitive file**

kubectl exec -it <any-pod> -- bash

echo "test" > /etc/passwd

* Falco alert: Write below /etc.

1. **Run network scan**

kubectl exec -it <any-pod> -- nmap -p 22 10.0.0.0/24

* Falco alert: network port scanning detected.

**3.3 Lab 3: Custom rules**

**Objective:** Create environment-specific rules.

**2. Falco Rule Structure**

Every rule in Falco follows this logical structure:

- rule: <rule\_name>

desc: <description of what this rule detects>

condition: <boolean expression evaluated on events>

output: <alert message with dynamic fields>

priority: <severity level>

tags: [optional\_tags]

**2.1 Example:**

- rule: Shell Spawned Inside Container

desc: Detect when a shell is launched inside a running container

condition: container.id != host and proc.name in (bash, sh, zsh, ksh)

output: "Shell spawned in container (user=%user.name command=%proc.cmdline container=%container.name)"

priority: WARNING

tags: [process, container]

**3. Rule Building Blocks**

**3.1 Events**

Events are **activities** Falco monitors — mainly **syscalls** (like open, execve, connect) or **Kubernetes audit logs**.

| **Event Type** | **Description** | **Example** |
| --- | --- | --- |
| **System Call Events** | Generated whenever a process performs an OS operation. | evt.type=execve (program executed) |
| **File System Events** | Detected when a file is read, written, or opened. | evt.type=open |
| **Network Events** | Triggered on socket operations. | evt.type=connect, evt.type=accept |
| **Kubernetes Audit Events** | Captured when API objects change. | ka.verb=create, ka.target.resource=pod |

**Example:**  
A syscall event where a container writes to /etc/passwd  
→ Falco sees an event: evt.type=open, fd.name=/etc/passwd, evt.arg.flags=O\_WRONLY

**3.2 Fields**

Falco exposes hundreds of **fields** that represent attributes of the event, such as the process, user, file, network socket, etc.

| **Category** | **Field** | **Description** |
| --- | --- | --- |
| **Event** | evt.type | Type of syscall (open, execve, connect) |
| **File Descriptor** | fd.name | File path accessed |
| **Process** | proc.name, proc.cmdline, proc.exepath | Name, full command, executable path |
| **User** | user.name, user.uid | Linux user context |
| **Container** | container.id, container.name, container.image.repository | Container information |
| **Kubernetes** | k8s.ns.name, k8s.pod.name, k8s.pod.labels | Kubernetes context |
| **Network** | fd.sip, fd.sport, fd.lip, fd.lport | Source/destination IPs and ports |

**3.3 Conditions**

A **condition** is a Boolean expression combining fields and operators.  
If it evaluates to **true** for an event, the rule triggers an alert.

**Operators:**

* Equality: =, !=
* Logical: and, or, not
* Set membership: in, contains
* String patterns: startswith, endswith

**Example Conditions:**

evt.type = open and fd.name startswith /etc and evt.arg.flags contains O\_WRONLY

container.id != host and proc.name in (bash, sh)

**3.4 Macros and Lists**

To avoid repetition, Falco supports reusable components:

**Macro:**

A reusable condition.

- macro: open\_write

condition: evt.type=open and evt.arg.flags contains O\_WRONLY

**List:**

A reusable list of values.

- list: shell\_binaries

items: [bash, sh, zsh, ksh]

**Using them in rules:**

- rule: Shell Spawn in Container

desc: Detect shell execution in containers

condition: container.id != host and proc.name in (shell\_binaries)

output: "Shell executed in container %container.name"

priority: WARNING

**Example 1 – Detect access to secret volumes:**

**Cutom Falco Rule custom-falco-rules.yaml**

# Your custom rules!

- rule: Access Secret Volumes

desc: Detect access to Kubernetes secret volume files

condition: >

(evt.type in (open, openat, read)) and

container and

fd.name startswith /var/run/secrets

output: >

Secret volume accessed by container=%container.name (user=%user.name process=%proc.name file=%fd.name)

priority: CRITICAL

tags: [k8s, secrets, data\_access, mitre\_exfiltration]

**Steps:**

kubectl create configmap falco-custom-rules --from-file=custom-falco-rules.yaml -n falco

kubectl patch daemonset falco -n falco --type='json' -p='[

{

"op": "add",

"path": "/spec/template/spec/volumes/-",

"value": {

"name": "custom-rules",

"configMap": {

"name": "falco-custom-rules"

}

}

}

]'

kubectl patch daemonset falco -n falco --type='json' -p='[

{

"op": "add",

"path": "/spec/template/spec/containers/0/volumeMounts/-",

"value": {

"name": "custom-rules",

"mountPath": "/etc/falco/rules.d/custom-falco-rules.yaml",

"subPath": "custom-falco-rules.yaml"

}

}

]'

kubectl rollout restart daemonset falco -n falco

**check Falco logs**

kubectl logs -n falco -l app.kubernetes.io/name=falco --tail=50 -f

1. Reload Falco:

kubectl exec -n falco -it <falco-pod> -- pkill -HUP falco

1. Test by reading a secret:

kubectl exec -it <any-pod> -- cat /var/run/secrets/kubernetes.io/serviceaccount/token

* Falco should generate the custom alert.

**3.4 Lab 4: Simulate Attacks**

**Objective:** Map alerts to attack phases (Recon, Execution, Privilege Escalation).

| **Attack** | **Trigger** | **Expected Falco Alert** |
| --- | --- | --- |
| Run shell in pod | kubectl exec -it pod -- bash | Shell spawned in container |
| Modify sensitive file | echo test > /etc/passwd | Write below /etc |
| Privilege escalation | Run sudo in container | Setuid binary executed |
| Network scan | nmap -p 22 10.0.0.0/24 | Network port scanning detected |
| Access secret | cat /var/run/secrets/... | Secret volume accessed |

**3.5 Lab 5: Alert Analysis**

**Objective:** Investigate Falco alerts for forensic understanding.

1. Collect logs:

kubectl logs -n falco <falco-pod> > falco\_logs.txt

1. Identify:
   * **Who triggered the alert** (user.name)
   * **Which container/pod**
   * **Command executed**
2. Map alerts to **Kubernetes workloads and attack phases**.

**4. Hands-On Labs: Writing Custom Rules**

Now that you understand Falco’s building blocks, let’s create and test some **custom rules**.

**Lab 1 – Detect File Write in /etc**

**Objective:** Detect any file write under /etc.

**Rule:**

- rule: Custom Write Below etc

desc: Detect any process writing to /etc

condition: evt.type=open and fd.name startswith /etc and evt.arg.flags contains O\_WRONLY

output: "Sensitive file write detected! (file=%fd.name command=%proc.cmdline user=%user.name)"

priority: CRITICAL

**Steps:**

1. Edit the Falco custom rule file:

kubectl exec -it -n falco <falco-pod> -- bash

vi /etc/falco/falco\_rules.local.yaml

1. Add the rule above.
2. Reload Falco:

pkill -HUP falco

1. Trigger the alert:

kubectl exec -it <any-pod> -- bash

echo "test" > /etc/hostname

1. Check Falco logs:

kubectl logs -n falco <falco-pod> | grep "Sensitive file write"

**Lab 2 – Detect Shell Execution Inside Container**

**Objective:** Detect when a shell process (bash, sh) starts inside any container.

**Rule:**

- list: shell\_procs

items: [bash, sh, zsh, ksh]

- rule: Shell Spawned in Container

desc: Detect shell processes in containers

condition: container.id != host and proc.name in (shell\_procs)

output: "Shell spawned in container (user=%user.name container=%container.name command=%proc.cmdline)"

priority: WARNING

**Test:**

kubectl exec -it <your-pod> -- bash

Falco logs should show the custom alert.

**Lab 3 – Detect Access to Kubernetes Secret Files**

**Objective:** Detect when a container reads files under /var/run/secrets/.

**Rule:**

- rule: Access Secret Volumes

desc: Detect containers accessing Kubernetes secret volumes

condition: (evt.type in (open, openat, read)) and container.id != host and fd.name startswith /var/run/secrets

output: "Container %container.name accessed secret volume file %fd.name"

priority: CRITICAL

**Test:**

kubectl exec -it <pod> -- cat /var/run/secrets/kubernetes.io/serviceaccount/token

You’ll get an alert in Falco logs.

**Lab 4 – Detect Unexpected Network Connections**

**Objective:** Alert when a container connects to external IPs.

**Rule:**

- rule: Outbound Connection from Container

desc: Detect outbound connections from any container

condition: evt.type = connect and container.id != host and fd.sip != 127.0.0.1

output: "Container %container.name made outbound connection to %fd.sip:%fd.sport"

priority: NOTICE

**Test:**

kubectl exec -it <pod> -- curl https://example.com

**Lab 5 – Detect Privileged Pod Creation (Kubernetes Audit Event)**

**Objective:** Detect creation of privileged pods using Kubernetes audit logs.

**Rule:**

- rule: Create Privileged Pod

desc: Detect creation of a privileged pod

condition: ka.verb=create and ka.target.resource=pods and ka.req.pod.containers.privileged=true

output: "Privileged Pod Created: user=%ka.user.name pod=%ka.target.name namespace=%ka.target.namespace"

priority: CRITICAL

**Test:**

1. Deploy a privileged pod:
2. kubectl run privpod --image=alpine --privileged -it sh
3. Observe Falco audit log alert.

**6. Best Practices for Custom Rule Development**

| **Practice** | **Description** |
| --- | --- |
| **Start Broad, Then Refine** | Begin with general conditions, then narrow down to avoid false positives. |
| **Use Macros & Lists** | Reuse common conditions for cleaner rules. |
| **Add Context in Output** | Include %container.name, %proc.cmdline, %user.name in alerts. |
| **Use Correct Priority** | Assign severity based on risk (INFO, NOTICE, WARNING, CRITICAL). |
| **Version Control** | Store falco\_rules.local.yaml in Git for audit and change tracking. |

**Attack Phase Correlation & Threat Actor Investigation**

This section teaches **how to connect Falco alerts into a full attack narrative** — identifying where an attack began, how it progressed, and who executed it.

**2.1 Lab: Simulate Full Attack Chain**

**Goal:** Observe Falco catching multiple stages of a simulated intrusion.

**Scenario:**

1. Attacker gains shell access in a pod.
2. Modifies /etc/passwd (privilege escalation attempt).
3. Reads a Kubernetes secret (data theft).
4. Makes an outbound connection to exfiltrate data.

**Sequence of Actions:**

# Step 1: Shell access

kubectl exec -it <nginx-pod> -- bash

# Step 2: Modify system file

echo "test" > /etc/hostname

# Step 3: Read secret

cat /var/run/secrets/kubernetes.io/serviceaccount/token

# Step 4: Outbound connection

curl http://8.8.8.8

**Expected Falco Alerts:**

| **Phase** | **Falco Rule Triggered** | **Description** |
| --- | --- | --- |
| Exploitation | Shell Spawned in Container | Attacker gained shell |
| Privilege Escalation | Write Below /etc | Modified system files |
| Credential Access | Access Secret Files | Read service account tokens |
| Exfiltration | Outbound Connection | Attempted data exfiltration |

**2.2 Lab: Identify the Bad Actor**

**Objective:** Use Falco logs and Kubernetes metadata to trace the source of the activity.

**Steps:**

1. Retrieve Falco logs:

kubectl logs -n falco <falco-pod> > falco\_analysis.txt

1. Identify key fields in the alerts:
   * %user.name → who executed the action
   * %container.name → in which container
   * %k8s.ns.name → namespace context
   * %proc.cmdline → exact command executed
2. Correlate to Kubernetes objects:

kubectl describe pod <container-name>

* + Find service account or image used.
  + Identify owner (Deployment/Job/User).

1. Build attack timeline:
   * Which rule triggered first?
   * Which namespace or workload was impacted?
   * Was this caused by a compromised app, insider, or misconfiguration?