sysdig SECURE EVERY SECOND.

E-BOOK

OWASP Top 10 for Kubernetes

In-depth analysis and mitigation steps

Table of Contents

Introduction

Cloud attacks are happening faster than ever before; bad actors need only 10 minutes or less to execute an attack. With the increasing adoption of Kubernetes to provide the infrastructure that powers modern cloud-native applications, it's important to ensure that these environments are secure and resilient against potential cyberthreats. The Open Web Application Security Project (OWASP) Top 10 for Kubernetes is a set of security risks specific to Kubernetes environments that organizations should address in order to ensure the security of cloud-native applications.

As Kubernetes acts as the brain (or orchestrator) for distributed container deployment, it manages service-oriented applications using containers distributed across clusters of hosts. Kubernetes provides mechanisms for application deployment, service discovery, scheduling, updating, maintenance, and scaling. However, these new layers of infrastructure complexity also add complexities for day-to-day tasks such as managing application performance, gaining visibility into services, and monitoring and troubleshooting workflows.

In addition to increased infrastructure complexity, many applications are now being rearchitected using microservices. Multiple components that provide singular functionality communicate with each other, and it is possible to distribute each service across several instances. This distribution and high workload volume of microservices make it more challenging to monitor Kubernetes environments effectively.

This e-book on OWASP Top 10 for Kubernetes provides valuable information and best practices beyond the original OWASP guidance, and that applies broadly to most Kubernetes environments. It covers the Kubernetes security basics of golden signal collection, observability, security monitoring, authentication, authorization, and vulnerability management. The e-book calls out relevant incidents that highlight the dangers of each risk and provides technical guidance on how to achieve basic mitigation. Additionally, it provides useful alerts that can notify you when something is not quite right. By addressing the OWASP Top 10 for Kubernetes security risks, organizations can ensure that their containerized environments are secure and resilient against potential cyberthreats.

OWASP Top 10 for Kubernetes Risk Assessment

One of the biggest concerns when using Kubernetes is whether you are complying with security requirements or guaranteeing an adequate security posture that takes into account all possible threats. For this reason, OWASP members created the OWASP Top 10 for Kubernetes, which helps identify the most likely risks.

OWASP Top 10 projects are useful awareness and guidance resources designed for security practitioners and engineers. They can also map to other security frameworks that help incident response engineers understand Kubernetes threats. MITRE ATT&CK techniques are also commonly used to register the attacker's techniques and help blue teams understand the best ways to protect an environment. In addition, you can check the Kubernetes threat model to understand all of the attack surfaces and main attack vectors.

The OWASP Kubernetes Top 10 puts all possible risks in an order of overall commonality or probability. In this e-book, we modified the order slightly, grouping some of the risks in the same category, such as misconfigurations, monitoring, or vulnerabilities. We also recommend some tools or techniques to audit your configuration and make sure that your security posture is the most appropriate.

What is OWASP Kubernetes?

OWASP is a nonprofit foundation that works to improve software security. Initially, OWASP focused on web application security (hence its name), but its scope has broadened over time because of the nature of modern systems design.

As applications development moves from monolithic architectures running traditionally on virtual machines hidden behind firewalls to modern-day microservice workloads running on cloud infrastructures, it's important to update the security requirements for each application environment. That's why the OWASP Foundation created the OWASP Top 10 for Kubernetes – a list of the 10 most common attack vectors specifically for the Kubernetes environment

The visual below spotlights which component or part is impacted by each of the risks that appear in OWASP Kubernetes mapped to a generalized Kubernetes threat model. This analysis also dives into each OWASP risk, providing technical details on why the threat is prominent, as well as common mitigations. It's helpful to group the risks into three categories and order of likelihood. The risk categories are:

Misconfigurations

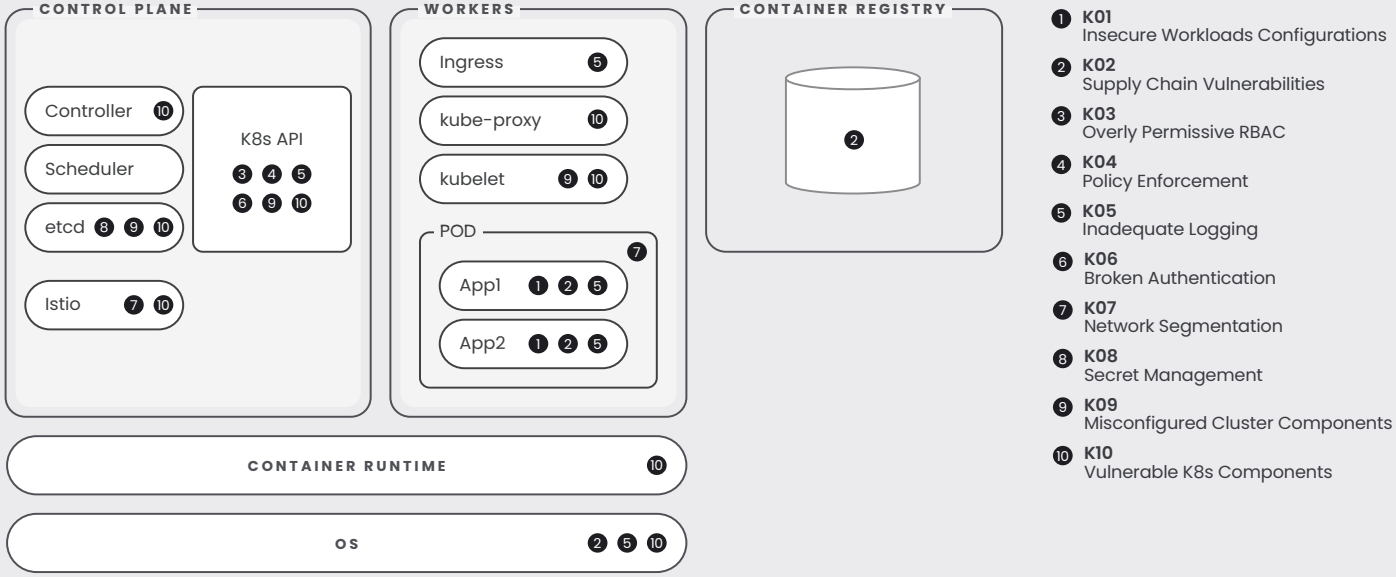
- K01:2022 Insecure Workload Configurations
- K09:2022 Misconfigured Cluster Components
- K03:2022 Overly Permissive RBAC Configurations
- K07:2022 Missing Network Segmentation Controls

Lack of visibility

- K05:2022 Inadequate Logging and Monitoring
- K04:2022 Lack of Centralized Policy Enforcement
- K08:2022 Secrets Management Failures

Vulnerability management

- K02:2022 Supply Chain Vulnerabilities
- K06:2022 Broken Authentication Mechanisms
- K10:2022 Outdated and Vulnerable Kubernetes Components

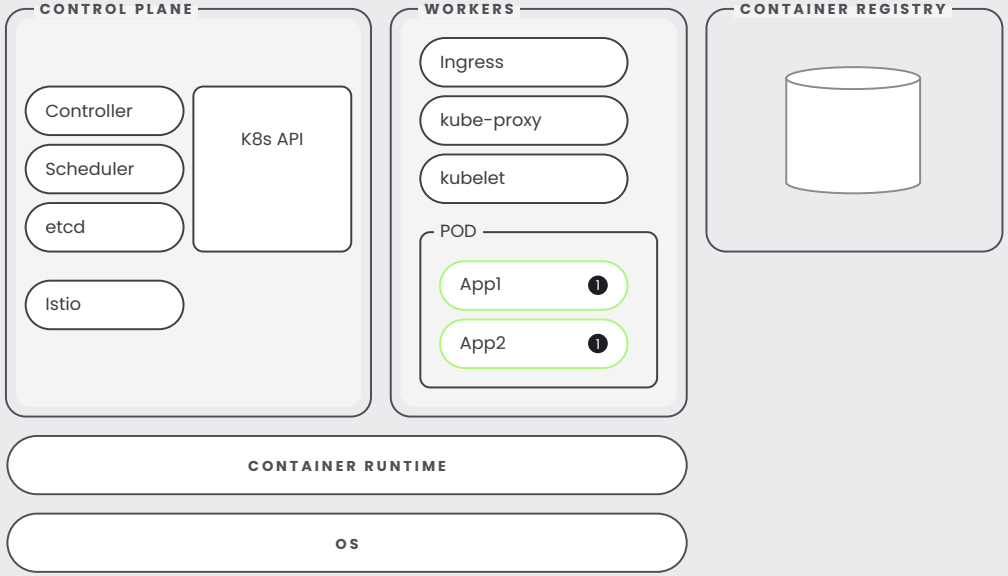


K01:2022

Insecure workload configurations

Security is at the forefront of all cloud provider offerings. Cloud service providers such as Amazon Web Services (AWS), Google Cloud Platform, and Microsoft Azure implement an array of sandboxing features, virtual firewall features, and automatic updates to underlying services in order to ensure that your business stays secure whenever and wherever possible. These measures also alleviate some of the traditional security burdens of on-premises environments. However, cloud environments apply what is known as a shared security model, which means that part of the responsibility is on the cloud service consumer to implement these security guardrails in their response environment. Responsibilities also vary based on the cloud consumption model and type of offering.

The administrators of a tenant have to ultimately ensure that workloads are using safe images, run on a patched/updated operating system (OS), and ensure the continuous auditing and remediation of infrastructure configurations. Misconfigurations in cloud-native workloads are one of the most common approaches for adversaries to gain access to your environment.



1 K01

Insecure Workloads Configurations

- App processes should not run as root
- Read-only filesystems should be used
- Privileged containers should be disallowed

Operating system

The nice thing about containerized workloads is that the images you choose often come pre-loaded with the dependencies necessary to function with your applications' base image, which is built for a particular OS.

These images pre-package some general system libraries and other third-party components that are not exactly required for the workload. And in some cases, such as within a microservices architecture (MSA), a given container image may be too bloated to facilitate a performant container that operates the microservice.

We recommend running minimal, streamlined images in your containerized workloads, such as Alpine Linux images, which are much smaller in file size. These lightweight images are ideal in most cases. Since there are fewer components packaged into it, there are also fewer possibilities for compromise. If you need additional packages or libraries, consider starting with the base Alpine image, and gradually adding packages/libraries where needed to maintain the expected behavior/performance.

Audit workloads

Consider using the Center for Internet Security (CIS) Benchmark for Kubernetes as a starting point for discovering misconfigurations. The open source project kube-bench, for instance, can check your cluster against the CIS Kubernetes benchmark using YAML files to set up the tests.

Example CIS benchmark control

Minimize the admission of root containers (5.2.6)

Linux container workloads can run as any Linux user. However, containers that run as the root user increase the possibility of container escape (privilege escalation and then lateral movement in the Linux host). The CIS benchmark recommends running all containers as a defined non-UID 0 user.

One example of a Kubernetes auditing tool that can help minimize the admission of root containers is kube-admission-webhook. This is a Kubernetes admission controller webhook that allows you to validate and mutate incoming Kubernetes API requests. You can use it to enforce security policies, such as prohibiting the creation of root containers in your cluster.

How to prevent workload misconfigurations with OPA

You can use tools such as Open Policy Agent (OPA) as a policy engine to detect these common misconfigurations. The OPA admission controller gives you high-level declarative language to author and enforce policies across your stack.

Let's say that you want to build an admission controller for the previously mentioned `alpine` image. However, one of the users of Kubernetes wants to set the securityContext to privileged=true.

```
- rule: DB program spawned process
  desc: >
   a database-server related program spawned a new process other
than itself.
   This shouldn\'t occur and is a follow on from some SQL
injection attacks.
 condition: >
   proc.pname in (db_server_binaries)
   and spawned_process
   and not proc.name in (db_server_binaries)
   and not postgres_running_wal_e
   and not user_known_db_spawned_processes
 output: >
   Database-related program spawned process other than itself
(user=%user.name user_loginuid=%user.loginuid
   program=%proc.cmdline pid=%proc.pid parent=%proc.pname
container_id=%container.id image=%container.image.repository)
 priority: NOTICE
 tags: [host, container, process, database, mitre_execution, T1190]
```

This is an example of a privileged pod in Kubernetes. Running a pod in privileged mode means that the pod can access the host's resources and kernel capabilities. To prevent privileged pods, the .rego file from the OPA Gatekeeper admission controller should look something like this:

```
package kubernetes.admission
deny[msg] {
    c := input_containers[_]
    c.securityContext.privileged
    msg := sprintf("Privileged container is not allowed: %v,
securityContext: %v",
[c.name, c.securityContext])
}
```

In this case, the output should look something like this:

```
Error from server (Privileged container is not allowed: alpine,
securityContext: {"privileged": true}): error when creating "STDIN":
admission webhook "validating-webhook.openpolicyagent.org"
```

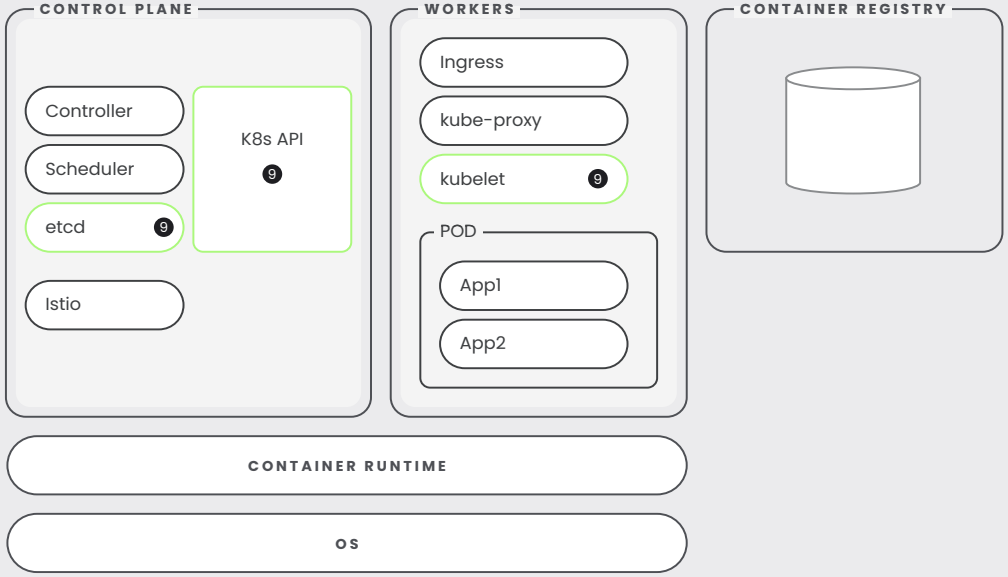
K09:2022

Misconfigured cluster components

Misconfigurations in core Kubernetes components are much more common than expected. Continuous and automatic auditing of infrastructure-as-code (IaC) and Kubernetes (YAML) manifests instead of checking them manually will reduce configuration errors.

One of the riskiest misconfigurations is the Anonymous Authentication setting in Kubelet, which allows nonauthenticated requests to the Kubelet. We strongly recommend checking your Kubelet configuration and ensuring that the flag described below is set to `false`.

When auditing workloads, it's important to keep in mind that there are different ways in which to deploy an application. With the configuration file of the various cluster components, you can authorize specific read/write permissions on those components. In the case of Kubelet, by default, all requests to the Kubelet's HTTPS endpoint that are not rejected by other configured authentication methods are treated as anonymous requests, and given a username of system:anonymous and a group of system:unauthenticated.



1 K09

Misconfigured Cluster Components

- kubelet
- etcd
- kube-apiserver

To disable the anonymous access for these unauthenticated requests, simply start Kubelet with the feature flag --anonymous-auth=false. When auditing cluster components like Kubelet, you can see that Kubelet authorizes API requests using the same request attributes approach as the API server. As a result, you can define the permissions such as:

- POST
- GET
- PUT
- PATCH
- DELETE

However, there are many other cluster components to focus on, not just Kubelet. For instance, kubectl plug-ins run with the same privileges as the kubectl command itself, so if a plug-in is compromised, it could potentially be used to escalate privileges and gain access to sensitive resources in your cluster.

Based on the CIS benchmark report for Kubernetes, we would recommend enabling the following settings for all cluster components.

etcd

The etcd database offers a highly available key/value store that Kubernetes uses to centrally house all cluster data. It is important to keep etcd safe, as it stores config data as well as Kubernetes Secrets. We strongly recommend regularly backing up etcd data to avoid data loss.

Thankfully, etcd supports a built-in snapshot feature. The snapshot can be taken from an active cluster member with the etcdctl snapshot save command. Taking the snapshot will have no performance impact. Here is an example of taking a snapshot of the keyspace served by $ENDPOINT to the file snapshotdb:

```
ETCDCTL_API=3 etcdctl --endpoints $ENDPOINT snapshot
save snapshotdb
```

kube-apiserver

The Kubernetes API server validates and configures data for API objects, which include pods, services, ReplicationControllers, and others. The API server services representational state transfer (REST) operations and provides the front end to the cluster's shared state through which all other components interact. It's critical to cluster operation and has high value, as an attack target it can't be understated. From a security standpoint, all connections to the API server, communication made inside the control plane, and communication between the control plane and kubelet components should only be provisioned to be reachable using Transport Layer Security (TLS) connections.

By default, TLS is unconfigured for the kube-apiserver. If this is flagged within the kube-bench results, simply enable TLS with the feature flags `--tls-cert-file=[file]` and `--tls-private-key-file=[file]` in the kube-apiserver. Since Kubernetes clusters tend to scale up and scale down regularly, we recommend using the TLS bootstrapping feature of Kubernetes. This allows automatic certificate signing and TLS configuration inside a Kubernetes cluster, rather than following the above manual workflow.

It is also important to regularly rotate these certificates, especially for long-lived Kubernetes clusters. Fortunately, there is automation to help rotate these certificates in Kubernetes v.1.8 or higher versions. You should also authenticate API server requests, which we cover later in the Broken Authentication Mechanisms section.

CoreDNS

CoreDNS is a DNS server technology that can serve as the Kubernetes cluster DNS and is hosted by the Cloud Native Computing Foundation (CNCF). CoreDNS superseded kube-dns since version v.1.11 of Kubernetes. Name resolution within a cluster is critical for locating the orchestrated and ephemeral workloads and services inherent in Kubernetes.

CoreDNS addressed a bunch of security vulnerabilities found in kube-dns, specifically in dnsmasq (the DNS resolver). This DNS resolver was responsible for caching responses from SkyDNS, the component responsible for performing the eventual DNS resolution services.

Aside from addressing security vulnerabilities in kube-dns's dnsmasq feature, CoreDNS addresses performance issues in SkyDNS. When using kube-dns, it also involves a sidecar proxy to monitor health and handle the metrics reporting for the DNS service.

CoreDNS addresses a lot of these security- and performance-related issues by providing all of the functions of kube-dns within a single container. However, it can still be compromised. As a result, it's important to again use kube-bench for compliance checks on CoreDNS.

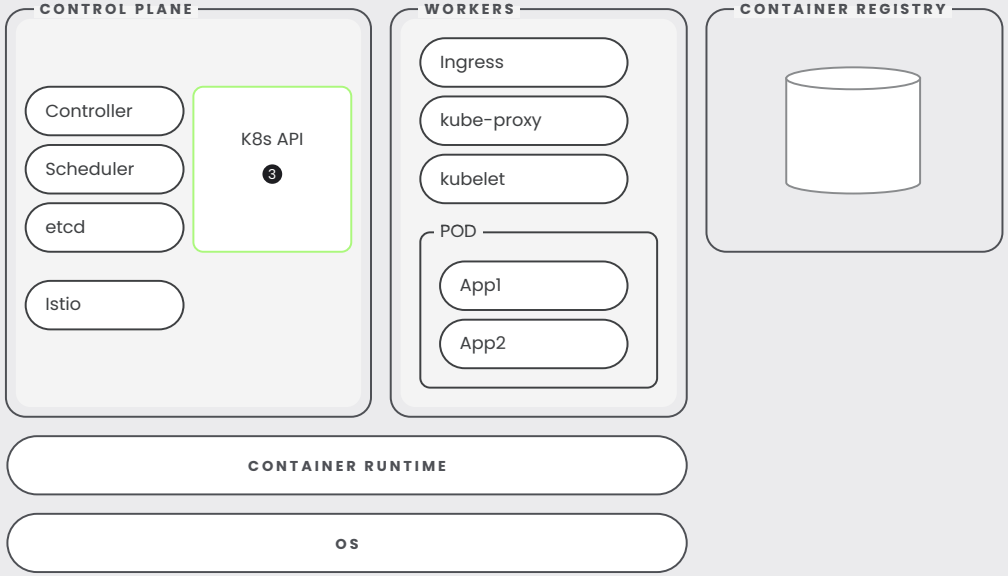
K03:2022

Overly permissive RBAC configurations

Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within your organization. A RBAC misconfiguration could allow an attacker to elevate privileges and gain full control of the entire cluster.

Creating RBAC rules is rather straightforward. For instance, to create a permissive policy to allow read-only create, read, update, delete (CRUD) actions (i.e., get, watch, list) for pods in the Kubernetes cluster's default network namespace, but to prevent create, update, or delete actions against those pods, the policy would look something like this:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
Metadata:
  namespace: default
  name: pod-reader
Rules:
- apiGroups: [“”] # “” indicates the core API group
  resources: [“pods”]
  verbs: [“get”, “watch”, “list”]
```



1 K03
Overly Permissive RBAC

Issues arise when managing these RBAC rules in the long run. Admins will likely need to manage ClusterRole resources to avoid building individual roles on each network namespace, as seen above. ClusterRoles allow you to build cluster-scoped rules for granting access to those workloads.

You can then use RoleBindings to bind the above-mentioned roles to users.

Similar to other identity and access management (IAM) practices, you will need to ensure that each user has the correct access to resources within Kubernetes without granting excessive permissions to individual resources. The manifest below shows how we recommend binding a role to a service account or user in Kubernetes:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
Metadata:
  name: read-pods
  namespace: default
Subjects:
- kind: User
  name: nigeldouglas
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

By scanning for RBAC misconfigurations, you can proactively bolster the security posture of your cluster and simultaneously streamline the process of granting permissions. One of the major reasons cloud-native teams grant excessive permissions is because of the complexity of managing individual RBAC policies in production. In other words, there may be too many users and roles within a cluster to manage by manually reviewing manifest code. That's why there are tools specifically built to handle the management, auditing, and compliance checks of your RBAC.

RBAC Audit

RBAC Audit is a tool created by the team at CyberArk. This tool is designed to scan the Kubernetes cluster for risky roles within RBAC and requires Python 3.0. This python tool can be run via a single command:

```
ExtensiveRoleCheck.py --clusterRole clusterroles.
json  --role Roles.json --rolebindings rolebindings.json
--cluseterolebindings clusterrolebindings.json
```

The output should look somewhat similar to:

```
[*] Started enumerating risky ClusterRoles:
[!][ClusterRole]-> Cluster-pod-creator Has permission to create pods!
[!][ClusterRole]-> Cluster-Secret-reder Has permission to
list secrets!
[!][ClusterRole]-> resource-reader Has permission to use get on
any resource!
[!][ClusterRole]-> nginx-1b-nginx-ingress Has permission to
list secrets!
[!][ClusterRole]-> prometheus-adapter-server-resources Has
Admin-Cluster permission!
[!][ClusterRole]-> prometheus-kube-state-metrics Has permission to
list secrets!
[!][ClusterRole]-> prometheus -prometheus-oper-operator Has
permission to access statefulsets with any verb!
[!][ClusterRole]-> prometheus -prometheus-oper-operator Has
permission to list secrets!
[!][ClusterRole]-> prometheus-prometheus-oper-operator Has permission
to access secrets with any verb!
[*] Started enumerating risky Roles:
[!][Role]-> nginx-1b-nginx-ingress Has permission to list secrets!
[!][Role]-> kubesystem-pod-creator Has permission to create pods!
[!][Role]-> default-admin Has Admin-Cluster permission!
[!][Role]-> res-reader Has permission to use get on any resource!
[!][Role]-> Random-user Has permission to use get on any resource!
[!][Role]-> local-secret-reader Has permission to list secrets!
[*] Started enumerating risky ClusterRoleBinding:
[!][ClusterRoleBinding]-> nginx-1b-nginx-ingress is binded to
nginx-lb-nginx-ingress ServiceAccount.
[!][ClusterRoleBinding]-> sal-resources is binded to
sal ServiceAccount.
[!][ClusterRoleBinding]-> secret-reader is binded to
sa-secret-reader ServiceAccount.
[!][ClusterRoleBinding]-> sa-pod-creator is binded to
sa-pod-creator ServiceAccount.
[!][ClusterRoleBinding]-> prometheus-adapter-hpa-controller is
binded to prometheus-adapter ServiceAccount.
[!][ClusterRoleBinding]-> prometheus-kube-state-metrics is binded
to prometheus-kube-state-metrics ServiceAccount.
[!][ClusterRoleBinding]-> prometheus-prometheus-oper-operator is
binded to prometheus-prometheus-oper-operator ServiceAccount.
[*] Started enumerating risky RoleRoleBindings:
[!][RoleBinding]-> nginx-1b-nginx-ingress is binded to
nginx-1b-nginx-ingress ServiceAccount.
[!J[RoleBinding]-> local-secret is binded to
kubesystem-secret-reader ServiceAccount.
[*] Started enumerating risky Roles:
[*] [Role] -> default-admin Has Admin-Cluster permissions!
[*] Started enumerating risky ClusterRoles:
[!] [ClusterRole]-> Cluster-Secret-reader Has permission to
list secrets!
```

```
[*] Started enumerating risky ClusterRoleBindings:
[!] [ClusterRoleBinding]-> secret-reader Is Binded to
sa-secret-reader ServiceAccount
[*] Started enumerating risky RoleBindings:
[!] [RoleBinding]-> nginx-1b-nginx-ingress Is binded to
nginx-lb-nginx-ingess ServiceAccount
```

Kubiscan

Kubiscan is another tool built by the team at CyberArk. Unlike RBAC Audit, this tool is designed for scanning Kubernetes clusters for risky permissions in the Kubernetes RBAC authorization model – not the RBAC roles. Again, Python 3.6 or higher is required for this tool to work.

Since Kubiscan is a Python tool, all commands start with 'python3' when run locally.

To see all the examples, run `python3 KubiScan.py -e` or, within the container, run `kubiscan -e`.

This table lists available Kubiscan commands.

```
# Get all risky ClusterRoles
python3 KubiScan.py --risky-clusterroles

# Get all risky Roles
python3 KubiScan.py --risky-roles

# Get all risky Roles and ClusterRoles
python3 KubiScan.py --risky-any-roles

# Get all risky RoleBindings
python3 KubiScan.py --risky-rolebindings

# Get all risky ClusterRoleBindings
python3 KubiScan.py --risky-clusterrolebindings

# Get all risky RoleBindings and ClusterRoleBindings
python3 KubiScan.py --risky-any-rolebindings

# Get all risky Subjects (Users, Groups or Service Accounts)
python3 KubiScan.py --risky-subjects

# Get all risky Pods1Containers
python3 KubiScan.py --risky-pods
```

Kubiscan
Risky Users List

```
+----------+----------------+-------------+-------------------------------------+
| Priority | Kind           | Namespace   | Name                                |
+----------+----------------+-------------+-------------------------------------+
| CRITICAL | Group          | None        | system:masters                      |
| CRITICAL | ServiceAccount | default     | kubisa                              |
| CRITICAL | ServiceAccount | kube-system | default                             |
| CRITICAL | ServiceAccount | default     | sa4                                 |
| CRITICAL | ServiceAccount | default     | risky-sa                            |
| CRITICAL | ServiceAccount | default     | root-sa2                            |
| CRITICAL | ServiceAccount | Kube-system | clusterrole-aggregation-controller  |
| HIGH     | ServiceAccount | kube-system | daemon-set-controller               |
| HIGH     | ServiceAccount | kube-system | deployment -controller              |
| CRITICAL | ServiceAccount | kube-system | generic-garbage-collector           |
| CRITICAL | ServiceAccount | kube-system | horizontal -pod-autoscaler          |
| HIGH     | ServiceAccount | kube-system | job-controller                      |
| CRITICAL | ServiceAccount | kube-system | namespace-controller                |
| CRITICAL | ServiceAccount | kube-system | persistent-volume-binder            |
| HIGH     | ServiceAccount | kube-system | replicaset-controller               |
| HIGH     | ServiceAccount | kube-system | replication-controller              |
| CRITICAL | ServiceAccount | Kube-system | resourcequota-controller            |
| HIGH     | ServiceAccount | kube-system | statefulset-controller              |
| CRITICAL | User           | None        | system:kube-controller -manager     |
| CRITICAL | ServiceAccount | default     | root-sa                             |
| CRITICAL | ServiceAccount | kube-system | bootstrap-signer                    |
| CRITICAL | ServiceAccount | kube-system | token-cleaner                       |
+----------+----------------+-------------+-------------------------------------+
```

Krane

Krane is a static analysis tool for Kubernetes RBAC. Similar to Kubiscan, it identifies potential security risks in Kubernetes RBAC design and makes suggestions on how to mitigate them. The major difference between these tools is the way Krane provides a dashboard of the cluster's current RBAC security posture and lets you navigate through its definition.

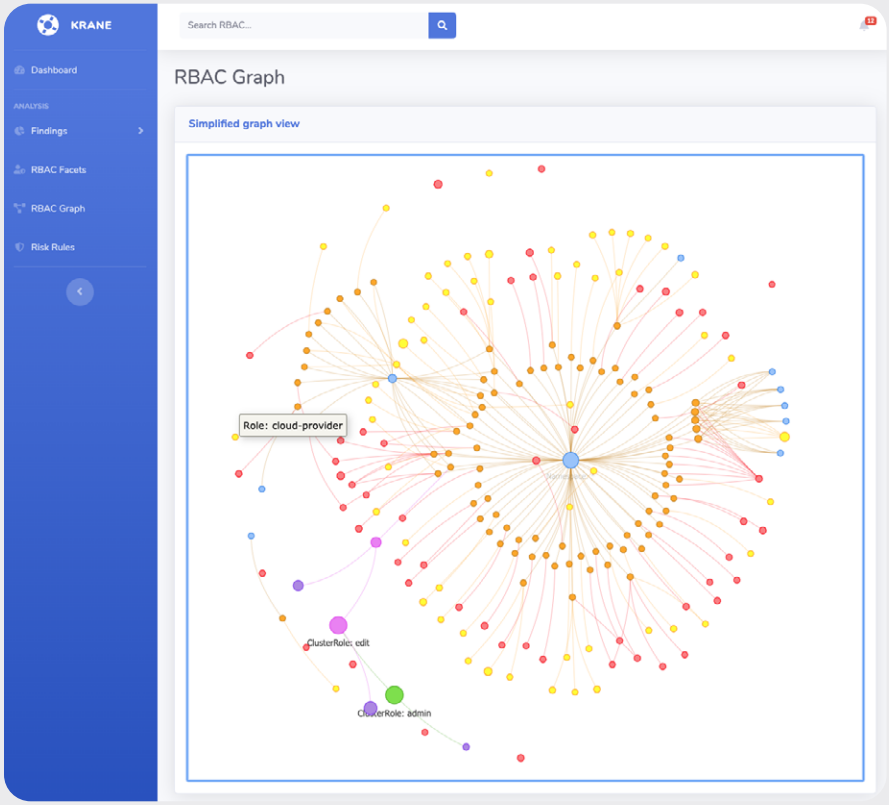
If you'd like to run an RBAC report against a running cluster, you must provide a kubectl context, as shown below:

```
krane report -k <kubectl-context>
```

If you'd like to view your RBAC design in the tree design above, with a network topology graph and the latest report findings, you need to start dashboard server via this command:

```
krane dashboard -c nigel-eks-cluster
```

The `-c` feature flag points to a cluster name in your environment. If you would like a dashboard of all clusters, simply drop the `-c` reference from the above command.



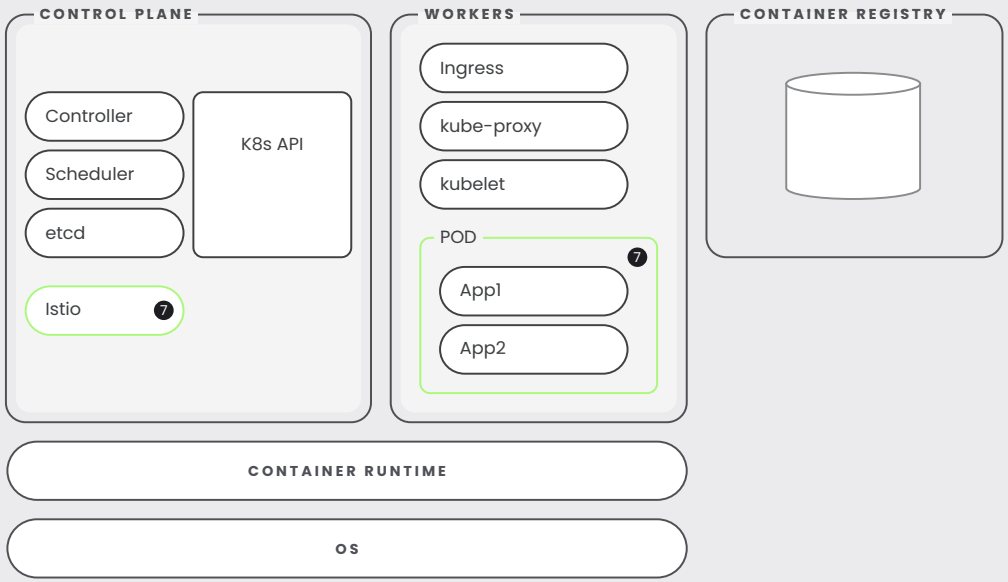
K07:2022

Missing network segmentation controls

Kubernetes, by default, defines what is known as a “flat network” design.

This allows workloads to freely communicate between each other without any prior configuration. However, they can do this without any restrictions. If an attacker were able to exploit a running workload, they would essentially have access to perform data exfiltration against all other pods in the cluster. Cluster operators that are focused on a zero trust architecture in their organization will want to take a closer look at Kubernetes network policy to ensure properly restricted services.

Kubernetes offers solutions to address the right configuration of network segmentation controls. Here, we show you two of them.



1 K07

Network Segmentation

- Native Controls (Multi-Cluster)
- Native Controls (NetworkPolicies)
- Service Mesh
- CNI Plugins

Service mesh with Istio

Istio provides a service mesh solution that allows security and network teams to manage traffic flow across microservices, enforce policies, and aggregate telemetry data in order to enforce microsegmentation on network traffic going in and out of your microservices.

At the time of this writing, the service relies on implementing a set of sidecar proxies to each microservice in your cluster. However, the Istio project is looking to move to a sidecar-less approach sometime in the year.

The sidecar technology is called Envoy, which handles ingress/egress traffic between services in the cluster and from a service to external services in the service mesh architecture. The clear advantage of using proxies is that they provide a secure microservice mesh, offering functions like traffic mirroring, discovery, rich Layer-7 (L7) traffic routing, circuit breakers, policy enforcement, telemetry recording/reporting functions, and – most importantly – automatic mutual TLS (mTLS) for all communication with automatic certificate rotation.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
Metadata:
  name: httpbin
  namespace: default
Spec:
  action: DENY
  Rules:
  - from:
    - source:
        namespaces: [“prod”]
    To:
    - operation:
        methods: [“POST”]
```

The below Istio AuthorizationPolicy sets action to DENY on all requests from the prod production namespace to the POST method on all workloads in the default namespace.

This policy is incredibly useful. Unlike Calico network policies that can only drop the traffic based on the IP address and port at the L3/L4 (the network layer), the authorization policy is denying the traffic based on HTTP/S verbs such as POST/GET at L7 (the application layer). This is important when implementing a web application firewall (WAF).

Discover how Istio monitoring can help you guarantee that your Istio services are in a good shape.

CNI

It's worth noting that although there are huge advantages to a service mesh, such as encryption of traffic between workloads via mTLS as well as HTTP/S traffic controls, there are also some complexities to managing a service mesh. The use of sidecars beside each workload adds additional overhead in your cluster, as well as unwanted issues troubleshooting those sidecars when they experience issues in production.

Many organizations opt to only implement the Container Network Interface (CNI) by default. The CNI, as the name suggests, is the networking interface for the cluster. CNIs like Project Calico and Cilium come with their own policy enforcement. While Istio enforces traffic controls on L7 traffic, the CNI tends to be focused more on network-layer traffic (L3/L4).

The following CiliumNetworkPolicy, as an example, limits all endpoints with the label `app=frontend` to only be able to emit packets using Transmission Control Protocol (TCP) on port 80, to any L3 destination:

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
Metadata:
  name: "l4-rule"
Spec:
  endpointSelector:
    matchLabels:
      app: frontend
  Egress:
    - toPorts:
      - ports:
        - port: "80"
          protocol: TCP
```

We mentioned using the Istio AuthorizationPolicy to provide WAF-like capabilities at the L7/application layer. However, a distributed denial-of-service (DDoS) attack can still happen at the network layer if the adversary floods the pods/endpoint with excessive TCP/User Datagram Protocol (UDP) traffic. Similarly, it can prevent compromised workloads from speaking to known/malicious command and control (C2) servers based on fixed IPs and ports.

Do you want to dig deeper? Learn more about how to prevent a DDoS attack in Kubernetes with Calico and Falco.

K05:2022

Inadequate logging and monitoring

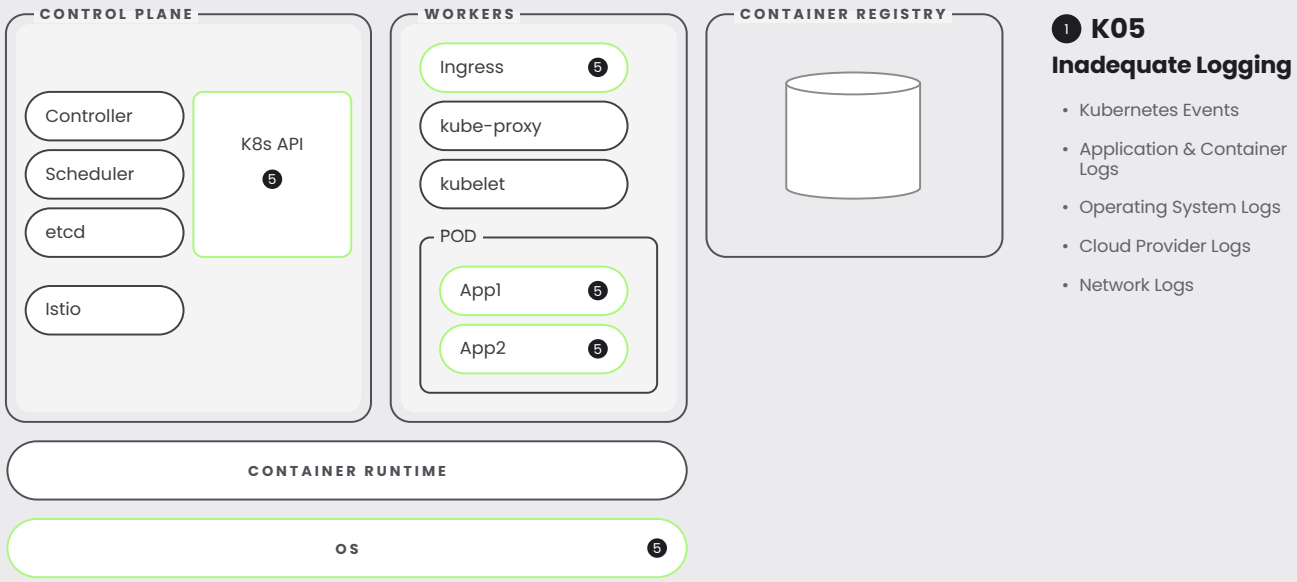
Kubernetes provides an audit logging feature by default. Audit logging shows a variety of security-related events in chronological order. These activities can be generated by users, by applications that use the Kubernetes API, or by the control plane itself.

However, there are other log sources to focus on – not limited to Kubernetes audit logs. They can include host-specific OS logs, network activity logs (such as DNS, for which you can monitor the Kubernetes add-ons CoreDNS), and cloud providers that also work as the foundation for the Kubernetes cloud.

Without a centralized tool for storing all of these sporadic log sources, you would have a hard time using them in the case of a breach. That's where tools like Prometheus, Grafana, and Falco are useful.

Prometheus

Prometheus is an open source, community-driven project for monitoring modern cloud-native applications and Kubernetes. It is a graduated member of the CNCF and has an active developer and user community.



Grafana

Like Prometheus, Grafana is an open source tool with a large community backing. Grafana allows you to query, visualize, alert on, and understand your metrics no matter where they are stored. Users can create, explore, and share dashboards with their teams.

Falco (runtime detection)

Falco, a recent graduate project of the Cloud Native Compute Foundation (CNCF) is the de facto standard for Kubernetes threat detection. Falco detects threats at runtime by observing the behavior of your applications and containers. Falco extends threat detection across cloud environments with Falco Plugins.

Falco was the first runtime security project to join the CNCF as an incubation-level project. Falco acts as a security camera, detecting unexpected behavior, intrusions, and data theft in real time in all Kubernetes environments. Falco v.0.13 added Kubernetes Audit Events to the list of supported event sources. This is in addition to the existing support for system call events. Kubernetes v1.11 introduced an improved implementation of audit events and provides a log of requests and responses to kube-apiserver.

Because almost all of the cluster management tasks are performed through the API server, the audit log can effectively track the changes made to your cluster.

Examples of this include:

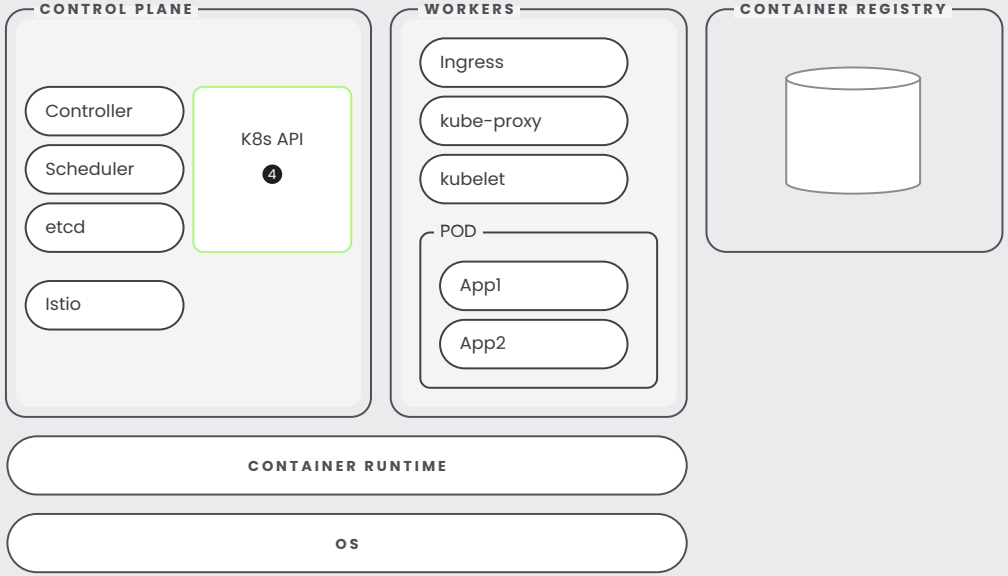
- Creating and destroying pods, services, deployments, DaemonSets, etc.
- Creating, updating, and removing ConfigMaps or secrets.
- Subscribing to the changes introduced to any endpoint.

K04:2022

Lack of centralized policy enforcement

Enforcing security policies becomes a difficult task when you need to enforce rules across multicluster and multicloud environments. By default, security teams need to manage risk across each of these heterogeneous environments separately.

There's no default way to detect, remediate, and prevent misconfigurations from a centralized location, meaning that clusters could potentially be left open to compromise.



1 K04
Policy Enforcement

Admission controller

An admission controller intercepts requests to the Kubernetes API server prior to persistence. The request must first be authenticated and authorized, and then the controller decides whether to allow the request to be performed. For example, you can create this admission controller configuration:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
  - name: ImagePolicyWebhook
    configuration:
      imagePolicy:
        kubeConfigFile: <path-to-kubeconfig-file>
        allowTTL: 50
        denyTTL: 50
        retryBackoff: 500
        defaultAllow: true
```

The `ImagePolicyWebhook` configuration is referencing a kubeconfig formatted file that sets up the connection to the back end. The point of this admission controller is to ensure that the back end communicates over TLS.

The `allowTTL: 50` sets the amount of time in seconds to cache the approval; similarly, the `denyTTL: 50` sets the amount of time in seconds to cache the denial. Admission controllers can be used to limit requests to create, delete, modify objects, or connect to proxies.

Unfortunately, the `AdmissionConfiguration` resource still needs individual managing on each cluster. If you forget to apply this file on one of your clusters, it will lose this policy condition. Thankfully, projects like OPA's Kube-Mgmt tool help manage the policies and data of OPA instances within Kubernetes instead of managing admission controllers individually.

The kube-mgmt tool automatically discovers policies and JSON data stored in ConfigMaps in Kubernetes and loads them into OPA. The feature flag `--enable-policy=false` can easily disable policies, or you could similarly disable data via a single flag: `--enable-data=false`.

Admission control is an important element of container security strategy to enforce policies that need Kubernetes context and create a last line of defense for your cluster. We touch on image scanning later in this research, but know that image scanning can also be enforced via a Kubernetes admission controller.

Runtime detection

It's important to standardize the deployment of security policy configurations to all clusters if they mirror the same configuration. In the case of radically different cluster configurations, they might require uniquely designed security policies. In either instance, how do you know which security policies are deployed in each cluster environment? That's where Falco comes into play.

Let's assume that the cluster is not using kube-mgmt, and there's no centralized way to manage the admission controllers. A user accidentally creates a ConfigMap with private credentials exposed within the ConfigMap manifest. Unfortunately, no admission controller was configured in the newly created cluster to prevent this behavior. In a single rule, Falco can alert administrators when this very behavior occurs:

```
- rule: Create/Modify Configmap With Private Credentials
  desc: >
    Detect creating/modifying a configmap containing a
private credential
  condition: kevt and configmap and kmodify
and contains_private_credentials
  output: >-
    K8s configmap with private credential
(user=%ka.user.name verb=%ka.verb

configmap=%ka.req.configmap.name namespace=%ka.target.namespace)
  priority: warning
  source: k8s_audit
  append: false
  exceptions:
  - name: configmaps
    fields:
    - ka.target.namespace
    - ka.req.configmap.name
```

This Falco rule sources the Kubernetes audit logs to show examples of private credentials that might be exposed in ConfigMaps in any namespace. The private credentials are defined as any of these conditions:

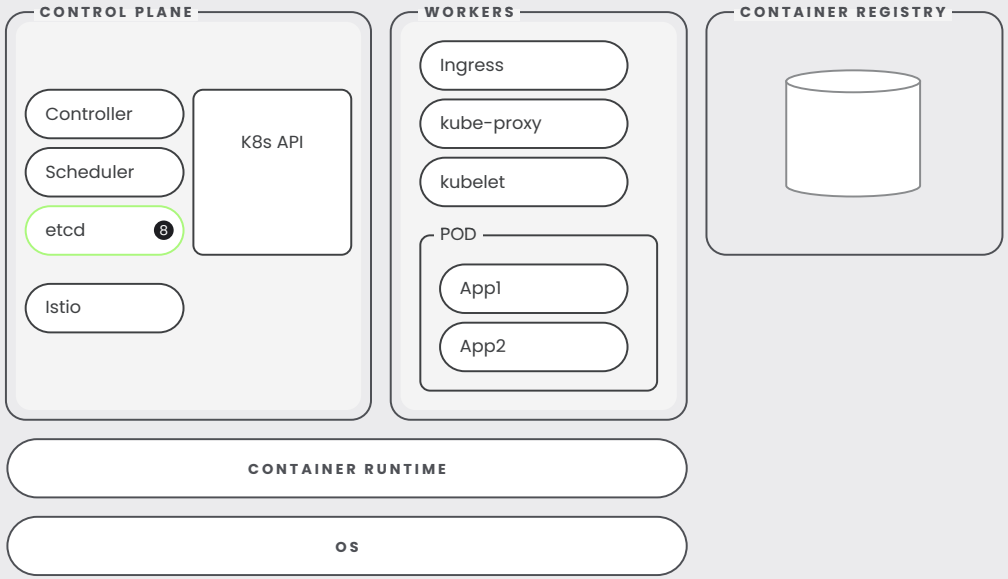
```
condition: (ka.req.configmap.obj contains "aws_access_key_id" or
ka.req.configmap.obj contains "aws-access-key-id" or
ka.req.configmap.obj contains "aws_s3_access_key_id" or
ka.req.configmap.obj contains "aws-s3-access-key-id" or
ka.req.configmap.obj contains "password" or
ka.req.configmap.obj contains "passphrase")
```

K08:2022

Secrets management failures

In Kubernetes, a Secret is an object designed to hold sensitive data, like passwords or tokens. To avoid putting this type of sensitive data in your application code, you can simply reference the Kubernetes Secret within the pod specification. This enables engineers to avoid hard coding credentials and sensitive data directly in the pod manifest or container image.

Regardless of this design, Kubernetes Secrets can still be compromised. The native Kubernetes Secrets mechanism is essentially an abstraction – the data still gets stored in the aforementioned etcd database, and it's turtles all the way down. As such, it's important for businesses to assess how credentials and keys are stored and accessed within Kubernetes Secrets as part of a broader secrets management strategy. Kubernetes provides other security controls, which include data-at-rest encryption, access control, and logging.



1 K08

Secret Management

- Encrypt secrets at rest
- Address Security Misconfigurations
- Ensure Logging and Auditing is in Place

Secrets at Rest – Encryption

One major weakness with the etcd database used by Kubernetes is that it contains all data accessible via the Kubernetes API, and therefore can allow an attacker extended visibility into secrets. That's why it's incredibly important to encrypt secrets at rest.

As of v.1.7, Kubernetes supports encryption at rest. This option will encrypt secret resources in etcd, preventing parties that gain access to your etcd backups from viewing the content of those secrets. While this feature is currently in beta and not enabled by default, it offers an additional level of defense when backups are not encrypted, or an attacker gains read access to etcd.

Here's an example of creating the EncryptionConfiguration custom resource:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
    providers:
      - aescbc:
          Keys:
            - name: key1
              secret: <BASE 64 ENCODED SECRET>
      - identity: {}
```

Addressing security misconfigurations

Aside from ensuring secrets are encrypted at rest, you need to prevent secrets from getting into the wrong hands. We discussed how vulnerability management, image scanning, and network policy enforcement can help protect applications from compromise. However, to prevent secrets (sensitive credentials) from being leaked, you should lock down RBAC wherever possible.

Keep all service account and user access to least privilege. There should be no scenario where users are "credential sharing" – essentially using a service account like "admin" or "default." Each user should have clearly defined service account names such as "Nigel," "William," or "Douglas." If a service account is doing something that it shouldn't be, you can easily audit the account activity and/or audit the RBAC configuration of third-party plug-ins and software installed in the cluster to ensure that access to Kubernetes Secrets is not granted unnecessarily to a user like Nigel, who does not require full elevated administrative privileges.

In the following scenario, a ClusterRole grants read access to secrets in the test namespace. In this case, the user assigned to this cluster role will have no access to secrets outside of this oddly specific namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
Metadata:
  name: secret-reader
  namespace: test
Rules:
- apiGroups: [“”]
  resources: [“secrets”]
  verbs: [“get”, “watch”, “list”]
```

Ensuring that logging and auditing is in place

Application logs help developers and security teams better understand what is happening inside the application. The primary use case for developers is to assist with debugging problems that affect their application's performance. In many cases, shipping logs to a monitoring solution like Grafana or Prometheus improves the time to respond to cluster events such as availability or performance issues. Most modern applications, including container engines, have some kind of logging mechanism supported by default.

The easiest and most adopted logging method for containerized applications is writing to standard output (stdout) and standard error streams. In the below example for Falco, a line is printed for each alert:

```
stdout_output:
  enabled: true
```

To identify potential security issues that arise from events, Kubernetes admins can simply stream event data like cloud audit logs or general host syscalls to the Falco threat detection engine.

By streaming the standard output (stdout) from the Falco security engine to Fluentd or Logstash, additional teams such as platform engineering or security operations can capture event data easily from cloud and container environments. Organizations can store the more useful security signals as opposed to just raw event data in Elasticsearch or other security information and event management (SIEM) solutions.

Dashboards can also be created to visualize security events and alert incident response teams:

```
10:20:22.408091526: File created below /dev by untrusted program
(user=nigel.douglas command=%proc.cmdline file=%fd.name)
```

K02:2022

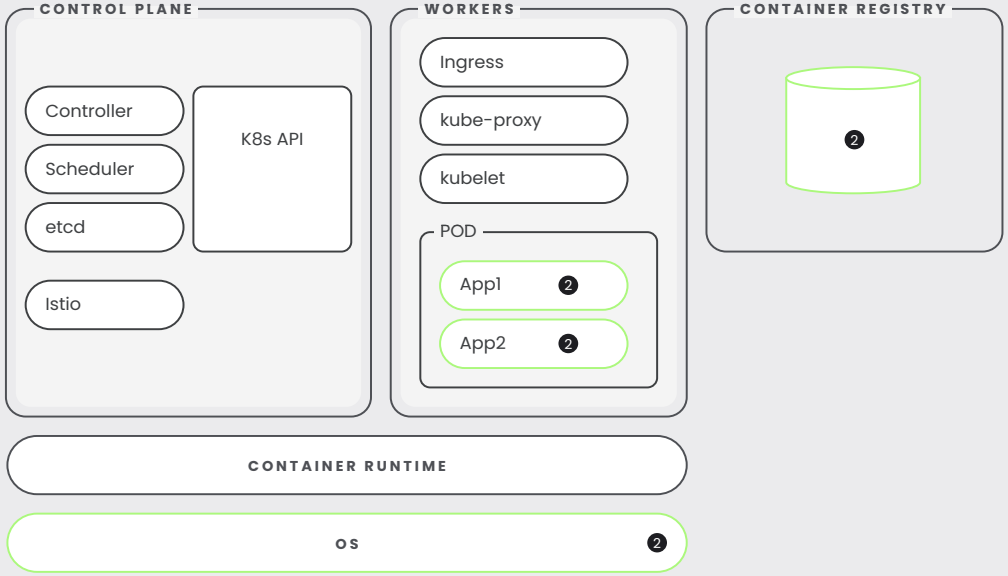
Supply chain vulnerabilities

After the four risks arising from misconfigurations, we will now detail those related to vulnerabilities.

Supply chain attacks are on the rise, as seen with the SolarWinds breach, in which their Orion software solution was compromised by the Russian threat group APT29 (commonly known as Cozy Bear). This was a long-running zero-day attack, which means that the SolarWinds customers who had Orion running in their environments were not aware of the compromise. APT29 adversaries would potentially have access to non-air-gapped Orion instances via this SolarWinds exploit.

SolarWinds is just one example of a compromised solution within the enterprise security stack. In the case of Kubernetes, a single containerized workload alone can rely on hundreds of third-party components and dependencies, making trust of origin at each phase extremely difficult. These challenges include, but are not limited to, image integrity, image composition, and known software vulnerabilities.

Let's dig deeper into each of these.



1 K02

Supply Chain Vulnerabilities

- Image Integrity
- Image Composition
- Known Software Vulnerabilities

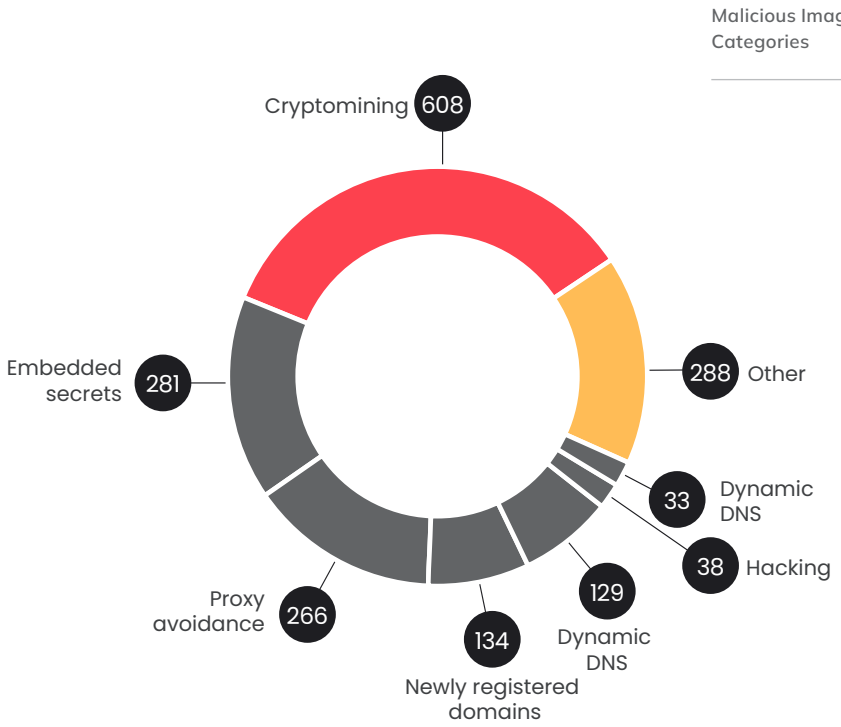
Images

A container image represents binary data that encapsulates an application and all of its software dependencies. Container images are executable software bundles that can run stand-alone (once instantiated into a running container) and make very well-defined assumptions about their runtime environment.

The Sysdig Threat Research Team (TRT) performed an analysis of over 250,000 Linux images in order to understand what kind of malicious payloads are hiding in the containers' images on Docker Hub.

The Sysdig TRT collected malicious images based on several categories, as shown above. The analysis focused on two main categories: malicious IP addresses or domains, and secrets. Both represent threats for people downloading and deploying images that are available in public registries, such as Docker Hub, exposing their environments to high risks.

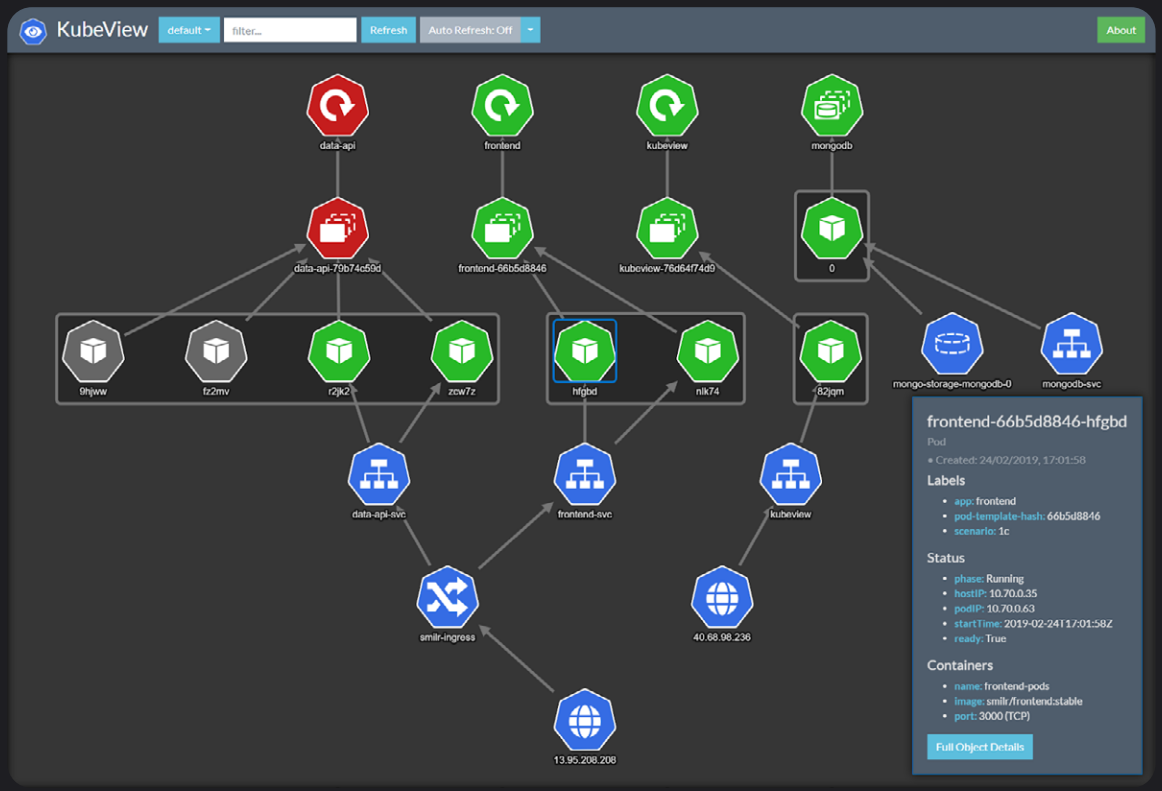
Additional guidance on image scanning can be found in the research of 12 image scanning best practices. This advice is useful whether you're just starting to run containers and Kubernetes in production, or you want to embed more security into your current DevOps workflows.



Dependencies

When you have a large number of resources in your cluster, you can easily lose track of all relationships between them. Even “small” clusters can have way more services than anticipated by virtue of containerization and orchestration. Keeping track of all services, resources, and dependencies is even more challenging when you’re managing distributed teams over multicluster or multicloud environments.

Kubernetes doesn’t provide a mechanism by default to visualize the dependencies between your deployments, services, or persistent volume claims (PVCs). KubeView is a great open source tool to view and audit intracluster dependencies. It maps out the API objects and how they are interconnected. Data is fetched in real time from the Kubernetes API. The status of some objects (pods, ReplicaSets, deployments) is color-coded red/green to represent their status and health.



Registry

The registry is a stateless, scalable server-side application that stores and lets you distribute container images.

Kubernetes resources that implement images such as pods, deployments, etc., will use imagePull secrets to hold the credentials necessary to authenticate to the various image registries. Like many of the problems we have discussed in this section, there's no inherent way to scan images for vulnerabilities in standard Kubernetes deployments.

But even on a private, dedicated image registry, you should scan images for vulnerabilities. But Kubernetes doesn't provide a default, integrated way to do this out of the box. You should scan your images in the continuous integration/continuous delivery (CI/CD) pipelines used to build them as part of a shift-left security approach. See the research about shift-left developer-driven security for more details.

Sysdig has authored detailed, technical guidance with examples on how to scan images for common CI/CD services, providing another layer of security to prevent vulnerabilities in your pipelines:

- Github actions
- Gitlab pipelines
- Azure pipelines
- Jenkins

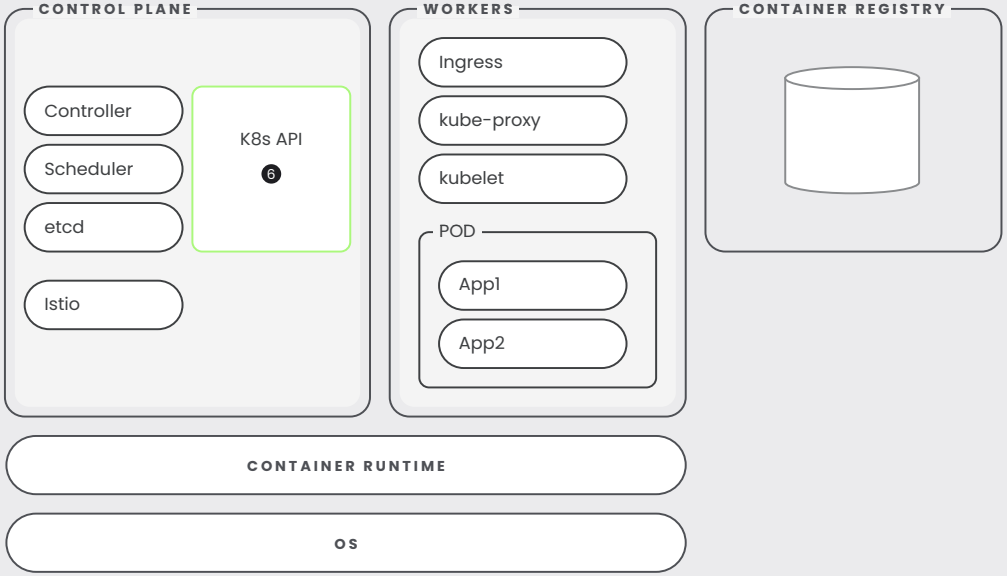
Another layer of security is a process of signing and verifying the images sent to registries or repositories. This reduces supply chain attacks by ensuring authenticity and integrity. It protects Kubernetes development and deployments, and provides better control of the inventory of containers running at any given time.

K01:2022

Broken authentication mechanisms

How to securely access your Kubernetes cluster should be a priority, and proper authentication in Kubernetes is key to avoiding most threats in the initial attack phase. Kubernetes administrators may interact with a cluster directly through Kubernetes APIs or the Kubernetes dashboard. Technically speaking, the Kubernetes dashboard in turn communicates to those APIs, such as the API server or Kubelet APIs. Enforcing authentication universally is a critical security best practice.

As seen with the Tesla cryptomining incident in 2019, the attacker infiltrated the Kubernetes dashboard, which was not protected by a password. Since Kubernetes is highly configurable, many components end up not being enabled, or use basic authentication so that they can work in a number of different environments. This presents challenges when it comes to cluster and cloud security postures.



1 K06

Broken Authentication

- Avoid using certificates for end-user authentication
- Never roll your own authentication
- Enforce MFA when possible
- Don't use Service Account tokens from outside of the cluster
- Authenticate users and external services using short-lived tokens

If a person wants to authenticate against a cluster, a main area of concern will be credentials management. The most likely case is that they will be exposed by an accidental error, leaking in one of the configuration files such as `.kubeconfig`.

Inside your Kubernetes cluster, the authentication between services and machines is based on service accounts. It's important to avoid using certificates for end-user authentication or service account tokens from outside of the cluster, because that would increase the risk. Therefore, we recommend continuously scanning for secrets or certificates that may be exposed by mistake.

OWASP recommends that, no matter what authentication mechanism you choose, **you should force humans to provide a second method of authentication**. If you use a cloud IAM capability and two-factor authentication is not enabled, for instance, you should be able to detect it at runtime in your cloud or Kubernetes environment to speed up detection and response. For this purpose, you can use Falco, an open source threat detection engine that triggers alerts at runtime according to a set of YAML-formatted rules.

```
- rule: Console Login Without Multi Factor Authentication
  desc: Detects a console login without using MFA.
  condition: >-
    aws.eventName="ConsoleLogin" and not aws.errorCode exists and
    jevt.value[/userIdentity/type]!="AssumedRole" and
    jevt.value[/responseElements/ConsoleLogin]="Success" and
    jevt.value[/additionalEventData/MFAUsed]="No"
  output: >-
    Detected a console login without MFA (requesting
user=%aws.user, requesting
    IP=%aws.sourceIP, AWS region=%aws.region)
  priority: critical
  source: aws_cloudtrail
  append: false
  exceptions: []
```

Falco helps you identify where insecure logins exist. In this case, it's a login to the AWS console without multifactor authentication (MFA). However, if an adversary were able to access the cloud console without additional authorization, they would likely be able to then access Amazon's Elastic Kubernetes Service (EKS) via the CloudShell.

That's why it's important to have MFA for cluster access, as well as the managed services powering the cluster – Google Kubernetes Engine (GKE), EKS, Azure Kubernetes Service (AKS), Intersight Kubernetes Service (IKS), etc.

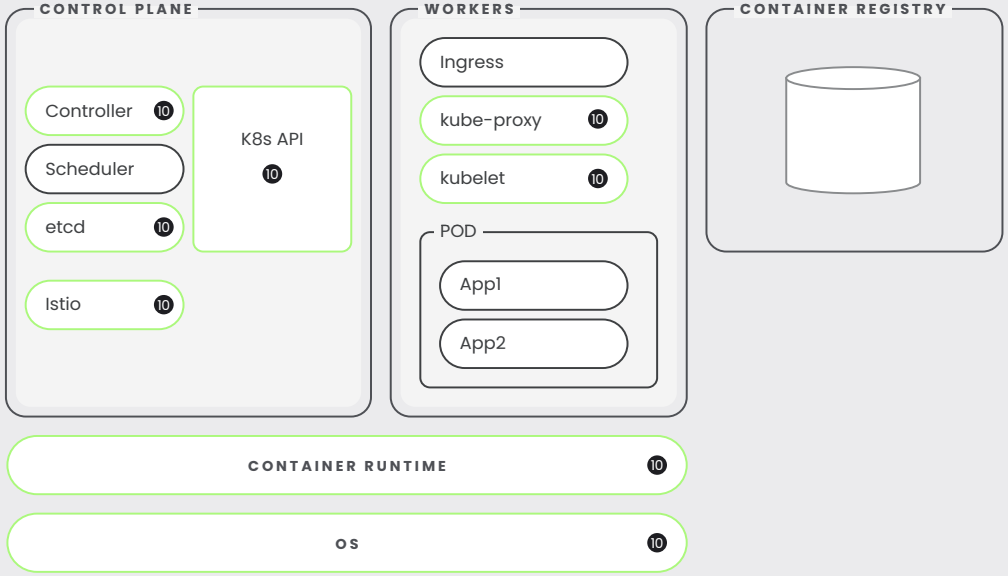
But it is not only important to protect access to Kubernetes. If you use other tools on top of Kubernetes to, for example, monitor events, you must protect those as well. As explained at KubeCon 2022, an attacker could exploit an exposed Prometheus instance and compromise your Kubernetes cluster.

K10:2022

Outdated and vulnerable Kubernetes components

Effective vulnerability management in Kubernetes is difficult. However, there are a set of best practices to follow.

Kubernetes admins must follow the latest up-to-date common vulnerabilities and exposures (CVE) databases, monitor vulnerability disclosures, and apply relevant patches where applicable. If not, Kubernetes clusters may be exposed to these known vulnerabilities, making it easier for an attacker to perform techniques that take full control of your infrastructure and potentially pivot to your cloud tenant where you've deployed clusters.



1 K10
Vulnerable K8s Components

- kubelet
- etcd
- kube-apiserver

The large number of open source components in Kubernetes, as well as the project release cadence, make CVE management particularly difficult. In v. 1.25 of Kubernetes, a new security feed was released to Alpha that groups and updates the list of CVEs that affect Kubernetes components.

Here is a list of the most famous ones:

- CVE-2021-25735 – Kubernetes validating admission webhook bypass.
- CVE-2020-8554 – Unpatched man-in-the-middle (MITM) attack in Kubernetes.
- CVE-2019-11246 – High-severity vulnerability affecting the kubectl tool. If exploited, it could lead to a directory traversal.
- CVE-2018-18264 – Privilege escalation through the Kubernetes dashboard.

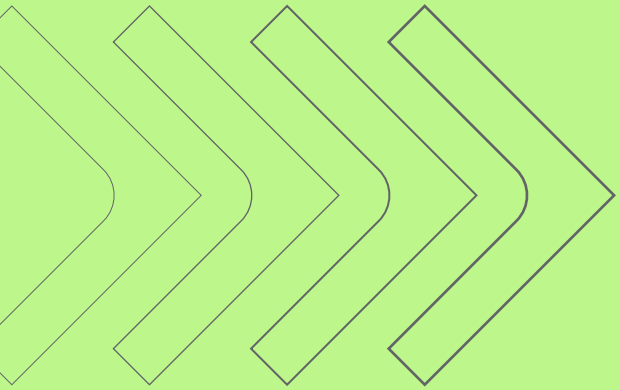
To detect these vulnerable components, you should use tools that check or scan your Kubernetes cluster, such as kubescape or kubeclarity, or look to a commercial platform offering such as Sysdig Secure.

Today, the vulnerabilities released directly target the Linux kernel, affecting the containers running on the cluster rather than the Kubernetes components themselves. Even so, you must keep an eye on each new vulnerability discovered and have a plan to mitigate the risk as soon as possible.

Conclusion

In this e-book, we presented important information and best practices for addressing the OWASP Top 10 for Kubernetes, a list of the most critical security risks impacting Kubernetes environments. Deploying and operating Kubernetes is a complex journey, let alone securing it. With the right tools and practices, you can effectively address these security risks and protect your applications. Follow the information provided in this e-book along with our companion resource, the Kubernetes Security Guide, to be well on your way to effectively monitoring and securing your Kubernetes environments.

In addition to OWASP Top 10, the "Sysdig 2023 Cloud-Native Security and Usage Report" provides valuable insights into the latest threats and trends in cloud-native environments, and can help you gain a deeper understanding of the evolving landscape and how to take proactive steps to secure your environment. Don't wait until it's too late. Stay ahead of the game with Sysdig.



See Sysdig in action.

Take the next step.

REQUEST A DEMO →

sysdig

E-BOOK

EBK-010 REV. B 4/24

About Sysdig

In the cloud, every second counts. Attacks move at warp speed, and security teams must protect the business without slowing it down. Sysdig stops cloud attacks in real time, instantly detecting changes in risk with runtime insights and open source Falco. Sysdig correlates signals across cloud workloads, identities, and services to uncover hidden attack paths and prioritize real risk. From prevention to defense, Sysdig helps enterprises focus on what matters: innovation.

Sysdig. Secure Every Second.