ALEVSK

Kubernetes Security

Attacking And Defending Modern Infrastructure

Lenin Alevski

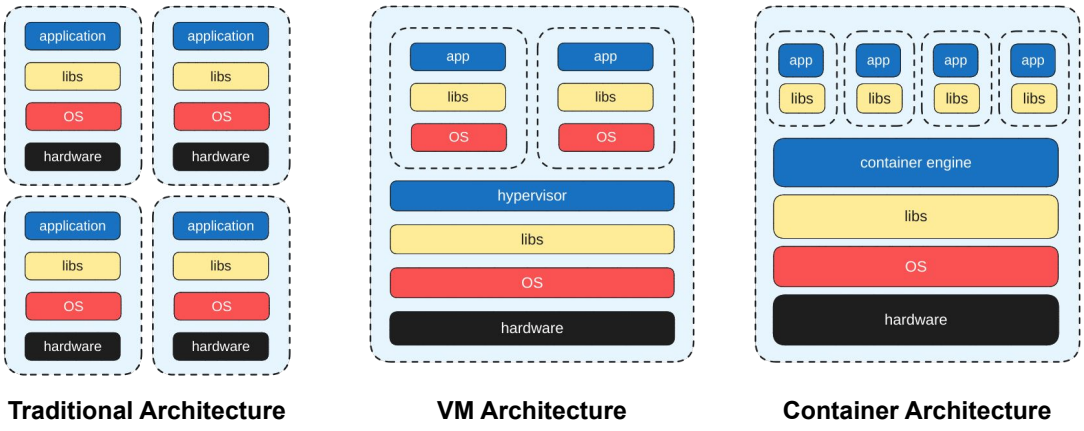
About Me

- Lenin Alevski 🇲🇽
- Security Engineer @ Google 🇺🇸
- Open Source Contributor
- Corporate & Startup world
- I’m obsessed with cybersecurity ❤️

Agenda

- Introduction to Containers
- Introduction to Kubernetes
- Kubernetes Threat Model
- Most common attack techniques in K8S
- Build-In Defenses
- Kubernetes Evolution

Application Compartmentalization



Traditional Architecture **VM Architecture** **Container Architecture**

ALEVSK

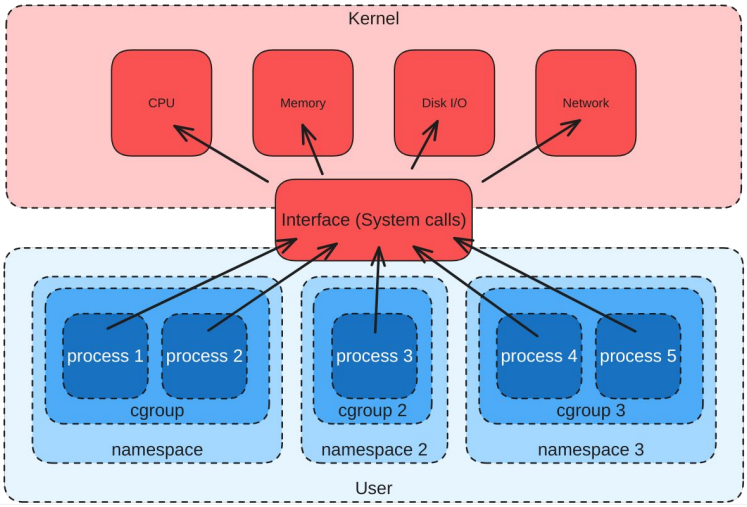
But first, let's talk about a little bit of the history of application compartmentalization.

- Traditional Architecture: In the traditional model if we want to have isolation between different applications and programs the solution was simple, put everything on separated machines, this was a straightforward solution however it come with a high cost, mostly on the hardware side. Migrating programs and regular maintenance was a challenge.
- VM Architecture: In the VM model a special type of software is introduced called the hypervisor, the hypervisor is capable of running multiple instances of an Operating System on top of the same hardware. This solution is more cost efficient compared to the previous one and offer a high level of Isolation between the different applications, however there's still some challenges when it come to the size of the virtual machines and having to manage multiple virtual operating systems.
- Container Architecture: Finally in the container model, applications packaged with all of its dependencies, libraries and configuration files together into a format called "a container". These container are run using a container runtime. Containerized applications are in general faster and cheaper to run and have many advantages in comparison to the previews models (such as size) however using containers is not a silver bullet when it comes to process isolation.

I’ve been discussing about containers and some of their benefits and drawbacks, but what if I tell you there is no container, is just another process running on your machine

So, What Containers Really Are?

- Namespaces
- Cgroups
- Capabilities



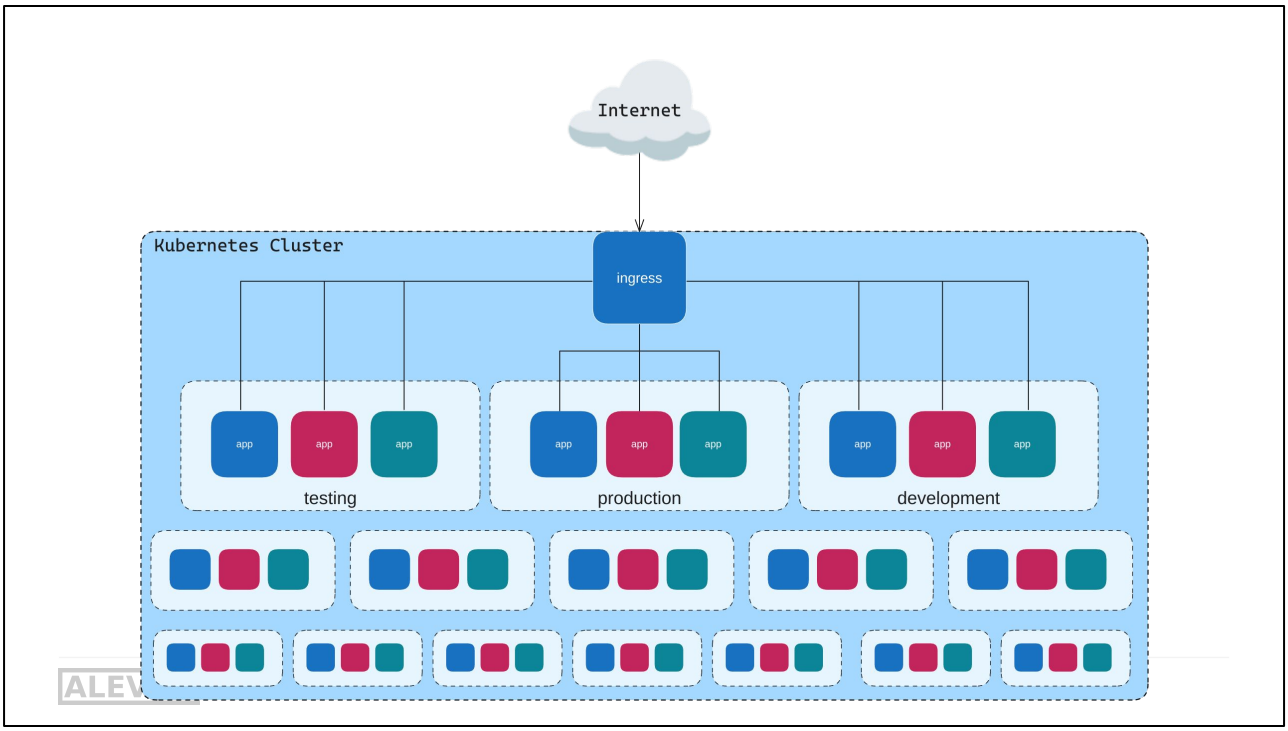
ALEVSK

So, what containers really are? Containers are a very clever operating system level virtualization for running multiple “isolated” processes on your machine. You can think of a Sandbox for processes

This isolation is achieved using multiple features of the Linux kernel, ie:

- Namespaces: controls what the process can “see”
- cgroups (short for control groups) is a Linux kernel feature that limits and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.
- Capabilities: controls the different privileges a process can have, ie: CAP_NET_ADMIN (Allows various network-related operations), CAP_SETUID (Allows changing the UID (User ID) and GID (Group ID))

What Is Kubernetes Anyways?



If Docker is a shipping container, Kubernetes is a fleet of cargo ships.

- Orchestration Tool
- Framework for building distributed systems
- Alternative way to deploy your Applications
- Operating system of the internet

As many containers as you want, at lightning speed!

Kubernetes Primitives

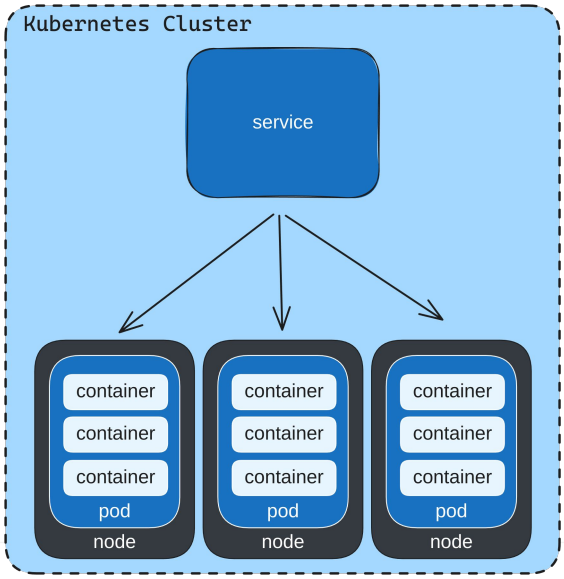
ALEVSK

Let's break it down

- Service: The way Kubernetes allows users to access your applications (the public IP address and port)
- Pod: Group of containers deployed together, they share a network namespace and can also share a file system namespace (webserver+backend+database+cache)
- Node: A virtual machine used to run pods (and by extension, containers)
- Cluster: A set of nodes that acts as the foundation of a Kubernetes application. Containers are grouped in pods which are allocated to and organized between nodes in a cluster.
- Deployment: Your cluster configuration, describes how your pods are allocated and managed

Kubernetes Application

```
apiVersion: v1
kind: Service
metadata:
  name: my-app
spec:
  selector:
   ...
---
apiVersion: apps/v1
kind: Deployment
metadata:
   ...
spec:
   ...
  Replicas: 3
  template:
   ...
   spec:
     containers:
       ...
....
....
```



You have a single service, which all the user sees.
But behind that service is any number of nodes running any number of pods wrapping any number of containers each.

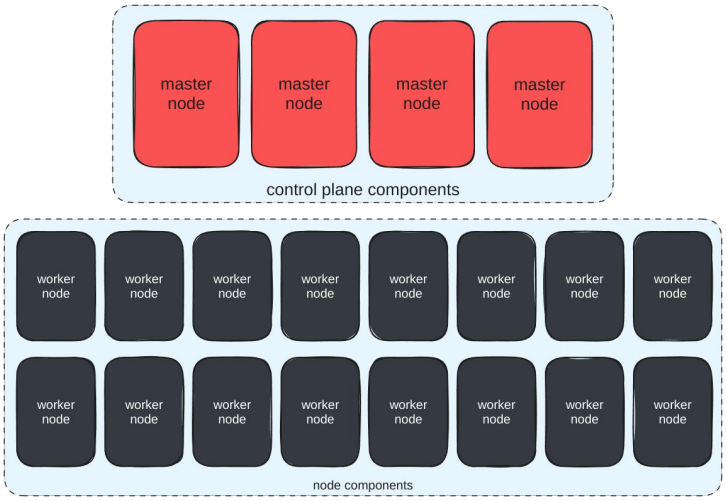
Kubernetes Components

Control Plane Components

- kube-apiserver
- etcd
- kube-scheduler
- kube-controller

Node Components

- kube-proxy
- kubelet
- container runtime



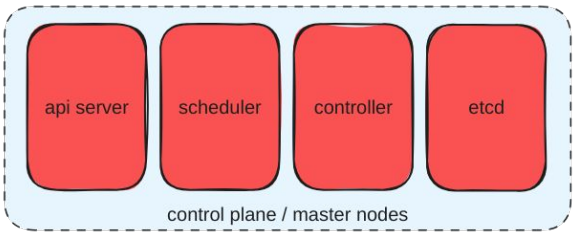
ALEVSK

Jumping down a further layer of abstraction and we have the components, divided into two sections: control plane components (one per cluster) and node components (one per node)

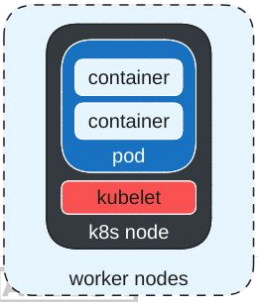
- Kube-apiserver: The Kubernetes API server is the control interface. It validates and configures data for the api objects, including pods, services, replicationcontrollers, etc. The API Server exposes REST operations and provides the frontend to the cluster's shared state through which all other components interact.
- Etcd: data storage containing configmaps, secrets, states, etc
- Kube-scheduler: Responsible for scheduling newly created pods to the best available nodes in the cluster. It keeps the application workload evenly distributed among nodes depending on resource needs.
- Kube-controller: Ensures that the current state of running pods matches the desired state of deployments/statefulset/replica-set/etc.
- Kube-proxy: Proxies traffic between services and nodes.
- Kubelet: The primary "node agent", communicates with the API server and manages the node.
- Container runtime: Software that is responsible for running containers (Docker).

https://www.educba.com/kubernetes-control-plane/

Kubernetes Threat Model: If An Attacker Controls ...



Control plane nodes: Attacker controls your cluster. They can modify, access and destroy everything



Pod / Container: Attacker controls application and may be able to escape and attack the node

Kubelet: Attacker controls running pods

Worker nodes: Attacker controls running pods. They can attack master nodes

Components have a hierarchy, and therefore a varying attack surface.
Pod/Container -> Kubelet -> Worker Node -> Control Plane Node

OWASP Kubernetes Risks Top (2022)

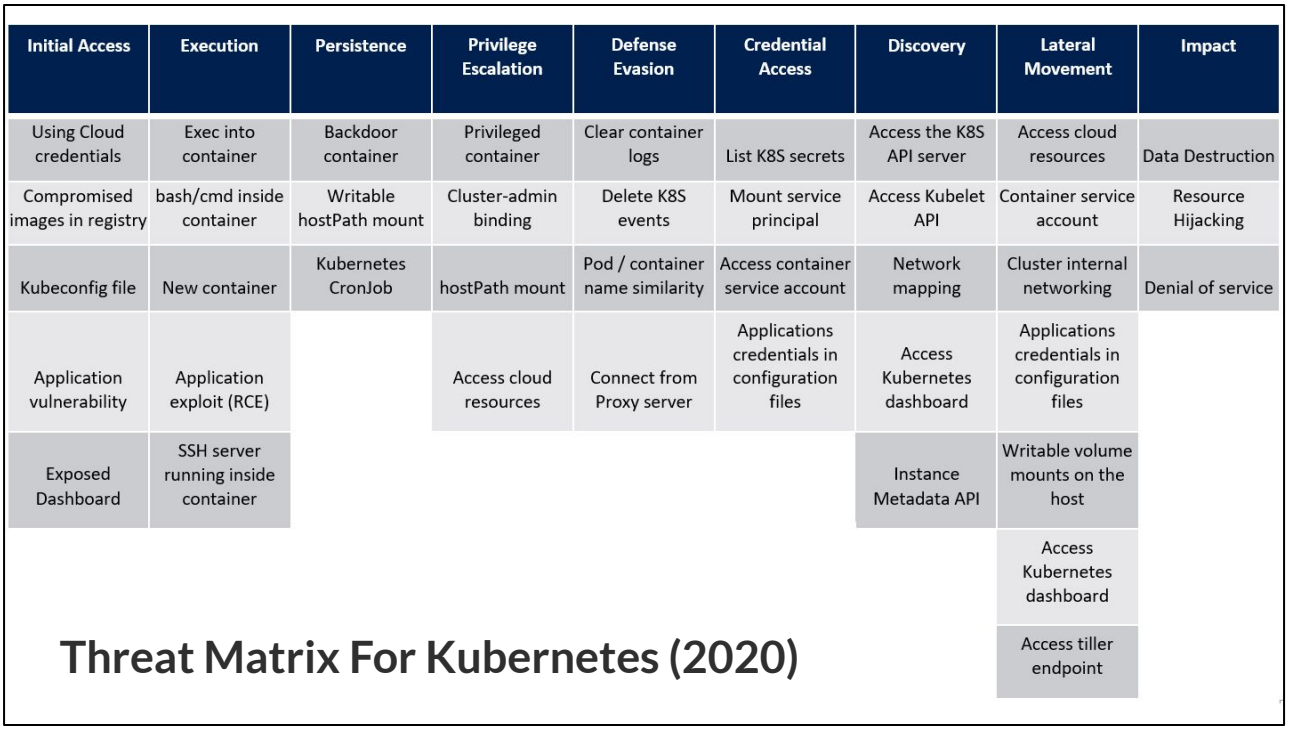
- K01: Insecure Workload Configurations
- K02: Supply Chain Vulnerabilities
- K03: Overly Permissive RBAC Configurations
- K04: Lack of Centralized Policy Enforcement
- K05: Inadequate Logging and Monitoring
- K06: Broken Authentication Mechanisms
- K07: Missing Network Segmentation Controls
- K08: Secrets Management Failures
- K09: Misconfigured Cluster Components
- K10: Outdated and Vulnerable Kubernetes Components

ALEVSK

With new technologies comes new attack surfaces, and with that comes new standards for defense.

Kubernetes Most Common Attack Techniques

ALEVSK

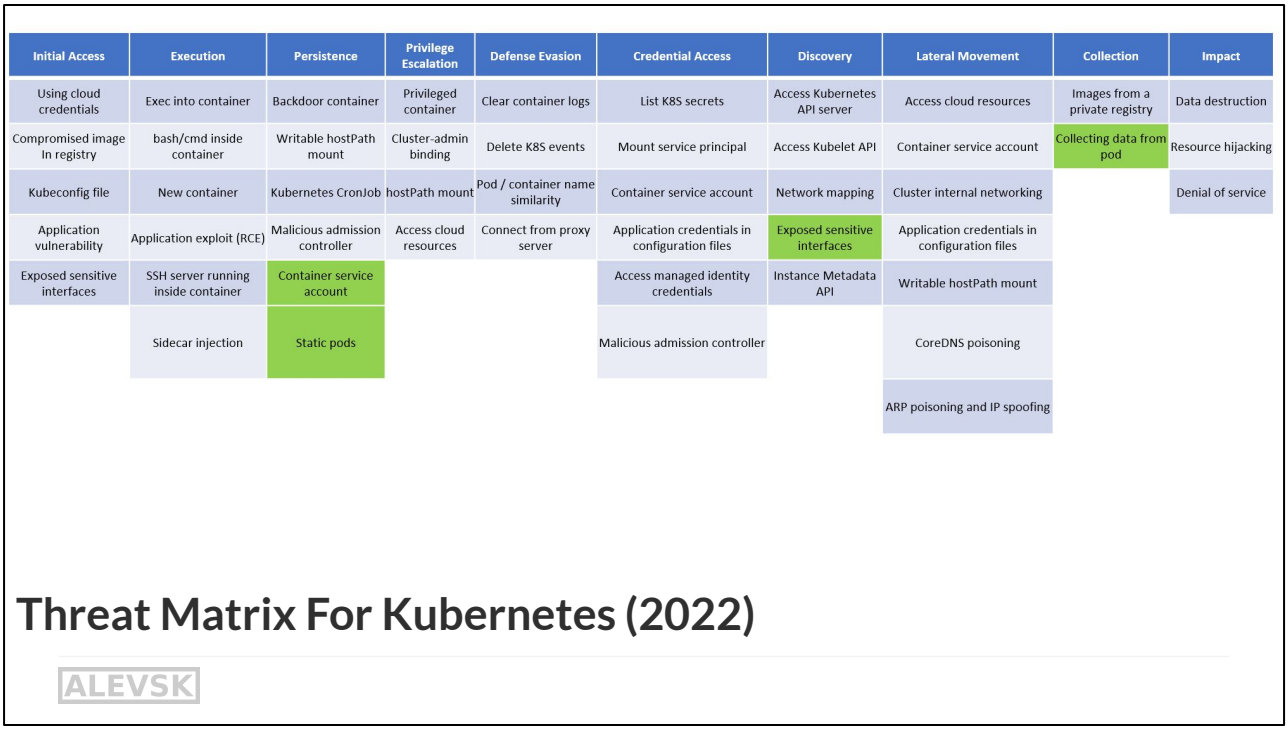


In 2020, security researchers at Microsoft published the first version of the “Threat Matrix For Kubernetes”, a MITRE ATT&CK-like matrix comprising the major techniques that are relevant to container orchestration, with focus on Kubernetes.

Although the attack techniques are different than those that target Linux or Windows systems, the tactics are actually similar, ie:

- “initial access to the computer” is similar to “initial access to the cluster”
- “malicious code on the computer” is similar to “malicious activity on the containers”
- “maintain access to the computer” is similar to “maintain access to the cluster”
- “gain higher privileges on the computer” similar to “gain higher privileges in the cluster”

https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/



Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Collection	Impact
Using cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access Kubernetes API server	Access cloud resources	Images from a private registry	Data destruction
Compromised image In registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account	Collecting data from pod	Resource hijacking
Kubeconfig file	New container	Kubernetes CronJob	hostPath mount	Pod / container name similarity	Container service account	Network mapping	Cluster internal networking		Denial of service
Application vulnerability	Application exploit (RCE)	Malicious admission controller	Access cloud resources	Connect from proxy server	Application credentials in configuration files	Exposed sensitive interfaces	Application credentials in configuration files		
Exposed sensitive interfaces	SSH server running inside container	Container service account			Access managed identity credentials	Instance Metadata API	Writable hostPath mount		
	Sidecar injection	Static pods			Malicious admission controller		CoreDNS poisoning		
							ARP poisoning and IP spoofing		

Threat Matrix For Kubernetes (2022)

ALEVSK

A second version of the matrix was published on 2022. The updated matrix add new techniques that were found by Microsoft researchers, as well as techniques that were suggested by the community. Also, several techniques that do not apply anymore were deprecated.

https://www.microsoft.com/en-us/security/blog/2022/12/07/mitigate-threats-with-the-new-threat-matrix-for-kubernetes/#:~:text=The%20threat%20matrix%20for%20Kubernetes%20can%20help%20organizations%20to%20have,required%20to%20prevent%20those%20threats
https://microsoft.github.io/Threat-Matrix-for-Kubernetes/

Initial Access

- Using cloud credentials
- Compromised images and registry
- Kubeconfig file
- Application Vulnerability
- Exposed sensitive interfaces

ALEVSK

In Kubernetes environments, the initial access tactic consists of techniques that are used for enabling attackers their first access into the cluster. This access can be achieved directly via the cluster management layer or by gaining access to a malicious or vulnerable resource that is deployed on the cluster.

Examples of this techniques include phishing cloud credentials, supply chain attacks using compromised container registries and even exposed cluster credentials on public repositories.

Using Cloud Credentials

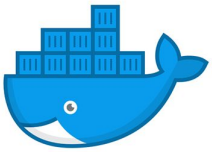
ALEVSK

When kubernetes is deployed on the public cloud (AKS, EKS, or GCP) compromised cloud credentials can lead to cluster takeover via the management console

Attackers may try to gain access to those management consoles by using different phishing and social engineering techniques. Even in cases where the cloud accounts are protected with MFA attackers can still try to trick the users into giving away access using techniques like MFA bombing or MFA spamming (MFA fatigue) and get access to those privileged accounts.

Compromised Registry And Images

- Supply Chain Attacks
- Vulnerable dependencies on images



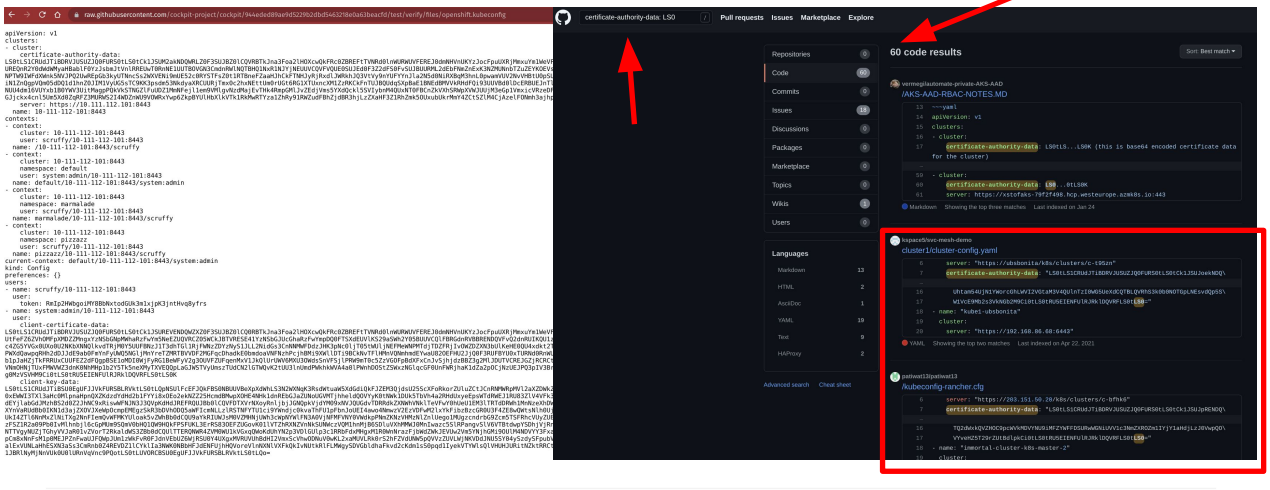
ALEVSK

Attackers who compromise a private registry can plant their own compromised images, running a compromised image could be the first step on the supply chain attack and could lead to compromise the entire cluster.

Examples of container registries:

- Docker hub
- jFrog artifactory
- Harbor

Kubeconfig File - Hunting For .kube/config Files



- If attackers get access to the ~/.kube/config credentials, for instance via a compromised developer machine, they can use them to access clusters with the same level of privileges as the user to whom the credentials belong
- These credentials are often exposed on public repositories like Github

Application Vulnerabilities

- OWASP Top 10
- SQLi
- RCE
- Command injection
- Etc

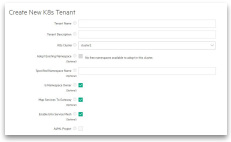
A01:2021	Broken Access Control
A02:2021	Cryptographic Failures
A03:2021	Injection
A04:2021	Insecure Design
A05:2021	Security Misconfiguration
A06:2021	Vulnerable and Outdated Components
A07:2021	Identification and Authentication Failures
A08:2021	Software and Data Integrity Failures
A09:2021	Security Logging and Monitoring Failures
A10:2021	Server-Side Request Forgery

At the end of the day we are running applications in our kubernetes clusters, running public-facing vulnerable applications (ie: OWASP Top 10) could be exploited by an attacker to gain initial access

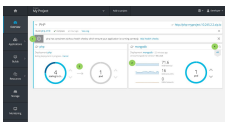
Exposed Sensitive Interfaces

RANCHER

OPENSHIFT



VMware Tanzu



ALEVSK

Finally, exposing a sensitive interface to the internet poses a security risk. Examples of such interfaces that were seen exploited include Rancher, Kubeflow, Argo Workflows, and the Kubernetes API server.

Execution

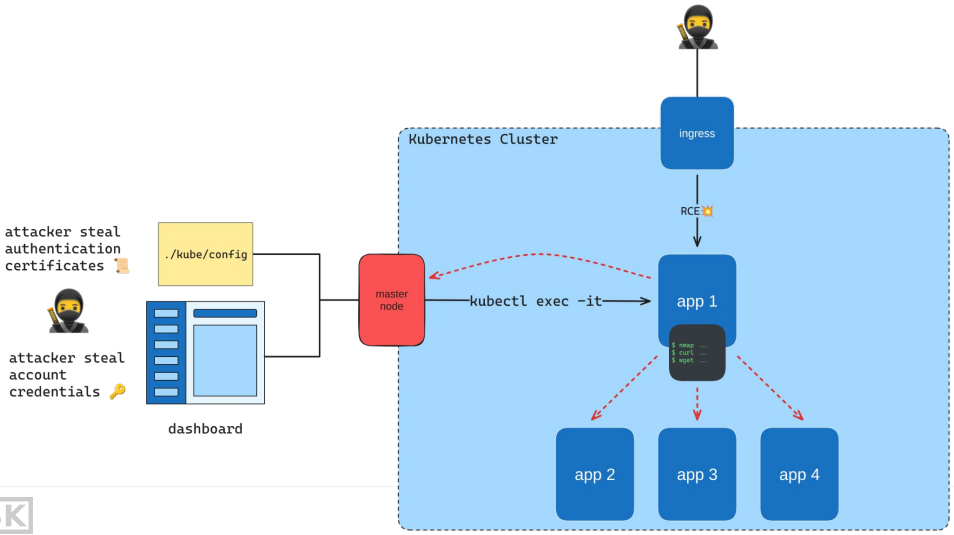
- Application exploit (RCE)
- Exec into container
- New container
- Sidecar Injection

ALEVSK

The execution tactic consists of techniques that are used by attackers to run their code inside a cluster. It allows an attacker to gain a foothold in a cluster for further exploitation.

For example, a container may be running a vulnerable application that can be exploited remotely, allowing an attacker to take over said container.

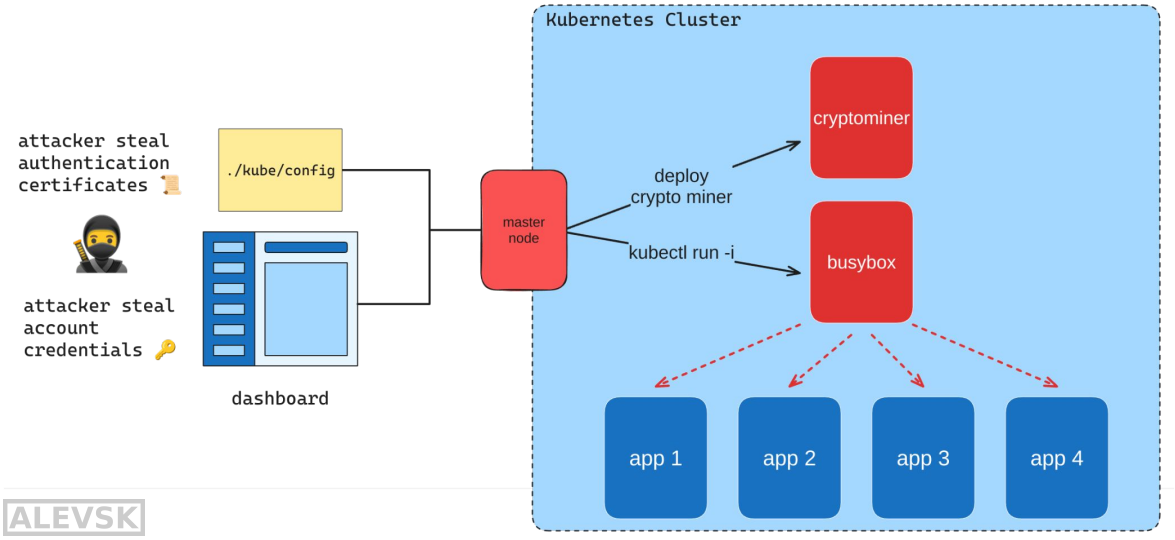
Exec Into The Container



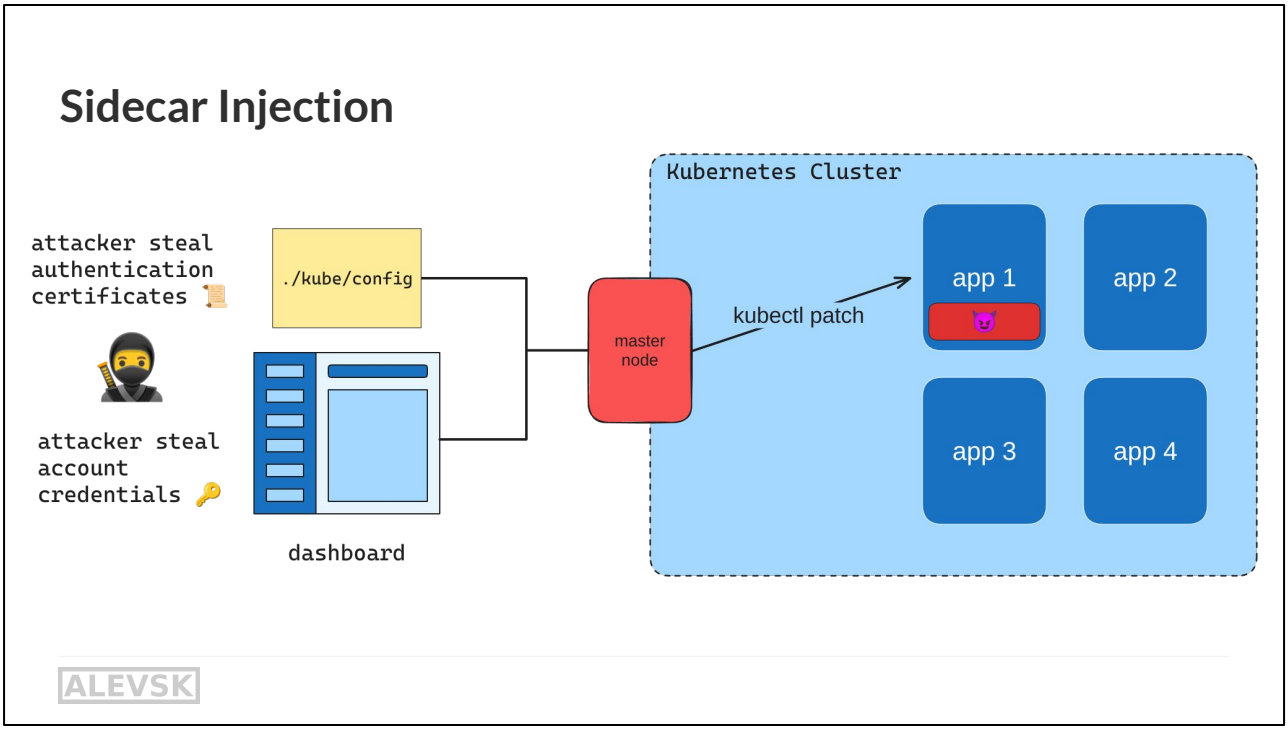
ALEVSK

If an attacker obtains permissions to connect to a container through the API, they can run malicious commands remotely using “kubectl exec”. These commands give the attacker full control of said container, allowing them to execute arbitrary code and read secret keys.

Deploying New Containers



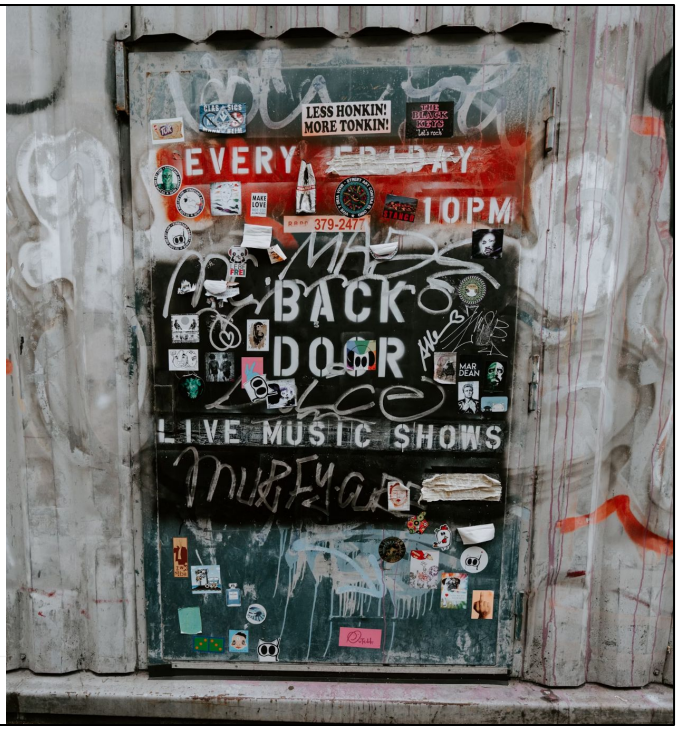
Sometimes, an attacker may have access to the kubernetes API but cannot exec into existing containers, for whatever reason (ie: scratch containers). In these cases, the attacker can simply deploy their own container to execute code and interact with internal services.



If an attacker wants to be especially sneaky, they can add a sidecar container to an existing pod in the cluster. Usually, pods have one main container, but they can also have many “sidecar” containers that run alongside it. These are typically harder to detect because they run in existing legitimate pod configurations.

Persistence

- Backdoor container
- Kubernetes CronJob
- Writable hostPath mount
- Malicious admission controller

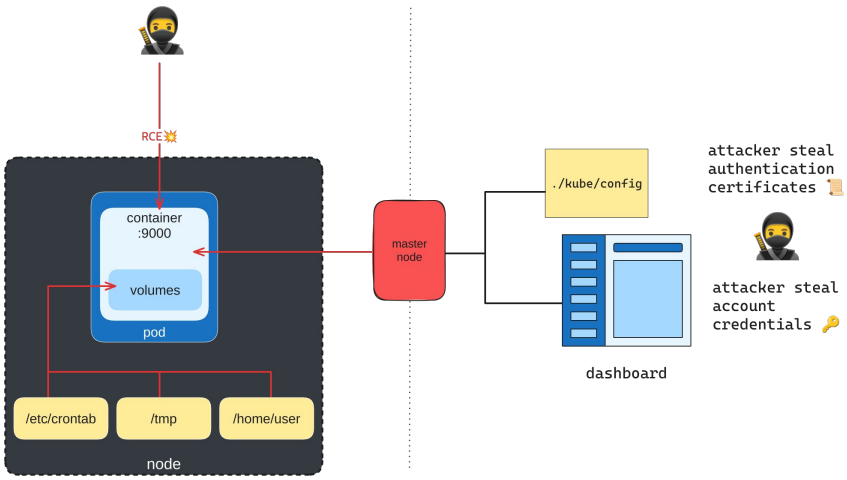


ALEVSK

The persistence tactic consists of techniques that are used by attackers to keep access to the cluster in case their initial foothold is lost.

- Backdoor container: Kubernetes is all about high availability, unfortunately that applies to attacker code as well. Attackers can use the Kubernetes controllers, such as DaemonSets or Deployments, to automatically deploy and maintain malicious containers running in the cluster.
- Kubernetes CronJob: Kubernetes Job is a controller that can make short-term, single run containers that perform specific jobs, and Kubernetes CronJob is used to schedule these jobs. Attackers can hijack CronJob and schedule their own malicious code to run in a container inside the cluster.

Writable Hostpath Mount

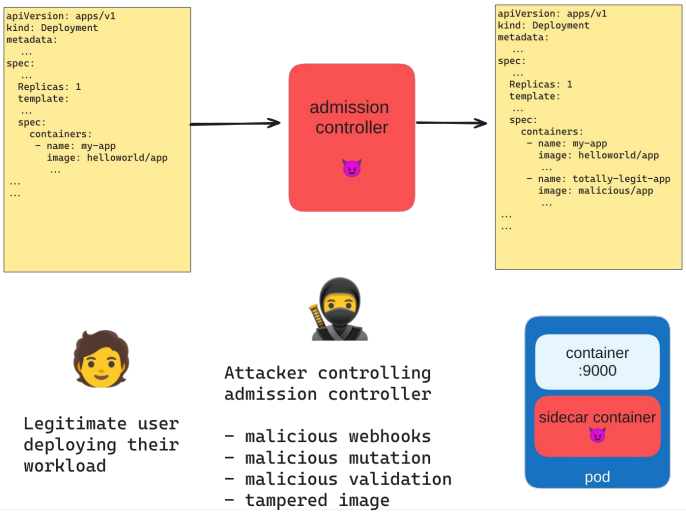


ALEVSK

A hostPath volume mounts a directory or a file from the host to the container. If an attacker can create a new container for code execution, they can potentially also create it with a writable hostPath volume. This gives the attacker access to host files and directories, which can provide persistence in a multitude of ways:

- A cronjob can be configured on the host to run malicious code
- Arbitrary commands can be added to the .bashrc file
- An SSH key can be added to the “authorized_keys” file on the host

Malicious Admission Controller

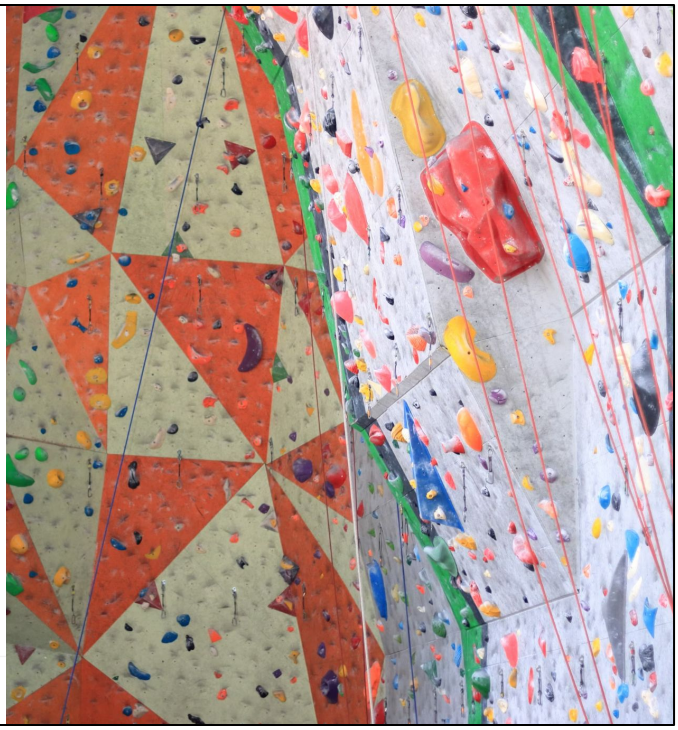


ALEVSK

The admission controller is a Kubernetes component that intercepts, and possibly modifies, requests to the Kubernetes API server. Attackers can use such webhooks for gaining persistence in the cluster. For example, attackers can intercept and modify the pod creation operations in the cluster and add their malicious container to every created pod.

Privilege Escalation

- HostPath mount
- Access cloud resources
- Privileged container
- Cluster-admin binding



ALEVSK

The privilege escalation tactic consists of techniques that are used by attackers to get higher privileges in the environment than those they currently have. In containerized environments, this can include getting access to the node from a container, gaining higher privileges in the cluster, and even getting access to the cloud resources

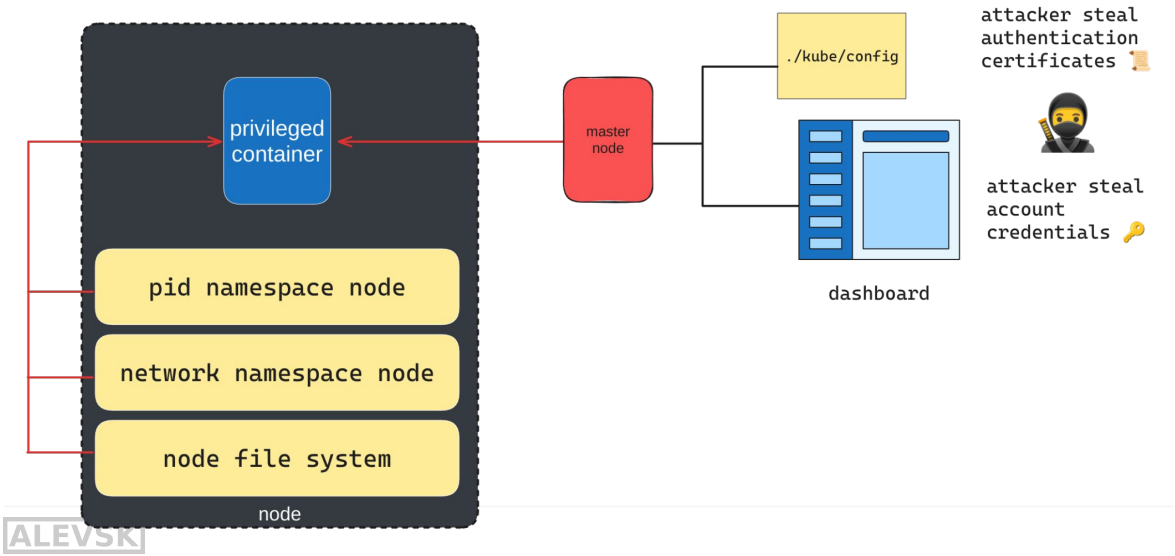
- HostPath mount: hostPath mount can be used by attackers to get access to the underlying host and thus break from the container to the host.
- Access cloud resources: depending on the cloud provider different services might be available from inside the cluster
- Privileged container: Attackers who gain access to a privileged container, or have permissions to create a new privileged container (by using the compromised pod's service account, for example), can get access to the host's resources.
- Cluster-admin binding: RBAC can restrict the allowed actions of the various identities in the cluster. Cluster-admin is a built-in high privileged role in Kubernetes

Privileged Container

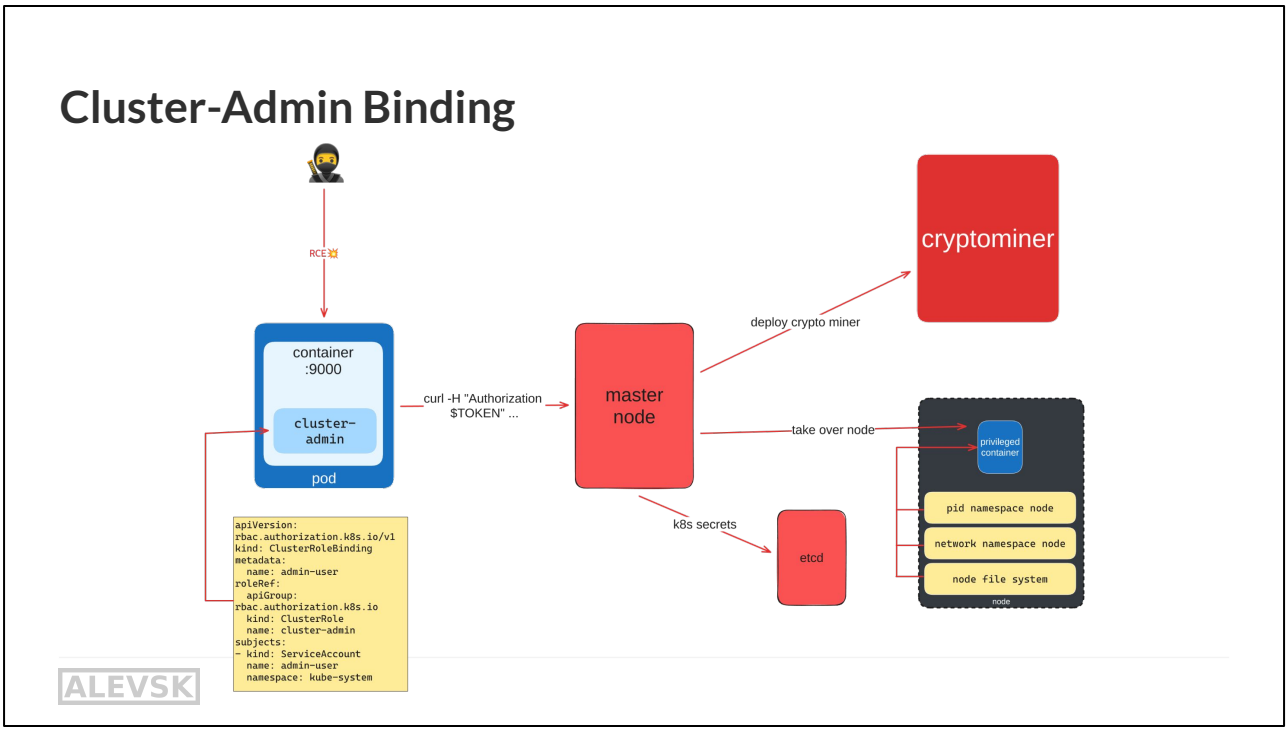
ALEVSK

- The infamous 280 characters tweet that will allow you to hack Kubernetes clusters:
- kubectl run r00t --restart=Never -ti --rm --image lol --overrides '{"spec":{"hostPID": true, "containers":[{"name":"1","image":"alpine","command":["nsenter","--mount=/proc/1/ns/mnt","--","/bin/bash"],"stdin": true,"tty":true,"securityContext":{"privileged":true}}]}}'
- Take a look at this command, take a picture and try to run it later on your kubernetes cluster if you want

A Container That Doesn't Contain Anything



- So, how the privilege escalation command works?
- Basically an attacker with enough permissions could (via the management console or using kubectl directly) deploy a privileged container in the cluster
- A privileged container would have access to all the operating system processes and the node network namespace. Additionally, the node file system could be mounted directly into the privilege container giving the attacker access to every file on the system.
- This is essentially a container that doesn't contain anything
- From there an attacker could try to move laterally to other nodes and even try to reach the control plane



- Sometimes privileged containers are already running inside the cluster. Containers running using a privilege role such as the “Cluster-admin” are high value targets for attackers
- The Cluster-admin role would be the equivalent to a “superuser” on a Linux system because it can perform any action on any resource in a cluster.
- In this example, an attacker that successfully steal the service account token associated to the cluster-admin role may be able to perform any task on the cluster, including, stealing secrets from etcd, deploying cryptominers, and take over the entire cluster.

Defense Evasion

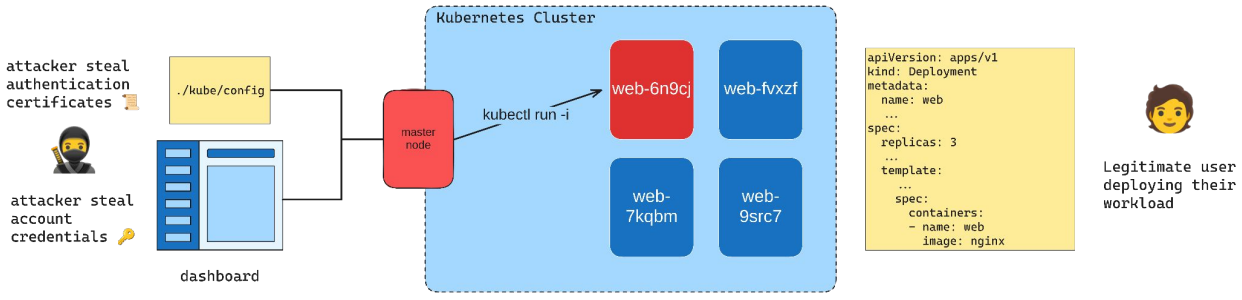
- Clear container logs
- Delete Kubernetes events
- Pod / Container name similarity
- Connect from Proxy server

ALEVSK

The defense evasion tactic consists of techniques that are used by attackers to avoid detection and hide their activity.

- Clear container logs: Attackers can delete the application or OS logs on a compromised container, hiding their container activity.
- Delete Kubernetes events: Kubernetes events track changes that occur inside the cluster. Attackers can delete these events (“kubectl delete events–all”) to hide their cluster activity.
- Pod / Container name similarity: Attackers can name their pods as if they were created by the existing controllers. For example, an attacker could create a malicious pod named coredns-{random suffix} which would look related to the CoreDNS Deployment.

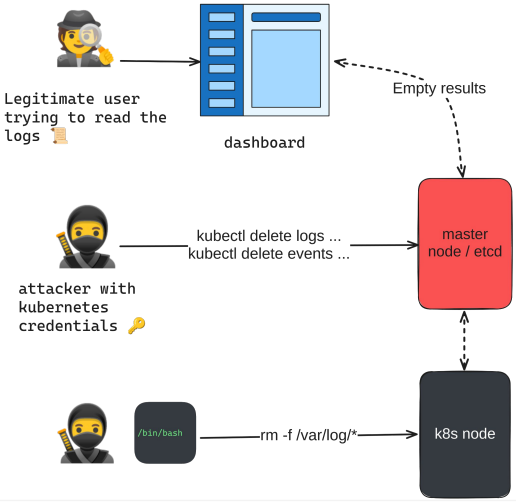
Pod / Container Name Similarity



ALEVSK

By exploiting the naming convention of Kubernetes pods, such as the random suffix (e.g., web-9src8) added to each pod in a ReplicaSet, attackers may attempt to create malicious pods that mimic the names of legitimate applications running in the cluster. The goal is to deceive and confuse cluster administrators. This evasion technique is especially effective when the cluster lacks proper monitoring tools

Delete Container Logs & Kubernetes Events



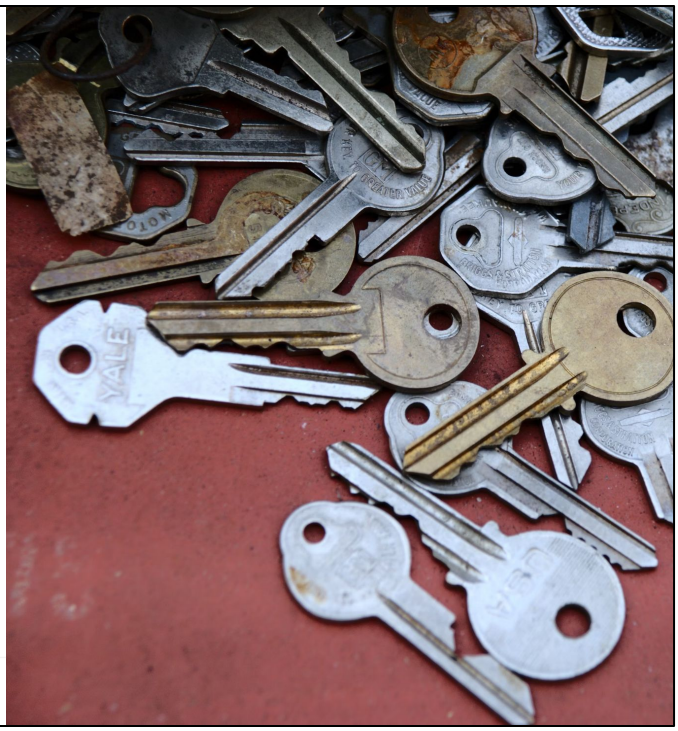
ALEVSK

An attacker with sufficient privileges in the cluster can delete container logs and Kubernetes events to cover their tracks, complicating the investigation for incident response teams. This is particularly problematic if the logs are not replicated outside the cluster, such as being sent to a SIEM or another log analysis solution.

Credential Access

- List Kubernetes secrets
- Mount Service Principal
- Access container service account
- Applications credentials in configuration files
- Access managed identity credential
- Malicious admission controller

ALEVSK

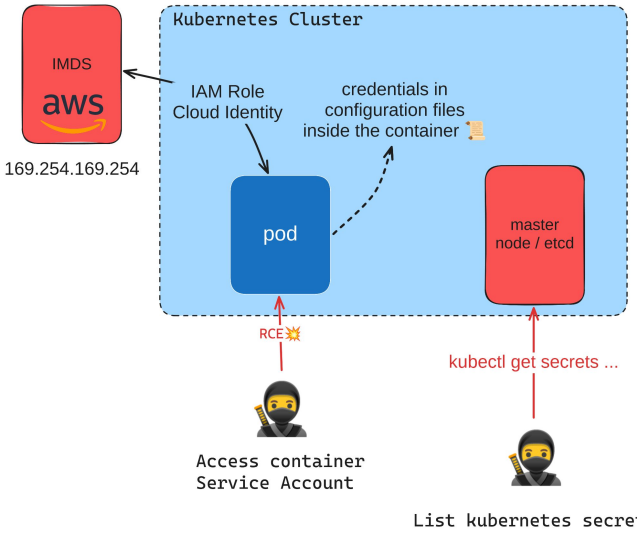


The credential access tactic consists of techniques that are used by attackers to steal credentials.

In kubernetes environments, this includes:

- Kubernetes secrets – objects that store sensitive information
- Service Principal credentials – cloud credentials. Attackers can sometimes leverage their access to a container to obtain these credentials. In AKS, for example, each node contains a service principal credential, so access to a node means access to this credential.
- Container service account credentials: Service Accounts represent applications in Kubernetes. By default, an SA is mounted to every created pod in the cluster.
- Applications credentials: Developers sometimes store secrets in the Kubernetes configuration files, such as environment variables in the pod configuration. This makes them very easy to steal, please don't do this!
- managed identity credentials – cloud credentials. Applications can obtain the token for a managed identity by accessing the Instance Metadata Service (IMDS). Attackers with access to a pod can query the IMDS to obtain a managed identity token and access cloud resources.
- Malicious admission controller: In addition to maintaining persistence, the admission controller can be used to read sensitive information in API calls.

Access managed identity credential



ALEVSK

Attackers may attempt to gather credentials from various sources. The most common method, if the attacker has sufficient Kubernetes permissions, is to list all secrets in the cluster using the kubectl get secrets command. Alternatively, if the attacker has arbitrary command execution on a running container, they will try to locate credentials mounted within the container, such as configuration files, database credentials, API keys, OAuth tokens, etc.

If Kubernetes is running on a managed service like EKS, GKE, or AKS, attackers may attempt to retrieve the cloud identity associated with the running pod by fetching identity credentials from the metadata service (e.g., IMDS in the case of EKS).

Discovery

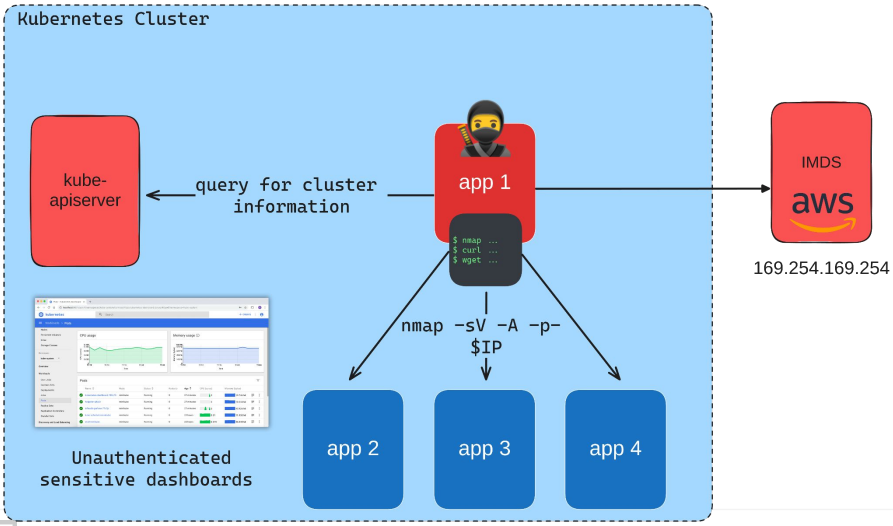
- Access the Kubernetes API server
- Access Kubelet API
- Network mapping
- Access Kubernetes dashboard
- Instance Metadata API

ALEVSK

The discovery tactic consists of techniques that are used by attackers to explore the environment to which they gained access. This exploration helps the attackers to perform lateral movement and gain access to additional resources.

- Access the Kubernetes API server: The Kubernetes API server is the gateway to the cluster. Attackers can send requests to the API server to probe the cluster and get information about pods, containers, secrets, and other resources.
- Access Kubelet API: the Kubelet exposes a read-only API service that does not require authentication (TCP port 10255). Attackers with network access to the host can send API requests to the Kubelet API and retrieve information about running pods and the node itself.
- Network mapping: By default, there is no restriction on pod communication in Kubernetes. Attackers can freely map the cluster network to get information on the running applications.
- Access Kubernetes dashboard: If an attacker steals service account credentials, they can use the Kubernetes dashboard (a Web-based UI) to perform actions in the cluster.
- Instance Metadata API: Cloud providers provide an instance metadata service for retrieving information about a virtual machine. Attackers who gain access to a container can query this service and obtain information about the underlying node the container is running on.

Kubernetes Service



ALEVSK

Once an attacker reaches the discovery phase and gains internal access to the cluster, they can use traditional network scanning methods to enumerate Kubernetes services running inside the cluster, hoping to find services with unauthenticated sensitive interfaces to escalate privileges. Additionally, if the attacker has credentials, they can query the kube-apiserver to gather information about workloads, such as container image versions, which can be used to identify known vulnerabilities. For Kubernetes clusters running on EKS, GKE, AKS, or other managed offerings, attackers may query the Instance Metadata Service (IMDS) to obtain information about the cloud identity of the current workload.

Lateral Movement

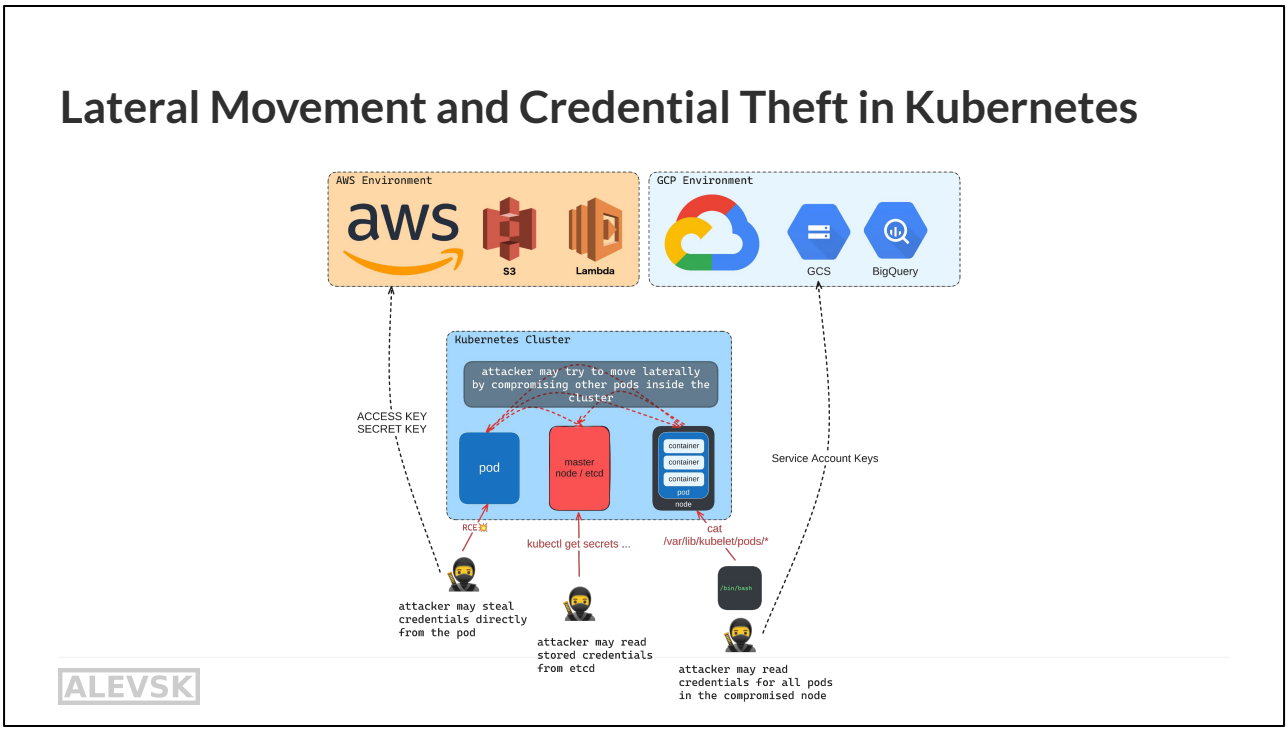
- Access cloud resources
- Container service account
- Cluster internal networking
- Applications credentials in configuration files
- Writable volume mounts on the host
- CoreDNS poisoning

ALEVSK

The lateral movement tactic consists of techniques that are used by attackers to move through the victim's environment. This includes gaining access to various resources in the cluster, gaining access to the underlying node from a container, or gaining access to the cloud environment itself.

- Access cloud resources: Attackers may move from a compromised container to the cloud environment.
- Container service account: Attackers who gain access to a container in the cluster may use the mounted service account token for sending requests to the API server, and gaining access to additional resources in the cluster.
- Cluster internal networking: Kubernetes networking behavior allows traffic between pods in the cluster as a default behavior. Attackers who gain access to a single container may use it for network reachability to another container in the cluster.
- Applications credentials in configuration files: Developers store secrets in the Kubernetes configuration files, for example, as environment variables in the pod configuration. Using those credentials attackers may gain access to additional resources inside and outside the cluster. (See "6: Application credentials in configuration files" for more details.)
- Writable volume mounts on the host: Attackers may attempt to gain access to the underlying host from a compromised container.
- CoreDNS poisoning: CoreDNS is the main DNS service used in Kubernetes. The configuration file (coorefile) is stored in a ConfigMap object, located at the kube-system namespace. If attackers have permissions to modify the

- take the network identity of other services.



Lateral Movement and Credential Theft in Kubernetes

- An attacker with arbitrary command execution on a container can steal credentials stored within the running pod.
- An attacker with sufficient privileges can retrieve credentials directly from etcd using the kubectl get secrets command.
- If an attacker compromises a worker node, they automatically gain access to all secrets associated with the pods running on that node.

By obtaining additional credentials, an attacker can attempt lateral movement to compromise more workloads within the cluster or gain access to resources in other cloud environments.

In Kubernetes, collection consists of techniques that are used by attackers to collect data from the cluster or through using the cluster.

- Images from private registry: The images that are running in the cluster can be stored in a private registry. For pulling those images, the container runtime engine (such as Docker or containerd) needs to have valid credentials to those registries. If the registry is hosted by the cloud provider, in services like Azure Container Registry (ACR) or Amazon Elastic Container Registry (ECR), cloud credentials are used to authenticate to the registry. If attackers get access to the cluster, in some cases they can obtain access to the private registry and pull its images.

Impact

- Data destruction
  - Deleting PV, logs, events, etc
- Resource Hijacking
  - Deploying cryptominers, torrents, etc
- Denial of service
  - Deleting pods, deployments, sts, services or ingress rules

ALEVSK

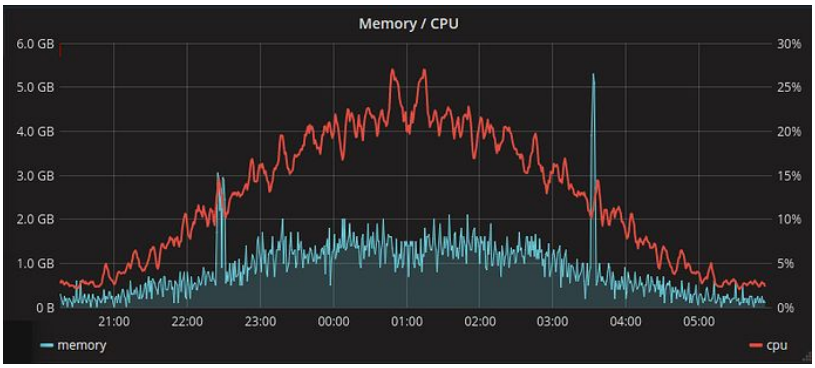
The Impact tactic consists of techniques that are used by attackers to destroy, abuse, or disrupt the normal behavior of the cluster.

- Data destruction: Attackers may attempt to destroy data and resources in the cluster. This includes deleting deployments, configurations, storage, and compute resources.
- Resource Hijacking: Attackers may abuse a compromised resource for running tasks. A common abuse is to use compromised resources for running digital currency mining. Attackers who have access to a container in the cluster or have permissions to create new containers may use them for such activity.
- Denial of service: Attackers may attempt to perform a denial of service attack, which makes the service unavailable to the legitimate users. In container clusters, this include attempts to block the availability of the containers themselves, the underlying nodes, or the API server.

Kubernetes Build-In Defenses

Resource Quotas and Limits

- Limits
  - CPU, Memory, etc.
- Requests
  - CPU, Memory, etc.



ALEVSK

When you configure a Pod, you can optionally specify how much of each resource each container needs. Most commonly, CPU and memory resources.

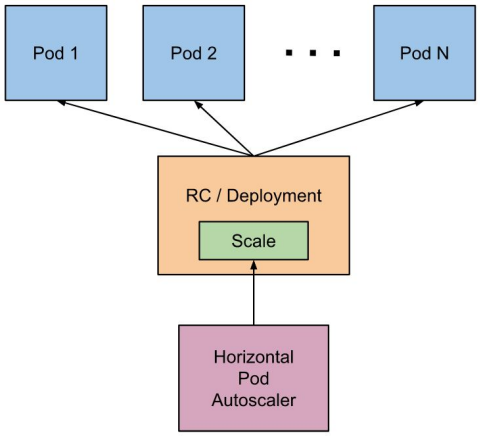
- The “resource request” configuration tells the kubelet how many resources to reserve specifically for a given container (lower limit).
- The “resource limit” configuration tells the kubelet that this container is not allowed to use any more of a resource than the limit you set (upper limit).
- The kubelet is in charge of enforcing these two values.
- This is a simply way to prevent malicious cryptominers.

https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

Horizontal Pod Autoscaling

Horizontal Pod Autoscaling (HPA) in Kubernetes is a feature that automatically adjusts the number of pods in a deployment, replica set, or stateful set based on the current load or resource usage.

Tip: You can also use **PodDisruptionBudget** to avoid accidental outages



ALEVSK

Horizontal Pod Autoscaling (HPA) in Kubernetes is a feature that automatically adjusts the number of pods in a deployment, replica set, or stateful set based on the current load or resource usage.

SecurityContext For Pods

- Privilege and Access Control
  - allowPrivilegeEscalation, privileged, runAsNonRoot, runAsUser, runAsGroup
- Capabilities Management
  - Capabilities: add, drop
- Filesystem and Mount Settings
  - readOnlyRootFilesystem
- Security Profiles
  - seLinuxOptions, seccompProfile, windowsOptions

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

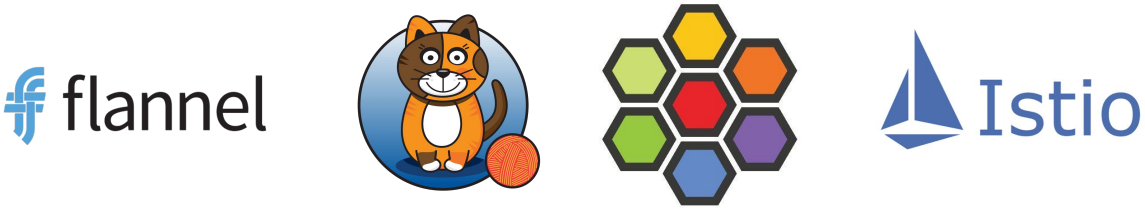
ALEVSK

Security context defines privilege and access control settings for a Pod or Container. Security context settings include, but are not limited to:

- AllowPrivilegeEscalation: controls whether processes in a container can gain more privileges than their parent process
- privileged: processes in privileged containers essentially have root control over the host Node.
- runAsNonRoot, runAsUser, runAsGroup: enforce process owners.
- Capabilities: give a user a subset of root privileges
- Filesystem: control which filesystems and volumes are read-only
- Security Profiles: Apply SELinux, system call filtering with seccomp, etc.

https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

Network Policies



ALEVSK

Network Policies allow you to specify how a pod is allowed to communicate with various "endpoints" and "services" over the network. They control who can talk to who.

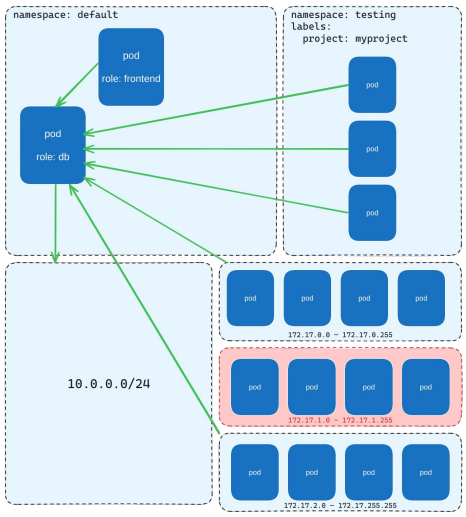
Network policies are implemented by the network plugin or CNI. To use network policies, that CNI must support network policies.

- kubenet is the default CNI plugin for kubernetes
- Popular CNIs: flannnel, calico, Cilium, istio CNI

https://kubernetes.io/docs/concepts/services-networking/network-policies/

Network Policies

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 172.17.0.0/16
            except:
              - 172.17.1.0/24
        - namespaceSelector:
            matchLabels:
              project: myproject
        - podSelector:
            matchLabels:
              role: frontend
      ports:
        - protocol: TCP
          port: 6379
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
      ports:
        - protocol: TCP
          port: 5978
```

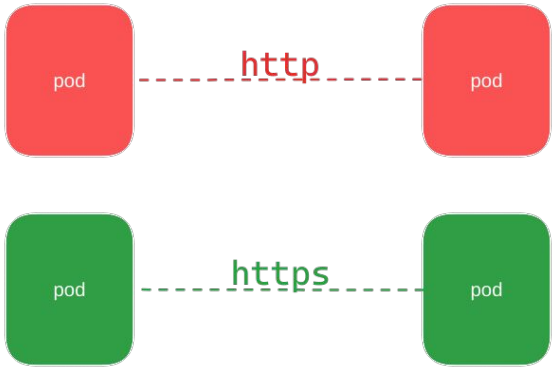


ALEVSK

- By default, pods are non-isolated, they accept traffic from any source.
- Once there is any NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not explicitly allowed by the NetworkPolicy.
- Here we have a podSelector that selects all pods with the role “db”. We can enforce both inbound and outbound traffic in our policy.
- For ingress, we only accept connections from 3 places: the IP block 172.17.0.0/16 excluding 172.17.1.0/24, the kubernetes project “myproject”, and pods with the role “frontend”. We also only accept inbound connections to TCP port 6379.
- For egress, we only allow connections made to the IP block 10.0.0.0/24, and only coming from port 5978.

Encryption In Transit

- Kubernetes Certificate Authority
  - certificates.k8s.io
- Cert-manager.io
- Service mesh
  - Istio, Linkerd, Consul, etc



ALEVSK

- By default, traffic between Pods is unencrypted, meaning kubernetes traffic can be sniffed for secrets.
- Pods can be configured with TLS certificates, allowing their traffic to be encrypted. The CA for these certs is `certificates.k8s.io`.
- These certs can also be generated and managed automatically by services like cert-manager.io.
- Service mesh?

https://kubernetes.io/docs/tasks/tls/managing-tls-in-a-cluster/

Encryption At Rest With EncryptionConfiguration

- Enable at-rest encryption for:
  - secrets, configmaps, events, *.apps, **, etc.
- Supports:
  - Identity (plain text), AESCBC, AESGCM
- External KMS

ALEVSK

EncryptionConfiguration covers at-rest encryption for resource data stored using the Kubernetes API.

You can also use an external KMS for secret management.

- Use a storage integration that provides encrypted volumes, or
- Encrypt the data within your own application

IMPORTANT: This will NOT encrypt filesystems mounted to containers. To do that you need to:

https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/

RBAC & Pod Security Policies (PSPs)

PSPs were deprecated in Kubernetes v1.21. Removed from Kubernetes in v1.25.

ALEVSK

When it comes to access control, PSP used to be a mechanisms to manage access to the different resources in the cluster

- PSPs objects are cluster-level resources that define the conditions pods must satisfy in order to be admitted into the cluster
- PSPs are enforced using an optional Kubernetes admission controller, if enabled, any attempts to create pods that do not satisfy authorized PSPs will be denied
    - PodSecurityPolicy: This was like a set of rules that determined what security conditions a pod must meet to run in your Kubernetes cluster
    - ClusterRole: Define a list of permissions that can be applied cluster-wide
    - ClusterRoleBinding: Ties the ClusterRole to users, groups, or service accounts, allowing them to use the permissions defined in the ClusterRole (doesn't limit to a single namespace)
    - RoleBinding: A RoleBinding connects roles to users, groups, or service accounts within a namespace.
    - Role: A Role is also a set of permissions but limited to a specific namespace within the cluster. It defines what actions a user, group, or service account can perform within that namespace

Protect Cluster Against Privilege Pods

PodSecurityPolicy

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restrictive
spec:
  privileged: false
  hostNetwork: false
  allowPrivilegeEscalation: false
  defaultAllowPrivilegeEscalation: false
  hostPID: false
  hostIPC: false
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
  - 'configMap'
  - 'downwardAPI'
  - 'emptyDir'
  - 'persistentVolumeClaim'
  - 'secret'
  - 'projected'
```

ClusterRole

```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-restrictive
rules:
- apiGroups:
  - extensions
  resources:
  - podsecuritypolicies
  resourceNames:
  - restrictive
  verbs:
  - use
```

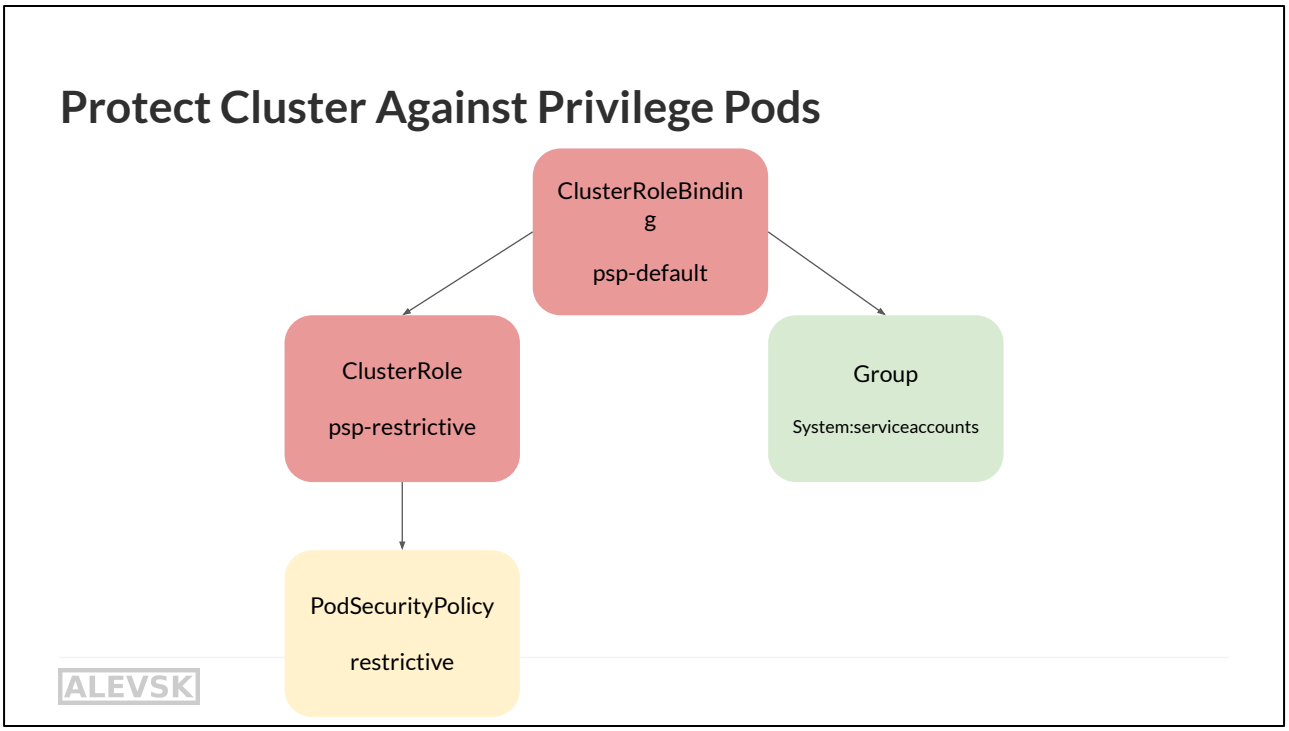
ClusterRoleBinding

```yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-default
subjects:
- kind: Group
  name: system:serviceaccounts
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: psp-restrictive
  apiGroup: rbac.authorization.k8s.io
```

ALEVSK

Let say we want to prevent users to run privilege pods in your cluster.

- The first step would be to create a PSP that add all necessary rules to restrict our pod from using the host network and process namespace, among other things
- Second, we will have to create a ClusterRole to allow the use of the restrictive PSP
- Third, we create a ClusterRoleBinding to tie together the ClusterRole and a group, user or service account at the cluster level



Visually the PSP configuration would look like the following tree structure

Protect Cluster Against Privilege Pods

```
kind: Deployment
metadata:
  name: nginx-hostnetwork-deployment
  namespace: default
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.4
      hostNetwork: true
```

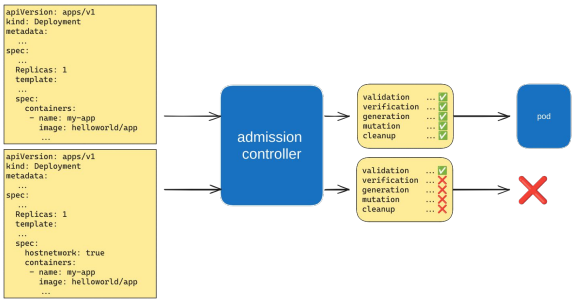
```
Controlled By:  Deployment/nginx-hostnetwork-deployment
Replicas:       0 current / 1 desired
Pods Status:    0 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=nginx
           pod-template-hash=597c4cff45
  Containers:
   nginx:
    Image:        nginx:1.15.4
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type             Status  Reason
  ----             ------  ------
  ReplicaFailure   True    FailedCreate
Events:
  Type     Reason        Age                From                   Message
  ----     ------        ----               ----                   -------
  Warning  FailedCreate  13s (x13 over 33s)  replicaset-controller  Error creating: pods "nginx-hostnetwork-deployment-597c4cff45-" is forbidden: unable to validate against any pod security policy: [spec.securityContext.hostNetwork: Invalid value: true: Host network is not allowed to be used]
```

ALEVSK

- In this example, If a malicious user try to run a container using the **hostNetwork: true** to share the host's network namespace, the restrictive PSP in place will prevent the pod from running
- **hostNetwork: true** provides a pod with greater network access at the cost of increasing security risk

Pod Security Admission & Pod Security Standards

- Less flexible than PSP, but easier to use and implement
- Privileged, Baseline & Restricted
- Validation
  (Validating Admission Webhook)
- Mutation
  (Mutating Admission Webhook),
- Third-party: OPA GateKeeper, Kyverno, jsPolicy, etc



ALEVSK

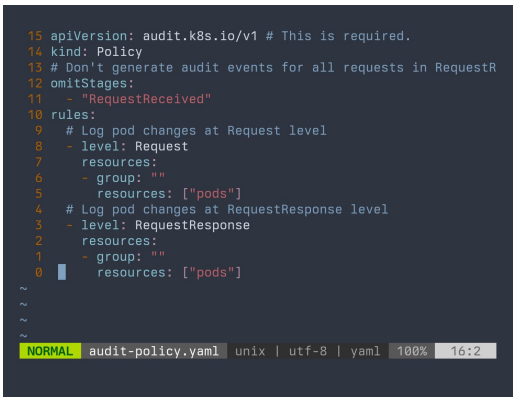
A modern approach for access control would be to use Pod Security Admission

- Pod Security Admission is a build-in feature in Kubernetes that helps ensure your Pods meet certain Pod Security Standards (Privileged, Baseline & Restricted) before they're allowed to run. Think of it as a security checkpoint that a Pod has to pass through
- Imagine a gatekeeper for your Kubernetes cluster. Before any request (like creating a Pod, Service, etc.) is fully processed, it has to go through various checkpoints. Admission Controllers are these checkpoints, each with its own set of rules on what's allowed and what's not. They can modify requests or reject them if they don't meet certain criteria, helping to enforce security, governance, and operational policies in your cluster.

https://kubernetes.io/docs/concepts/security/pod-security-admission/
https://kubernetes.io/docs/concepts/security/pod-security-standards/

Audit Log

- When to Log:
  - RequestReceived, ResponseStarted, ResponseComplete, Panic
- What to Log:
  - None, Metadata, Request, RequestResponse
- Where to Log:
  - Filesystem, Webhook



ALEVSK

Next, Kubernetes has an audit logging feature that, when enabled, keeps a detailed record of all the actions that happen inside your cluster, such as who accessed it, what changes they made, and when these activities occurred. It's very customizable and it allow you to send this logs to other tools for further analysis, however there are a couple things to consider:

- It increases the memory footprint of the API server because additional resources are required when processing each new request.
- Audit Logs do not directly capture application-level events such as incoming HTTP requests
- These application-specific logs are managed separately and typically involve a different logging mechanism or setup.
- Tools like Fluentd, Elastic Stack (ELK), or Prometheus with Grafana are popular choices in the Kubernetes ecosystem for this purpose.

https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/

Kubernetes Cluster Hardening

- Isolate Kubernetes Nodes
  - Don't expose nodes directly, use ingress controller, separate control and data flow
- Protect Kubelet
  - Disable anonymous authentication
  - Enable mTLS authentication on the kubelet's HTTPS endpoint
- Protect etcd with TLS, Firewall and Encryption
- Use Third-Party Authentication for API Server

ALEVSK

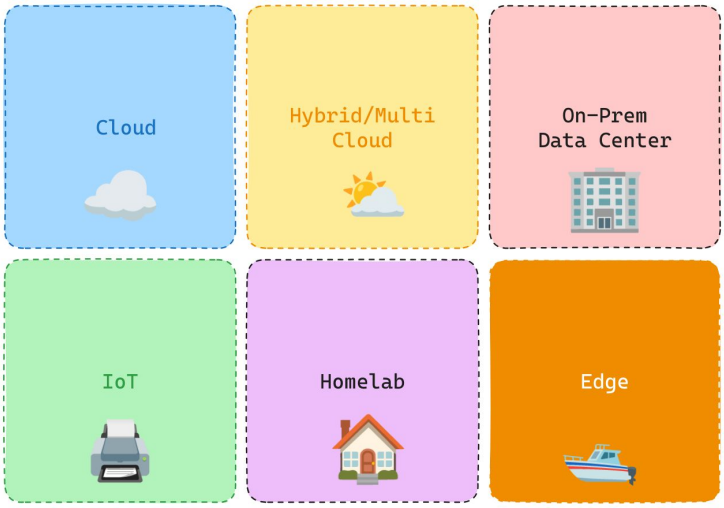
Finally, there's different methods, best practices and industry standards to follow when securing kubernetes, you can start by:

- Isolate kubernetes Nodes
  - Don't expose node directly to the Internet and instead use an ingress controller
  - If your environment supports it, configure separate network interfaces on your nodes for different types of traffic. For example:
    - One network interface handles public internet traffic (ingress traffic to ports 80 and 443).
    - Another interface handles internal cluster communications, including kube-api server traffic and node management traffic.
- Protect Kubelet
  - Disable anonymous authentication
  - Enable mTLS authentication on the kubelet's HTTPS endpoint
- Use Third-Party Authentication for API Server:
  - OIDC or implement a custom proxy for authentication

What's Next?

Kubernetes Evolution (Edge & Hybrid Cloud)

- Kubeedge
- SuperEdge
- Akri
- OpenYurt
- K3s
- Microk8s
- Minikube
- ...



ALEVSK

- We are still facing many security challenges when it comes to defending Kubernetes, and it's becoming increasingly popular to deploy Kubernetes at the "edge" for use cases like hybrid environments, IoT and Edge computing.
- Tools like kubeedge, SuperEdge, K3s, MicroK8s and Minikube have a low CPU and memory footprint and make it very easy to deploy Kubernetes on low energy consumption devices, but at the cost of increasing our attack surface

Next Time You Deploy Kubernetes – Part 1

1. Don't deploy images from untrusted registries in your cluster, regularly scan images for vulnerabilities and enable binary authorization
2. Don't store sensitive data on kubernetes secrets, instead use an **external KMS** to manage secrets or enable **EncryptionConfiguration**
3. Encryption in transit: generate TLS certificates for your workloads using the **certificates.k8s.io** api or use a third party solution like **cert-manager.io**

Next Time You Deploy Kubernetes – Part 2

4. Enforce **SecurityContext** and **limit resource consumption** for your workloads
5. Network Segmentation, choose **CNI plugin** that supports **network policies** to limit communication between your **services** and monitor network traffic
6. Configure Role-Based Access Control (**RBAC**) policies via an **Admission Controller** or **PSPs** always following the principle of least privilege. Never use **cluster-admin** in your workloads

Next Time You Deploy Kubernetes – Part 3

7. Turn on audit logging and monitor unusual or unwanted api calls, e.g. authentication failures.
8. Keep Kubernetes version up to date, update and patch regularly
9. Hardening control plane components and lock down kubelet
10. Adopt GitOps practices for managing your Kubernetes clusters to enhance security, auditability, and consistency through declarative configuration, automation, and tight integration with version control systems

Thanks

- @alevsk
- https://github.com/alevsk
- https://www.linkedin.com/in/alevsk/
- https://www.alevsk.com