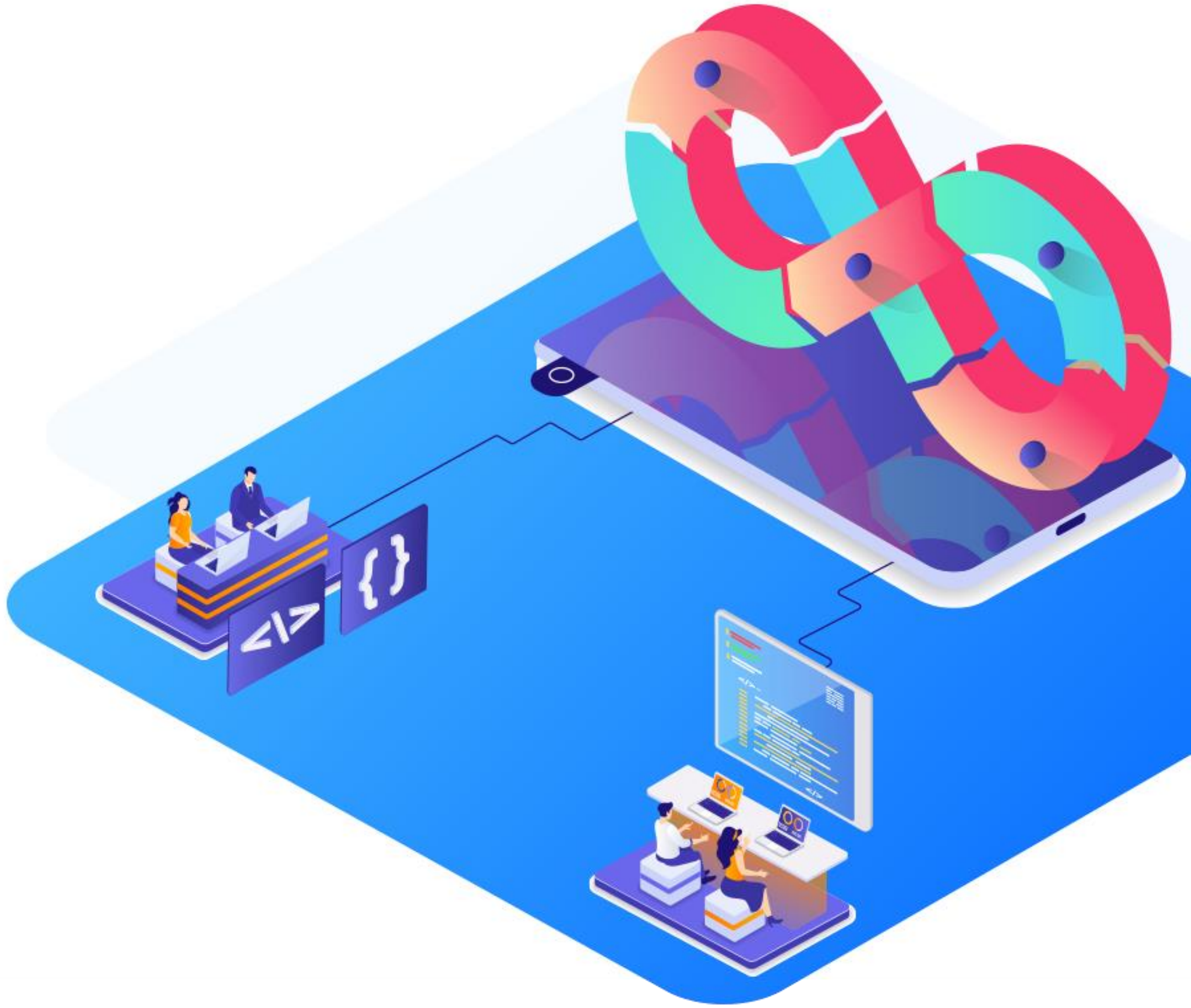
Container Orchestration Using Kubernetes



DevSecOps in Kubernetes

Learning Objectives

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

- Apply the principles of DevSecOps in Kubernetes to integrate security throughout the software development lifecycle
- Implement best practices for securing Kubernetes environments by configuring network policies, managing secrets, and enforcing Pod Security Policies
- Deploy security tools and strategies in Kubernetes to ensure compliance, threat detection, and efficient incident response
- Utilize Open Policy Agent (OPA) and Gatekeeper to enforce security policies within Kubernetes clusters
- Develop continuous monitoring and auditing strategies for Kubernetes environments to ensure compliance and rapid response to security incidents

Introduction to Security in Kubernetes

Kubernetes Security

Security in Kubernetes is a multi-faceted domain that involves securing the cluster, the applications running within it, and the network.

Security Infrastructure in Kubernetes

It is a comprehensive set of security measures and practices implemented across various layers of a Kubernetes environment to protect against threats and vulnerabilities.

Categories of security infrastructure in Kubernetes are:

- Code
- Container
- Cluster
- Cloud

Code Security

Code security ensures that the application code running inside containers is secure and free from vulnerabilities.

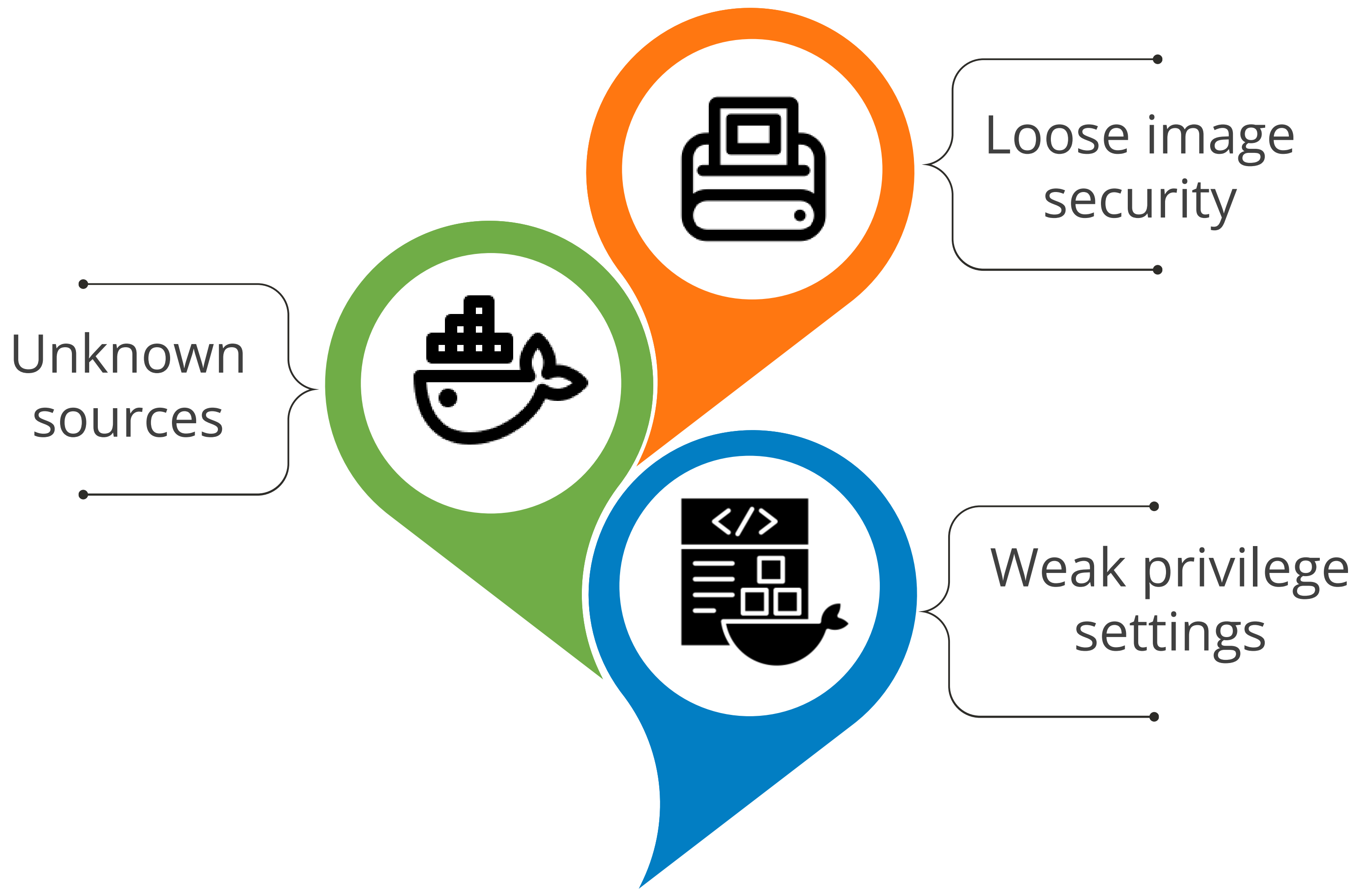
Common challenges in code security and their solutions:

Insecure code	Embed static code analysis (SCA) tool in the development pipeline to quickly detect security issues within the code
Inadequate application risk assessment	Ensure that applications are resilient against common attacks
Risk of software dependencies	Use tools like the **OWASP dependency check** to check outdated or vulnerable libraries

Container Security

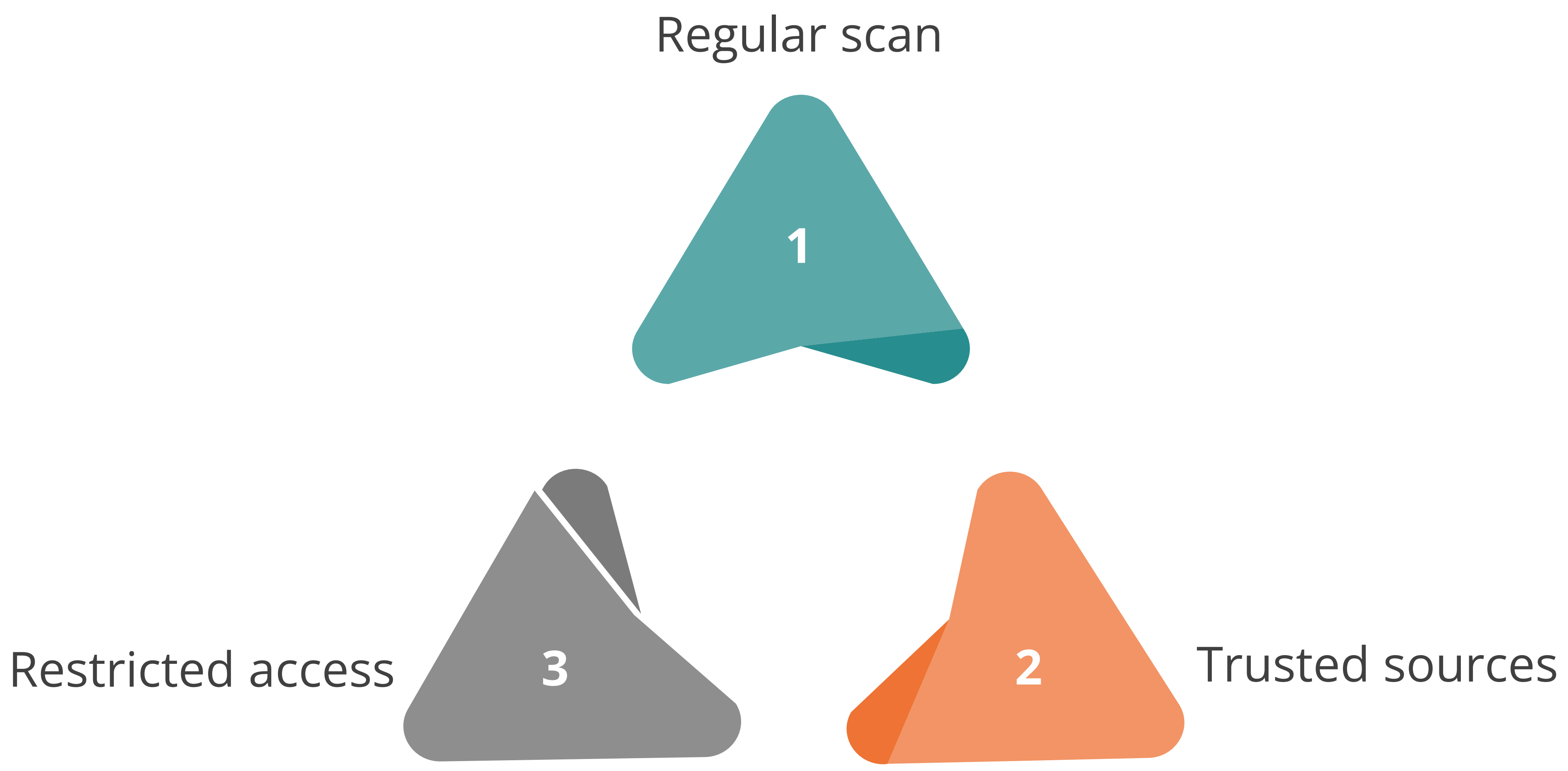
It ensures that containerized applications are protected from vulnerabilities, attacks, and unauthorized access.

The primary areas of focus for organizations to strengthen container security include:



Container Security: Best Practices

Following are the best practices for securing containerized applications:



Container Security: Best Practices

It is important to regularly scan containers for vulnerabilities to identify and address potential threats in images and running instances.

Regular scan

It is prioritized when there are frequent updates to images or when security compliance is critical.

Container Security: Best Practices

It is essential to always pull container images from trusted, verified repositories to avoid malicious code or vulnerabilities, especially with open-source or third-party images.

Trusted sources

It is prioritized when using many external resources or third-party integrations.

Container Security: Best Practices

It is important to limit access to containers and container resources, ensuring only authorized users and processes can interact with them.

Restricted access

It is especially useful in environments with multiple teams managing containers or when minimizing access to reduce the attack surface is needed.

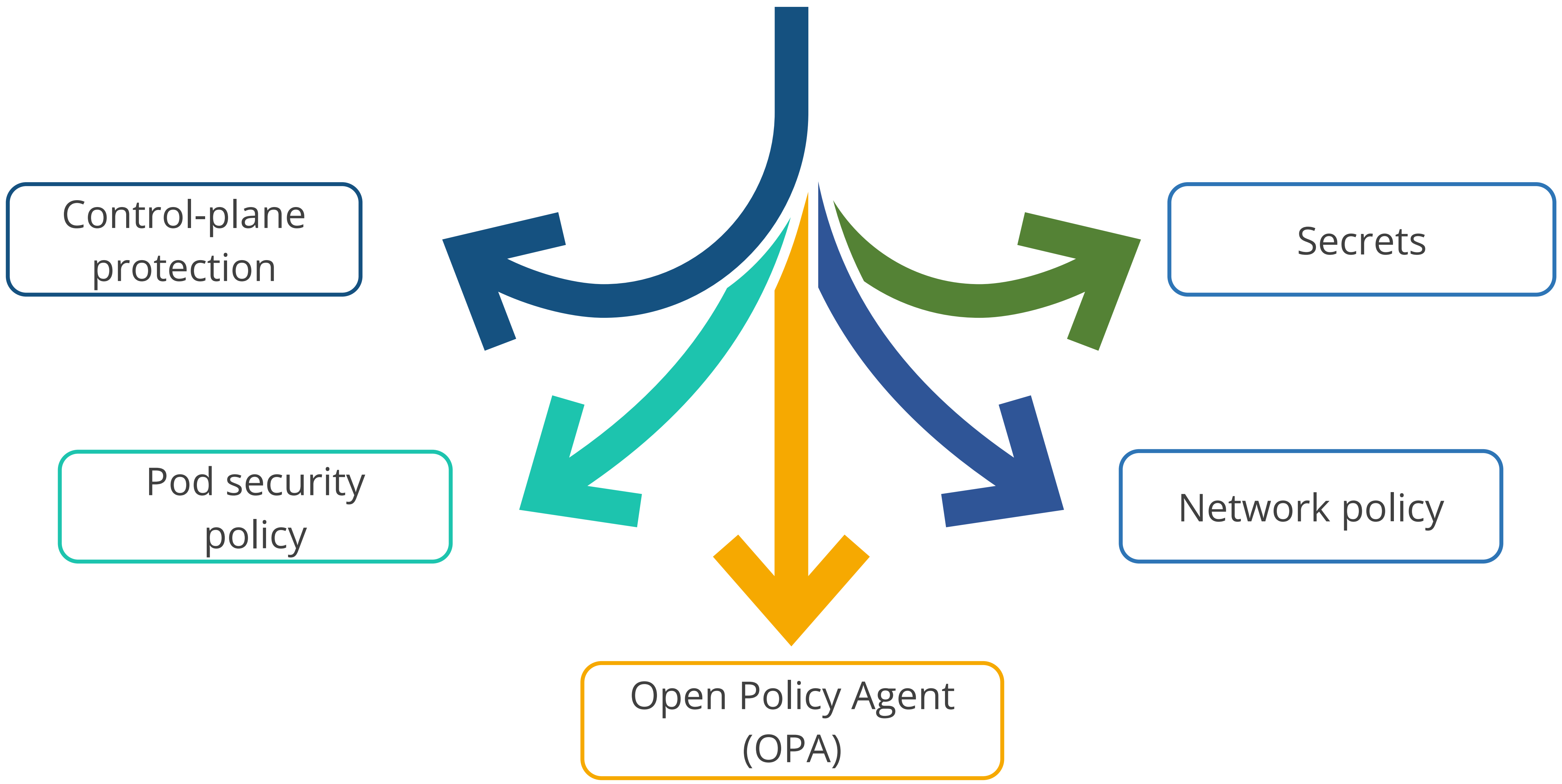
Cluster Security

It is the process of protecting a Kubernetes cluster and its components from unauthorized access, vulnerabilities, and attacks.

This protection involves securing the cluster infrastructure, enforcing access controls, applying network policies, encrypting sensitive data, and continuously monitoring and logging cluster activities.

Cluster Security: Components

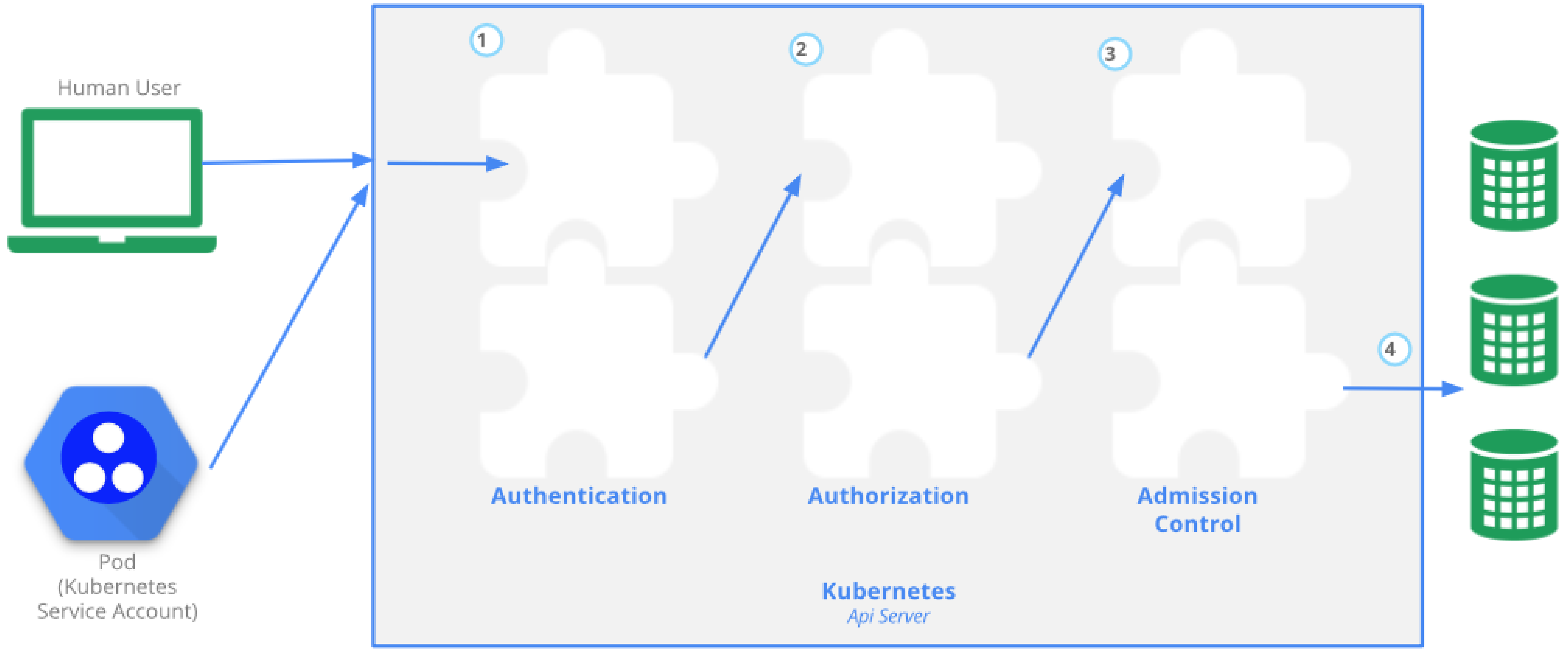
The key components of cluster security include:



Control Plane Protection

It secures the critical components of the Kubernetes control plane, such as the API server, scheduler, controller manager, and etcd.

The key stages involved in control-plane protection are:



Source: https://rafay.co/the-kubernetes-current/securing-kubernetes-applying-zero-trust-principles-to-your-kubernetes-environment/

Control Plane Protection: Stages

- **Request initiation:** A human user or pod initiates a request to the Kubernetes API server.
- **Authentication:** The system verifies the identity of the requester.
- **Authorization**: It checks if the authenticated user or service account has the required permissions.
- **Admission control**: It applies additional policies and validations to the request.
- **Request processing**: It processes the request and applies changes to the cluster state stored in etcd.

Pod Security Policy

The Pod security standards define three different policies.
These policies are cumulative and range from highly permissive to highly restrictive.

Privileged	Baseline	Restricted
It is an unrestricted policy that allows known privilege escalations with minimal security restrictions for Pod configurations.	It is a minimally restrictive policy that blocks privilege escalations while allowing default Pod configurations.	It is a heavily restricted policy enforcing strict security controls and Pod hardening best practices.

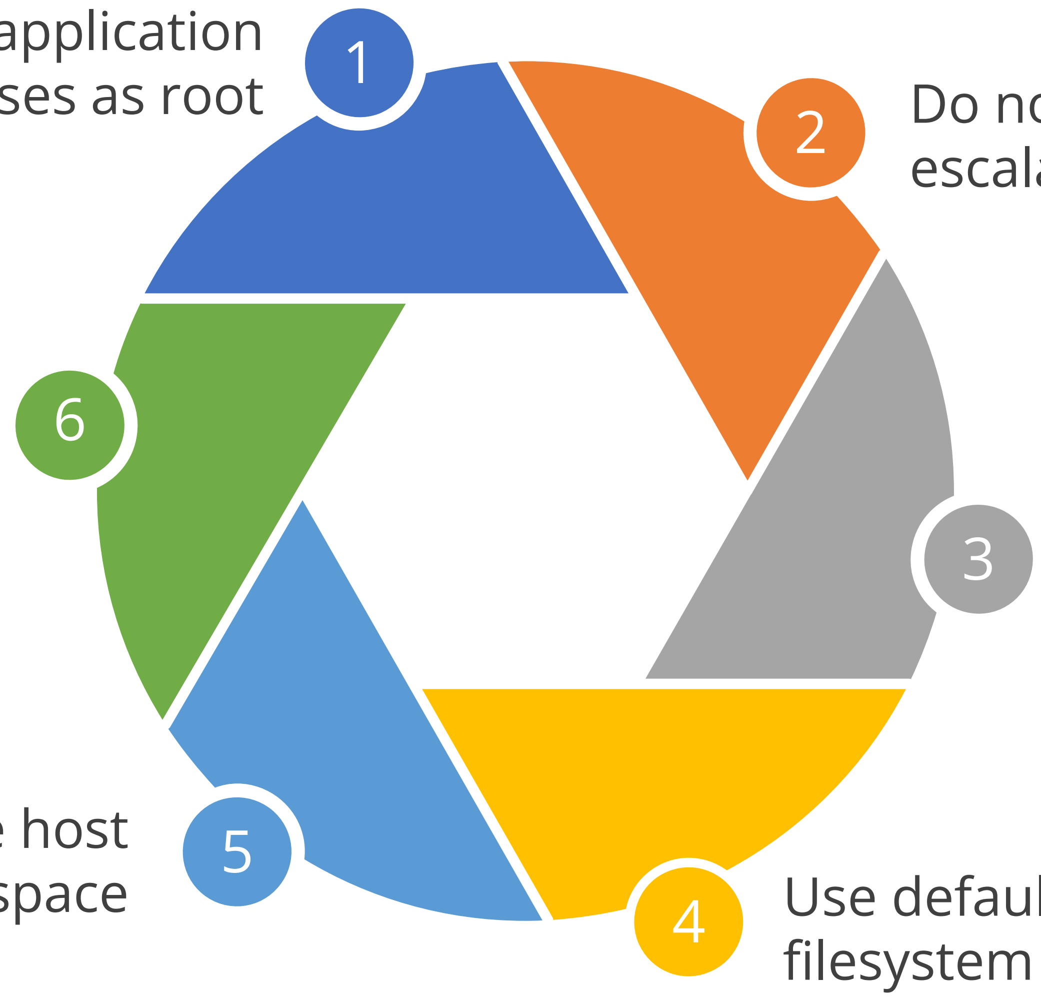
Pod Security Policy: Use Cases

Below are the use cases for three different pod security policies:

Privileged	Baseline	Restricted
Suitable for scenarios where pods need extensive permissions to perform their functions	Ideal for general-purpose workloads where some level of security is required	Best suited for highly sensitive or critical workloads that require stringent security measures

Pod Security Policy: Levels

These policies are necessary to manage the security attributes of Pods, focusing on controlling container privilege levels:



1. Do not run application processes as root
2. Do not allow privilege escalation
3. Use root-only filesystem
4. Use default **/proc** filesystem mount
5. Do not use the host network or process space
6. Enforce service account control

Pod Security Policy: Complexity

Complexities of implementing Pod Security Policies (PSPs) include:

Confusing application	Users often struggle to apply PSPs correctly, leading to broader permissions than intended.
Limited visibility	There is difficulty in determining which Pod Security Policy (PSP) applies to a specific pod, making it challenging to track and understand the policies.
Limited support for choosing Pod defaults	Modifications to pod default values through PSPs have a limited scope, causing inconsistencies.
Challenges in enabling PSP by default	The complexity and associated risks make it impractical to enable PSPs by default, hindering widespread adoption.

Open Policy Agent (OPA): Gatekeeper

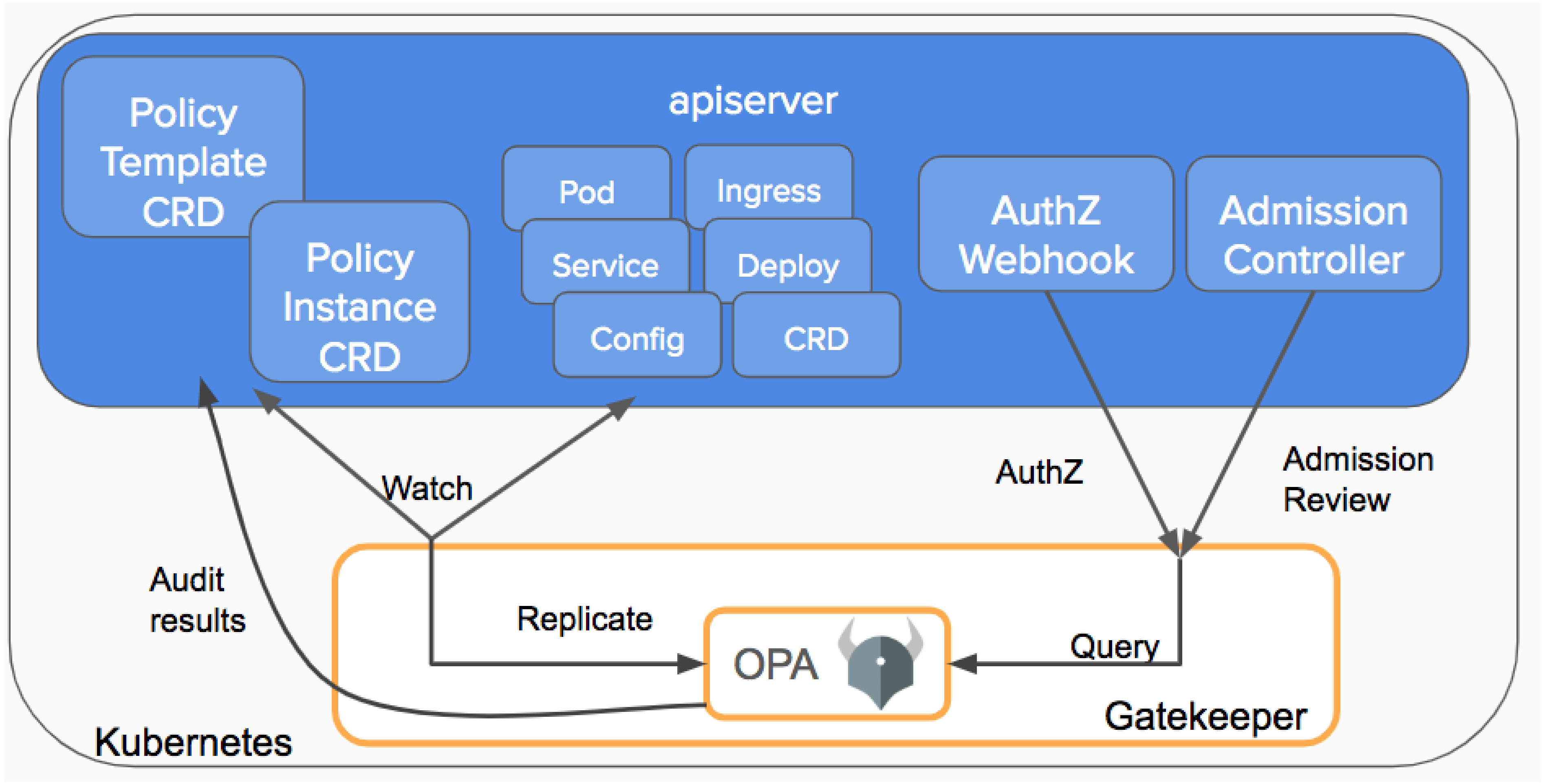
Open Policy Agent (OPA) is a general-purpose policy engine that can be used to enforce policies in various systems, including Kubernetes.

Open Policy Agent

Gatekeeper is an extension of OPA that integrates with Kubernetes to provide policy enforcement and auditing capabilities.

How OPA Gatekeeper Enforces Policies in Kubernetes Clusters

OPA Gatekeeper integrates with the Kubernetes API server to enforce policies during resource creation, update, or deletion. The diagram below illustrates this process:



Source: https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/

How OPA Gatekeeper Enforces Policies in Kubernetes Clusters

The following outlines the process OPA Gatekeeper uses to enforce policies within Kubernetes clusters:

- **Policy template CRD**: Defines policy templates specifying rules and constraints
- **Policy instance CRD**: Specifies which policies apply to which resources in the cluster
- **Kubernetes API server:** Handles all requests to create, update, or delete resources
- **Gatekeeper admission webhook:** Is triggered by the API server for resource requests and intercepts requests and sends them to OPA for evaluation

How OPA Gatekeeper Enforces Policies in Kubernetes Clusters

The following outlines the process OPA Gatekeeper uses to enforce policies within Kubernetes clusters:

- **OPA policy evaluation:** Evaluates requests against defined policies using the Rego language, enabling complex and flexible rule definitions
- **Decision-making:** Decides whether to allow or deny the request, returning the decision to Gatekeeper
- **Admission control:** Enforces OPA's decision and allows or blocks requests based on compliance with policies
- **Audit and monitoring:** Monitors changes, audits resources, and flags any violations or non-compliant resources for review

Network Policy

A network policy controls network traffic between pods or between pods and external networks.

Network policies apply to connections involving a pod at one or both ends, and do not affect other network connections.

Network Policy: Example

The following example demonstrates a network policy that controls ingress and egress traffic based on pod labels and specific IP blocks:

Example:

```
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
```

Example:

```
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
```

Network Policy: Example

Pod selector	Selects pods based on labels; an empty selector applies to all pods in the namespace.
Policy types	Specifies whether the policy applies to ingress, egress, or both; the default is ingress.
Ingress	Allows traffic matching from specified sources and ports. Example: single port, multiple sources (ipBlock, namespaceSelector, podSelector)
Egress	Allows traffic matching to specified destinations and ports. Example: single port to destination in 10.0.0.0/24

Secrets

A Secret is an object that stores sensitive data, such as passwords, tokens, or keys. Using Secrets prevents the need to include confidential information directly in the application code.

Creating secrets:

Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
data:
  username: bXktYXBw
  password: Mzk1MjgkdmRnN0pi
```

Using Secrets as Environment Variables

It involves injecting secret data directly into the environment variables of containers within pods.

This method enables applications to access sensitive information through environment variables, which may be easier for some applications to use compared to file system volumes.

Using Secrets as Environment Variables: Example

The following example demonstrates how to inject a secret into a container's environment variable by referencing a key from a Kubernetes secret:

Example:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-single-secret
spec:
  containers:
  - name: envars-test-container
    image: nginx
    env:
    - name: SECRET_USERNAME
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: username
```

Using Secrets as Volume

It involves mounting a secret as a file system volume within a pod, making the secret's data accessible as files to the containers running in that pod.

Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
        # name must match the volume name below
        - name: secret-volume
          mountPath: /etc/secret-volume
          readOnly: true
  # The secret data is exposed to Containers in the Pod
through a Volume.
  volumes:
    - name: secret-volume
      secret:
        secretName: db-secret
```

Quick Check

As a DevOps engineer, you use OPA Gatekeeper to enforce compliance policies in your Kubernetes cluster. A developer's attempt to create a new deployment is denied due to policy violations. How does OPA Gatekeeper enforce the policies that caused the denial?

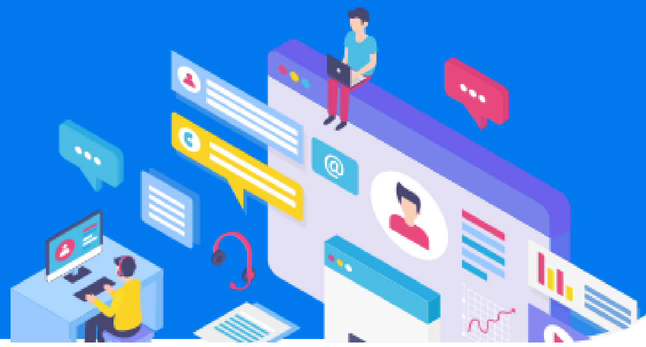
A. By scanning the container images for vulnerabilities

B. By integrating with the Kubernetes API server to enforce policies during resource creation, update, or deletion

C. By monitoring network traffic between pods

D. By periodically checking the state of the cluster and applying policies retroactively

Assisted Practice



Implementing Kubernetes Container Security **Duration: 10 Min.**

Problem statement:

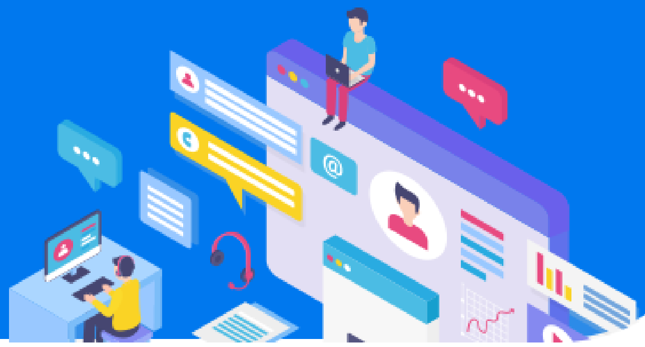
You have been asked to implement Kubernetes container security for enhancing the security of containerized applications.

Outcome:

By the end of this demo, you will be able to implement Kubernetes container security measures to enhance the protection of containerized applications.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



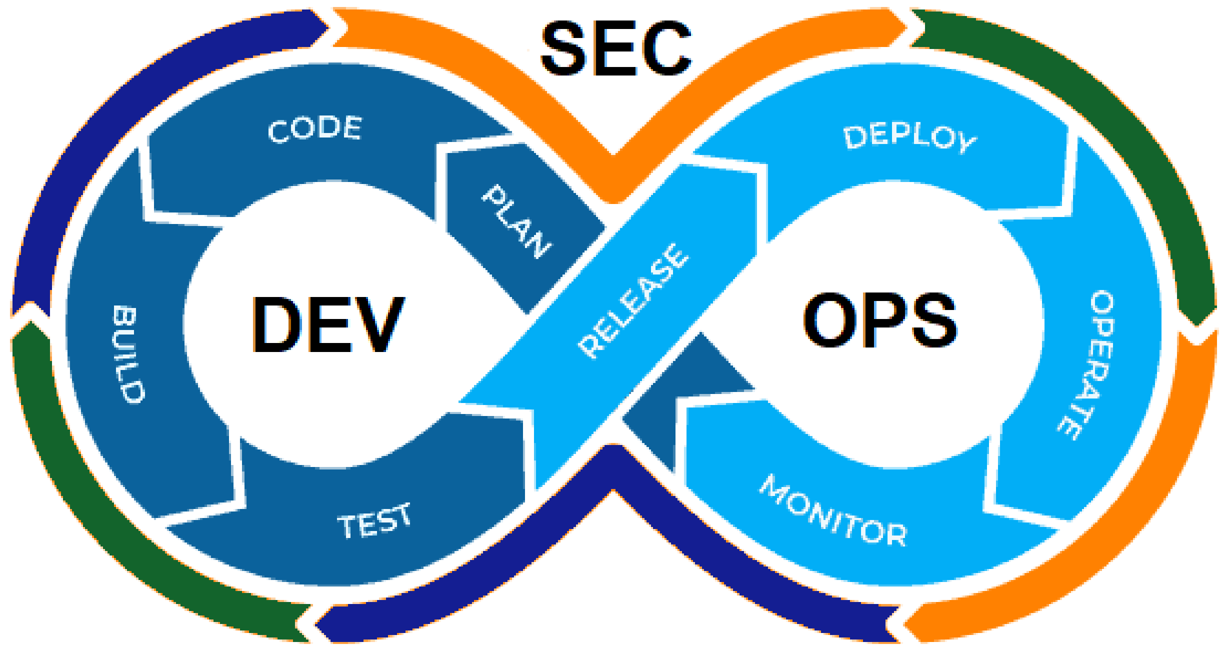
Steps to be followed:

1. Implement network security
2. Secure pods
3. Enhance Pod security
4. Ensure container security

DevSecOps in Kubernetes

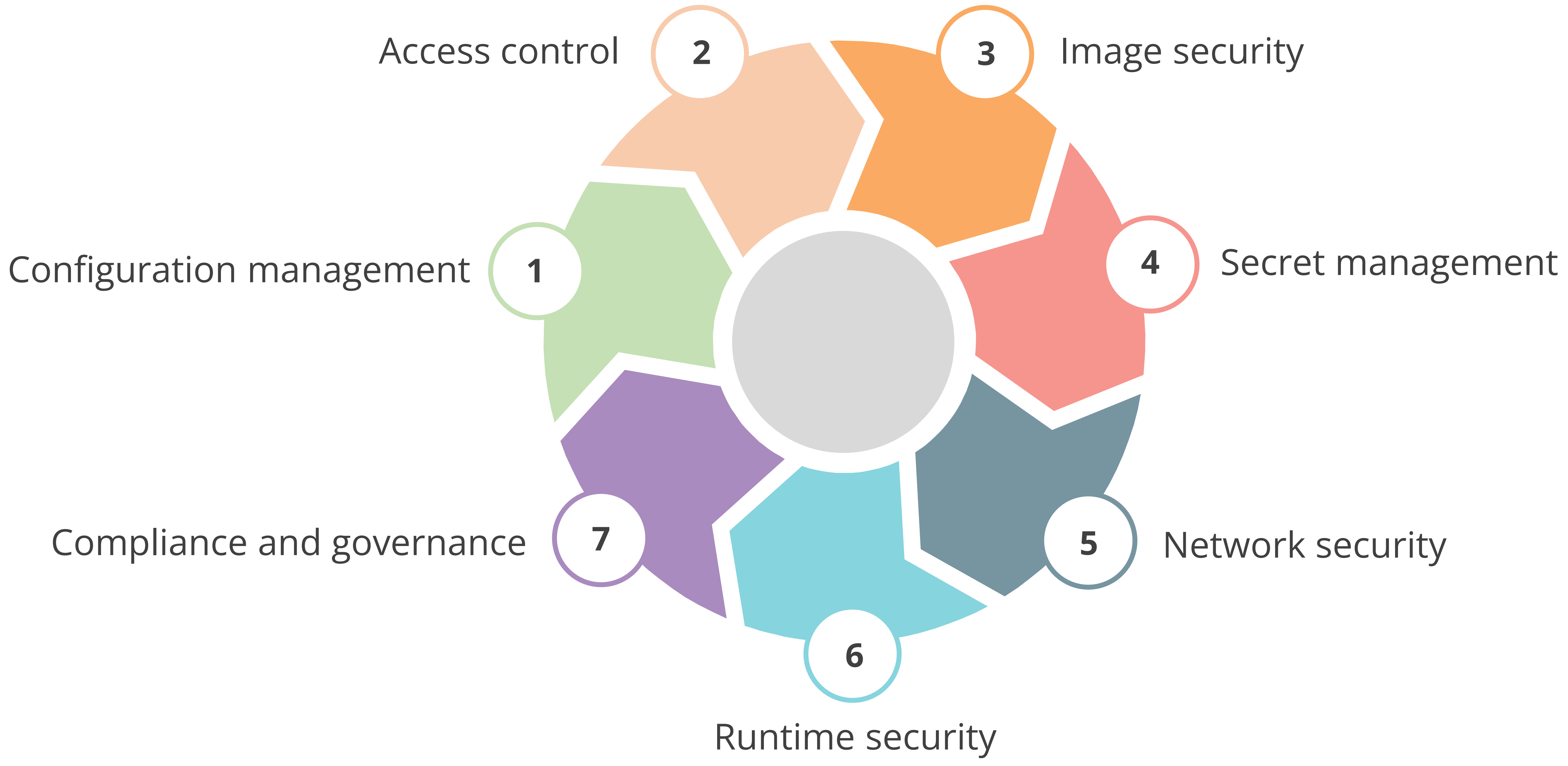
What Is DevSecOps in Kubernetes?

DevSecOps in Kubernetes integrates security practices throughout the software development lifecycle, enhancing agility and reducing vulnerabilities.



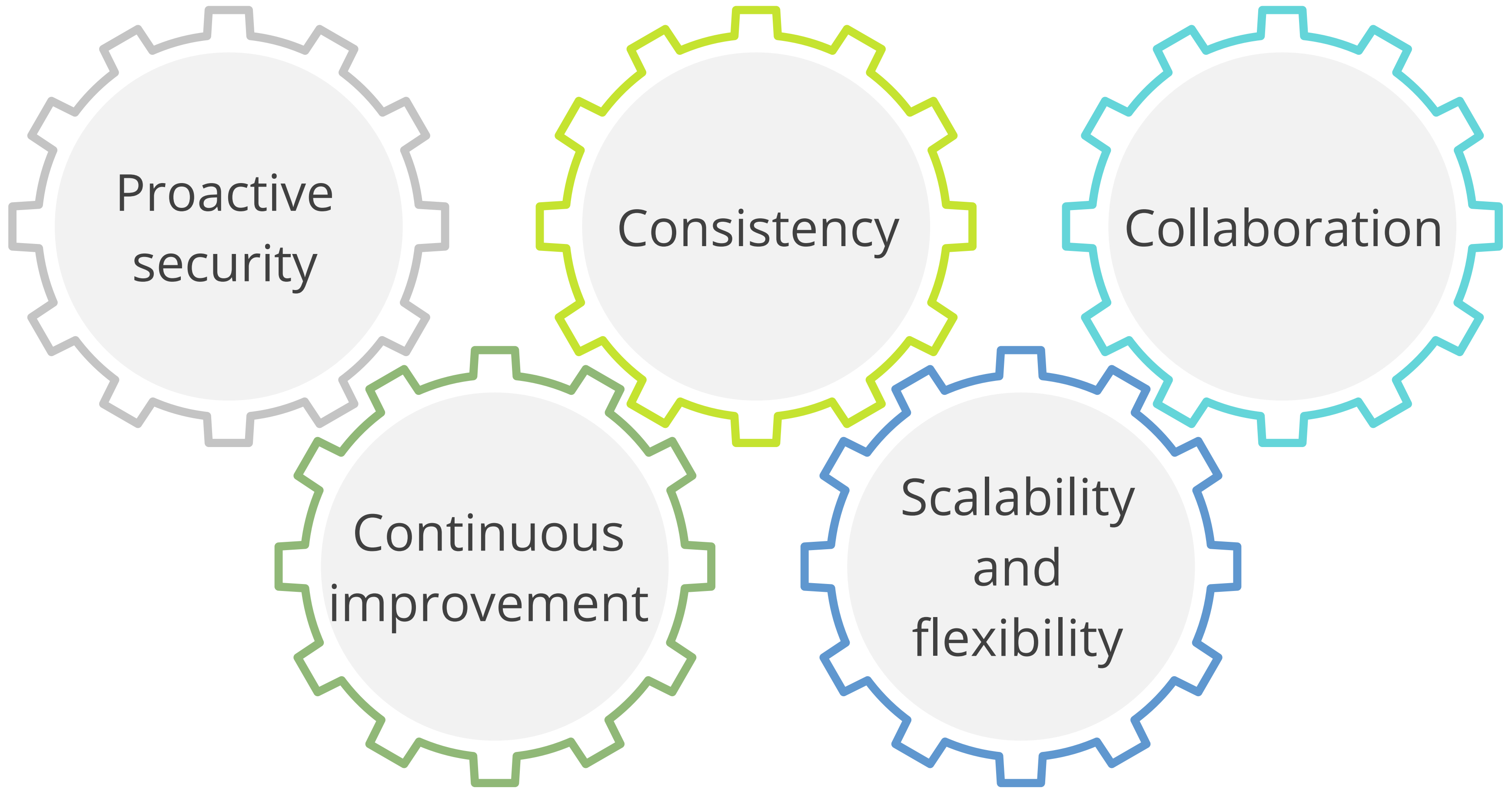
How DevSecOps Addresses Kubernetes Security Challenges

DevSecOps addresses the following Kubernetes security challenges by implementing automated processes and security best practices throughout the development lifecycle:



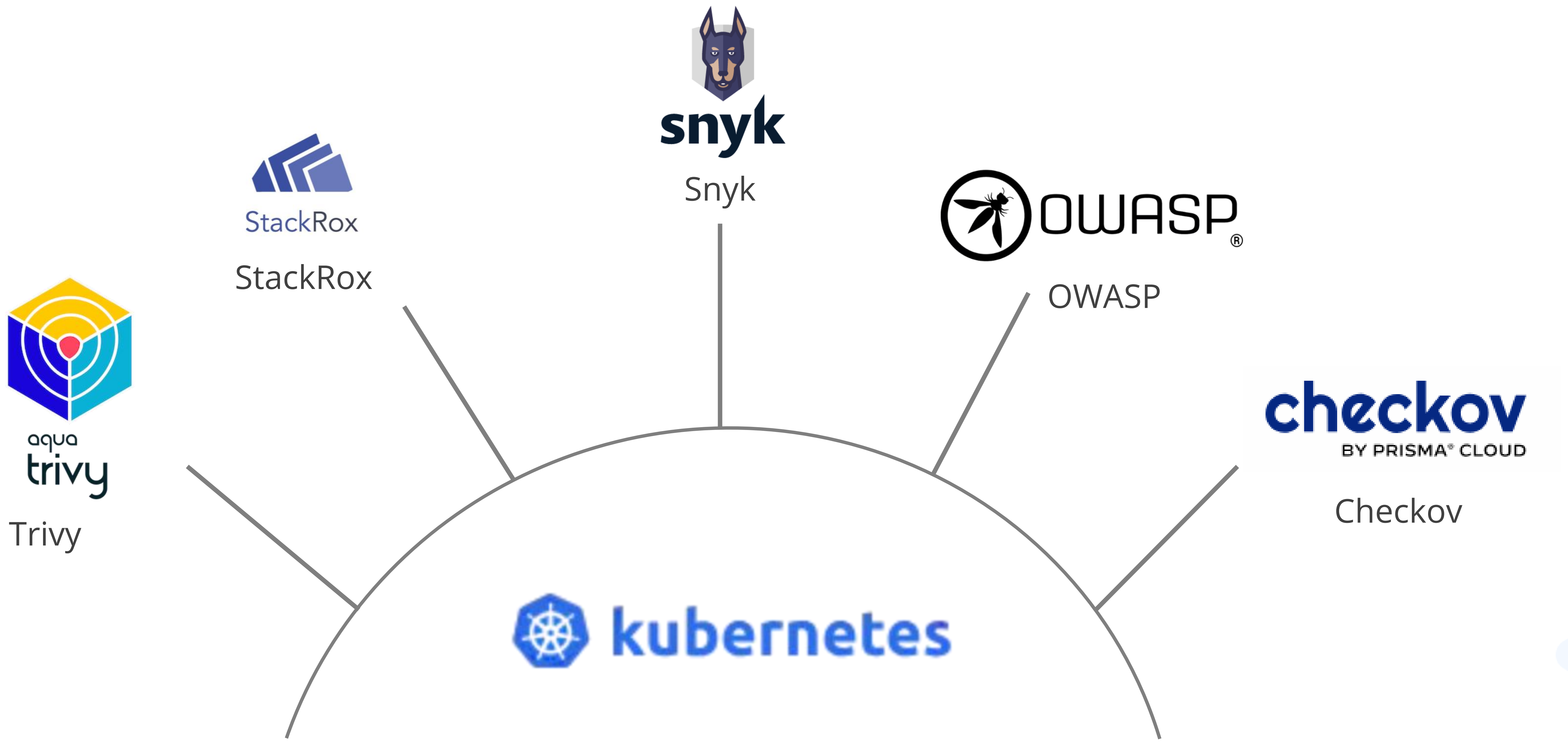
Benefits of DevSecOps in Kubernetes

Here are the key benefits of DevSecOps in Kubernetes:



DevSecOps in Kubernetes: Tools

Here are five top DevSecOps tools for Kubernetes:



DevSecOps in Kubernetes: Tools

Trivy

It is a comprehensive security scanner for Kubernetes environments, identifying vulnerabilities in container images, filesystems, and configuration files.

StackRox

It focuses on container and Kubernetes security and provides visibility, compliance, and threat detection capabilities to secure Kubernetes environments.

DevSecOps in Kubernetes: Tools

snyk

Snyk

It is a developer-first security platform that helps find and fix vulnerabilities in code, dependencies, containers, and infrastructure as code.

OWASP®

OWASP

It is a widely used security and compliance tool which provides a set of best practices and automated checks to identify and remediate potential security vulnerabilities and compliance issues within Kubernetes clusters.

DevSecOps in Kubernetes: Tools

checkov BY PRISMA® CLOUD

Checkov

It is an open-source security and compliance tool that provides a set of best practices and automated checks to identify and remediate potential security vulnerabilities and compliance issues within Kubernetes clusters.

DevSecOps in Kubernetes: Tools Comparison

Tool	Key Features	When to Choose	Benefits
aqua trivy	Comprehensive scanner for vulnerabilities, misconfigurations, and compliance checks	Used for its wide scanning across images, containers, and Kubernetes clusters	Detects vulnerabilities, supports multiple formats, and integrates easily
StackRox	Kubernetes-native security platform for runtime security, network segmentation, and compliance	Ideal for advanced runtime security and detailed network visibility	Strong Kubernetes integration, robust runtime security, compliance enforcement, and auditability

DevSecOps in Kubernetes: Tools Comparison

Tool	Key Features	When to Choose	Benefits
snyk	Tool for scanning vulnerabilities in open-source dependencies, container images, and Kubernetes apps	Perfect for DevOps teams detecting vulnerabilities early in CI/CD pipelines	Continuous scanning, early detection, and CI/CD pipeline integration
OWASP®	Open-source tools and guidelines for identifying security risks in web apps and containers	Helps follow security standards and identify web app vulnerabilities	Offers trusted security guidelines and tools, supporting comprehensive vulnerability detection

DevSecOps in Kubernetes: Tools Comparison

Tool	Key Features	When to Choose	Benefits
checkov	IaC security scanner focused on Terraform, Kubernetes, and CloudFormation	Ideal for securing and ensuring compliance in IaC workflows	Automates IaC security checks enables fast deployment and prevents misconfigurations

Vulnerability Scanning with Trivy

Trivy is a versatile Kubernetes security tool that can be used for:

Comprehensive vulnerability scanning: It identifies vulnerabilities in container images, Kubernetes manifests, and configuration files.

Misconfiguration and secret detection: It scans Kubernetes clusters for configuration issues and exposed secrets. This helps ensure compliance with security standards.

Integration: It integrates with CI/CD pipelines and other Kubernetes tooling, providing automated scanning during development and deployment processes.

Vulnerability Scanning with Trivy

Trivy provides CLI-based scanning capabilities for Kubernetes, allowing users to scan Kubernetes resources such as pods, deployments, and DaemonSets for vulnerabilities and misconfigurations.

The following are some of the important commands for scanning Kubernetes resources:

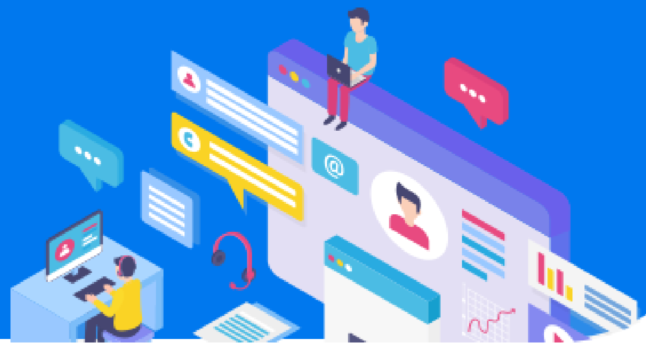
trivy k8s -n kube-system --report=summary	Scan all resources in a namespace
trivy k8s --report=summary	Scan a full cluster and generate a summary report

Vulnerability Scanning with Trivy

The following are some of the important commands for scanning Kubernetes resources:

trivy k8s deployment/appname	Scan a specific resource and get all the output
trivy k8s --severity=CRITICAL --report=all	Filter vulnerabilities by severity
trivy k8s --format json -o results.json	Display JSON output on a full cluster scan

Assisted Practice



Implementing Pod Security Policies **Duration: 10 Min.**

Problem statement:

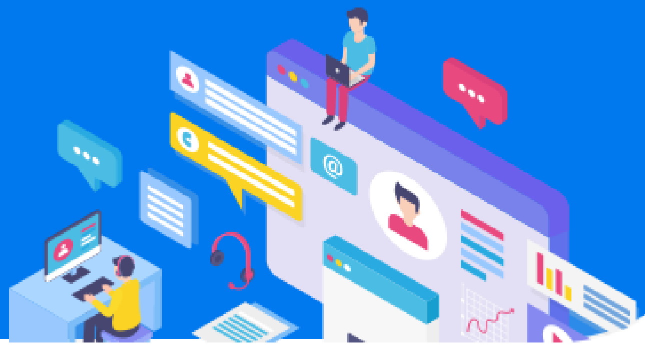
You have been asked to implement Pod security policies to enhance container security.

Outcome:

By the end of this demo, you will be able to implement Pod Security Policies to enhance the security of containerized applications in Kubernetes.

Note: Refer to the demo document for detailed steps

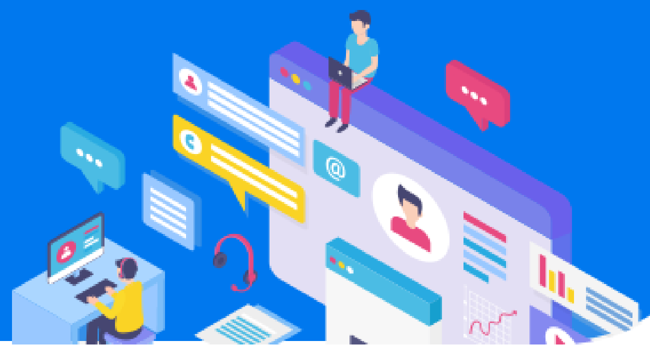
Assisted Practice: Guidelines



Steps to be followed:

1. Create a Pod Security Policy
2. Create a role and role binding for PSP
3. Test the Pod Security Policy
4. Delete the resources

Assisted Practice



Scanning Kubernetes Cluster Resources Using the Trivy CLI **Duration: 10 Min.**

Problem statement:

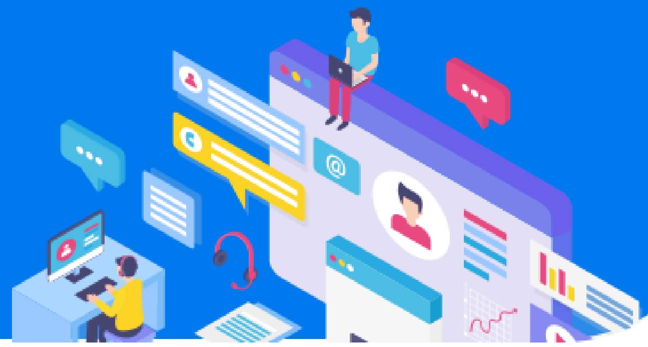
You have been asked to deploy an application on a Kubernetes cluster and scan its resources to detect dynamic runtime vulnerabilities in both application and cluster configuration.

Outcome:

By the end of this demo, you will be able to deploy an application on a Kubernetes cluster and use Trivy CLI to scan its resources, detecting dynamic runtime vulnerabilities in both the application and cluster configuration.

Note: Refer to the demo document for detailed steps

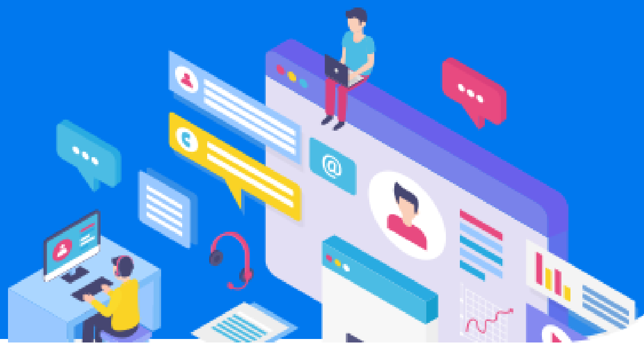
Assisted Practice: Guidelines



Steps to be followed:

1. Create a namespace and deploy an application in the cluster
2. Verify the application deployment
3. Install Trivy for Kubernetes
4. Scan the application resources within a cluster using the Trivy CLI

Assisted Practice



Enforcing Kubernetes Policies Using Gatekeeper **Duration: 10 Min.**

Problem statement:

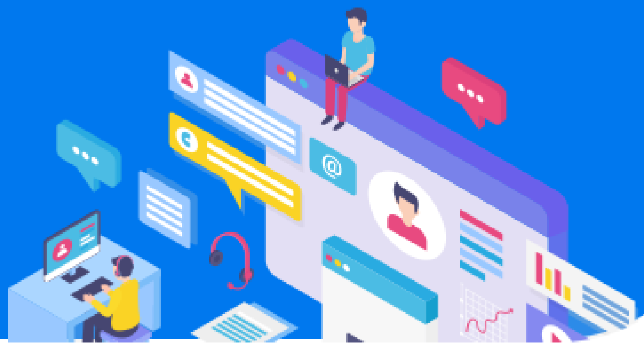
You have been asked to enforce Kubernetes policies for pod resource limits and namespace creation using the open policy agent (OPA) Gatekeeper.

Outcome:

By the end of this demo, you will be able to enforce Kubernetes policies for pod resource limits and namespace creation using the Open Policy Agent (OPA) Gatekeeper.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Install OPA Gatekeeper
2. Create a ConstraintTemplate and constraint for the resource limit policy
3. Test the resource limit policy
4. Create a ConstraintTemplate and constraint for the namespace policy
5. Test the namespace policy

Quick Check

A company is implementing DevSecOps to enhance the security of their Kubernetes environment. They have identified several security challenges including access control, image security, secret management, and runtime security. Which of the following best demonstrates how DevSecOps can address these challenges?

A. Implementing automated vulnerability scans in CI/CD pipelines

B. Increasing the number of manual security checks

C. Isolating Kubernetes clusters from CI/CD pipelines

D. Limiting access to Kubernetes dashboards to specific IP addresses

Key Takeaways

- Common challenges in code security include insecure code, dependencies, and inadequate risk assessments.
- Best practices for container security involve regular scanning of containers, using trusted images, and restricting access by avoiding privileged user accounts.
- Cluster security measures include protecting the control plane and nodes, implementing strict access controls, using network policies to control traffic, and continuously monitoring and logging activities.
- DevSecOps tools for Kubernetes include Devtron, StackRox, Snyk, OWASP, and Checkov.

Key Takeaways

- Benefits of DevSecOps in Kubernetes include proactive security, continuous improvement, consistent security practices, enhanced collaboration, scalability, and flexibility.
- Implementing security practices involves proactive measures and continuous monitoring using steps and tools for best practices in Kubernetes environments.

Thank You