CADT

ASEAN CYBER SHIELD

Building a Kubernetes Cloud Security Infrastructure

Submitted by: AnonyMeow

Published date: 30th August 2024

asean

KISA
Korea Internet & Security Agency

AKCF

Table of Contents

1. Introduction

1.1 Background

Kubernetes, a platform for orchestrating containers that are source has changed the game when it comes to how applications are put into action and handled in cloud settings. By taking charge of deploying, adjusting the scale of, and managing application containers across groups of hosts Kubernetes provides an adaptable toolkit for up-to-date DevOps methods. As companies increasingly move towards designs Kubernetes has become a key element of their systems offering notable benefits in scalability, flexibility and resilience. Nonetheless, as reliance on Kubernetes grows, ensuring the security of these deployments has turned into a worry. Cloud security. Which involves practices and technologies aimed at safeguarding data, applications, and infrastructure, in cloud environments. Now needs to tackle the challenges posed by Kubernetes. These challenges involve protecting containerized applications, the foundation infrastructure they run on and the fluid environment of Kubernetes where workloads can be swiftly initiated, modified or erased.

1.2. Importance of Securing Kubernetes Infrastructure

Securing the Kubernetes infrastructure is crucial, due to its role in managing business applications. Any weaknesses or errors in the Kubernetes setup can result in outcomes, like data breaches, service interruptions or unauthorized entry. With the increasing number of cyber threats aimed at Kubernetes organizations must prioritize implementing security protocols.

Securing a Kubernetes environment involves addressing various aspects of security, including container security, network security, authentication and authorization, and compliance with industry standards. Organizations must also consider the security of the entire software supply chain, from code development to deployment. Ensuring that security is integrated into every stage of the Kubernetes lifecycle is essential for maintaining the integrity, availability, and confidentiality of applications and data.

1.3. Objective

This project aims to provide a comprehensive guide to building a secure Kubernetes cloud infrastructure. It will explore the key components necessary for securing a Kubernetes environment, along with best practices that organizations can adopt to mitigate risks and enhance security. The report will cover various aspects of Kubernetes security, including Container security, Network security, Authentication and Authorization, Compliance, and Monitoring.

2. Understanding Kubernetes and Cloud Security Basics

2.1 Introduction to Kubernetes

Kubernetes is an open-source platform that helps you orchestrate and manage your container infrastructure on-premises or in the cloud. It’s a container-centric management environment. Google originated it and

donated it to the open-source community. Now it's a project of the vendor-neutral Cloud Native Computing Foundation. It automates the deployment, scaling, load balancing, logging, monitoring, and other management features of containerized applications. These are the features that are characteristic of typical platform-as-a-service solutions.

Kubernetes supports declarative configurations. When you administer your infrastructure declaratively, you describe the desired state you want to achieve, instead of issuing a series of commands to achieve that desired state. Kubernetes's job is to make the deployed system conform to your desired state and keep it there despite failures. Declarative configuration saves your work. Because the system's desired state is always documented, it also reduces the risk of error.

Kubernetes also allows imperative configuration, in which you issue commands to change the system's state. But administering Kubernetes at scale imperatively would be a big, missed opportunity. One of the primary strengths of Kubernetes is its ability to keep a system in a state you declare automatically. Experienced Kubernetes administrators use imperative configuration only for quick temporary fixes and as a tool in building a declarative configuration.

2.2 Kubernetes Components and Architecture

Kubernetes architecture is designed to ensure the automated and scalable management of containerized applications. The architecture is composed of a control plane and nodes, each playing a distinct role in the operation of the cluster.

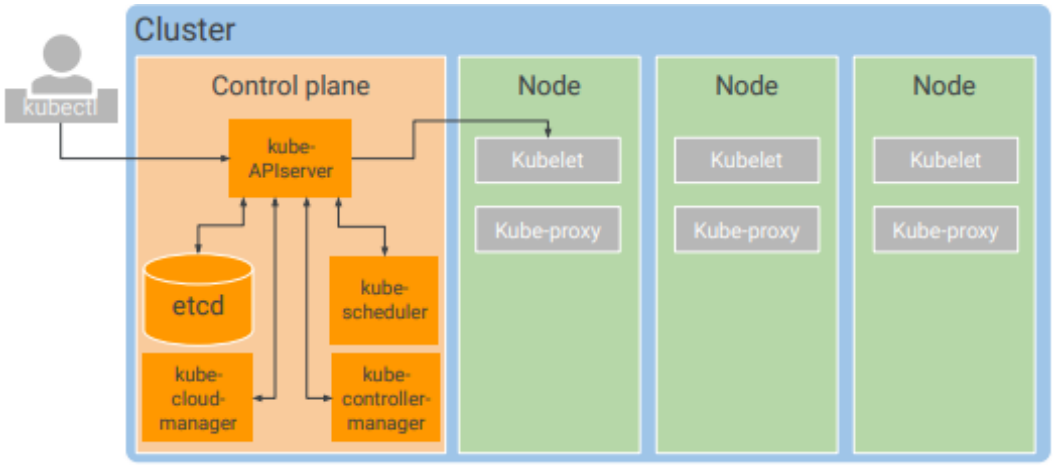

Figure 1: Kubernetes Architecture

Nowadays the computers that compose your clusters are usually virtual machines. One computer is called the "control plane," and the others are called simply "nodes." The job of the nodes is to run Pods. The job of the control plane is to coordinate the entire cluster. We will discuss its control-plane components first.

Several critical Kubernetes components run on the control plane. The single component that you interact with directly is the kube-apiserver. This component's job is to accept commands that view or change the state of

the cluster, including launching Pods. we will use the kubectl command frequently; this command's job is to connect to kube-apiserver and communicate with it using the Kubernetes API. kube-apiserver also authenticates incoming requests, determines whether they are authorized and valid, and manages admission control. But it's not just kubectl that talks with kube-apiserver. In fact, any query or change to the cluster's state must be addressed to the kube-apiserver.

etcd is the cluster's database. Its job is to reliably store the state of the cluster. This includes all the cluster configuration data; and more dynamic information such as what nodes are part of the cluster, what Pods should be running, and where they should be running. You never interact directly with etcd; instead, kube-apiserver interacts with the database on behalf of the rest of the system.
kube-scheduler is responsible for scheduling Pods onto the nodes. To do that, it evaluates the requirements of each Pod and selects which node is most suitable. But it doesn't do the work of launching Pods on Nodes. Instead, whenever it discovers a Pod object that doesn't yet have an assignment to a node, it chooses a node and simply writes the name of that node into the Pod object. Another component of the system is responsible for then launching the Pods.

kube-controller-manager has a broader job. It continuously monitors the state of a cluster through Kube-APIserver. Whenever the current state of the cluster doesn't match the desired state, kube-controller-manager will attempt to make changes to achieve the desired state. It's called the "controller manager" because many Kubernetes objects are maintained by loops of code called controllers. These loops of code handle the process of remediation. Controllers will be very useful to you. To be specific, you'll use certain kinds of Kubernetes controllers to manage workloads.

Kubelet is an agent that runs on each node, ensuring that containers are running in a pod as expected. It communicates with the control plane, receiving instructions and reporting on node status.

kube-proxy is responsible for maintaining network rules on nodes, allowing network communication to pods from inside or outside the cluster. It manages routing and load balancing within the cluster.

2.3 The Four C's of Cloud-Native Security

As organizations increasingly adopt cloud-native technologies, the need for robust security practices becomes paramount. Cloud-native security, which focuses on securing applications and infrastructure designed to run in cloud environments, is often conceptualized through the framework of the "Four C's": Code, Container, Cluster, and Cloud. This framework provides a comprehensive approach to securing every layer of the cloud-native stack, ensuring that security is integrated throughout the development and deployment lifecycle.

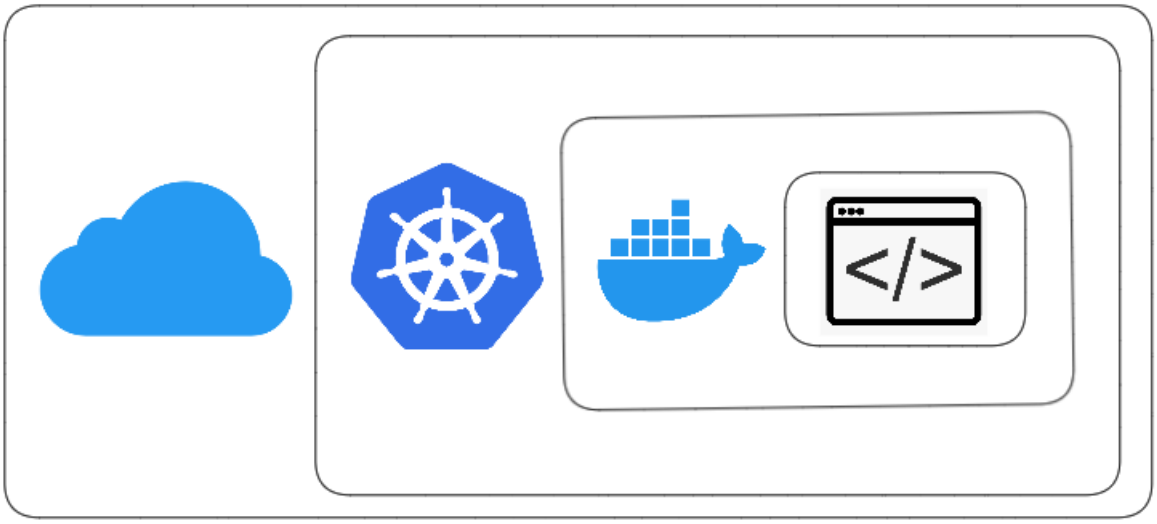

Figure 2: 4 C Cloud-Native Security

The first "C" in the cloud-native security framework is Code. Security begins at the code level, where developers are responsible for writing secure, clean, and efficient code. Vulnerabilities introduced at this stage can propagate throughout the entire cloud-native environment, making it crucial to establish secure coding practices from the outset. Secure coding practices include input validation, error handling, and the use of secure libraries and frameworks. Developers should also adhere to the principle of least privilege, ensuring that code executes with the minimum level of permissions necessary to function.

The second layer in the Four C's framework is Container security. Containers encapsulate applications and their dependencies, providing a consistent environment across development, testing, and production. However, the portability and flexibility of containers also introduce unique security challenges. Securing containers involves ensuring that the container images are free from vulnerabilities and are built from trusted sources. Organizations should use minimal base images to reduce the attack surface and regularly scan these images for known vulnerabilities.

The third "C" in the cloud-native security model is Cluster security, which focuses on securing the Kubernetes or other orchestration platforms that manage containerized applications. The cluster represents the operational environment where containers run, and securing this layer is crucial to maintaining the integrity and availability of cloud-native applications. cluster security begins with the proper configuration of the orchestration platform.

The final layer of the Four C's framework is Cloud security. This layer addresses the security of the underlying cloud infrastructure provided by cloud service providers (CSPs), such as AWS, Azure, or Google Cloud. The cloud represents the foundation upon which all other layers are built, making its security critical to the overall security of cloud-native applications. cloud security involves a shared responsibility model, where the CSP is responsible for securing the infrastructure, while the customer is responsible for securing the applications, data, and services running on that infrastructure. Organizations must understand and implement security

controls provided by the CSP, such as identity and access management (IAM), encryption, and monitoring tools.

3. Kubernetes Security Concepts

3.1 Kubernetes Security Fundamentals

3.1.1 Role-Based Access Control (RBAC)

Kubernetes RBAC is a key security control that ensures that cluster users and workloads have access only to resources required to execute their roles. It is important to ensure that, when designing permissions for cluster users, the cluster administrator understands the areas where privilege escalation could occur, to reduce the risk of excessive access leading to security incidents.

a. RBAC concepts

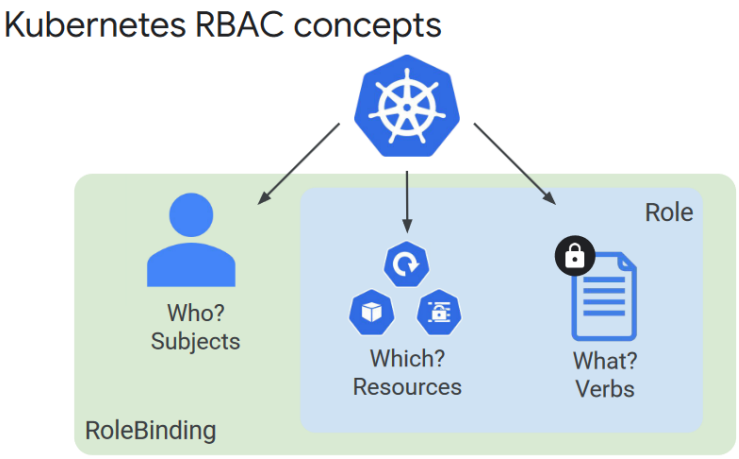

Figure 3

There are three main elements to Kubernetes role-based access control: subjects, resources, and verbs. With Kubernetes RBAC you define what operations (verbs) can be executed over which objects (resources) by who (subjects). A subject is a set of users or processes that can make requests to the Kubernetes API. A resource is a set of Kubernetes API objects, such as Pods, Deployments, Services, or PersistentVolumes.

A verb is a set of operations that can be performed on resources, such as get, watch, create, and describe. These three elements can be connected by creating two types of RBAC API objects: Roles and RoleBindings. Roles connect API resources and verbs. RoleBindings connect roles to subjects. Roles and RoleBindings are applied at the cluster level or namespace level.

b. Kubernetes RBAC components

Kubernetes RBAC is helpful if you need to grant users access to specific Kubernetes namespaces. You add users as subjects, configure the roles with the proper permissions, and then bind them to the objects—in this case, the namespaces in the cluster. In Kubernetes, there are two types of roles: Role and ClusterRole.

RBAC roles are defined at the namespace level, and RBAC ClusterRoles are defined at the cluster level.

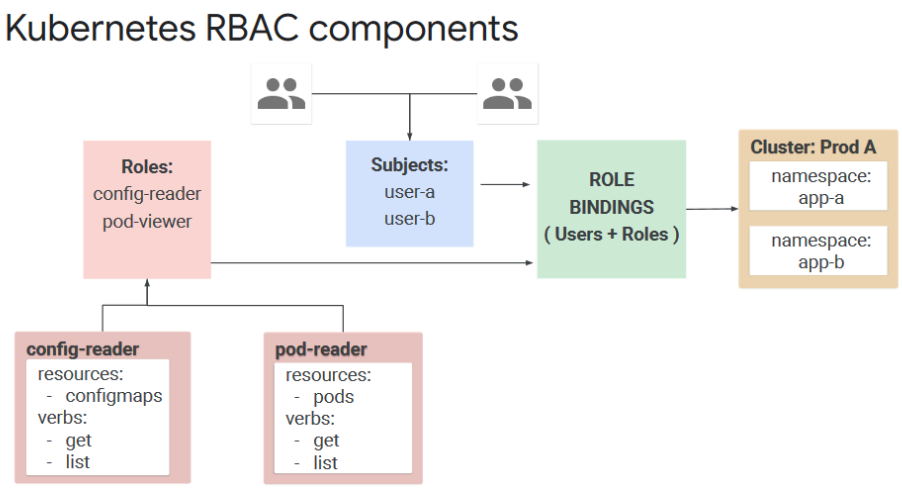

Figure 4

- **Role**: A Role is a collection of permissions that allow users to perform specific actions (verbs, such as "get", "create", "and "delete") on a defined set of Kubernetes resource types (such as Pods, Deployments, and Namespaces). Roles are namespaced objects; the permissions they grant only apply within the namespace that the Role belongs to.
- **ClusterRole**: ClusterRoles work similarly to Roles but are a non-namespaced alternative for cluster-level resources. You'll need to use a ClusterRole to control access to objects such as Nodes, which don't belong to any namespace. ClusterRoles also allows you to globally access namespaced resources across all namespaces, such as every Pod in your cluster.
- **RoleBinding**: RoleBindings represent the links between your Roles and Users or Service Accounts. A RoleBinding lets you reference a Role, and then grant those permissions to one or more users (termed Subjects).
- **ClusterRoleBinding**: ClusterRoleBinding is equivalent to RoleBinding, but targets ClusterRole resources instead of Roles.
- **Users:** In Kubernetes, a User represents a human who authenticates to the cluster using an external service. You can use private keys (the default method), a list of usernames and passwords, or an OAuth service such as Google Accounts. Users are not managed by Kubernetes; there is no API object for them so you can't create them using Kubectl. You must make changes to the external service provider.
- **Service Accounts:** Service Accounts are token values that can be used to grant access to namespaces in your cluster. They're designed for use by applications and system components. Unlike Users, Service Accounts are backed by Kubernetes objects and can be managed using the API.

3.1.2 Network Policies

Network Policies are a mechanism for controlling network traffic flow in Kubernetes clusters. They allow you to define which of your Pods are allowed to exchange network traffic. You should use them in your clusters to prevent apps from reaching each other over the network, which will help limit the damage if one of your apps is compromised.

Each Network Policy you create targets a group of Pods and sets the Ingress (incoming) and Egress (outgoing) network endpoints those Pods can communicate with.

a. Network Policies work

Network Policies can set a different list of allowed targets for their Ingress and Egress rules. It's also possible to use Network Policies to block all network communications for a Pod or restrict traffic to a specific port range.

Network Policies are additive, so you can have multiple policies targeting a particular Pod. The sum of the "allow" rules from all the policies will apply. Traffic from or to sources that don't match any of the "allow" rules will be blocked if the target Pod is also covered by a "deny" policy.

b. Network Policies use cases

Network Policies is the best practice for a secure Kubernetes configuration. They prevent Pod network access from being unnecessarily broad, such as in the following scenarios:

- Ensuring a database can only be accessed by the app it's part of: Databases running in Kubernetes are often intended to be solely accessed by other in-cluster Pods, such as the Pods that run your app's backend. Network Policies allow you to enforce this constraint, preventing other apps from communicating with your database server.
- Isolating Pods from your cluster's network: Some sensitive Pods might not need to accept any inbound traffic from other Pods in your cluster. Using a Network Policy to block all Ingress traffic to them will tighten your workload's security.
- Allow specific apps or namespaces to communicate with each other: Kubernetes namespaces are the primary mechanism for separating objects associated with different apps, teams, and environments. You can use Network Policies to network-isolate these resources and achieve stronger multi-tenancy.

3.1.3 Secrets management

Secrets Management is a process where you manage secrets, like SSH keys, database credentials, certificates, and API keys in a secure, centralized way. Access is controlled by role-based access control and policies.

In Kubernetes, secrets contain all kinds of sensitive information. For instance, database credentials or API keys. The term secrets management describes the centralized and secured management of these secrets.

Sebastiaan Kok, Innovation Engineer at True, explains the importance of secret management and why this should be a key focus. He also points out how True handles secrets management itself and thus supports its customers in keeping their secrets safe.

a. Kubernetes Secrets management work

Kubernetes Secrets allow users to store data in a secret object, which can then be accessed or modified by Kubernetes clients. A Secret is a key-value pair that includes a Secret ID and authenticates information. Kubelet uses the Secret ID to identify the credentials that must be provided to an application container when creating a pod.

Kubelet runs on each node in a cluster and manages pods and their containers. Secrets are only accessible by pods if they're explicitly part of a mounted volume or now kubelet pulls an image for use in a pod.

The Kubernetes API server stores Kubernetes Secrets. They're only accessible to specific users with permission to access the Kubernetes API server. The Kubernetes API server is a single point of failure for our application, so we must keep our data safe and secure.

b. Practices for Kubernetes Secrets Management

Secrets, like keys, passwords, tokens, and other configuration values, must be stored correctly. If our Kubernetes cluster is compromised, Secrets must remain secure. An attacker shouldn't be able to exploit Secrets to compromise sensitive data, build a botnet, or command and control (C2) servers.

Here are some techniques to help us keep Kubernetes Secrets safe:

- Enable encryption at rest
- Configure RBAC rules
- Encrypt etcd data
- Use a centralized Secrets store for easy management

3.1.4 API server, etcd security, and container runtime security

a. API server

The Kubernetes API is a resource-based (RESTful) programmatic interface provided via HTTP. It supports retrieving, creating, updating, and deleting primary resources via the standard HTTP verbs (POST, PUT, PATCH, DELETE, GET).

b. Etcd security

Etcd is an open-source distributed key-value store that is used to store and manage the information that distributed systems need for their operations. It stores the configuration data, state data, and metadata in Kubernetes.

The name “etcd” comes from a naming convention within the Linux directory structure: In UNIX, all system configuration files for a single system are contained in a folder called “/etc;” “d” stands for “distributed.”

3.2 Secure Configuration and Best Practices

3.2.1 Secure configuration of Kubernetes components

Securing Kubernetes components is crucial to maintaining a safe and stable cluster environment. Kubernetes is composed of several core components, including the API server, etcd, controller manager, scheduler, and kubelet, each of which requires careful configuration to ensure security. Here's a detailed breakdown of how to securely configure these components:

a. API Server

The API server (kube-apiserver) is the central control point for managing Kubernetes clusters. It handles all RESTful interactions with the cluster, making it a primary target for attackers.

Key Security Configurations:

- Authentication: Ensure that all clients interacting with the API server are authenticated. Kubernetes supports various authentication methods, including client certificates, bearer tokens, and external authentication providers like OpenID Connect. This helps verify the identity of users and service accounts accessing the cluster.
- RBAC (Role-Based Access Control): Enable and configure RBAC to define what authenticated users and service accounts can do within the cluster. RBAC policies should adhere to the principle of least privilege, ensuring that entities only have the permissions they need.
- TLS Encryption: Use TLS to encrypt all communications with the API server. This includes securing the API server's communication with clients, nodes, and etcd. Proper certificate management is critical for maintaining this encryption.
- Audit Logging: Enable audit logs to capture all API requests and responses. Audit logs are vital for tracking access and changes within the cluster, which helps in detecting unauthorized activities and diagnosing issues.
- Restrict API Access: Use network policies or firewall rules to restrict access to the API server, ensuring that only trusted sources, such as authorized users and internal components, can communicate with it.

b. Etcd

etcd is a key-value store that Kubernetes uses to store all cluster data, including configuration data and secrets. Given its critical role, securing etcd is essential.

Key Security Configurations:

- TLS for etcd: Ensure that all etcd client-server and peer-to-peer communications are encrypted using TLS. This protects the data in transit and helps prevent unauthorized access.
- Authentication and Authorization: Implement authentication and authorization controls to restrict access to etcd. Only the Kubernetes API server and necessary management components should be able to interact with etcd.
- Access Control: Limit access to etcd to minimize the attack surface. Use firewall rules or network segmentation to prevent unauthorized access.
- Data Encryption: Encrypt etcd data at rest to protect sensitive information, such as secrets, even if an attacker gains access to the storage.
- Regular Backups: Perform regular backups of etcd data and ensure that these backups are encrypted and stored securely. This allows for recovery in case of data corruption or loss.

c. Controller Manager

The kube-controller-manager is responsible for running various controllers that regulate the cluster's state, such as ensuring desired pod replicas or managing node lifecycle events.

Key Security Configurations:

- Secure Communication: Ensure that the controller manager communicates securely with the API server using TLS. This prevents man-in-the-middle attacks and eavesdropping on sensitive data.
- Principle of Least Privilege: Configure the controller manager with minimal permissions required to perform its functions. Avoid over-provisioning permissions, which could be exploited if the controller manager is compromised.
- Monitoring and Logging: Enable logging for the controller manager to monitor its activities. This can help in detecting anomalous behavior or potential security incidents.

d. Kubelet

The kubelet is an agent that runs on every node in the cluster, responsible for ensuring that containers are running as expected.

Key Security Configurations:

- Kubelet Authentication and Authorization: Secure the kubelet API with authentication and authorization to control who can interact with it. Unauthorized access to the kubelet can lead to node or pod compromises.
- TLS Encryption: Encrypt communication between the kubelet and the API server, as well as between the kubelet and the container runtime, using TLS. This protects data in transit and ensures the integrity of communications.
- Restrict Access: Limit access to the kubelet's API to only trusted components and users. Network policies or firewall rules can help enforce this restriction.
- Security Profiles: Enforce security profiles like Seccomp and AppArmor for containers managed by the kubelet. These profiles restrict the actions that containers can perform, reducing the risk of container escapes or other security breaches.

e. Network Security

Network security is essential for protecting communications between Kubernetes components, nodes, and external clients.

Key Security Configurations:

- Network Policies: Implement Kubernetes Network Policies to control traffic between pods, namespaces, and external services. This limits the attack surface by enforcing least-privilege network access.
- Secure Ingress and Egress: Manage ingress and egress traffic using firewalls, ingress controllers, and egress policies. Ensure that only necessary services are exposed and that they are securely configured.
- Pod-to-Pod Encryption: Consider encrypting communication between pods, especially for sensitive data or communications between critical components. This can be achieved using service mesh solutions or custom encryption layers.

3.2.2 Cluster hardening best practices

Securing a Kubernetes cluster is critical to ensuring the protection of applications and data. Below are key best practices for Kubernetes cluster hardening, designed to bolster security and minimize vulnerabilities:

- Restrict API Access
    - Control API Access: Limit access to the Kubernetes API server by implementing strong authentication and authorization mechanisms. Use Role-Based Access Control (RBAC) to ensure that only authorized users and services can interact with the API.
    - Use Secure Endpoints: Ensure that the API server is only accessible over secure connections (HTTPS).
- Secure etcd

    - Encrypt Data: Since etcd stores all cluster data, including sensitive information, it's vital to enable encryption for data at rest. Encrypt etcd communication using Transport Layer Security (TLS) to prevent unauthorized access.
    - Restrict Access: Limit access to etcd to only those components that need it, such as the API server. This minimizes the risk of unauthorized data retrieval.
- Pod Security Policies
    - Pod Security Standards: Use Pod Security Policies (PSPs) or adopt the newer Pod Security Standards to enforce rules about how pods are allowed to operate within the cluster. This includes restrictions on running privileged containers, using host namespaces, or allowing containers to run as root.
    - Namespace Segmentation: Segregate workloads into different namespaces and apply security policies specific to each namespace. This can prevent a breach in one namespace from affecting the entire cluster.
- Network Policies
    - Restrict Pod Communication: Implement network policies to control which pods can communicate with each other and with services outside the cluster. Network segmentation limits the potential spread of an attack within the cluster.
    - Limit External Exposure: Carefully manage ingress and egress rules to ensure that only necessary traffic is allowed in and out of the cluster. This reduces the attack surface exposed to external threats.
- Encrypt Secrets and Sensitive Data
    - Use Kubernetes Secrets: Store sensitive information, such as API keys and passwords, in Kubernetes Secrets rather than in plain text within environment variables or ConfigMaps. Ensure these secrets are encrypted both at rest and in transit.
    - Limit Secret Access: Use RBAC to restrict which users and services can access specific secrets, ensuring that only those with a legitimate need can retrieve sensitive data.
- Monitor and Audit Activity
    - Enable Logging: Implement comprehensive logging for all Kubernetes components, including the API server, etcd, and kubelet. Use centralized logging solutions to aggregate and analyze logs for suspicious activity.
    - Conduct Regular Audits: Regularly audit cluster activity, including API server requests and access to sensitive resources, to detect potential security incidents. Tools like kube-bench can help automate security checks against industry benchmarks.

3.3 Monitoring and Security Tools

3.3.1 DigitalOcean Monitoring

DigitalOcean Monitoring is a free, opt-in service that gathers and displays metrics about Droplet-level resource utilization. Monitoring supports configurable alert policies with integrated email and Slack notifications to help us track the operational health of your infrastructure.

3.3.2 Kubernetes-native security tool (Kube-bench)

Kube-bench is an open-source tool to assess the security of Kubernetes clusters by running checks against the Center for Internet Security (CIS) Kubernetes benchmark. It was developed in **GoLang** by Aqua Security, a provider of cloud-native security solutions.

Kubernetes CIS benchmarks cover security guidelines & recommendations for the following:

- **Control Plane Components**: Control plane node configurations & component recommendations.
- **Worker Nodes**: Worker node configurations and Kubelet.
- **Policies**: RBAC, service accounts, Pod security standards, CNI and network policies, Secret Management, etc.

Kube-bench can help with the following:

- **Cluster hardening**: Kube-bench automates the process of checking the cluster configuration as per the security guidelines outlined in CIS benchmarks.
- **Policy Enforcement**: Kube-bench checks for RBAC configuration to ensure the necessary least privileges are applied to service accounts, users, etc. it also checks for pod security standards and secret management.
- **Network segmentation:** Kube-bench checks for CNI and its support for network policy to ensure that network policies are defined for all namespaces.

You can run kube-bench checks against a cluster in two ways:

- From the command line using kube-bench CLI
- Run inside a pod

4. Practical Implementation

4.1 Infrastructure Overview

This project centers around deploying applications to a Kubernetes (K8s) cluster on Digital Ocean, leveraging a robust CI/CD pipeline that integrates GitHub Actions, Docker Registry, and ArgoCD.

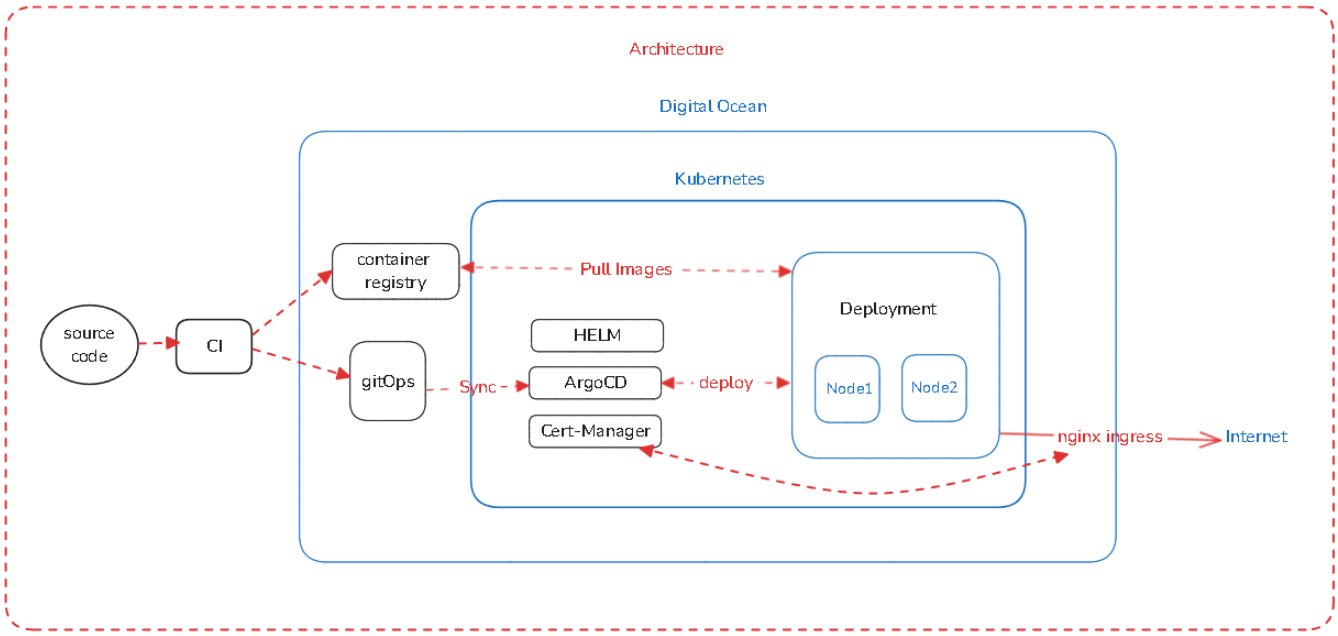

Figure 5: Architecture

CI/CD Pipeline:

- **GitHub Actions**: Automates the build, test, and deployment processes. Upon code changes in the GitHub repository, GitHub Actions triggers workflows that build Docker images, run tests, and push the images to the Docker Registry.
- **Docker Registry**: Hosted on Digital Ocean, this registry stores the Docker images that are later deployed to the Kubernetes cluster.
- **ArgoCD**: Manages the continuous deployment of applications to the Kubernetes cluster. ArgoCD continuously monitors the GitHub repository and syncs the application state to the desired state defined in the repository.

Kubernetes Infrastructure:

- **Helm**: A package manager for Kubernetes, Helm is used to manage Kubernetes manifests, making deployment and management of applications more efficient. Helm charts streamline the installation of packages like Cert-Manager and NGINX Ingress Controller.
- **Cert-Manager**: Automates the management and issuance of TLS certificates, ensuring that all traffic to and from the applications is encrypted.
- **NGINX Ingress Controller**: Facilitates the routing of external traffic to the Kubernetes services, ensuring secure access through HTTPS, utilizing the certificates managed by Cert-Manager.
- **Digital Ocean Load Balancer**: Distributes incoming network traffic across multiple servers, ensuring availability and reliability of the deployed applications.

4.2 DigitalOcean Kubernetes Cluster Setup

Setting up a Kubernetes cluster on DigitalOcean is a crucial part of this project. The process involves creating a managed Kubernetes cluster, configuring the necessary resources, and ensuring the cluster is ready for application deployment and security implementation.

4.2.1 Creating the Kubernetes Cluster

- **DigitalOcean Control Panel**: Begin by logging into the DigitalOcean control panel. Navigate to the "Kubernetes" section and click on "Create a Cluster."
- **Cluster Configuration**: Choose the desired region for the cluster, such as New York or Amsterdam, to minimize latency based on your application's user base. Select the version of Kubernetes, preferably the latest stable release, to ensure you have access to the most recent features and security updates.
- **Node Pools**: Configure the node pools by selecting the appropriate Droplet size (CPU and memory) and the number of nodes. A minimum of three nodes is recommended for high availability and load balancing.
- **Cluster Networking**: Opt for the default VPC network provided by DigitalOcean, which isolates your cluster within a private network. This ensures that your pods and services are securely networked within the cluster.

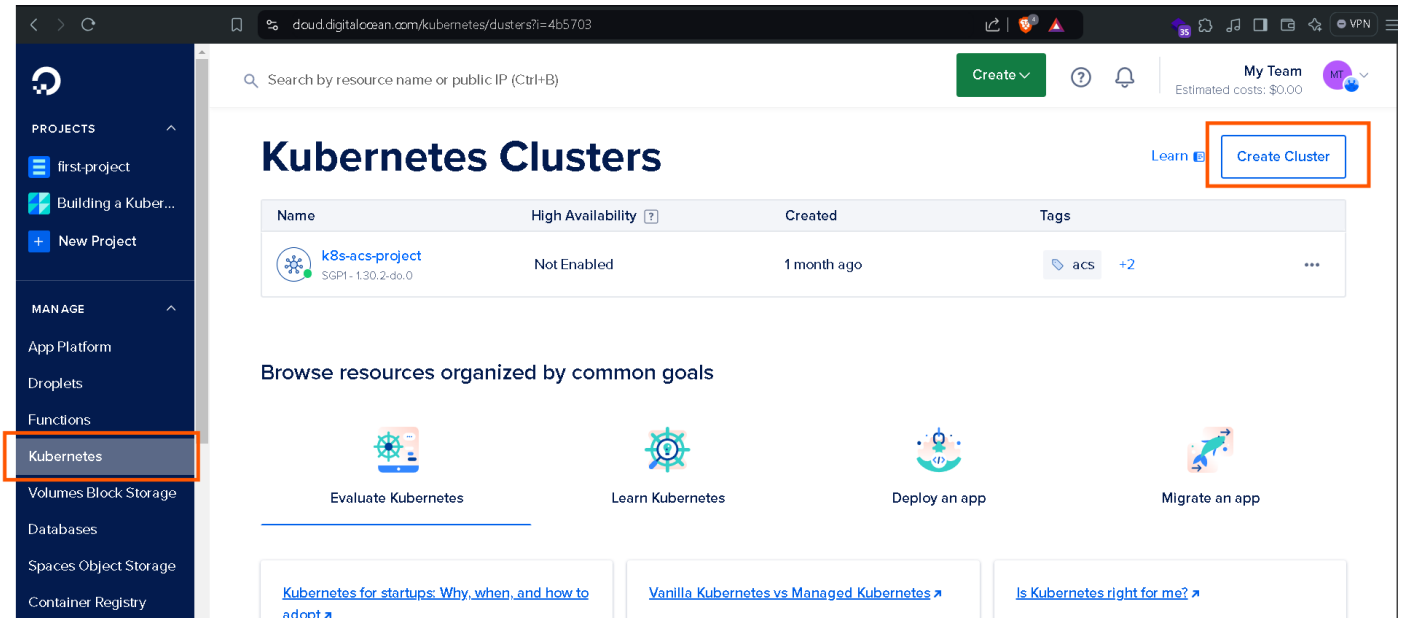

Figure 6: K8s inDigital Ocean

4.2.2 Install nginx ingress load balancer

The NGINX Ingress Controller is a vital component in a Kubernetes cluster, enabling the routing of external HTTP and HTTPS traffic to the services running within the cluster. Installing and configuring the NGINX Ingress Controller on your DigitalOcean Kubernetes cluster involves a series of steps, including setting up a Load Balancer to efficiently manage incoming traffic.

Add the NGINX Ingress Helm Repository: First, add the NGINX Ingress Helm repository and update it to ensure you're pulling the latest version of the chart:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
```

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginxhelm repo update ingress-nginx
```

Create the nginx-ingress-values.yml Configuration File: Create a custom YAML file to define specific configurations for the NGINX Ingress Controller:

```
controller:

replicaCount: 1

  resources:

    requests:

      cpu: 100m

      memory: 90Mi

  service:

    type: LoadBalancer

    annotations:

      service.beta.kubernetes.io/do-loadbalancer-enable-proxy-protocol: "true"

      service.beta.kubernetes.io/do-loadbalancer-tls-passthrough: "true"

      service.beta.kubernetes.io/do-loadbalancer-hostname: "acs.chivangoy.tech"

  config:

    use-proxy-protocol: "true"

    keep-alive-requests: "10000"

    upstream-keepalive-requests: "1000"

    worker-processes: "auto"

    max-worker-connections: "65535"

    use-gzip: "true"
```

Note: Update the service.beta.kubernetes.io/do-loadbalancer-hostname value to match your DNS configuration.

Install the NGINX Ingress Controller: Finally, install the NGINX Ingress Controller using Helm with the custom values specified in your YAML file:

```
helm install ingress-nginx ingress-nginx/ingress-nginx --namespace ingress-nginx --create-namespace -f nginx-ingress-values.yml
```

4.2.3 Install cert manager

Cert-Manager is a Kubernetes add-on that automates the management and issuance of TLS certificates from various sources, including Let's Encrypt. To set up Cert-Manager in your Kubernetes cluster, follow these steps:

Add the Jetstack Helm Repository: Start by adding the Jetstack Helm repository, which provides the Cert-Manager charts, and then update your Helm repositories:

```
helm repo add jetstack https://charts.jetstack.io

helm repo update jetstack
```

Create the cert-manager-values.yml Configuration File: Define the default values for Cert-Manager in a YAML file. This configuration ensures that the Custom Resource Definitions (CRDs) are installed, which are essential for Cert-Manager to function properly:

```
# Starter Kit default values for cert-manager.

# CRDs are required usually

installCRDs: true

# Required only if you want to monitor cert-manager activity

prometheus:

  enabled: false
```

Install Cert-Manager Using Helm: Install Cert-Manager with the custom values file. This step deploys Cert-Manager into your Kubernetes cluster:

```
helm install ingress-nginx ingress-nginx/ingress-nginx --namespace ingress-nginx --create-
namespace -f nginx-ingress-values.yml
```

Create the cert-issuer.yml Configuration File: Define a ClusterIssuer resource to configure Cert-Manager to use Let's Encrypt for issuing certificates. Create a YAML file with the following content:

```
apiVersion: cert-manager.io/v1

kind: ClusterIssuer

metadata:

  name: letsencrypt-prod

spec:

  # ACME issuer configuration

  # `email` - the email address to be associated with the ACME account (make sure it's a valid
one)

  # `server` - the URL used to access the ACME server's directory endpoint

  # `privateKeySecretRef` - Kubernetes Secret to store the automatically generated ACME account
private key

  acme:

    email: chiva.ngoy@student.cadt.edu.kh

    server: https://acme-v02.api.letsencrypt.org/directory

    privateKeySecretRef:

      name: letsencrypt-prod-private-key

    solvers:

      # Use the HTTP-01 challenge provider

      - http01:

          ingress:

            class: nginx
```

Apply the cert-issuer.yml Configuration: Apply the configuration to your cluster to create the ClusterIssuer resource:

```
kubectl apply -f cert-issuer.yml
```

4.2.4 Install argocd for continuous deployment

ArgoCD is a declarative, GitOps continuous delivery tool for Kubernetes that enables you to manage and deploy your applications using Git repositories. Follow these steps to install ArgoCD in your Kubernetes cluster:

Create the ArgoCD Namespace: Start by creating a dedicated namespace for ArgoCD:

```
kubectl create namespace argocd
```

Install ArgoCD: Apply the ArgoCD installation manifest to deploy ArgoCD components into the **argocd** namespace:

```
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

Create the argocd-ingress-http.yml Configuration File: Define an Ingress resource to expose the ArgoCD server externally and configure TLS using Let's Encrypt. Create a YAML file with the following content:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: argocd-server-ingress
  namespace: argocd
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
    kubernetes.io/ingress.class: nginx
    kubernetes.io/tls-acme: "true"
    nginx.ingress.kubernetes.io/ssl-passthrough: "true"
    # If you encounter a redirect loop or are getting a 307 response code
    # then you need to force the nginx ingress to connect to the backend using HTTPS.
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
spec:
  rules:
    - host: argocd.chivangoy.tech
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: argocd-server
```

```
                port:
                  name: https
  tls:
    - hosts:
        - argocd.chivangoy.tech
      secretName: argocd-secret # do not change, this is provided by Argo CD
```

Note: Adjust the host field with your actual DNS configuration.

Apply the Ingress Configuration: Apply the configuration to create the Ingress resource for ArgoCD:

```
kubectl apply -f argocd-ingress-http.yml
```

Retrieve the Admin Password: Get the initial admin password for ArgoCD from the Kubernetes secret:

```
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" |
base64 -d; echo
```

Access the ArgoCD Web Interface: Navigate to your configured hostname in a web browser. Log in using the username admin and the password retrieved from the previous step.

4.2.5 Install private container registry on DigitalOcean

A private container registry is essential for securely storing and managing container images used in your Kubernetes deployments. DigitalOcean offers a managed private container registry service, which integrates seamlessly with their Kubernetes service. Follow these steps to set up a private container registry on DigitalOcean:

Create a Private Container Registry: Log in to the DigitalOcean Control Panel and create a private container registry by following these steps:

- Navigate to the **Container Registry** section.
- Click on **Create Registry**.
- Choose a name for your registry and select the region closest to your Kubernetes cluster for optimal performance.
- Click **Create Registry** to complete the setup.

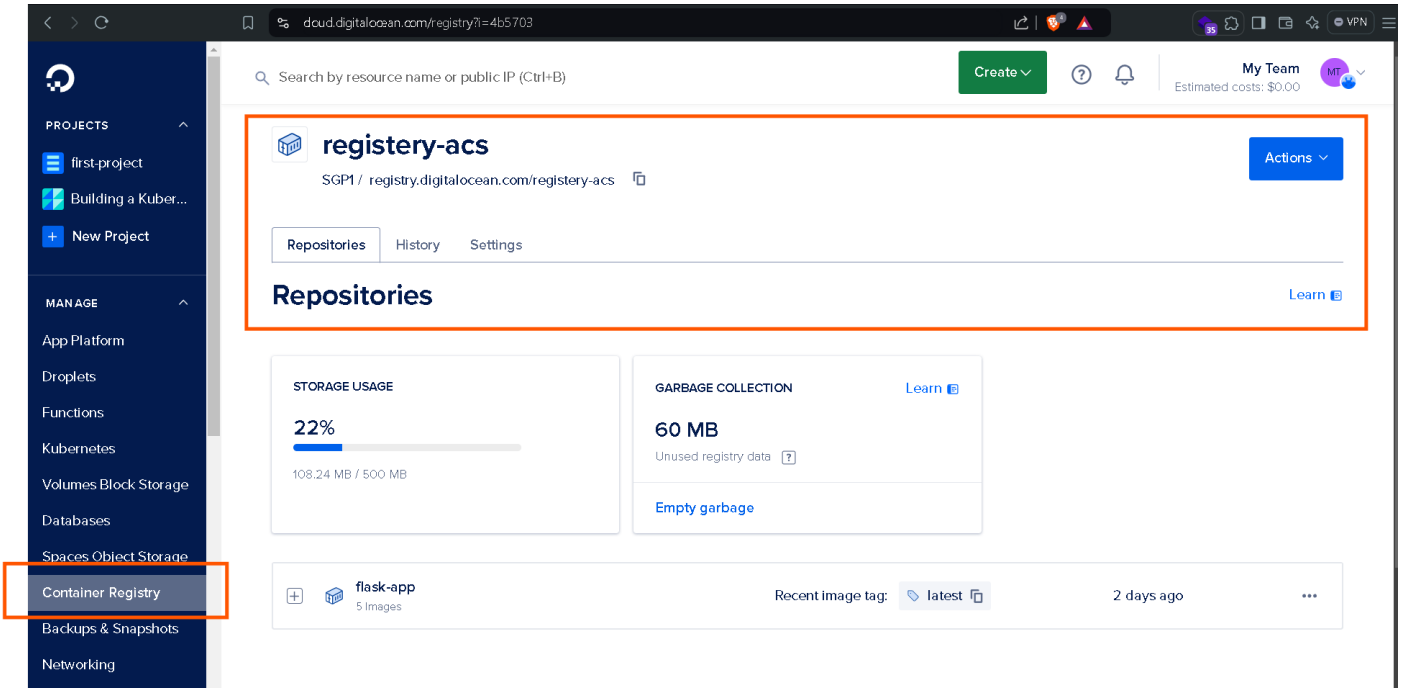

Figure 7: Container Registry

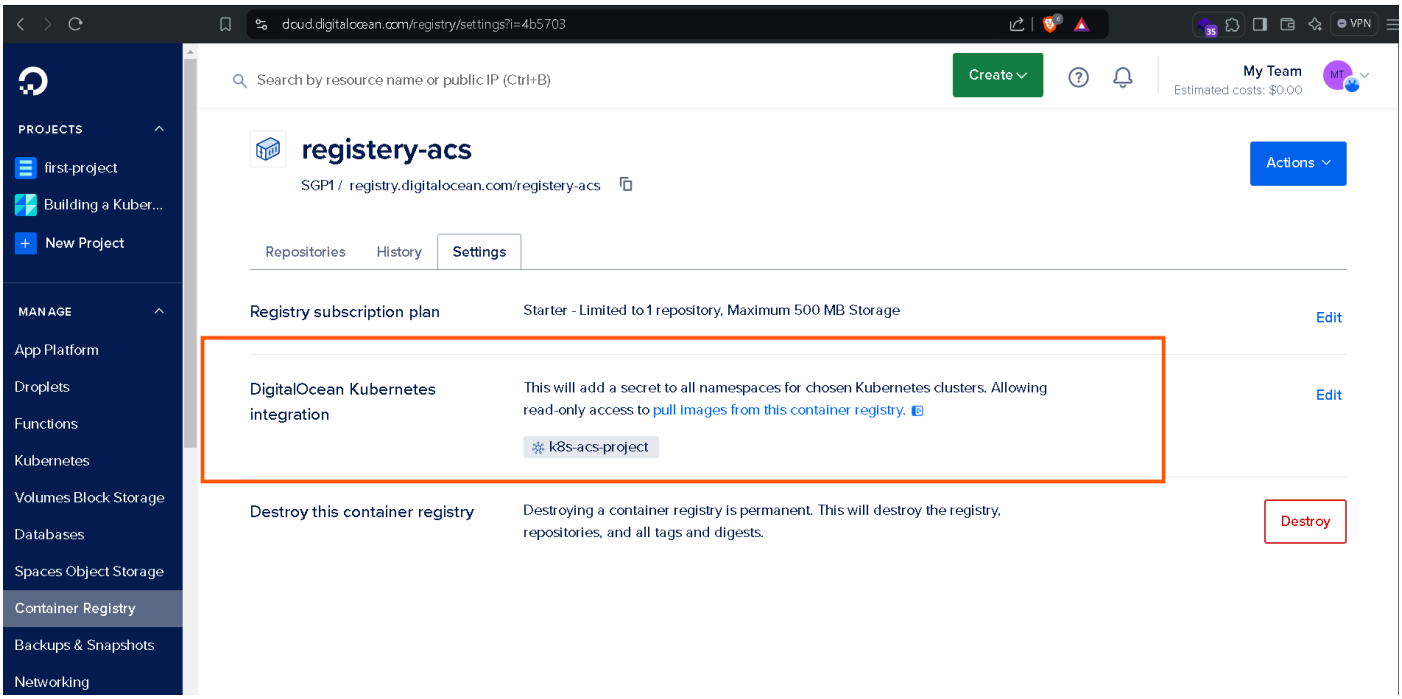

Figure 8: Integration container registry with k8s

4.3 Security implementation

4.3.1 Securing network

a. Using vpc on DigitalOcean

To establish a secure network environment for our Kubernetes cluster, we have utilized a Virtual Private Cloud (VPC) on DigitalOcean. A VPC allows us to create an isolated network environment, ensuring that our

Kubernetes resources are securely separated from other tenants on the cloud. The key features and benefits of using a VPC in our setup include:

- **Isolation:** The VPC provides network isolation, preventing unauthorized access to our resources from outside the defined network. This isolation is crucial for maintaining the confidentiality and integrity of our infrastructure.
- **Private Networking:** Within the VPC, our Kubernetes nodes communicate over a private network, eliminating the need to expose them to the public internet. This reduces the attack surface and enhances overall security.
- **Scalability:** The VPC setup on DigitalOcean allows us to easily scale our infrastructure without compromising security. As our needs grow, we can add more nodes and resources within the secure VPC environment.

b. Cilium and Hubble

In our Kubernetes infrastructure, network security and observability are critical components. To address these needs, we have implemented Cilium and Hubble, both of which are open-source tools designed to enhance network security and provide deep observability.

Cilium is an open-source networking, observability, and security solution that seamlessly integrates with Kubernetes. It leverages Linux's eBPF (extended Berkeley Packet Filter) technology to enforce network security policies, allowing fine-grained control over network communications between services. By doing so, Cilium enhances the security posture of our Kubernetes cluster, ensuring that only authorized services can communicate with each other.

To secure the network, Cilium was installed into the Kubernetes cluster with the following command:

```
cilium install --version 1.16.1
```

Once installed, we verified the status of Cilium and its components using:

```
cilium status --wait
    /¯¯\
 /¯¯\__/¯¯\    Cilium:             OK
 \__/¯¯\__/    Operator:           OK
 /¯¯\__/¯¯\    Envoy DaemonSet:    OK
 \__/¯¯\__/    Hubble Relay:       OK
    \__/       ClusterMesh:        disabled

Deployment             hubble-relay       Desired: 1, Ready: 1/1, Available: 1/1
DaemonSet              cilium-envoy       Desired: 3, Ready: 3/3, Available: 3/3
```

```
DaemonSet              cilium             Desired: 3, Ready: 3/3, Available: 3/3
Deployment             hubble-ui          Desired: 1, Ready: 1/1, Available: 1/1
Deployment             cilium-operator    Desired: 1, Ready: 1/1, Available: 1/1
Containers:            cilium             Running: 3
                       hubble-ui          Running: 1
                       cilium-operator    Running: 1
                       hubble-relay       Running: 1
                       cilium-envoy       Running: 3
Cluster Pods:          25/25 managed by Cilium
Helm chart version:
```

The status output confirmed that all Cilium components, including the Cilium DaemonSet, Envoy DaemonSet, and Hubble Relay, were successfully deployed and running:

Hubble is a fully distributed networking and security observability platform built on top of Cilium. It utilizes eBPF to provide comprehensive visibility into the communication and behavior of services within the Kubernetes cluster. Hubble allows us to monitor and observe network traffic in real time, helping us to identify potential security issues and optimize network performance. To access Hubble's observability features, we port-forwarded Hubble to our local terminal using the command:

```
cilium hubble port-forward
```

With the port-forwarding in place, we were able to observe network traffic by running:

```
hubble observe
```

For a more interactive experience, the Hubble UI was accessed by running:

```
cilium hubble ui
```

Through the Hubble UI, we could visualize the network traffic, analyze service communication patterns, and gain insights into the security of our network. This level of observability is crucial for maintaining the security and integrity of our Kubernetes environment, enabling us to detect and respond to potential threats swiftly.

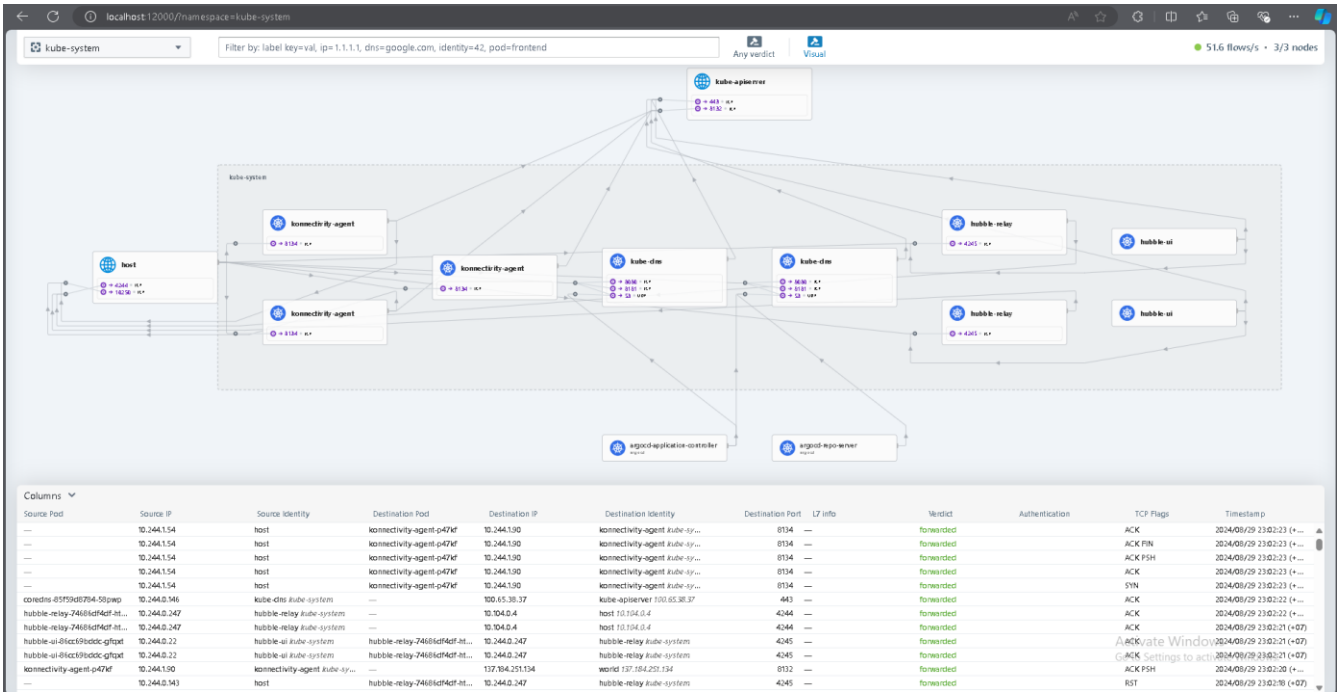

Figure 9: Network security and observability

we have implemented specific Cilium Network Policies to control and restrict traffic within the ArgoCD namespace. These policies ensure that only authorized communication is allowed between the various components of ArgoCD and between ArgoCD and external entities.

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: restrict-argocd-api
  namespace: argocd
spec:
  endpointSelector:
    matchLabels:
      app.kubernetes.io/name: argocd-server
  ingress:
  - fromEndpoints:
    - matchLabels:
        app.kubernetes.io/name: argocd-cli
    - matchExpressions:
      - {key: "k8s:io.cilium.k8s.policy.cluster", operator: "Exists"}
  - toPorts:
    - ports:
      - port: "443"
        protocol: TCP

---

apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: restrict-inter-component
  namespace: argocd
spec:
  endpointSelector:
    matchLabels:
      app.kubernetes.io/part-of: argocd
```

```
  egress:
  - toEndpoints:
    - matchLabels:
        app.kubernetes.io/name: argocd-repo-server
    toPorts:
    - ports:
      - port: "8081"
        protocol: TCP

---

apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: restrict-external-access
  namespace: argocd
spec:
  endpointSelector:
    matchLabels:
      app.kubernetes.io/name: argocd-server
  ingress:
  - fromEntities:
    - world
  - toPorts:
    - ports:
      - port: "443"
        protocol: TCP
  egress:
  - toEntities:
    - world

---

apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: deny-all
  namespace: argocd
spec:
  endpointSelector:
    matchLabels: {}
  ingress: []
  egress: []
```

These policies provide a robust framework for securing the network interactions within the ArgoCD namespace, ensuring that only necessary and authorized traffic is allowed, thereby reducing the attack surface of the Kubernetes cluster.

4.3.2 Protecting Pods Against Privilege Escalation

To enhance the security of Kubernetes pods and prevent privilege escalation attacks, specific security configurations were implemented. These configurations ensure that containers run with the minimum privileges necessary, reducing the risk of exploitation.

The following securityContext was applied:

```
securityContext:
```

```
  runAsNonRoot: true  # Ensure the container runs as a non-root user
  runAsUser: 1000
  runAsGroup: 3000
  fsGroup: 2000
  allowPrivilegeEscalation: false  # Prevent privilege escalation
  readOnlyRootFilesystem: true  # Make the root filesystem read-only
  seccompProfile:
    type: RuntimeDefault  # Use the default seccomp profile for additional security
  capabilities:
    drop:
    - ALL
```

To enhance the overall security of the Kubernetes environment, namespaces were configured to adhere to the highest security standards using Pod Security Admission. This involves enforcing the **restricted** Pod Security Standard, which applies stringent security controls to the pods within these namespaces. The restricted policy is designed to limit the potential attack surface by enforcing security best practices, such as disallowing privilege escalation, running as a non-root user, and restricting access to the host network and filesystem.

```
kubectl label --overwrite ns argocd pod-security.kubernetes.io/enforce=restricted
kubectl label --overwrite ns cert-manager pod-security.kubernetes.io/enforce=restricted
kubectl label --overwrite ns flask-app pod-security.kubernetes.io/enforce=restricted
kubectl label --overwrite ns ingress-nginx pod-security.kubernetes.io/enforce=restricted
kubectl label --overwrite ns default pod-security.kubernetes.io/audit=baseline
```

4.3.3 Securing Container Runtime

In our efforts to enhance container runtime security within the Kubernetes cluster, we have implemented stringent controls to ensure that only images from trusted sources are deployed. This practice is vital in mitigating the risks associated with the use of potentially compromised or malicious container images.

Integration with Private Container Registry

To reinforce image security, we have integrated a private container registry on DigitalOcean with our Kubernetes cluster. This setup involves several key actions and configurations:

1. **Private Container Registry Integration:** By configuring our Kubernetes cluster to use a private container registry, we ensure that all container images are pulled from a controlled and trusted source. This integration limits the risk of inadvertently deploying images that may contain vulnerabilities or malware.

2. **Enforcing Trusted Image Sources:** We have established policies and controls to enforce the use of container images only from our private registry. This involves setting up authentication mechanisms between Kubernetes and the private registry to validate and access images. This process includes:
    - **Registry Credentials:** Storing and managing credentials securely using Kubernetes Secrets to allow the cluster to authenticate and pull images from the private registry.
    - **ImagePullSecrets Configuration:** Specifying imagePullSecrets in our Kubernetes deployment manifests to ensure that images are pulled using the appropriate credentials.

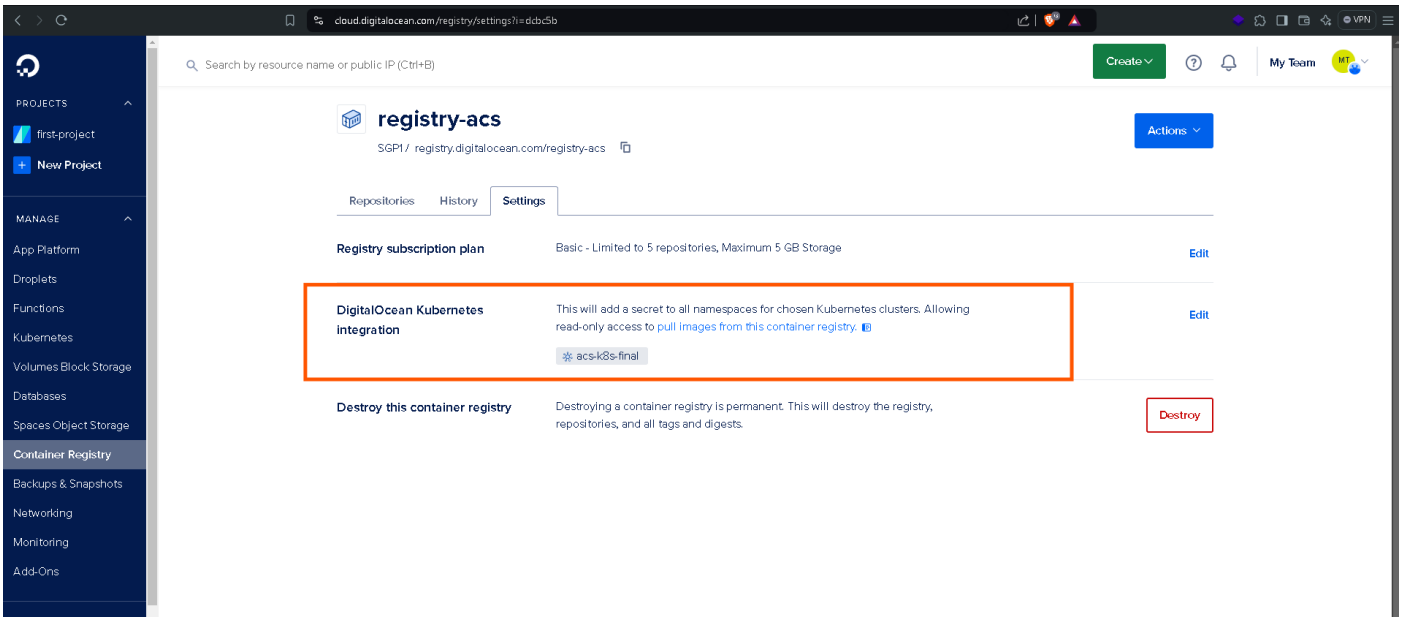

Figure 10: Private Container Registry

4.3.4 CI/CD Security

In our CI/CD pipeline, security is a top priority, especially when handling sensitive information such as credentials and secrets. To mitigate the risks associated with hardcoded credentials, we have implemented several best practices and security measures.

1. Use of Secret Variables in Git Repository

To avoid the inclusion of hardcoded credentials in our source code or configuration files, we use secret variables stored in our Git repository. This approach helps in maintaining the confidentiality and integrity of sensitive information. The process involves:

- **Storing Secrets Securely:** Secret variables, such as API keys, passwords, and other sensitive data, are stored securely within the Git repository's settings. These variables are managed through the repository's secret management features and are not exposed in the codebase.

- **Access Control:** Strict access controls are applied to ensure that only authorized personnel have the ability to view or manage these secrets. This helps in minimizing the risk of unauthorized access or leakage of sensitive information.

2. Deploying Applications via ArgoCD

To further enhance the security of our CI/CD pipeline, we deploy applications using ArgoCD. This approach provides several benefits in terms of security and operational efficiency:

- **Declarative Configuration:** ArgoCD allows us to manage our application deployments using declarative configuration files. This ensures that all deployment configurations are version-controlled and auditable.
- **Secure Deployment:** ArgoCD integrates with the Kubernetes cluster to securely deploy applications. It uses the defined configurations and secrets to ensure that sensitive information is managed securely during the deployment process.
- **Automated Deployments:** Automated deployment processes reduce the risk of human error and ensure that changes are consistently applied across different environments. ArgoCD's continuous deployment capabilities allow for secure and reliable application updates.

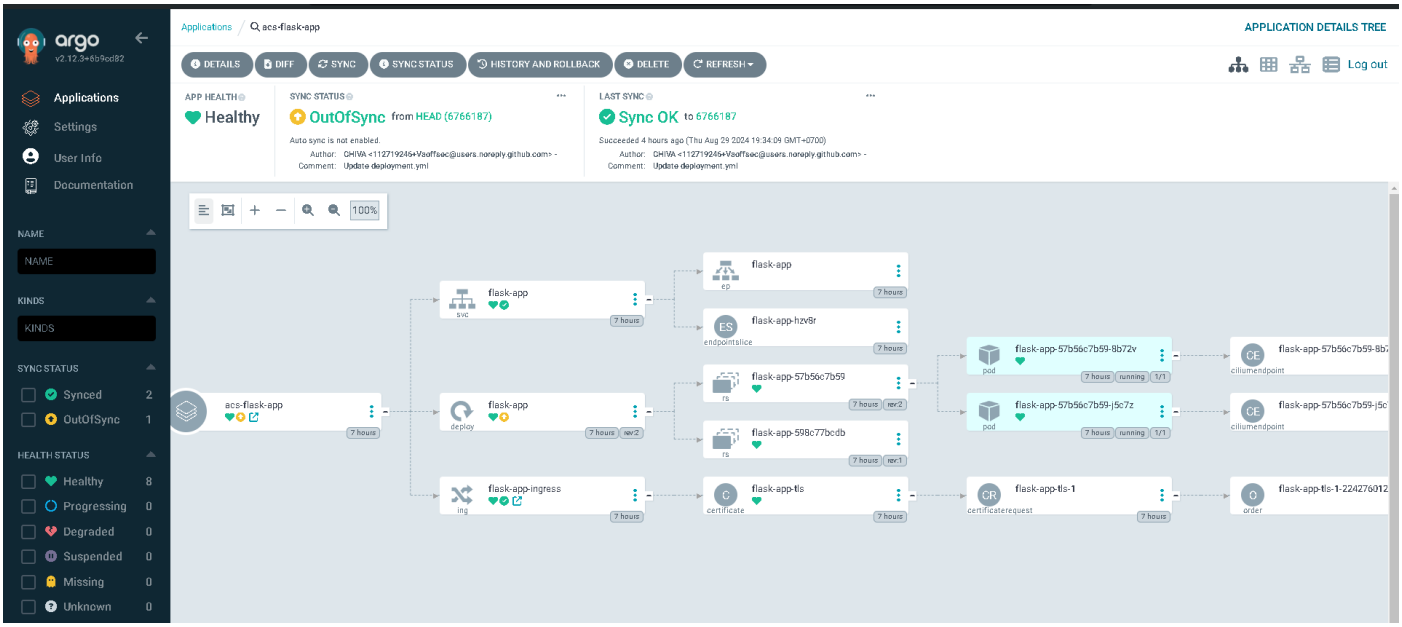

Figure 11: GitOps with argocd

4.3.5 Role-Based Access Control (RBAC)

In our Kubernetes infrastructure, we implemented Role-Based Access Control (RBAC) to enforce strict access policies, ensuring that each component and service only has the permissions necessary to perform its functions.

RBAC allows us to define roles and assign them to specific service accounts, which helps in managing the security of our cluster by minimizing the risk of privilege escalation and unauthorized access.

Service Accounts

We created dedicated service accounts for different namespaces and components within our Kubernetes cluster. Each service account is associated with specific roles and permissions, tailored to the needs of the component it serves. Below are the service accounts created for various namespaces:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: argocd-sa
  namespace: argocd

apiVersion: v1
kind: ServiceAccount
metadata:
  name: cert-manager-sa
  namespace: cert-manager

apiVersion: v1
kind: ServiceAccount
metadata:
  name: default-sa
  namespace: default

apiVersion: v1
kind: ServiceAccount
metadata:
  name: ingress-nginx-sa
  namespace: ingress-nginx

apiVersion: v1
kind: ServiceAccount
metadata:
  name: kube-node-lease-sa
  namespace: kube-node-lease
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kube-public-sa
  namespace: kube-public

apiVersion: v1
kind: ServiceAccount
metadata:
  name: kube-system-sa
  namespace: kube-system

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

Roles

We defined roles that specify the permissions associated with each service account. These roles define what resources the service accounts can access and what actions they are allowed to perform within their respective namespaces. Below are some examples of roles we created:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: argocd-role
  namespace: argocd
rules:
- apiGroups: [""]
  resources: ["pods", "services", "configmaps", "secrets"]
  verbs: ["get", "list", "watch", "create", "update", "delete"]

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: cert-manager-role
```

```
  namespace: cert-manager
rules:
- apiGroups: ["cert-manager.io"]
  resources: ["certificates", "issuers", "orders", "challenges"]
  verbs: ["get", "list", "watch", "create", "update", "delete"]

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: ingress-nginx-role
  namespace: ingress-nginx
rules:
- apiGroups: ["extensions"]
  resources: ["ingresses"]
  verbs: ["get", "list", "watch", "create", "update", "delete"]
```

RoleBindings

To apply the roles to the service accounts, we used RoleBindings, which associate a role with a specific service account within a namespace. Below are examples of the RoleBindings:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: argocd-binding
  namespace: argocd
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: argocd-role
subjects:
- kind: ServiceAccount
  name: argocd-sa
  namespace: argocd

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: cert-manager-binding
```

```
  namespace: cert-manager
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: cert-manager-role
subjects:
- kind: ServiceAccount
  name: cert-manager-sa
  namespace: cert-manager
```

ClusterRoles and ClusterRoleBindings

For components that require cluster-wide access, we defined ClusterRoles and ClusterRoleBindings. ClusterRoles are similar to Roles but are not limited to a specific namespace. They can be used to grant permissions across all namespaces. Below are examples of ClusterRoles and their bindings:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: argocd-clusterrole
rules:
  - apiGroups: ["", "apps", "batch"]
    resources: ["pods", "services", "deployments", "jobs", "cronjobs"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: argocd-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: argocd-clusterrole
subjects:
- kind: ServiceAccount
  name: argocd-sa
  namespace: argocd
```

4.3.6 Securing communication

a. Encryption in Transit with Cert-Manager

To secure communication between clients and services in our cluster, we implemented **TLS (Transport Layer Security)** encryption using Cert-Manager. Cert-Manager automates the management and issuance of TLS certificates, which are used to encrypt data in transit.

- **Certificate Issuance:** We used Cert-Manager to automatically request and manage TLS certificates from Let's Encrypt. These certificates are used to secure communication channels between clients and the services running on our Kubernetes cluster.
- **Auto-Renewal:** Cert-Manager ensures that certificates are automatically renewed before expiration, reducing the risk of service disruption due to expired certificates.
- **Application of Certificates:** The certificates issued by Cert-Manager are applied to Kubernetes Ingress resources, ensuring that all incoming traffic to the services is encrypted.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: flask-app-ingress
  namespace: flask-app
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
    kubernetes.io/tls-acme: "true"
    nginx.ingress.kubernetes.io/ssl-passthrough: "true"
    nginx.ingress.kubernetes.io/backend-protocol: "HTTP"
spec:
  ingressClassName: nginx
  rules:
    - host: flask-app.chivangoy.tech
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: flask-app
```

```
                port:
                  number: 80
  tls:
    - hosts:
        - flask-app.chivangoy.tech
      secretName: flask-app-tls
```

b. Ingress NGINX Load Balancer

The NGINX Ingress Controller plays a crucial role in managing and securing incoming HTTP and HTTPS traffic to the Kubernetes cluster.

- **Ingress Controller:** The NGINX Ingress Controller is used to route external traffic to the appropriate services within the cluster. It provides load balancing, SSL termination, and name-based virtual hosting.
- **SSL/TLS Termination:** The NGINX Ingress Controller terminates SSL/TLS at the ingress point, ensuring that traffic between clients and the Kubernetes services is encrypted.
- **Configuration with Cert-Manager:** The Ingress resources are configured to use the TLS certificates managed by Cert-Manager. This ensures that all traffic handled by the NGINX Ingress Controller is encrypted.
- **Secure Load Balancing:** The Ingress NGINX Controller also provides load balancing, ensuring that incoming traffic is distributed evenly across the backend services, enhancing both performance and reliability.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: flask-app-ingress
  namespace: flask-app
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
    kubernetes.io/tls-acme: "true"
    nginx.ingress.kubernetes.io/ssl-passthrough: "true"
    nginx.ingress.kubernetes.io/backend-protocol: "HTTP"
spec:
```

```
ingressClassName: nginx
rules:
  - host: flask-app.chivangoy.tech
    http:
      paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: flask-app
              port:
                number: 80
tls:
  - hosts:
      - flask-app.chivangoy.tech
    secretName: flask-app-tls
```

4.3.7 Security Testing with Kloudle

To ensure our Kubernetes infrastructure on DigitalOcean meets the highest security standards, we integrated **Kloudle** as part of our security testing process. Kloudle is a cloud-native security tool that helps in identifying and mitigating security risks across cloud environments, including Kubernetes.

Objectives of Security Testing:

- **Identify Misconfigurations:** Detect potential misconfigurations in our Kubernetes setup that could lead to security vulnerabilities.
- **Assess Security Posture:** Evaluate the overall security posture of our Kubernetes infrastructure and ensure it aligns with best practices.
- **Remediate Security Issues:** Provide actionable insights and recommendations for fixing any detected issues to maintain a secure environment.

Testing Process:

- **Integration with Kloudle:** Kloudle was integrated with our Kubernetes environment on DigitalOcean, allowing for continuous monitoring and scanning of assets.
- **Automated Security Scans:** Kloudle performed automated security scans across all assets within the Kubernetes cluster. These scans were scheduled to run at regular intervals to ensure ongoing compliance.
- **Analysis of Scan Results:** The results of the scans were analyzed to identify any misconfigurations or vulnerabilities. Special attention was given to high-severity issues that could pose significant risks.

5. Results

Complete Cluster Setup: We fully set up the Kubernetes cluster environment, which included the installation and configuration of essential components such as ArgoCD for continuous deployment and Cert-Manager for automated certificate management.

Security Implementations:

- **Pod Security Admission (PSA):** We enforced Pod Security Admission policies to protect against privilege escalation within the cluster. This ensures that pods operate with the least privileges necessary, reducing the attack surface.
- **Network Security and Observability:** We implemented a Virtual Private Cloud (VPC) on DigitalOcean for network isolation and security. Additionally, Cilium and Hubble were deployed for enhanced network security and deep observability within the cluster, allowing us to monitor and secure network traffic effectively.
- **Securing Communications:** Encryption in transit was established using Cert-Manager, ensuring that all communication between services and external clients is encrypted. The NGINX Ingress Controller was configured to handle SSL/TLS termination, providing secure load balancing for incoming traffic.
- **Role-Based Access Control (RBAC):** We implemented a comprehensive RBAC strategy, ensuring that each service account and role had appropriate permissions based on the principle of least privilege. This helped in managing access to cluster resources securely.
- **Container Runtime Security:** We enforced security measures at the container runtime level, including the use of images from trusted registries and implementing security contexts to prevent privilege escalation within containers.

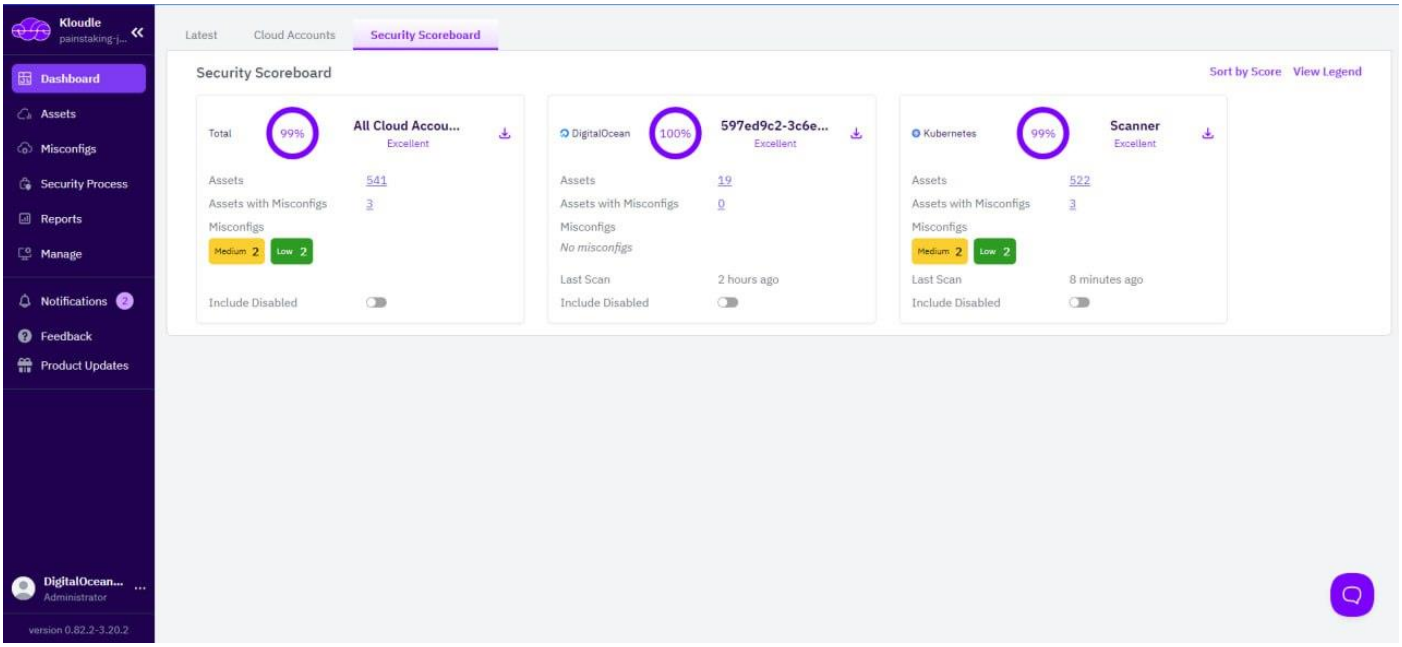

Figure 12 : Result from Kloudle

As part of our security testing, we utilized Kloudle to scan our Kubernetes infrastructure hosted on DigitalOcean. The results from the scan are as follows:

- **Overall Security Score:** Our DigitalOcean Kubernetes infrastructure achieved a **100%** security score, indicating excellent security posture with no detected misconfigurations.
- **Assets Scanned:** A total of **19 assets** were scanned within the Kubernetes infrastructure. The scan confirmed that **none of these assets had any misconfigurations**.
- **Misconfigurations:** The scan results reported **no misconfigurations** across all assets, which underscores the effectiveness of our security implementation.
- **Security Overview for Kubernetes:**
    - **Score:** The Kubernetes cluster received a security score of **99%**.
    - **Misconfigurations:** The scan detected **3 misconfigurations**:
        - **Medium Severity:** 2 misconfigurations.
        - **Low Severity:** 2 misconfigurations

6. Summary and Recommendations

6.1 Summary

The project focused on enhancing the security of a Kubernetes cluster by implementing several key security measures. The main components secured include the control plane, network policies, service accounts, and pod-level security.

- Role-Based Access Control (RBAC): RBAC was implemented to restrict access to Kubernetes resources based on roles, ensuring that users and services have only the permissions necessary to perform their tasks.
- Network Policies: Network policies were used to control the flow of traffic between different pods and services within the cluster. This ensured that only authorized communication is allowed, reducing the risk of unauthorized access.
- TLS Encryption: TLS encryption was enforced for all communication between Kubernetes components to protect data in transit from eavesdropping and tampering.
- Service Account Management: Service accounts were carefully managed and assigned minimal permissions required for their tasks, following the principle of least privilege.
- Pod Security: Pod security was enhanced through the use of Pod Security Policies (PSP) or the PodSecurity admission controller, which enforced security standards at the pod level, such as restricting root access and preventing privilege escalation.
- Compliance Checks: The security configuration was regularly audited using tools like kube-bench to ensure compliance with Kubernetes security best practices.

6.2 Recommendations

With this project, we have some recommendations for the implementation of Kubernetes security infrastructure:

- **Network Policy Implementation**:
    - **Refine Network Policies**: While the diagram shows that traffic is being successfully forwarded, it is crucial to ensure that network policies are fine-tuned to allow only necessary traffic between pods. Any unnecessary or overly permissive rules should be tightened to reduce the potential attack surface.
    - **Segmentation**: Implement strict network segmentation between different namespaces and components to isolate workloads and minimize the blast radius in case of a security incident.
- **Security Logging and Monitoring**:
    - **Enhanced Monitoring**: Integrate with a centralized logging system and security information and event management (SIEM) solution to enhance the monitoring capabilities. This would allow for real-time alerts on any suspicious traffic patterns or anomalies in the network flows.

    - **Audit Logs**: Regularly review the audit logs generated by Kubernetes and the associated network flow data to detect any unauthorized access or configuration changes that could compromise the cluster's security.
- **Service Account Management**:
    - **Least Privilege Principle**: Ensure that service accounts used within the kube-system namespace adhere to the principle of least privilege. Assign only the necessary permissions required for each service account to perform its tasks, and review these permissions regularly.
- **TLS and Encryption**:
    - **Enforce TLS**: Verify that all communication between Kubernetes components and within the network is encrypted using Transport Layer Security (TLS). This is especially important for sensitive components like kube-apiserver and etcd.
    - **Cert Management**: Ensure that TLS certificates are managed properly, with regular renewals and secure storage. Tools like cert-manager can help automate this process within Kubernetes.
- **Pod Security Policies and Admission Controllers:**
    - **Pod Security Enhancements**: Apply Pod Security Policies (PSP) or use the PodSecurity admission controller to enforce security best practices at the pod level. This includes restricting root access, controlling privilege escalation, and limiting container capabilities.
    - **Admission Controllers**: Use admission controllers to validate and enforce security policies before resources are created or updated in the cluster. This adds another layer of security by preventing misconfigurations and enforcing compliance.
- **Cluster Audits and Compliance**:
    - **Regular Audits**: Conduct regular security audits of the cluster using tools like kube-bench to ensure compliance with Kubernetes security benchmarks. Any deviations from the security standards should be addressed promptly.
    - **Compliance Monitoring**: Implement continuous compliance monitoring to ensure that the cluster remains secure over time, especially as new services and applications are deployed.

7. Reference

- https://kubernetes.io/
- https://github.com/aquasecurity/kube-bench
- https://cloud.google.com/kubernetes-engine?hl=en#the-most-scalable-and-fully-automated-kubernetes-service
- https://devopscube.com/kube-bench-guide/

- https://snyk.io/blog/kubernetes-network-policy-best-practices/
- https://medium.com/@saurabh_adhau/mastering-kubernetes-role-based-access-control-rbac-a-comprehensive-guide-to-access-control-d2bc27ad2bed
- https://medium.com/@tamerbenhassan/mastering-kubernetes-secret-management-practical-use-cases-bba1d3ecdde1
- https://www.youtube.com/watch?v=d9xfB5qaOfg
- https://github.com/maxim04/do-k8s-cluster
- https://www.youtube.com/watch?v=wqsUfvRyYpw
- https://www.simform.com/blog/kubernetes-architecture/