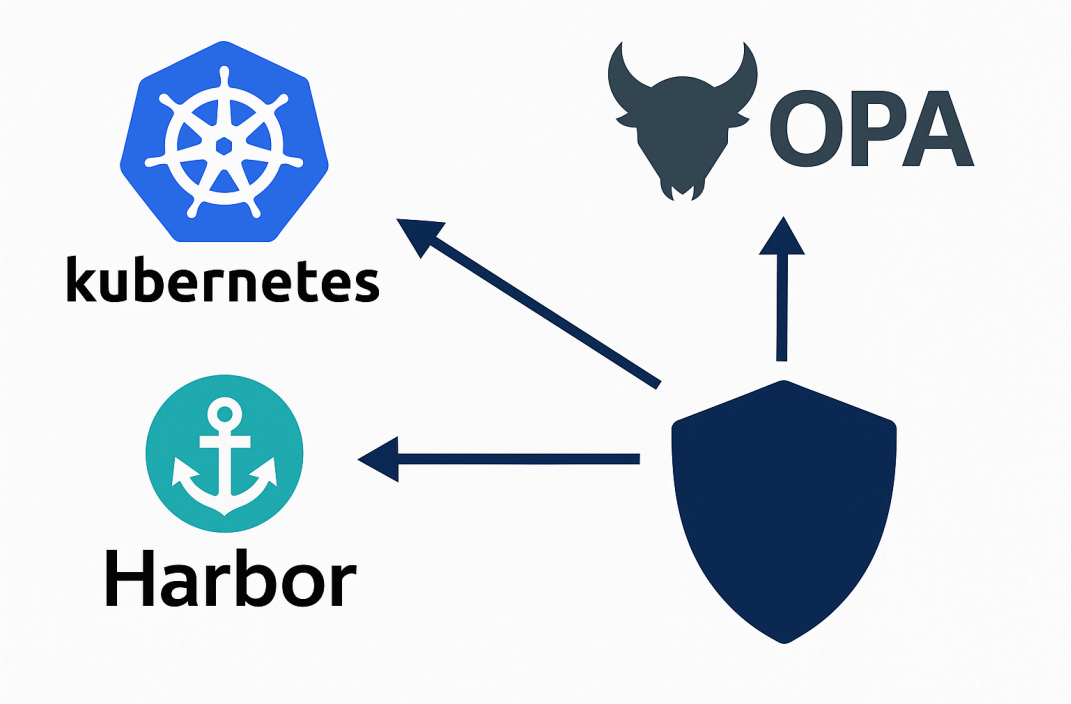
Gobernanza y Seguridad de Contenedores con Harbor, Kubernetes y OPA



Alejandro Liáñez Frutos

2º Administración de Sistemas Informáticos en Red

IES Gonzalo Nazareno

ÍNDICE

1. Introducción

El uso de contenedores para el desarrollo, despliegue y gestión de aplicaciones se ha vuelto una práctica muy utilizada. Herramientas como Docker y plataformas como Kubernetes han cambiado la forma de gestionar las aplicaciones y su correspondiente infraestructura. Siguiendo este contexto, podemos decir, que es necesario contar con un registro de contenedores seguro, privado y controlado, que nos permita almacenar y gestionar las imágenes de forma eficiente y adecuada a las necesidades de seguridad de cada organización.

Este proyecto tiene como objetivo desplegar un sistema de registro privado de contenedores utilizando la herramienta Harbor, una solución de código abierto que no solo nos permitirá almacenar y distribuir las imágenes de los contenedores, sino que también incluye características avanzadas, como un control de acceso y el escaneo de posibles vulnerabilidades de las imágenes. Además, incorporaremos OPA Gatekeeper para aplicar políticas de seguridad dentro de nuestro clúster de Kubernetes, garantizando que solo se desplieguen aplicaciones que cumplan con los criterios de seguridad que hayamos predefinido con esta herramienta.

El proyecto se llevará a cabo en un entorno local, utilizando Minikube sobre KVM como tecnología de virtualización, lo que nos permitirá demostrar la posibilidad de crear un entorno seguro y funcional sin necesidad de depender de servicios en la nube. Llevar a cabo todo esto nos permitirá obtener un conocimiento práctico de la gestión de sistemas complejos, aplicable a entornos reales de trabajo.

Este trabajo no solo tiene como objetivo crear un sistema funcional, sino también profundizar en la seguridad y la gobernanza dentro de los entornos de contenedores.

2. Objetivos

2.1. Objetivo general

El objetivo principal del proyecto es diseñar y desplegar una solución completa de registro de contenedores privada, segura y gobernada mediante tecnologías de código abierto, todo esto llevado a cabo en un entorno local basado en Kubernetes, que nos permita gestionar imágenes Docker con control de acceso, escaneo de vulnerabilidades y políticas de seguridad aplicadas de forma automatizada.

2.2. Objetivos específicos

Tras conocer el objetivo principal del proyecto, podemos considerar que, los objetivos finales a conseguir tras la completa realización del mismo, son los siguientes:

- ➢ Configurar un clúster de **Kubernetes local con Minikube**, ejecutado sobre máquinas virtuales creadas con **KVM**.

- ➢ Instalar y desplegar **Harbor** como registro de contenedores privado accesible mediante interfaz web y con soporte para autenticación y control de acceso basado en roles (RBAC).

- ➢ Integrar el escáner **Trivy** en Harbor para realizar análisis automáticos de vulnerabilidades en las imágenes almacenadas.

- ➢ Implementar **cert-manager** con **Let's Encrypt** para la generación automática de certificados TLS que aseguren las comunicaciones con Harbor a través de HTTPS.

- Configurar **Ingress NGINX** para exponer el registro de forma segura dentro del clúster.

- Incorporar **OPA Gatekeeper** para aplicar políticas de seguridad que limiten el despliegue de imágenes que no cumplan criterios predefinidos.

- Demostrar el funcionamiento de la plataforma mediante el despliegue de una aplicación de ejemplo que deberá cumplir todas las políticas definidas para poder ejecutarse en el clúster.

3. Escenario necesario para la realización del proyecto

Para la implementación de este proyecto, se utilizará un entorno local, aprovechando la virtualización con QEMU/KVM y Minikube para desplegar el clúster de Kubernetes. Este enfoque nos permitirá replicar un entorno de producción a pequeña escala sin necesidad de utilizar recursos externos como la nube, asegurando así que el proyecto sea viable en entornos con recursos limitados.

3.1. Requisitos de hardware

- **8 GB de RAM** (preferiblemente 12-16 GB para un rendimiento óptimo).

- **CPU de al menos 2 núcleos** para manejar las cargas de trabajo de Kubernetes y los servicios desplegados.

- **20 GB de almacenamiento** como mínimo para alojar el sistema operativo, las imágenes de los contenedores y los servicios necesarios.

3.2. Requisitos de software

- **Sistema operativo base:** Debian 12 Bookworm o Ubuntu 22.04 LTS. Ambos son sistemas operativos estables y adecuados para entornos de servidor, con amplia documentación y soporte en la comunidad de contenedores y Kubernetes.

- **Minikube** para desplegar Kubernetes localmente. Minikube proporcionará un entorno de Kubernetes adecuado para llevar a cabo las pertinentes pruebas y demostraciones.

- **QEMU/KVM** para crear las máquinas virtuales. Este software de virtualización se usará para ejecutar Minikube y Kubernetes en un entorno controlado y aislado, facilitando la administración de los recursos virtualizados.

- **Docker** para la creación de imágenes personalizadas que se cargarán en el registro de contenedores Harbor.

- **Helm** como gestor de paquetes de Kubernetes, para simplificar la instalación y configuración de los servicios, como Harbor y cert-manager.

- **Harbor** como registro privado de contenedores. Se configurará en el clúster de Kubernetes, proporcionando una interfaz web y capacidades avanzadas como el escaneo de vulnerabilidades y control de acceso.

- **OPA Gatekeeper** para aplicar políticas de seguridad en Kubernetes, asegurando que las imágenes desplegadas cumplan con las normativas de seguridad de la organización.

- **Trivy** como herramienta de escaneo de vulnerabilidades en imágenes Docker. Trivy se integrará con Harbor para realizar un escaneo automático de todas las imágenes subidas al registro, detectando vulnerabilidades de seguridad antes de que las imágenes sean desplegadas en el clúster.

3.3. Entorno de red y servicios

- **Exposición segura de servicios mediante Ingress NGINX**: Para exponer Harbor de manera segura a través de HTTPS dentro del clúster de Kubernetes. Ingress NGINX también será utilizado para gestionar el enrutamiento y el acceso a los servicios desplegados.

- **Automatización de certificados TLS con cert-manager y Let's Encrypt**: Cert-manager se encargará de obtener y renovar los certificados TLS de manera automática, integrándose con Let's Encrypt para proporcionar certificados gratuitos.

4. Fundamentos y conceptos teóricos

4.1. Contenedores y Docker

Un contenedor es una unidad de software que empaqueta una aplicación y todas sus dependencias, como bibliotecas, configuraciones y herramientas necesarias para ejecutarse de forma aislada, independiente del entorno.

Docker es una plataforma de contenedores que permite crear, desplegar y ejecutar aplicaciones dentro de estos contenedores. La ventaja principal de los contenedores es que aseguran que una aplicación funcione en cualquier entorno, desde desarrollo hasta producción, sin necesidad de configuraciones adicionales.

4.2. Kubernetes

Kubernetes es un sistema de orquestación de contenedores de código abierto que automatiza la implementación, el escalado y la gestión de aplicaciones contenerizadas. Kubernetes permite gestionar clústeres de contenedores, asegurando que las aplicaciones estén siempre disponibles y que los recursos del sistema se utilicen de manera eficiente.

4.3. Harbor

Harbor es un registro de contenedores empresarial y de código abierto, que proporciona un almacenamiento centralizado para las imágenes Docker. A diferencia de Docker Hub, Harbor está diseñado para organizaciones que necesitan un control más exhaustivo sobre las imágenes que suben y descargan.

Además de las funciones básicas de un registro, Harbor incluye características avanzadas como:

- **Autenticación**: Soporta diferentes métodos de autenticación, incluidos control de acceso basado en roles (RBAC).

- **Escaneo de vulnerabilidades**: Integrado con herramientas como Trivy, permite escanear las imágenes almacenadas en busca de vulnerabilidades de seguridad.

- **Replicación**: Permite replicar las imágenes entre diferentes instancias de Harbor, facilitando la gestión de imágenes en entornos distribuidos.

4.4. OPA Gatekeeper

OPA Gatekeeper es una implementación de Open Policy Agent (OPA) que se integra con Kubernetes para aplicar políticas de gobernanza. Con OPA Gatekeeper, es posible definir reglas que controlen qué imágenes de contenedor pueden ser desplegadas, cómo deben configurarse los recursos de Kubernetes, y demás parámetros importantes. Esto se logra utilizando políticas Rego, un lenguaje específico de OPA para definir las reglas.

Algunas de las políticas que pueden implementarse incluyen:

- **Control de acceso**: Asegura que solo usuarios autorizados puedan realizar ciertos despliegues.

- **Validación de imágenes**: Como ejemplo, podemos exigir que las imágenes provengan de registros internos autorizados o que sean escaneadas en busca de vulnerabilidades antes de su despliegue.

La integración de OPA Gatekeeper proporciona una capa adicional de seguridad y conformidad, ayudando a las organizaciones a garantizar que sus entornos de contenedores sean seguros y cumplan con las políticas definidas.

4.5. Trivy

Trivy es una herramienta de escaneo de vulnerabilidades para imágenes Docker. Está diseñada para detectar vulnerabilidades conocidas en las dependencias de una aplicación contenida, incluidos los sistemas operativos base y las bibliotecas que utiliza la aplicación. Es capaz de escanear tanto las imágenes locales como las almacenadas en un registro, como Harbor, y reporta cualquier vulnerabilidad que pueda comprometer la seguridad de las aplicaciones.

Su integración con Harbor permite un análisis automático de las imágenes al ser subidas, ayudando a prevenir el despliegue de imágenes inseguras.

4.6. Cert-Manager y Let's Encrypt

Cert-manager es una herramienta que facilita la gestión y renovación automática de certificados TLS en Kubernetes. Se integra con Let's Encrypt para emitir certificados gratuitos y seguros para los servicios desplegados dentro del clúster. Esto permite que Harbor y otros servicios que se exponen al exterior utilicen HTTPS para asegurar las comunicaciones y proteger los datos en tránsito.

4.7. Ingress NGINX

Ingress NGINX es un controlador de Ingress para Kubernetes que gestiona el enrutamiento de las solicitudes HTTP(S) a los servicios dentro del clúster. Este controlador facilita la exposición de los servicios de Kubernetes al exterior de manera segura mediante el uso de HTTPS, utilizando certificados gestionados por cert-manager.

5. Preparación del escenario

El entorno para el desarrollo de este proyecto se ha creado completamente de forma local, sobre una máquina virtual basada en Debian 12 con conexión a internet y, al menos 12 GB de RAM y 2 CPUs.

A continuación, prepararemos el escenario previo, indicando las instalaciones del sistema de virtualización usado, docker como herramienta para crear las imágenes que cargaremos posteriormente en Harbor y Minikube, para utilizar Kubernetes localmente; además, instalaremos la herramienta kubectl, con la que gestionaremos el clúster y helm, que facilitará el despliegue de aplicaciones como Harbor.

5.1. Virtualización con QEMU/KVM

Para trabajar con el sistema de virtualización QEMU/KVM + libvirt en nuestra distribución Linux Debian/Ubuntu, vamos a instalar los siguientes paquetes:

```
apt install qemu-system libvirt-clients libvirt-daemon-system
```

Podemos obtener las versiones de las aplicaciones que hemos instalado con el comando *virsh version*:

```
[~] (master)
alejandro$ virsh version
Compilada con biblioteca: libvirt 9.0.0
Uso de biblioteca: libvirt  9.0.0
Utilizando API: QEMU 9.0.0
Ejecutando hypervisor: QEMU 7.2.15
```

Para que nuestro usuario sin privilegios pueda hacer conexiones privilegiadas, el usuario debe pertenecer el grupo *libvirt*, para ello, ejecutaremos el siguiente comando con el usuario *root*:

```
adduser usuario libvirt
```

5.2. Instalación de Docker

Hay varios métodos de instalación disponibles, en nuestro caso, usaremos los repositorios apt de Docker y seguiremos los siguientes pasos:

Actualizamos los repositorios e instalamos los paquetes necesarios;

```
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
```

Añadimos la clave GPG oficial de Docker:

```
  sudo install -m 0755 -d /etc/apt/keyrings
  curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg
--dearmor -o /etc/apt/keyrings/docker.gpg
  sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Añadimos el repositorio oficial de Docker:

```
  echo \
  "deb [arch=$(dpkg --print-architecture)
signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/debian
\
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Instalamos Docker Engine:

```
  sudo apt-get update
  sudo apt-get install docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin
```

Añadimos nuestro usuario sin privilegios al grupo *docker*:

```
sudo usermod -aG docker $USER
```

Volvemos a iniciar una terminal con nuestro usuario para recargar la configuración:

```
su - $USER
```

Utilizamos el siguiente comando para comprobar que Docker se ha instalado con éxito, comprobando así la versión instalada:

```
docker --version
```

```
[~] (master)
alejandro$ docker --version
Docker version 20.10.24+dfsg1, build 297e128
```

5.3. Instalación de Minikube

Instalaremos Minikube directamente desde su binario; para ello, seguiremos los siguientes pasos:

Descargamos como usuario sin privilegios y con ayuda del comando *curl*, la última versión del binario de minikube:

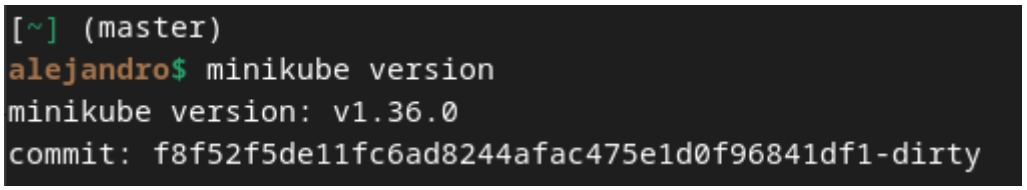
```
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
```

Movemos el binario descargado a un directorio del PATH, estableciendo permisos de ejecución:

```
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Comprobamos que se ha instalado correctamente con el siguiente comando:

```
minikube version
```



5.4. Creación del clúster de k8s

Tras la instalación de Minikube, crearemos el clúster de Kubernetes de un solo nodo; Minikube puede hacer esto en diversos sistemas de virtualización, en nuestro caso, utilizaremos QEMU/KVM por lo que, debemos especificar el driver que queremos utilizar:

```
minikube start --driver=kvm2
```

```
[~] (master)
alejandro$ minikube start --driver=kvm2
😄  minikube v1.36.0 en Debian 12.10
    ▪ KUBECONFIG=/home/alejandro/.kube/config
✨  Using the kvm2 driver based on user configuration
👍  Starting "minikube" primary control-plane node in "minikube" cluster
🔥  Creando kvm2 VM (CPUs=2, Memory=3900MB, Disk=20000MB) ...
❗  Image was not built for the current minikube version. To resolve this you can delete and recreate yo
ur minikube cluster using the latest images. Expected minikube version: v1.35.0 -> Actual minikube versi
on: v1.36.0
🐳  Preparando Kubernetes v1.33.1 en Docker 28.0.4...
    ▪ Generando certificados y llaves
    ▪ Iniciando plano de control
    ▪ Configurando reglas RBAC...
🔗  Configurando CNI bridge CNI ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Complementos habilitados: storage-provisioner, default-storageclass

❗  /usr/local/bin/kubectl is version 1.31.0, which may have incompatibilities with Kubernetes 1.33.1.
    ▪ Want kubectl v1.33.1? Try 'minikube kubectl -- get pods -A'
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

El comando anterior crea una máquina virtual o un contenedor en el sistema escogido e instala kubernetes en ella; además, se configura kubectl (si está instalado) para que utilice el clúster recién creado.

Podemos comprobar el estado de minikube con el siguiente comando:

```
minikube status
```

```
[~] (master)
alejandro$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Para reiniciar minikube debemos utilizar los siguientes comandos:

```
minikube stop
```

```
minikube start
```

5.5. Instalación de kubectl

Kubectl es la herramienta de línea de comandos utilizada para interactuar con la API de Kubernetes. Es por tanto la herramienta fundamental que vamos a utilizar para gestionar nuestros objetos en el clúster recién creado con minikube.

Instalaremos esta herramienta descargando su binario e instalándolo en el directorio */usr/local/bin*:

```
curl -LO
"https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bi
n/linux/amd64/kubectl"
sudo install kubectl /usr/local/bin/kubectl
```

Tras esto, debemos reiniciar minikube para que se configure kubectl de forma automática.

Comprobamos su correcta instalación con el siguiente comando:

```
kubectl version
```

```
[~] (master)
alejandro$ kubectl version
Client Version: v1.33.1
Kustomize Version: v5.6.0
Server Version: v1.33.1
```

Comprobamos que, ya aparece la versión del servidor, por lo tanto, la conexión con el clúster que gestiona minikube ha resultado exitosa. Además, podemos ejecutar nuestro primer comando propiamente de kubectl:

```
kubectl get nodes
```

```
[~] (master)
alejandro$ kubectl get nodes
NAME       STATUS   ROLES           AGE   VERSION
minikube   Ready    control-plane   10m   v1.33.1
```

5.6. Instalación de Helm

Utilizaremos el siguiente comando para instalar esta herramienta:

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

Verificamos su correcta instalación con el siguiente comando:

```
helm version
```

```
[~] (master)
alejandro$ helm version
version.BuildInfo{Version:"v3.18.2", GitCommit:"04cad4610054e5d546aa5c5d9c1b1d5cf68ec1f8", GitTreeState:
"clean", GoVersion:"go1.24.3"}
```

6. Descripción detallada de lo que se ha realizado

Una vez que tengamos instaladas todas las herramientas previas que necesitaremos; procederemos a llevar a cabo el despliegue de cada una de las nuevas tecnologías que vamos a utilizar; además, configuraremos las herramientas principales, como son Harbor y OPA, con las que realizaremos diferentes pruebas para comprender mejor su uso y, posteriormente, procederemos a realizar las demostraciones prácticas del proyecto.

6.1. Instalación y despliegue de herramientas

Todas estas instalaciones y despliegues las realizaremos en Minikube gracias a la herramienta Helm, siguiendo prácticamente siempre los mismos pasos.

6.1.1 Despliegue de Ingress NGINX

Añadiremos el repositorio de Ingress NGINX a nuestro cliente Helm; tras esto, actualizaremos los índices locales de charts para asegurarnos de tener las última versión disponible y, por último, instalaremos la herramienta, creando un *namespace* con el mismo nombre:

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

helm install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx --create-namespace
```

```
alejandro$ helm install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx --create-namespace
NAME: ingress-nginx
LAST DEPLOYED: Mon Jun  9 21:22:28 2025
NAMESPACE: ingress-nginx
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The ingress-nginx controller has been installed.
It may take a few minutes for the load balancer IP to be available.
You can watch the status by running 'kubectl get service --namespace ingress-nginx ingress-nginx-controller --output wide --watch'
```

Podemos comprobar que el despliegue se ha realizado correctamente utilizando el siguiente comando:

```
helm list -A
```

```
[~] (master)
alejandro$ helm list -A
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS  CHART                   APP VERSION
ingress-nginx   ingress-nginx   1               2025-06-09 21:22:28.701181869 +0200 CEST        deployedingress-nginx-4.12.3    1.12.3
```

6.1.2. Instalación de cert-manager

Para instalar y desplegar cert-manager, debemos seguir los mismos pasos exactos que en el punto anterior; la única diferencia es, obviamente, el repositorio correspondiente:

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

helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
  --set installCRDs=true
```

```
[~] (master)
alejandro$ helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
  --set installCRDs=true
NAME: cert-manager
LAST DEPLOYED: Mon Jun  9 21:24:14 2025
NAMESPACE: cert-manager
STATUS: deployed
REVISION: 1
```

Comprobamos el correcto despliegue con el mismo comando que en el punto anterior:

```
helm list -A
```

```
[~] (master)
alejandro$ helm list -A
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS          CHART                   APP VERSION
cert-manager    cert-manager    1               2025-06-09 21:24:14.850609929 +0200 CEST        deployed        cert-manager-v1.17.2    v1.17.2
ingress-nginx   ingress-nginx   1               2025-06-09 21:22:28.701181869 +0200 CEST        deployed        ingress-nginx-4.12.3    1.12.3
```

6.1.3. Despliegue de Harbor

El despliegue de Harbor es parecido a los dos anteriores, pero no idéntico; comenzamos añadiendo el repositorio a nuestro cliente Helm y actualizando los índices locales de charts:

```
helm repo add harbor https://helm.goharbor.io
helm repo update
```

Tras esto, crearemos un fichero que tendrá la configuración personalizada de Harbor, incluyendo lo siguiente:

- ➢ Exposición a la red mediante *ingress*.
- ➢ Dominio personalizado.
- ➢ Contraseña del usuario admin.
- ➢ Certificados autofirmados.

➢ Persistencia de los volúmenes de todos los servicios que se crearán.

En el enlace siguiente, encontraremos el contenido completo del fichero que hemos usado para desplegar Harbor: harbor-values.yaml

El comando siguiente desplegará Harbor, aplicando la configuración definida en *harbor-values.yaml*:

```
helm install harbor harbor/harbor \
  --namespace harbor \
  --create-namespace \
  -f harbor-values.yaml
```

```
[~/proyecto_integrado] (main)
alejandro$ helm install harbor harbor/harbor \
  --namespace harbor \
  --create-namespace \
  -f harbor-values.yaml
I0609 22:05:27.811979   17000 warnings.go:110] "Warning: annotation \"kubernetes.io/ingress.class\" is d
eprecated, please use 'spec.ingressClassName' instead"
NAME: harbor
LAST DEPLOYED: Mon Jun  9 22:05:23 2025
NAMESPACE: harbor
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Comprobamos que el despliegue se ha realizado correctamente:

```
helm list -A
```

```
[~/proyecto_integrado] (main)
alejandro$ helm list -A

NAME            NAMESPACE       REVISION        UPDATED                                 STATUS  CHART                   APP VERSION
cert-manager    cert-manager    1               2025-06-09 21:24:14.850609929 +0200 CEST        deployedcert-manager-v1.17.2    v1.17.2
harbor          harbor          1               2025-06-09 22:05:23.970838121 +0200 CEST        deployedharbor-1.17.1           2.13.1
ingress-nginx   ingress-nginx   1               2025-06-09 21:22:28.701181869 +0200 CEST        deployedingress-nginx-4.12.3    1.12.3
```

Para asegurarnos que todos los pods se han creado y están funcionando correctamente, podemos usar el siguiente comando:

```
kubectl get pods -n harbor
```

```
[~/proyecto_integrado] (main)
alejandro$ kubectl get pods -n harbor
NAME                                READY   STATUS    RESTARTS      AGE
harbor-core-7d5b69fbd9-hrnlg        1/1     Running   0             2m5s
harbor-database-0                   1/1     Running   0             2m5s
harbor-jobservice-597c744cbf-g77nw  1/1     Running   2 (41s ago)   2m5s
harbor-portal-5b6b5f7494-2xlv7      1/1     Running   0             2m5s
harbor-redis-0                      1/1     Running   0             2m5s
harbor-registry-5d769958cf-k46ww    2/2     Running   0             2m5s
harbor-trivy-0                      1/1     Running   0             2m5s
```

Pasado un tiempo, todos los pods tendrán estado Running, lo que significa que Harbor se ha desplegado correctamente.

6.1.4 Instalación de OPA Gatekeeper

La instalación y despliegue de OPA es igual a las dos primeras herramientas que hemos hecho; añadimos su repositorio y actualizamos para verificar que tenemos la última versión de la herramienta:

```
helm repo add gatekeeper
https://open-policy-agent.github.io/gatekeeper/charts
helm repo update
```

Instalamos y desplegamos OPA con el siguiente comando:

```
helm install gatekeeper gatekeeper/gatekeeper --namespace
gatekeeper-system --create-namespace
```

```
[~/proyecto_integrado] (main)
alejandro$ helm install gatekeeper gatekeeper/gatekeeper --namespace gatekeeper-system --create-namespace
NAME: gatekeeper
LAST DEPLOYED: Mon Jun  9 22:47:41 2025
NAMESPACE: gatekeeper-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Comprobamos que se ha desplegado correctamente con el siguiente comando:

```
helm list -A
```

```
[~/proyecto_integrado] (main)
alejandro$ helm list -A
NAME            NAMESPACE              REVISION        UPDATED                                 STATUS          CHART                   APP VERSION
cert-manager    cert-manager           1               2025-06-09 21:24:14.850609929 +0200 CEST        deployed        cert-manager-v1.17.2    v1.17.2
gatekeeper      gatekeeper-system      1               2025-06-09 22:47:41.284805857 +0200 CEST        deployed        gatekeeper-3.19.1       v3.19.1
harbor          harbor                 1               2025-06-09 22:05:23.970838121 +0200 CEST        deployed        harbor-1.17.1           2.13.1
ingress-nginx   ingress-nginx          1               2025-06-09 21:22:28.701181869 +0200 CEST        deployed        ingress-nginx-4.12.3    1.12.3
```

Podemos comprobar que todos los pods creados están en estado *Running*, lo que significa que la herramienta está lista para usarse:

```
kubectl get pods -n gatekeeper-system
```

```
[~/proyecto_integrado] (main)
alejandro$ kubectl get pods -n gatekeeper-system
NAME                                             READY   STATUS    RESTARTS   AGE
gatekeeper-audit-58fdd95bc6-lp8vt                1/1     Running   0          108s
gatekeeper-controller-manager-6886548cc7-dqc5m   1/1     Running   0          108s
gatekeeper-controller-manager-6886548cc7-hgrbw   1/1     Running   0          108s
gatekeeper-controller-manager-6886548cc7-j54th   1/1     Running   0          108s
```

6.2. Configuración y pruebas con Harbor

Antes de comenzar con Harbor, debemos configurar una serie de cosas para que la aplicación sea accesible desde el navegador:

Añadiremos el dominio de nuestra aplicación Harbor al archivo */etc/hosts* junto a la dirección ip de minikube; para ello, podemos ejecutar el siguiente comando para hacerlo de manera sencilla y rápida:

```
echo "$(minikube ip) harbor.local" | sudo tee -a /etc/hosts
```

```
[~/proyecto_integrado] (main)
alejandro$ echo "$(minikube ip) harbor.local" | sudo tee -a /etc/hosts
192.168.39.176 harbor.local
```

Cómo harbor expone su servicio a través de Ingress, debemos asegurarnos de que está bien configurado; para ello usaremos el siguiente comando:

```
kubectl get ingress -n harbor
```

```
[~/proyecto_integrado] (main)
alejandro$ kubectl get ingress -n harbor
NAME             CLASS    HOSTS          ADDRESS   PORTS     AGE
harbor-ingress   <none>   harbor.local             80, 443   5m41s
```

Comprobamos los servicios expuestos de nuestro controlador Ingress usando el siguiente comando:

```
kubectl get svc -n ingress-nginx
```

```
[~/proyecto_integrado] (main)
alejandro$ kubectl get svc -n ingress-nginx
NAME                                 TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx-controller             LoadBalancer   10.105.51.159   <pending>     80:30732/TCP,443:30523/TCP   49m
ingress-nginx-controller-admission   ClusterIP      10.111.25.154   <none>        443/TCP                      49m
```

Como vemos, no tiene dirección ip externa, por lo que, vamos parchear el servicio para usar la ip de minikube y, poder así, acceder de forma externa a Harbor; para ello, usaremos el siguiente comando:

```
kubectl patch svc ingress-nginx-controller -n ingress-nginx   -p
'{"spec": {"externalIPs": ["192.168.39.176"]}}'
```

```
[~/proyecto_integrado] (main)
alejandro$ kubectl patch svc ingress-nginx-controller -n ingress-nginx   -p '{"spec": {"externalIPs": ["192.168.39.176"]}}'
service/ingress-nginx-controller patched
```

Si volvemos a comprobar los servicios del Ingress vemos que, la dirección ip de minikube se ha establecido como ip externa para que Harbor sea accesible desde el exterior:

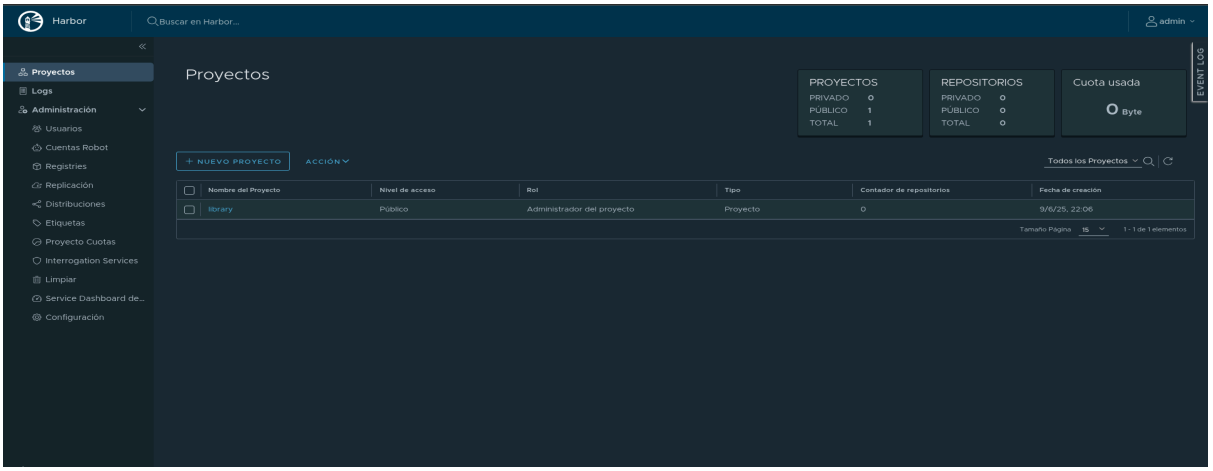
```
kubectl get svc -n ingress-nginx
```

```
[~/proyecto_integrado] (main)
alejandro$ kubectl get svc -n ingress-nginx
NAME                                 TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)                      AGE
ingress-nginx-controller             LoadBalancer   10.105.51.159   192.168.39.176   80:30732/TCP,443:30523/TCP   53m
ingress-nginx-controller-admission   ClusterIP      10.111.25.154   <none>           443/TCP                      53m
```

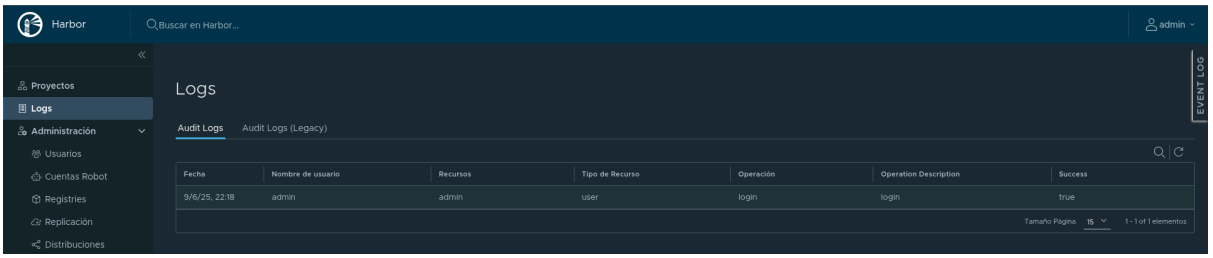
Para acceder a Harbor desde nuestro navegador favorito, debemos indicar el dominio configurado previamente, en este caso, harbor.local:

Como vemos, se nos abrirá la ventana de inicio de sesión de Harbor, nos conectaremos con el usuario admin y la contraseña que definimos previamente, en nuestro caso es admin.

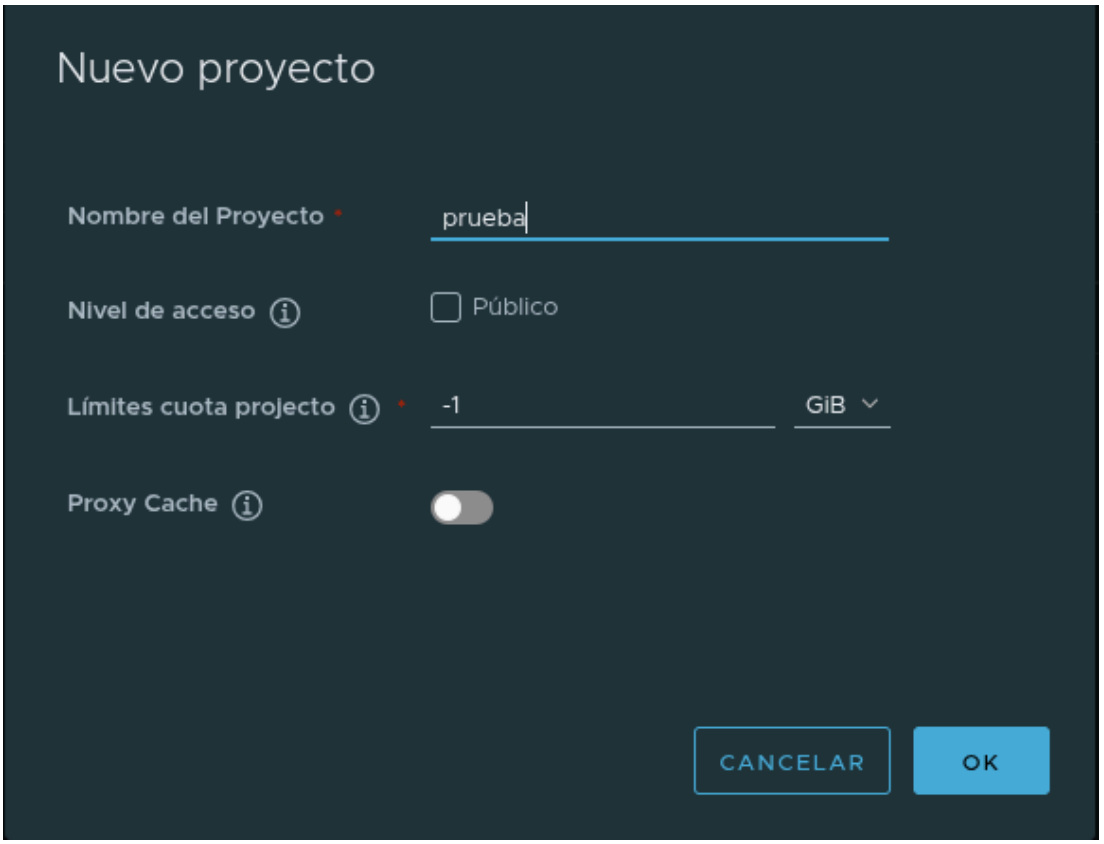
Tras iniciar sesión, se nos abrirá la ventana principal de Harbor, dónde podremos encontrar nuestros proyectos; además, tenemos un menú desplegable con diferentes apartados y configuraciones que podemos ejecutar en nuestra herramienta:



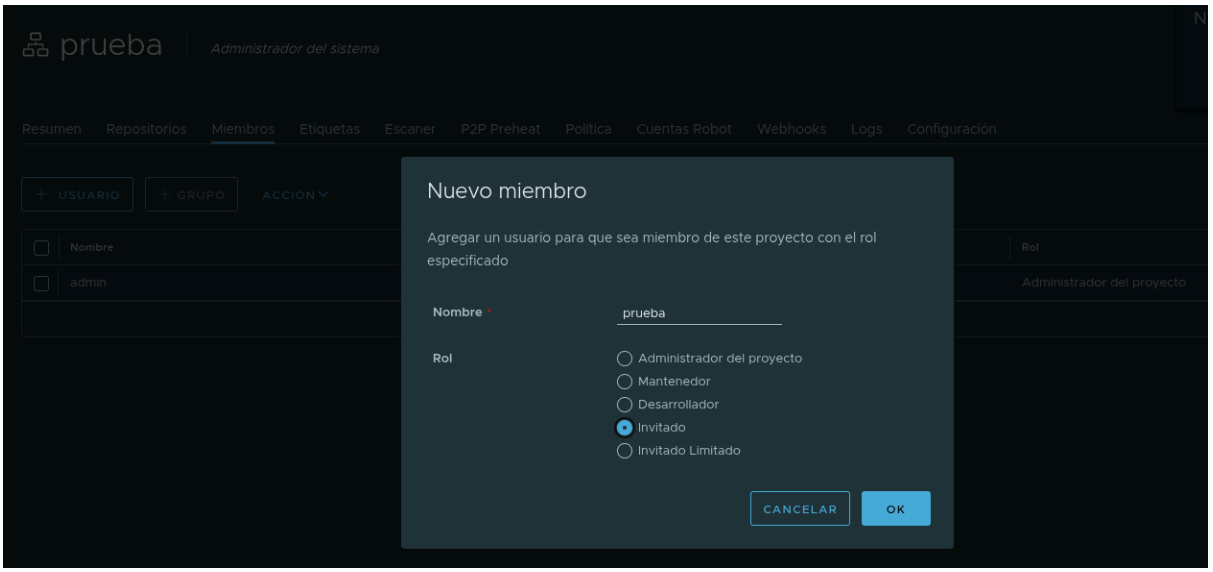
Si nos vamos a la pestaña *Logs*, podremos ver todos los cambios que ocurren en nuestra aplicación; como en este momento, solo nos hemos conectado con el usuario *admin*, solo nos aparece este registro.



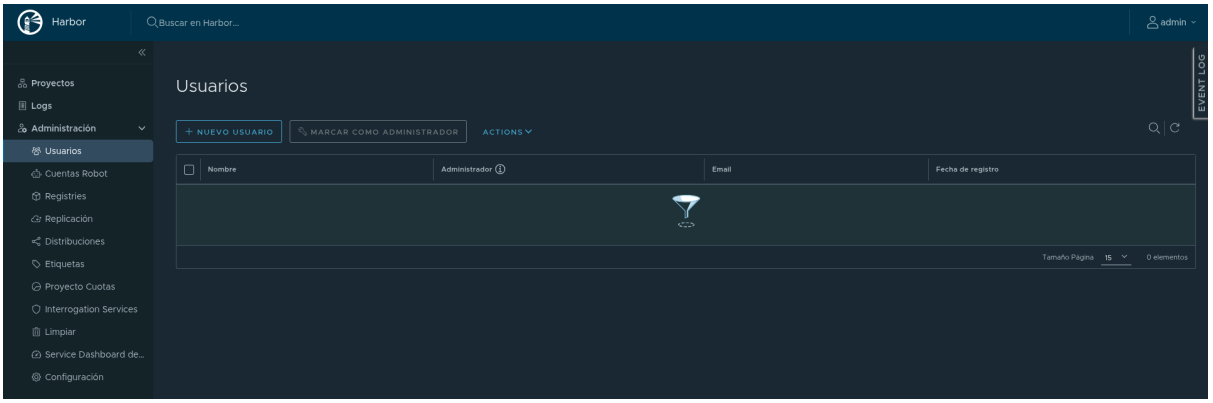
Para crear un nuevo proyecto, pincharemos en la pestaña Nuevo Proyecto, dónde nos aparecerá un pequeño formulario, en él, indicaremos el nombre del mismo, si es privado o público y el límite de cuotas correspondiente.



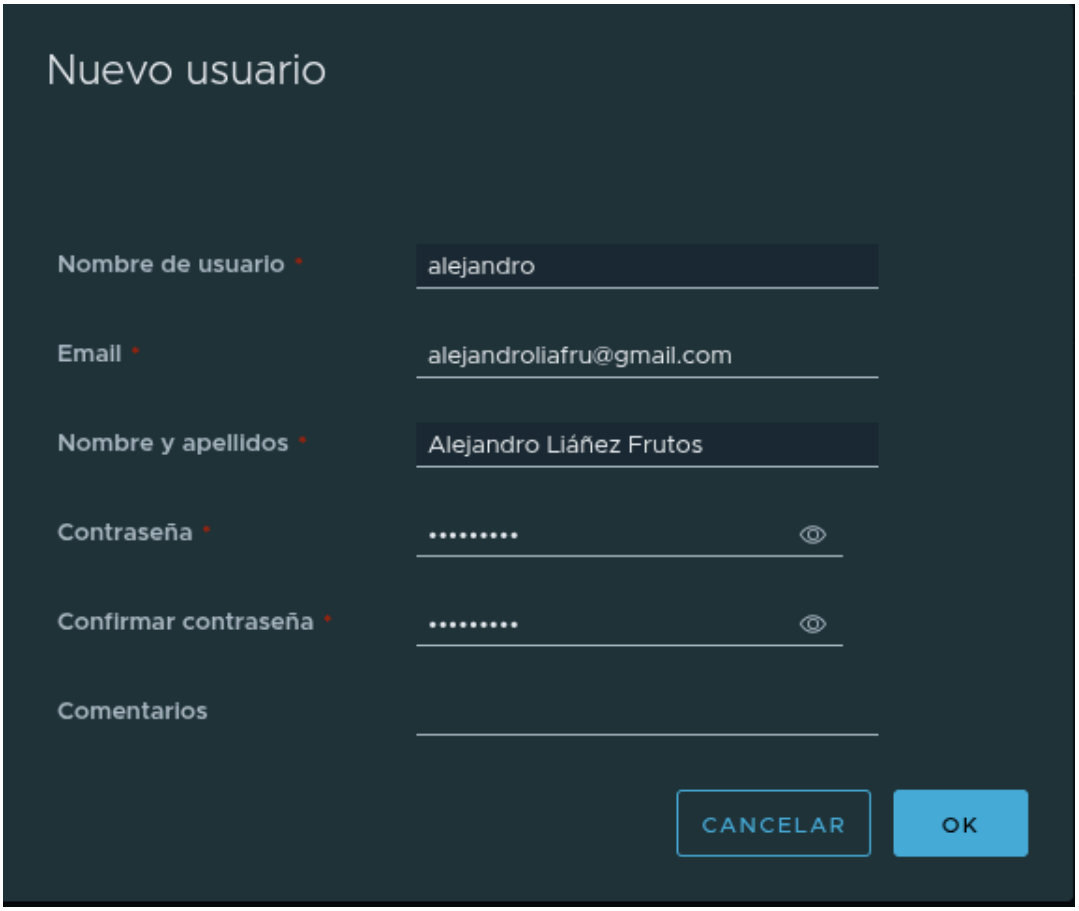
Dentro de nuestro nuevo proyecto llamado *Prueba*, podemos asignar diferentes usuarios con diferentes roles al mismo; dependiendo del rol que le asignemos, tendrá unos permisos u otros sobre el proyecto en cuestión.



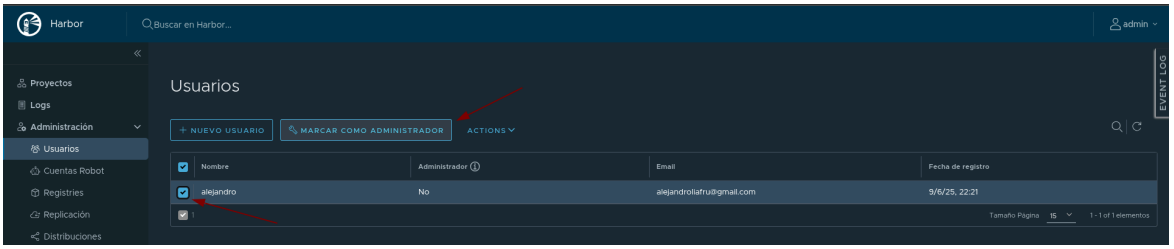
Para crear un nuevo usuario, nos dirigiremos a la pestaña *Usuarios* y pincharemos en *Nuevo Usuario*:



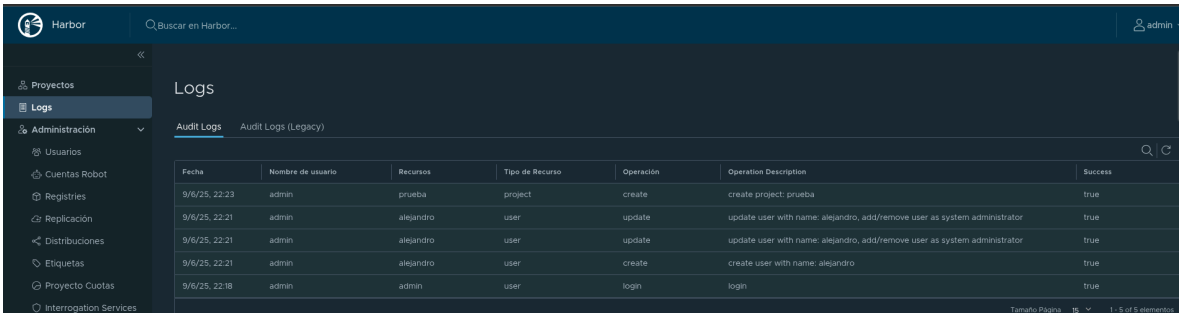
Tras esto, nos aparecerá un nuevo formulario a rellenar con los datos del usuario que queramos crear:



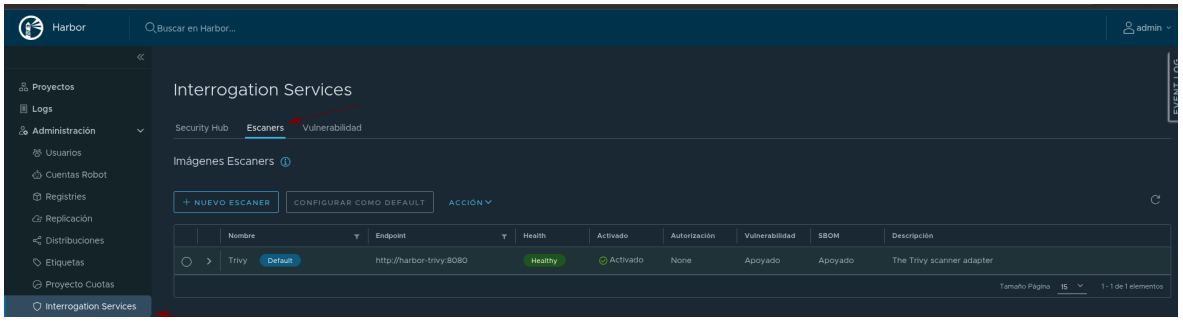
Como vemos, nuestro nuevo usuario ha sido creado exitosamente; si queremos que éste tenga el rol de administrador, debemos añadirlo como se muestra en la imagen siguiente:



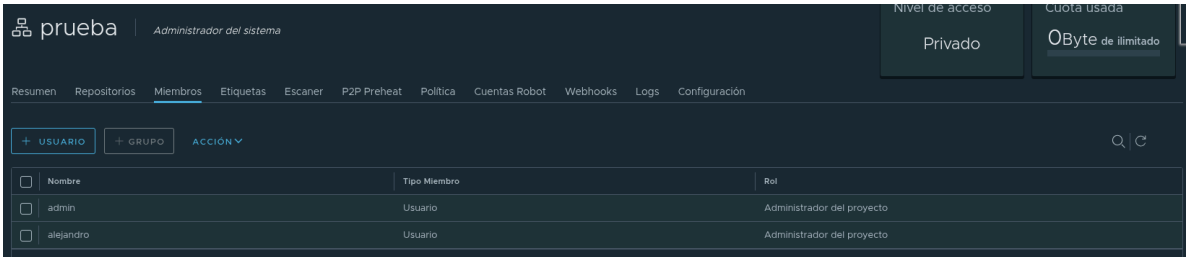
Si volvemos a la pestaña Logs, podemos ver que todos los cambios y configuraciones realizadas se han registrado correctamente.



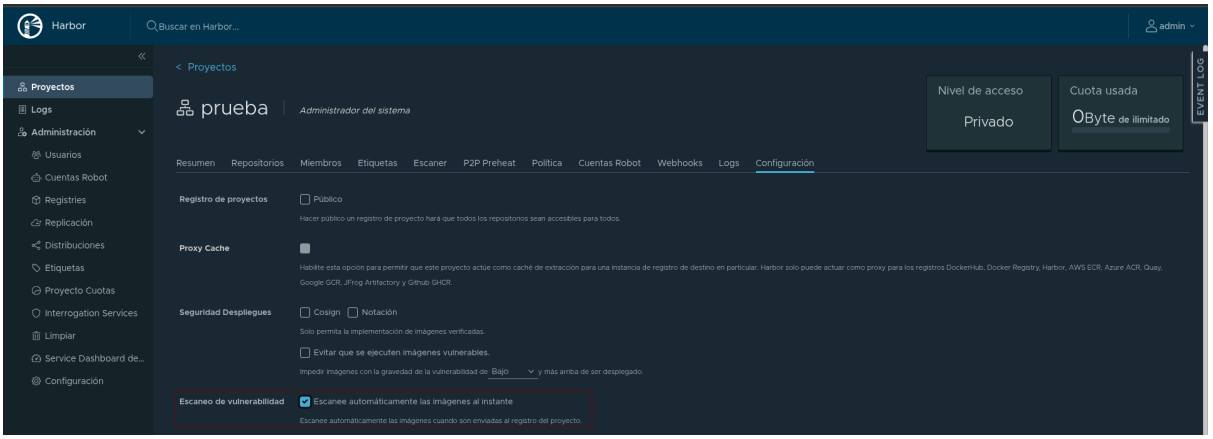
Si nos dirigimos a la pestaña Interrogation Services, podemos ver que, como dijimos anteriormente, el escáner Trivy se instaló junto a Harbor y está operativo y listo para usarse.



Para configurar escaneos automáticos de las imágenes que subiremos a Harbor, nos dirigiremos a nuestro proyecto Prueba, dónde ya tenemos asignado a nuestro usuario alejandro.



Si pinchamos en Configuración podremos marcar la opción de escaneos automáticos de vulnerabilidades con nuestro escáner Trivy:



Para probar el funcionamiento de esto, haremos una prueba sencilla, en la que almacenaremos una imagen Docker en Harbor.

Creamos dos ficheros para construir la imagen Docker, un Dockerfile y un index.html:

```
[~/proyecto_integrado] (main)
alejandro$ cat Dockerfile
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
```

```
[~/proyecto_integrado] (main)
alejandro$ cat index.html
<!DOCTYPE html>
<html>
<head><title>Hola desde Harbor</title></head>
<body><h1>Imagen personalizada de nginx</h1></body>
</html>
```

Construimos nuestra imagen Docker con el siguiente comando:

```
docker build -t harbor.local/prueba/nginx-personalizado:1.0 .
```

```
[~/proyecto_integrado] (main)
alejandro$ docker build -t harbor.local/prueba/nginx-personalizado:1.0 .
Sending build context to Docker daemon  79.36kB
Step 1/2 : FROM nginx:alpine
alpine: Pulling from library/nginx
f18232174bc9: Pull complete
61ca4f733c80: Pull complete
b464cfdf2a63: Pull complete
d7e507024086: Pull complete
81bd8ed7ec67: Pull complete
197eb75867ef: Pull complete
34a64644b756: Pull complete
39c2ddfd6010: Pull complete
Digest: sha256:65645c7bb6a0661892a8b03b89d0743208a18dd2f3f17a54ef4b76fb8e2f2a10
Status: Downloaded newer image for nginx:alpine
 ---> 6769dc3a703c
Step 2/2 : COPY index.html /usr/share/nginx/html/index.html
 ---> 9fc2a2a5be65
Successfully built 9fc2a2a5be65
Successfully tagged harbor.local/prueba/nginx-personalizado:1.0
```

Iniciamos sesión en Harbor desde Docker:

```
docker login harbor.local
```

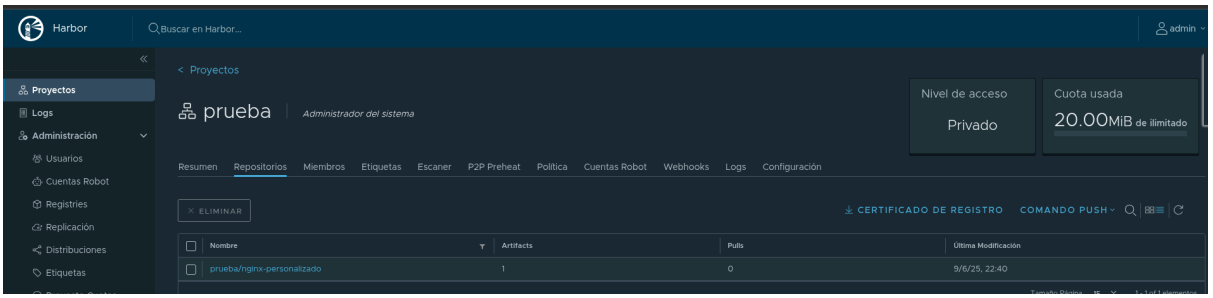
```
[~/proyecto_integrado] (main)
alejandro$ docker login harbor.local
Authenticating with existing credentials...
WARNING! Your password will be stored unencrypted in /home/alejandro/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

Subimos la imagen a Harbor con el siguiente comando:

```
docker login harbor.local
```

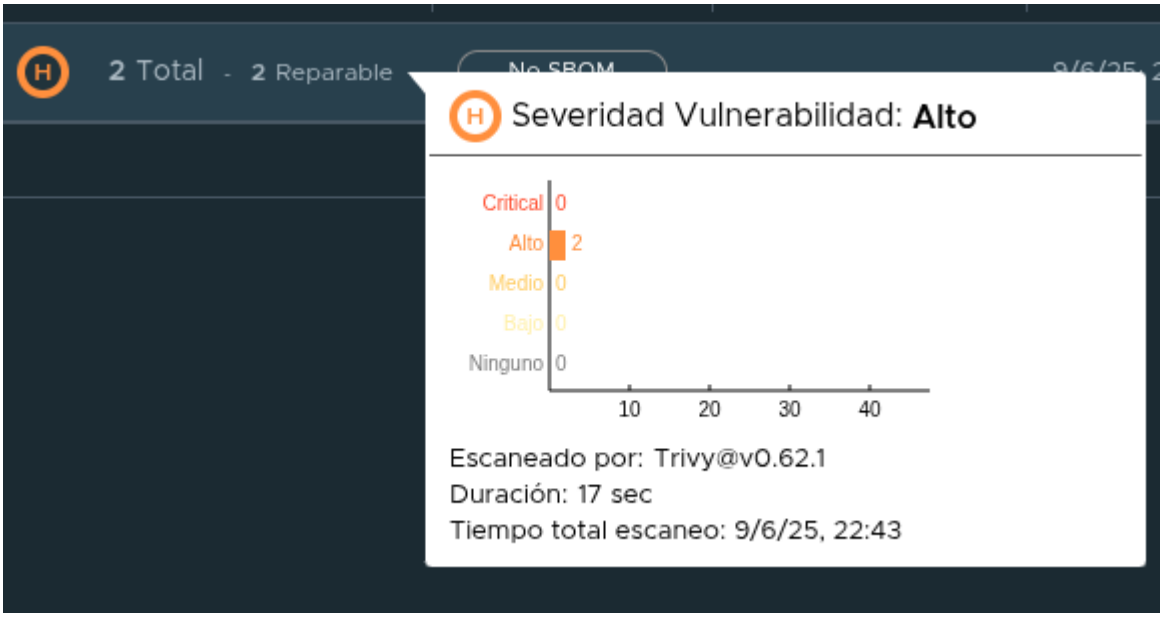
```
[~/proyecto_integrado] (main)
alejandro$ docker push harbor.local/prueba/nginx-personalizado:1.0
The push refers to repository [harbor.local/prueba/nginx-personalizado]
e2e6f6da6805: Pushed
0d853d50b128: Pushed
947e805a4ac7: Pushed
811a4dbbf4a5: Pushed
b8d7d1d22634: Pushed
e244aa659f61: Pushed
c56f134d3805: Pushed
d71eae0084c1: Pushed
08000c18d16d: Pushed
1.0: digest: sha256:4a2aa92192effc1ec3d6f3a991176c9103ccbc5b8c6e4f6099a60824f233f9de size: 2196
```

Volvemos a Harbor y nos dirigimos a nuestro proyecto *Prueba*, para así verificar sus repositorios y comprobar que podemos encontrar la imagen Docker que hemos creado previamente.

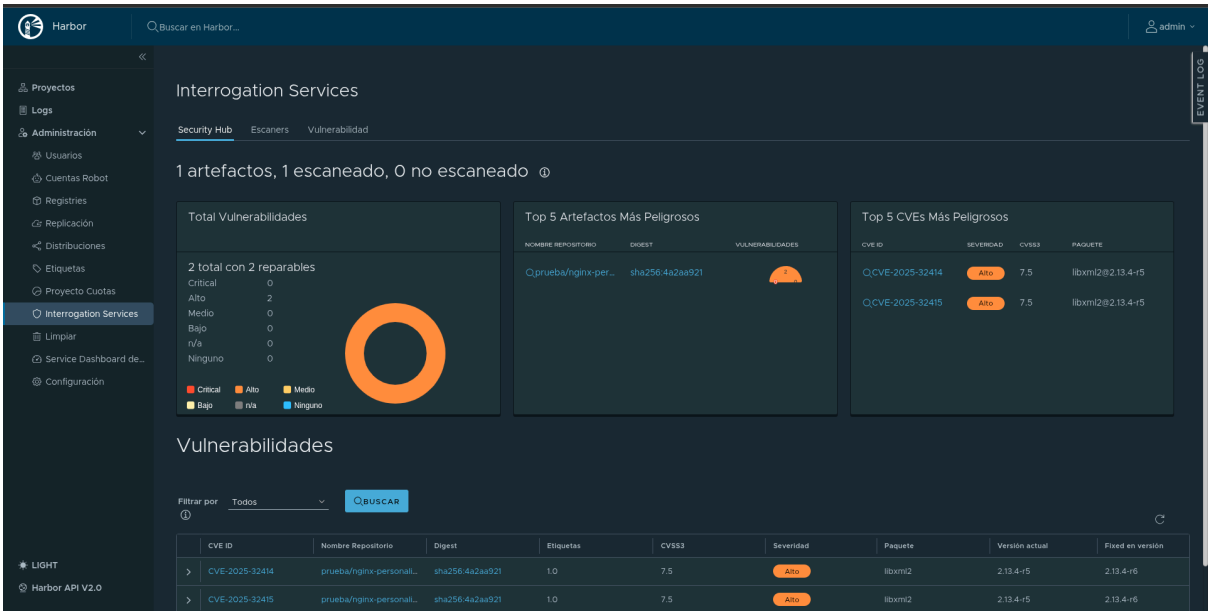


Si accedemos a la imagen subida, podemos ver que se ha escaneado automáticamente.

Si pinchamos sobre las vulnerabilidades encontradas, podemos ver la categoría que Trivy les ha establecido, según su gravedad, la fecha del escaneo y su duración.



Además, si nos dirigimos a *Interrogation Services*, podemos ver un análisis más exhaustivo sobre el escaneo realizado.



6.3. Configuración y pruebas con OPA GateKeeper

Comenzaremos realizando una pequeña prueba con OPA para comprobar su funcionamiento; esta prueba consistirá en crear una política de seguridad que evite el despliegue de imágenes que no sean pertenecientes a Harbor.

Para comenzar, crearemos un archivo que definirá una nueva política de seguridad; el archivo se encuentra en el siguiente enlace:
template-block-nonharbor.yaml

Estas políticas se nombran como Rego de forma oficial; el Rego que acabamos de crear define lo siguiente:

- Revisa todos los contenedores del objeto que se está intentando crear.
- Verifica si la imagen comienza con *harbor.local*.
- Si la imagen no cumple ese requisito, se genera un mensaje de violación de política que dice *"La imagen 'x' no está en Harbor"*.

Tras esto, crearemos un *Constraint*, es decir, una instancia concreta de la plantilla anterior, que definirá sobre qué recursos de Kubernetes se aplicará la lógica; este archivo se encuentra en el siguiente enlace:
constraint-block-nonharbor.yaml

Este archivo aplica la política a los objetos de tipo *Pod* y solo se valida cuando alguien intenta crear o modificar uno.

Aplicamos la política con los siguientes comandos:

```
kubectl apply -f template-block-nonharbor.yaml
kubectl apply -f constraint-block-nonharbor.yaml
```

```
[~/proyecto_integrado] (main)
alejandro$ kubectl apply -f template-block-nonharbor.yaml
constrainttemplate.templates.gatekeeper.sh/k8srequiredharbor created

[~/proyecto_integrado] (main)
alejandro$ kubectl apply -f constraint-block-nonharbor.yaml
k8srequiredharbor.constraints.gatekeeper.sh/solo-imagenes-harbor created
```

Podemos comprobar que la política se ha implementado correctamente con los siguientes comandos:

```
kubectl get constrainttemplates
kubectl get k8srequiredharbor
```

```
[~/proyecto_integrado] (main)
alejandro$ kubectl get constrainttemplates
NAME                AGE
k8srequiredharbor   6m24s

[~/proyecto_integrado] (main)
alejandro$ kubectl get k8srequiredharbor
NAME                    ENFORCEMENT-ACTION   TOTAL-VIOLATIONS
solo-imagenes-harbor    deny                 23
```

Ahora que OPA Gatekeeper tiene creadas:

- ➢ Una plantilla con lógica (*ConstraintTemplate*)
- ➢ Una política que aplica esa lógica sobre Pods (*Constraint*)

Vamos a crear un manifiesto con una imagen no permitida para probar la política de seguridad; esto lo haremos con un fichero que se encuentra en el siguiente enlace:
nginx-test.yaml

Este archivo intentará crear un *Pod* en nuestro clúster con la imagen llamada *nginx:latest*; para ello usaremos el siguiente comando:

```
kubectl apply -f nginx-test.yaml
```

```
[~/proyecto_integrado] (main)
alejandro$ kubectl apply -f nginx-test.yaml
Error from server (Forbidden): error when creating "nginx-test.yaml": admission webhook "validation.gate
keeper.sh" denied the request: [solo-imagenes-harbor] La imagen 'nginx:latest' no está en Harbor.
```

Como vemos, el *Pod* no se crea, mostrándonos el mensaje de error que definimos anteriormente, ya que esta imagen no está en nuestro registro Harbor.

Con esto, hemos conseguido implementar una política de seguridad que:

- ➢ Bloquea el uso de imágenes externas (como Docker Hub).
- ➢ Garantiza que todas las imágenes vengan del registro privado, lo que permite mayor control, trazabilidad y seguridad.

6.4. Demostraciones prácticas

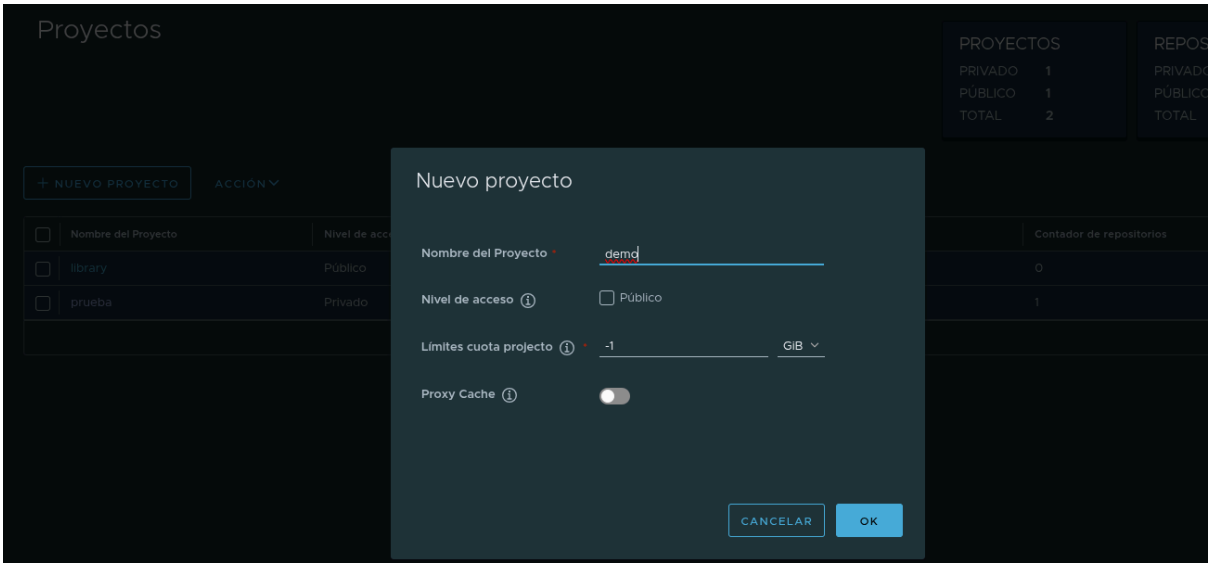
Aparte de las demostraciones que se indicarán en este documento, se realizará un despliegue del clúster en vivo en la presentación del proyecto; este punto lo obviaremos aquí, ya que se ha explicado previamente todo el despliegue.

El resto de demostraciones indicadas serán las que se realicen en la presentación.

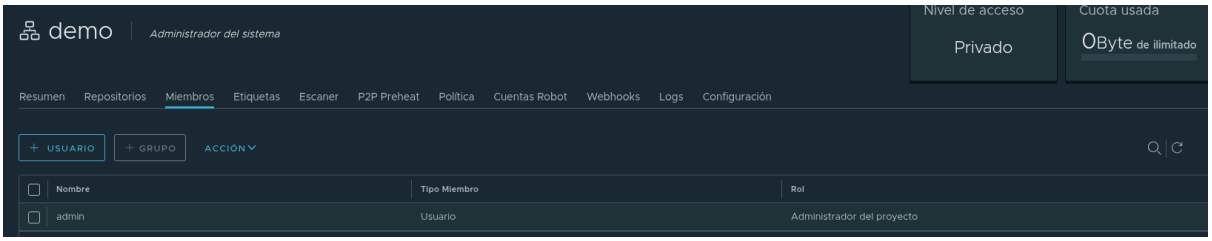
6.4.1. Subida y escaneo automático de una imagen personalizada

Para esta demostración, crearemos una imagen más completa que la prueba que hicimos anteriormente que, además de servir contenido, puede incluir dependencias para que Trivy tenga más material que analizar.

Comenzaremos creando un nuevo proyecto en Harbor llamado *Demo*, para realizar esta demostración.



Este proyecto estará asignado al usuario *admin*.



Creamos un Dockerfile nuevo, en este caso, se utiliza Python y Flask con versiones antiguas para que se muestren posibles vulnerabilidades; el Dockerfile se encuentra en el siguiente enlace:

Dockerfile

Con este fichero, crearemos la nueva imagen usando el siguiente comando:

```
docker build -t harbor.local/demo/flask-personalizado:1.0 .
```

```
[~/proyecto_integrado/demos] (main)
alejandro$ docker build -t harbor.local/demo/flask-personalizado:1.0 .
Sending build context to Docker daemon  3.584kB
Step 1/5 : FROM python:3.9-slim
3.9-slim: Pulling from library/python
61320b01ae5e: Pull complete
b73a592742ab: Pull complete
710d06b8662f: Pull complete
3a36dc9f4d24: Pull complete
Digest: sha256:657a140aae5f8eb61c69c3df950fade52f1a7924f88612071acccb863a9efe0f
Status: Downloaded newer image for python:3.9-slim
 ---> 131bd0653c5c
Step 2/5 : RUN pip install flask==2.0.0 requests==2.19.1
 ---> Running in 55d8760fc698
Collecting flask==2.0.0
  Downloading Flask-2.0.0-py3-none-any.whl (93 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 93.2/93.2 kB 2.7 MB/s eta 0:00:00
```

Nos loguearemos con Docker en Harbor usando el usuario *admin*:

```
docker login harbor.local
```

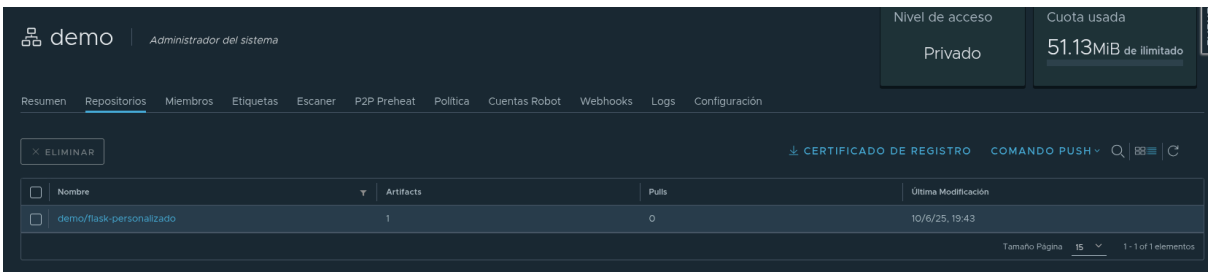
```
[~/proyecto_integrado/demos] (main)
alejandro$ docker login harbor.local
Authenticating with existing credentials...
WARNING! Your password will be stored unencrypted in /home/alejandro/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

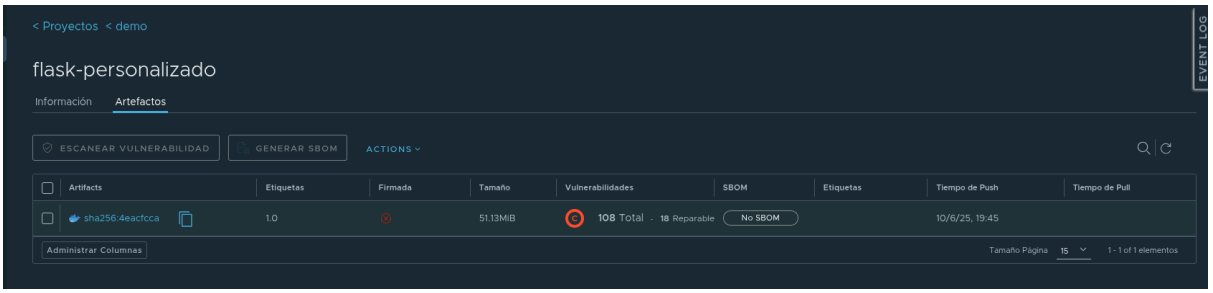
Subimos la imagen a Harbor con el siguiente comando:

```
docker push harbor.local/demo/flask-personalizado:1.0
```

```
[~/proyecto_integrado/demos] (main)
alejandro$ docker push harbor.local/demo/flask-personalizado:1.0
The push refers to repository [harbor.local/demo/flask-personalizado]
47213f55f632: Pushed
a10317d120fd: Pushed
978f260c1369: Pushed
9b5482944372: Pushed
9ad43ba78452: Pushed
ace34d1d784c: Pushed
1.0: digest: sha256:4eacfcca5760054ff7780c8d6a5bd93e9d556275e3ac742bfb83b4f8a6e088f5 size: 1577
```

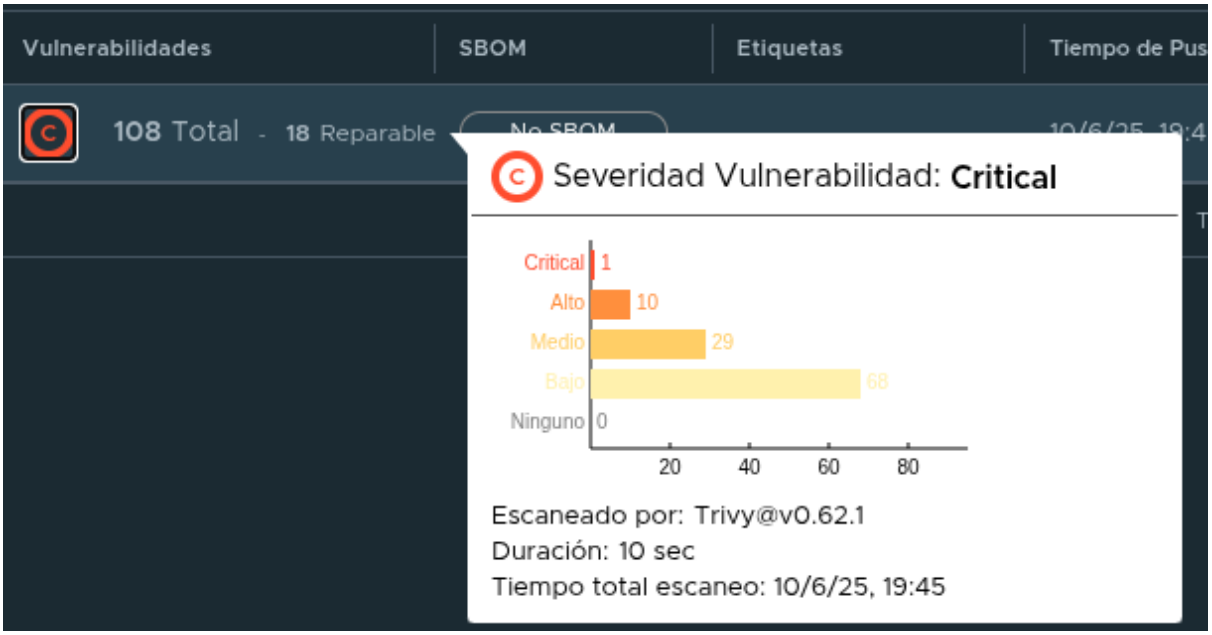
Comprobamos que la imagen se ha subido en Harbor correctamente y que el escaneo ha comenzado de forma automática:



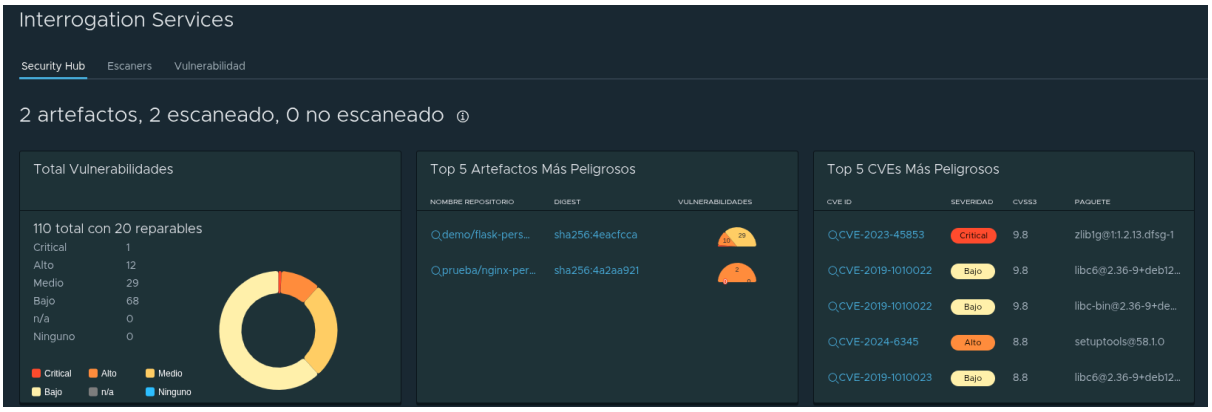


6.4.2. Visualización de vulnerabilidades desde la interfaz de Harbor

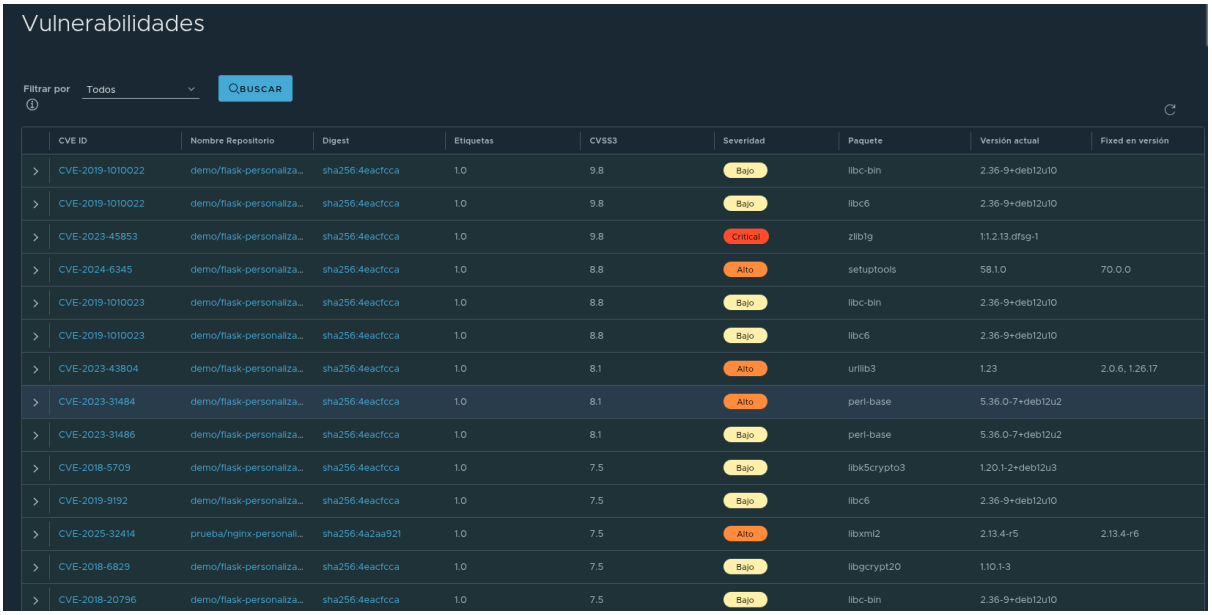
Nos dirigimos a nuestro proyecto Demo para ver, desde la imagen anteriormente subida, el informe que ha generado Trivy.



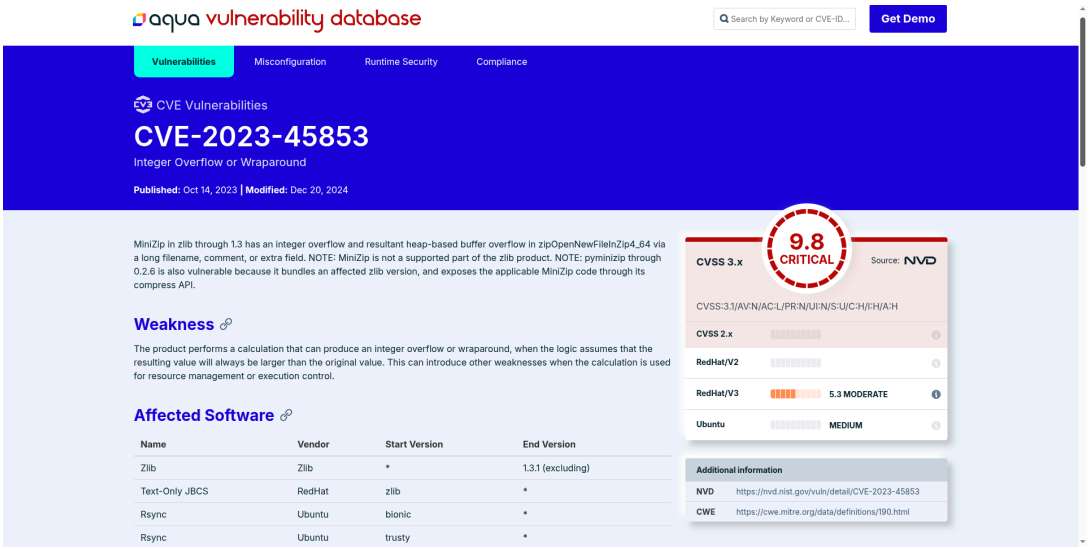
Si nos dirigimos a la pestaña *Interrogation Services* podemos ver todas las vulnerabilidades encontradas y un informe más detallado:



Como vemos en la siguiente imágen, todo está clasificado según su riesgo:



Cada vulnerabilidad presenta un enlace a una base de datos que las clasifica por un código identificativo; en este enlace podemos encontrar más información sobre la misma:

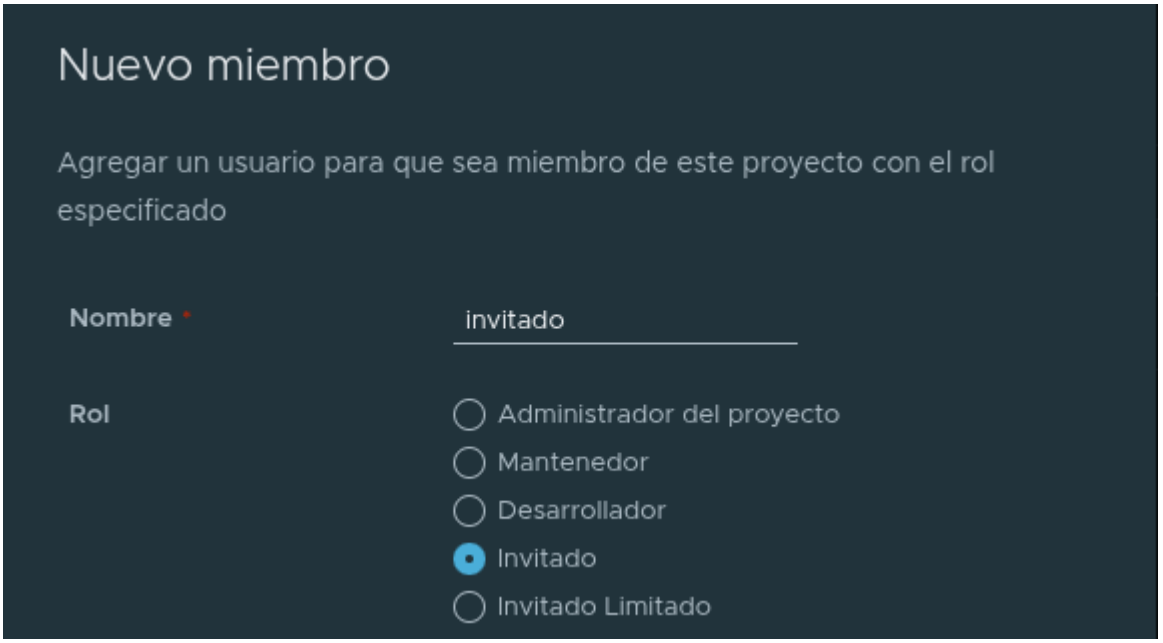


6.4.3. Control de acceso por roles en Harbor

Harbor tiene una funcionalidad diferente para cada usuario, dependiendo del rol que este mismo tenga asignado sobre un proyecto concreto.

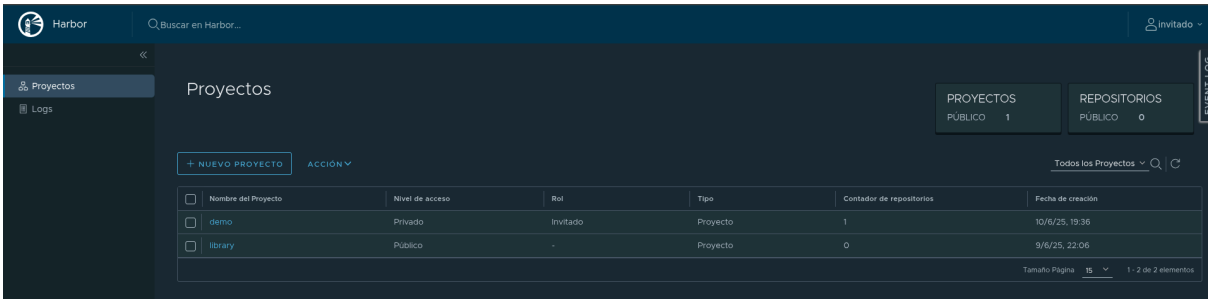
Para comprobar esto, nos loguearemos en Harbor con el usuario *admin* y nos dirigimos a *Usuarios* para crear uno nuevo llamado *invitado*:

Tras esto, nos dirigiremos a nuestro proyecto demo, y añadiremos a este nuevo usuario con el rol de *Invitado*:

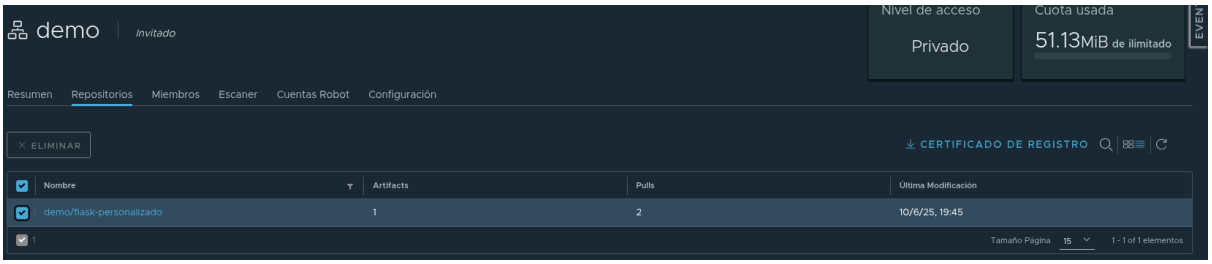


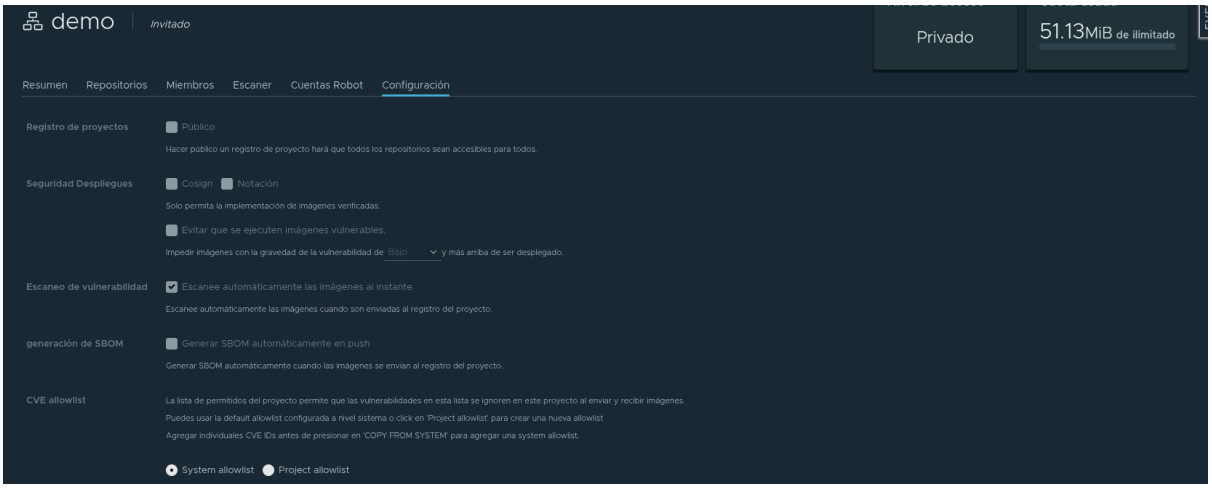
En una nueva pestaña de nuestro navegador, nos logueamos con este nuevo usuario:

Como podemos ver, al acceder con este usuario, solo tenemos acceso al proyecto asignado y al público que viene por defecto:



No podremos cambiar nada en la configuración del proyecto y, mucho menos borrar repositorios:





6.4.4. Aplicación de políticas con OPA

A continuación, se realizarán 3 demostraciones diferentes de aplicaciones de políticas con OPA Gatekeeper.

6.4.4.1. Bloqueo de imágenes externas

Con esta política, sólo se permitirá desplegar imágenes que provengan de *harbor.local*.

Creamos el *ConstraintTemplate* correspondiente, se encuentra en el siguiente enlace:
k8srequiredregistry-template.yaml

```
kubectl apply -f k8srequiredregistry-template.yaml
```

```
[~/proyecto_integrado/demos/demo1] (main)
alejandro$ kubectl apply -f k8srequiredregistry-template.yaml
constrainttemplate.templates.gatekeeper.sh/k8srequiredregistry created
```

Aplicamos la poltica de seguridad:
require-harbor.yaml

```
kubectl apply -f require-harbor.yaml
```

```
[~/proyecto_integrado/demos/demo1] (main)
alejandro$ kubectl apply -f require-harbor.yaml
k8srequiredregistry.constraints.gatekeeper.sh/only-allow-harbor-images created
```

Para probar el correcto funcionamiento de esta política, intentaremos desplegar un *Pod* con una imagen externa:

```
kubectl run flask-hub --image=python:3.9-slim
```

```
[~/proyecto_integrado/demos/demo1] (main)
alejandro$ kubectl run flask-hub --image=python:3.9-slim
Error from server (Forbidden): admission webhook "validation.gatekeeper.sh" denied the request: [only-a
low-harbor-images] La imagen 'python:3.9-slim' no proviene del registro permitido 'harbor.local'
```

Como vemos, la política bloquea el despliegue del *Pod*.

Probaremos ahora con una imagen válida:

```
kubectl run flask-valid
--image=harbor.local/demo/flask-personalizado:1.0
--image-pull-policy=Never
```

```
[~/proyecto_integrado/demos/demo1] (main)
alejandro$ kubectl run flask-valid --image=harbor.local/demo/flask-personalizado:1.0 --image-pull-policy
=Never
pod/flask-valid created
```

En este caso, el *Pod* se despliega correctamente, ya que está imagen se encuentra en nuestro registro de Harbor.

Podemos ver nuestras políticas creadas con el siguiente comando:

```
kubectl get constrainttemplates
```

```
[~/proyecto_integrado/demos/demo1] (main)
alejandro$ kubectl get constrainttemplates
NAME                  AGE
k8srequiredharbor     21h
k8srequiredregistry   82s
```

6.4.4.2. Restringir uso del hostPath

Esta política tiene como objetivo evitar que los *Pods* monten directorios del host, algo que puede ser muy riesgoso.

Creamos el *ConstraintTemplate* correspondiente, se encuentra en el siguiente enlace:
k8sdisallowhostpath-template.yaml

```
kubectl apply -f k8sdisallowhostpath-template.yaml
```

```
[~/proyecto_integrado/demos/demo2] (main)
alejandro$ kubectl apply -f k8sdisallowhostpath-template.yaml
constrainttemplate.templates.gatekeeper.sh/k8sdisallowhostpath created
```

Creamos el *Constraint* para activar la política:
disallow-hostpath.yaml

```
kubectl apply -f disallow-hostpath.yaml
```

```
[~/proyecto_integrado/demos/demo2] (main)
alejandro$ kubectl apply -f disallow-hostpath.yaml
k8sdisallowhostpath.constraints.gatekeeper.sh/restrict-hostpath created
```

Creamos un *Pod* de ejemplo que inclumpla la política recién creada:
pod-con-hostpath.yaml

```
kubectl apply -f pod-con-hostpath.yaml
```

```
[~/proyecto_integrado/demos/demo2] (main)
alejandro$ kubectl apply -f pod-con-hostpath.yaml
Error from server (Forbidden): error when creating "pod-con-hostpath.yaml": admission webhook "validatio
n.gatekeeper.sh" denied the request: [restrict-hostpath] No se permite el uso de hostPath: host-volume
```

Como vemos, el despliegue se bloquea por la política aplicada.

6.4.4.3. Bloquear pods privilegiados

Esta política hará que Gatekeeper evité el despliegue de cualquier *Pod* que tenga *securityContext.privileged: true*.

Creamos el *ConstraintTemplate* correspondiente, se encuentra en el siguiente enlace:
k8sno-privileged-containers-template.yaml

```
kubectl apply -f k8sno-privileged-containers-template.yaml
```

```
[~/proyecto_integrado/demos/demo3] (main)
alejandro$ kubectl apply -f k8sno-privileged-containers-template.yaml
constrainttemplate.templates.gatekeeper.sh/k8snoprivileged created
```

Creamos el *Constraint* para activar la política:

no-privileged-pods.yaml

```
kubectl apply -f no-privileged-pods.yaml
```

```
[~/proyecto_integrado/demos/demo3] (main)
alejandro$ kubectl apply -f no-privileged-pods.yaml
k8snoprivileged.constraints.gatekeeper.sh/no-privileged-pods created
```

Intentamos desplegar un *Pod* privilegiado:

```
kubectl apply -f pod-privileged.yaml
```

```
[~/proyecto_integrado/demos/demo3] (main)
alejandro$ kubectl apply -f pod-privileged.yaml
Error from server (Forbidden): error when creating "pod-privileged.yaml": admission webhook "validation.
gatekeeper.sh" denied the request: [no-privileged-pods] No se permite el uso de contenedores privilegiad
os en el pod pod-privileged
```

Como vemos, la política bloquea el despliegue.

Ahora, intentaremos desplegar un *Pod* sin privilegios:

```
kubectl apply -f pod-normal.yaml
```

```
[~/proyecto_integrado/demos/demo3] (main)
alejandro$ kubectl apply -f pod-normal.yaml
pod/pod-normal created
```

Como vemos, el despliegue se hace correctamente.

7. Conclusiones y propuestas de trabajo

Este proyecto ha permitido desplegar, de forma completamente local, una infraestructura moderna para la gestión de imágenes de contenedores basada en Kubernetes, Harbor y OPA Gatekeeper. Se ha demostrado la viabilidad de construir un entorno seguro, funcional y automatizado sin depender de servicios cloud ni herramientas de pago.

Entre los principales logros se encuentran:

- La integración exitosa de Harbor como registro privado de contenedores con escaneo de vulnerabilidades (Trivy).
- El despliegue automatizado de componentes mediante Helm.
- La aplicación de políticas de seguridad mediante OPA Gatekeeper para reforzar la gobernanza del clúster.
- La demostración práctica de todo el sistema, desde la creación de imágenes hasta su despliegue condicionado por políticas.

Este entorno resulta aplicable a contextos reales como desarrollo interno en empresas, formación profesional o entornos restringidos sin acceso a la nube pública.

Como posibles líneas futuras de mejora o ampliación del proyecto se proponen:

- ➢ Integrar un sistema CI/CD como Jenkins, Tekton o GitHub Actions para automatizar pruebas y despliegues.

- ➢ Añadir una base de datos de vulnerabilidades interna y políticas personalizadas más complejas en Rego.

- ➢ Explorar la integración de más registros (con federación) o replicación entre instancias de Harbor.

- ➢ Implementar autenticación externa en Harbor mediante LDAP o SSO.

8. Dificultades encontradas

Durante el desarrollo del proyecto se han encontrado diversas dificultades técnicas que han requerido ajustes en el planteamiento inicial:

- **Certificados TLS y HTTPS:** En el diseño original se planteó el uso de *cert-manager* junto con Let's Encrypt para generar automáticamente certificados válidos. Sin embargo, esto requiere que el dominio utilizado (*harbor.local*) sea accesible públicamente o apunte a una IP con resolución DNS válida, lo cual no es posible en un entorno totalmente local. El intento de emitir certificados para dominios *.local* o *.example.com* fue rechazado por la ACME de Let's Encrypt.

- **Problemas con el túnel de Minikube:** Fue necesario mantener *minikube tunnel* activo para que los servicios *LoadBalancer* expusieran correctamente los puertos. Esto obligó a mantener un proceso adicional en ejecución y a modificar rutas de red manualmente.

- **Resolución DNS dentro de Minikube:** El pod de Kubernetes no podía resolver *harbor.local* al intentar descargar imágenes desde Harbor. Fue necesario modificar el *ConfigMap* de CoreDNS para que la resolución del dominio se hiciera correctamente desde dentro del clúster.

- **Uso de HTTP en lugar de HTTPS:** Debido a las limitaciones mencionadas, se optó por deshabilitar TLS y utilizar HTTP en el entorno local. Esto simplificó el acceso y evitó errores con certificados autofirmados, pero sacrificó parte del objetivo inicial de seguridad automática.

- **Acceso al registro desde dentro del clúster:** Incluso tras configurar los secretos *imagePullSecrets*, fue necesario configurar el demonio de Docker dentro de Minikube para aceptar registros inseguros (*insecure-registries*), lo cual implicó modificar su configuración interna.

Estas complicaciones forman parte del aprendizaje real que proporciona el despliegue de sistemas en entornos complejos y no controlados, como ocurre con entornos empresariales o locales.

9. Bibliografía, enlaces y recursos utilizados

A continuación se listan las fuentes consultadas durante el desarrollo del proyecto, tanto para los fundamentos teóricos como para las guías prácticas de despliegue y solución de problemas:

9.1. Documentación oficial

- Harbor Documentación
- Trivy Documentación
- OPA Gatekeeper Documentación
- Minikube Documentación
- Helm Documentación
- cert-manager Documentación
- Ingress NGINX Controller

9.2. Referencias y guías técnicas

- Restringir registros de contenedores usando OPA
- Cómo añadir dominios locales en CoreDNS

- Configurar registros inseguros en Docker
- Kubernetes Secrets para imagePullSecrets
- Como instalar Harbor en Ubuntu
- Repositorio GitHub de OPA
- Kubernetes con OPA
- Instalación de QEMU/KVM en Linux
- Curso de Kubernetes
- Curso de Docker