Trabajo de fin de grado
-
Seguridad en K8s, Protección 360º
K8s security, 360 º protection

Autor: Óscar Anadón Olalla
Director: Francisco Borja Buera
Tutor: Francisco Javier Zarazaga

Convocatoria 2022 / 2023

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Universidad
Zaragoza
1542

Seguridad en K8s, protección 360 º

Resumen

Durante el desarrollo del presente trabajo de fin de grado, Seguridad en K8s, protección 360º, se ha llevado a cabo un estudio sobre la seguridad que emplean los sistemas de Kubernetes, comparando la seguridad nativa que dicha plataforma ofrece y las diferentes herramientas disponibles en el mercado. Para lograr este objetivo se han ejecutado distintas configuraciones, instalando el software pertinente y comparando diversos parámetros, así como la experiencia general de uso.

La compañía NTT Data centra gran parte de sus medios en proporcionar servicios cloud, diseñando e implementando aplicaciones alojadas en la nube. Debido a la facilidad que muestra Kubernetes para gestionar los recursos disponibles, su empleo dentro de la empresa es muy elevado, por lo que este trabajo facilitará la toma de decisiones a la hora de llevar a cabo la estructura inicial de un proyecto o escoger las tecnologías que este empleará, dependiendo de los requerimientos del mismo.

De primera instancia se puede pensar, como se ha hecho durante los últimos años, que Kubernetes es una plataforma segura donde realizar los despliegues necesarios de cualquier aplicación, sin embargo, aún implementando una configuración correcta y siendo minucioso con cada parámetro relativo a la seguridad, es posible que existan brechas o vulnerabilidades que podrían ser explotadas por un atacante.

Como se observa a lo largo del documento presente, es necesario el empleo de software especializado que permita realizar análisis periódicos del estado del sistema y alertar en caso de que exista un escenario defectuoso e incluso proporcionar nuevas herramientas y capas de seguridad necesarias para que ningún usuario indebido pueda acceder a datos que no le corresponden, manipularlos o impedir su utilización cuando se requiera.

Finalmente, se ha concluido qué herramientas son mejores para cada suceso, comparando sus características y observando cómo trabajan en casos concretos. A pesar de que sí es posible en ciertas ocasiones dar una respuesta certera sobre qué software es mejor, se ha observado que en muchas de las ocasiones no es posible obtener un veredicto absoluto que coloque a una por encima de otra, sino que dependerá mucho del contexto en el que se encuentren.

Declaración de autoría

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe remitirse a seceina@unizar.es dentro del plazo de depósito)

D./Dª. Óscar Anadón Olalla ,

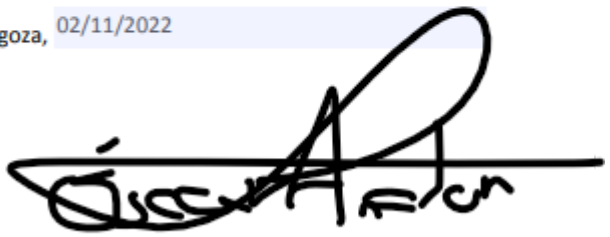
en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de Estudios de la titulación de Grado en Ingeniería Informática (Título del Trabajo)

Seguridad en K8's. protección 360º

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 02/11/2022



Fdo: Óscar Anadón O.

Índice

Agradecimientos

Aunque hay mucha gente que merece reconocimiento me gustaría mencionar entre todos ellos a mis compañeros de NTT Data, que tantas cosas me han explicado una y otra vez.

A todos los profesores que presentan una gran vocación por el aprendizaje y la informática, que logran crear interés para que incluso uno mismo siga investigando por su cuenta.

A mis amigo Sergio, sin ti todavía tendría algunas decenas de créditos pendientes. Gracias amigo.

A Marina, que tanto me apoya y confía en mí y en mis sueños como pocos lo hacen, además de haberse tragado 85 charlas sobre Kubernetes.

A mis yayas y yayos, con los que tanto he disfrutado y los cuales estarían ahora tan orgullosos de mi.

Por último y en especial,a mi madre y a mi padre, los que siempre me han apoyado y enseñado la importancia de los estudios, gracias por vuestros consejos y ayudas. Os quiero mucho.

1. Introducción

1.1 Contexto del Trabajo

Este proyecto ha sido realizado en NTT DATA Co., compañía japonesa de comunicaciones especializada en la integración de sistemas. NTT tiene actualmente diferentes sedes repartidas por España, una de las cuales se ubica en Zaragoza y centra gran parte de su operativa en el desarrollo de servicios en la nube, realizando diversos proyectos para clientes, en muchos de los cuales se emplea Kubernetes.
Es innegable el crecimiento que ha tenido la demanda de servicios en la nube, como puede ser el empleo de contenedores, microservicios y similares. Por ello, NTT ha decidido realizar una apuesta clara por Kubernetes, uno de los mayores orquestadores de contenedores actuales. Cada vez son más las compañías que emplean esta plataforma para desplegar sus servicios, sin embargo, en muchas ocasiones se lleva a cabo dicha tarea sin comprender a la perfección las implicaciones y características que porta, por ello existe una preocupación incipiente de la existencia de brechas de seguridad. Es por esto que ha decidido poner en marcha una línea de actuación dedicada a la evaluación de los riesgos que puede haber y a la planificación de las estrategias de contingencia. En dicha línea se encuentra este proyecto, en el que se investigarán posibles fallas de seguridad y metodologías de actuación para las mismas.

1.2 Contexto Tecnológico

Lo primero de todo, ¿qué es Kubernetes? Kubernetes, de manera abreviada K8s, es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios [1]. Su empleo provee al usuario de numerosas y heterogéneas utilidades que facilitan, en gran medida, la administración de sus aplicaciones. Así entonces, NTT realiza una apuesta decisiva por su uso, siendo este una parte fundamental de su producto, por lo que el desarrollo de este trabajo de fin de grado facilitará la toma de decisiones existente en diferentes proyectos. Esta tecnología presenta dos motivos principalmente que la hacen muy atractiva.

En primer lugar, el auge y crecimiento de los servicios cloud: Es creciente el número de empresas (y particulares) que deciden utilizar recursos en la nube. Gran parte de ello se debe a la economicidad de su uso y la adaptabilidad que proporciona. Por ejemplo, para una empresa que requiera la utilización de máquinas con un sistema operativo (SO) concreto únicamente en el desarrollo de un producto en particular, le sería muy ineficiente y costoso comprar dichas máquinas. En lugar de ello despliega dichos SO en contenedores mediante Kubernetes durante la duración que se necesite, ahorrando recursos y tiempo, lo que directamente implica una disminución de los costes. Además de esto, otra gran ventaja que posee es la escalabilidad, la potencia que ofrece K8s para aumentar sus capacidades es muy elevada, tanto de forma periódica como para momentos puntuales, de igual manera también es posible disminuir dichas capacidades de manera sencilla.

En segundo lugar Google, el desarrollador inicial de esta tecnología, la cual integra en su plataforma de cloud, Google Cloud Platform (GPC). A pesar de que hay grandes competidores en este sector como Amazon (AWS) o Microsoft (Azure), suele suceder que la tecnología ofrecida por Google acaba siendo una de las más predominantes del mercado. No se está afirmando rotundamente que vaya a ser el principal exponente de los servicios cloud, pero sí uno de los más importantes. Todo esto se encuentra recogido en el *Magic Quadrant* de Gartner [2] donde expone y explica los principales potenciales de servicios de infraestructura y plataforma en la nube. En la figura 1 se puede observar que Google se encuentra muy bien posicionado y con una espera de crecimiento notable.

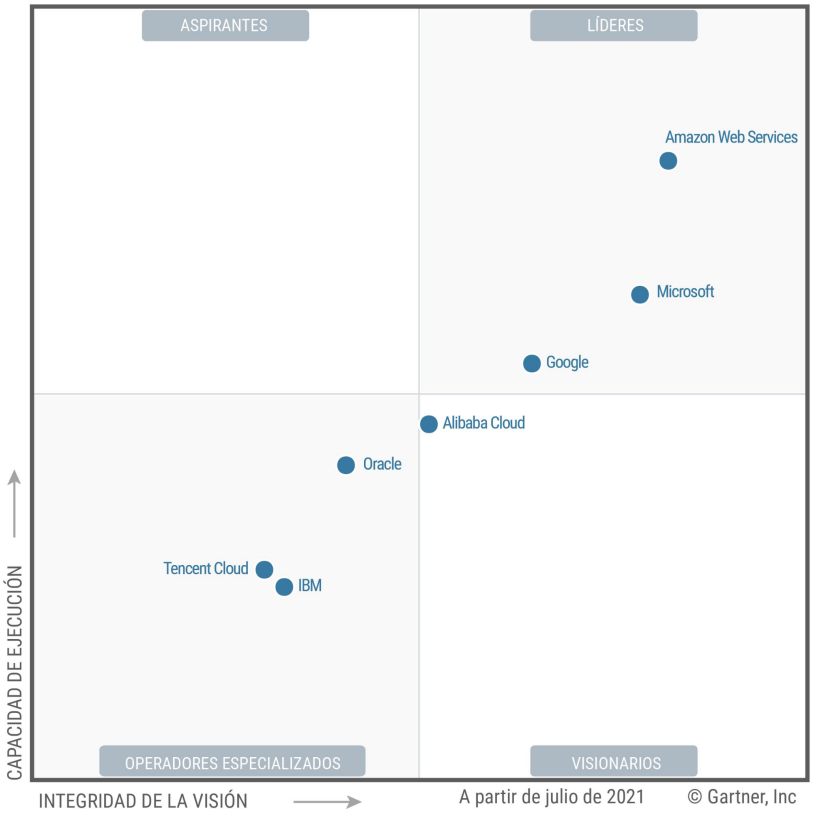


Figura 1. Gráfico de los principales competidores en Cloud [2]

1.3 Motivación y problema que se aborda

A la hora de escoger un tema sobre el trabajo de fin de grado han existido varias propuestas interesantes, entre las cuales aparecían numerosas relacionadas con el entorno cloud. Para poder establecer un primer contacto con el mundo laboral, se decidió realizarlo a través de una empresa, entre las cuales destacó NTT DATA. Tras analizar varios proyectos surgió la idea de gestionar la seguridad de un sistema que emplee Kubernetes, lo cual resultó interesante y práctico debido a que la utilización actual de este orquestador es muy elevada. Sumado a esto, se escogió NTT por el gran departamento (DAR, Digital Architecture) que tienen relacionado con las tecnologías cloud, ya que de manera directa permitiría contar con una amplia gama de recursos tanto materiales como de información y de personal.

El problema que se aborda es una cuestión de gestión de riesgos en la línea de lo visto en las asignaturas de ingeniería del software, sistemas distribuidos y seguridad informática. La dependencia creciente de la tecnología de Kubernetes para el despliegue de sistemas la convierte en un punto de riesgo cada vez más alto. Es por ello que en el TFG se plantea el estudio de los posibles riesgos, además de cómo elaborar estrategias de mitigación y planes de contingencia.

1.4 Alcance, objetivos y limitaciones

En el presente documento se va a exponer el análisis realizado a diversas configuraciones de Kubernetes(K8s), cuyo entendimiento total, logrará mediante el empleo de diferentes herramientas una comprobación 360º en lo respectivo a su seguridad, con la finalidad de intentar proteger al máximo el sistema pertinente, así como de comprender las vías que llevan a ello.
Tras el estudio de herramientas empleadas para el monitoreo y mantenimiento de la seguridad en sistemas sustentados por Kubernetes, se ha determinado cuál de ellas es mejor en relación al ambiente en el que se desarrollan así como también si es necesario su empleo en K8s o por si mismo genera intrínsecamente un sistema robusto.

1.5 Herramientas de trabajo

Durante la realización del proyecto se han empleado las siguientes herramientas:

Kubernetes: Utilizado como herramienta de gestión de contenedores, despliegues y similares (link ¿Qué es Kubernetes?)
Docker: Empleado para el manejo de imágenes en contenedores (link https://www.docker.com/)
Kubectl: Herramienta CLI para trabajar con K8s, desplegando y gestionando aplicaciones o inspeccionando recursos del cluster (link Instalar herramientas | Kubernetes)
Kube-bench: Software para el análisis de configuración de plataformas, que comprueba si K8s esta documentado de forma segura según el CIS Kubernetes Benchmark (link https://github.com/aquasecurity/kube-bench)

Kube-Hunter: Software para el análisis de configuración de plataformas, que comprueba si K8s esta documentado de forma segura según el CIS Kubernetes Benchmark (link https://github.com/aquasecurity/kube-hunter)
Kyverno: Motor de políticas diseñado para K8s (link Kyverno)
OPA Gatekeeper: Motor de políticas para proyectos nativos en la nube (link Introduction | Gatekeeper)
Falco: Herramienta utilizada para el manejo de seguridad activa dentro de contenedores, K8s y Cloud (link Open Source Container Security Tools: Falco - Sysdig)
Istio: Software empleado para la gestión, observabilidad e implementación de seguridad de comunicaciones. (link https://istio.io/)
Kubernetes Network Policies: Solución nativa de K8s para la aplicación de políticas en red (link Network Policies | Kubernetes)
Trivy: Herramienta para el análisis de imágenes (link https://github.com/aquasecurity/trivy)
Grype: Herramienta para el análisis de imágenes (link https://github.com/anchore/grype)
SealedSecrets: Software utilizado para el cifrado de secretos (link https://github.com/bitnami-labs/sealed-secrets)
Hashicorp-vault: Software empleado para el cifrado de secretos, generación y gestión de los mismos (link https://github.com/bitnami-labs/sealed-secrets)

1.6 Esquema general de la memoria del proyecto

Con el objetivo de facilitar la comprensión y desglosar de manera práctica el contenido, se han establecido las siguientes secciones.

- **Resumen**: Contiene de forma sintetizada todo el trabajo realizado
- **Introducción**: Se define el objetivo y el alcance del proyecto, el contexto en el que se realiza y la metodología llevada a cabo durante su desarrollo.
- **Trabajo desarrollado**: Se explica el procedimiento seguido durante el proyecto, así como una explicación amplia de los problemas estudiados, las vías mediante las cuales estos pueden ser resueltos y la comparativa de las mismas. Dentro de esta sección se encuentran las siguientes categorías:
    - *Análisis previo de plataforma*: Se realiza un estudio previo del sistema a emplear, comprobando que plantea una configuración correcta en situaciones como directorios con permisos adecuados, conexiones permitidas o similares.
    - *Aplicación de políticas de seguridad*: En esta sección se escribe sobre cómo aplicar políticas en Kubernetes, entendidas estas como el establecimiento de normas en cuanto a la utilización o acceso de recursos.
    - *Seguridad activa*: En dicho apartado se comprueba cómo mantener asegurado el sistema, mediante el análisis activo de las acciones que “están” ocurriendo.
    - *Seguridad en comunicaciones*: Esta sección estudia la interacción de componentes, comprobando las posibilidades de asegurarla, cómo monitorizar y otros empleos de utilidad en los sistemas distribuidos.
    - *Análisis estático de imágenes*: Se chequean algunas de las imágenes más populares en el empleo de contenedores, observando vulnerabilidades que pueden contener.

    - *Gestión de secretos*: En esta sección se trata el tema de los secretos, los cuales se emplean de manera directa para limitar el acceso a los datos.
    - *Ejemplo práctico*: Se plantea un caso de uso real, aplicable al entorno de NTT DATA Co. de cómo emplear lo estudiado anteriormente, viendo como funciona Kubernetes y las herramientas externas en suceso de un desarrollo real.
- **Conocimientos adquiridos**: En esta sección se describen las diferentes aptitudes obtenidas.
- **Ideas futuras**: Se habla sobre los posibles trabajos o estudios realizables a partir de este proyecto.
- **Conclusiones**: El último apartado en lo que respecta al trabajo, en él se aborda el delimitar qué herramienta es necesaria y cuál de ellas es mejor para cada posible caso que se plantee, así como el definir de manera amplia, si Kubernetes ofrece por sí mismo el poder manejar una seguridad competente para según qué ocasiones.
- **Bibliografía**: Sección a parte en la que aparecen los enlaces y material consultado y empleado

En los anexos se explican detalles a tener en cuenta sobre la realización del trabajo. En estas secciones se podrán encontrar pruebas de ejecuciones realizadas, diferentes procesos de instalación relacionados con las herramientas mencionadas anteriormente y ejemplos de utilización que presentan cada una de estas.

- **Anexo I:** Información relacionada con la aplicación de políticas
- **Anexo II:** Información relacionada con la seguridad activa
- **Anexo III:** Información relacionada con la aseguración de comunicaciones
- **Anexo IV:** Información relacionada con el análisis de imágenes
- **Anexo V:** Información relacionada con el empleo de secretos
- **Anexo VI:** Ejemplo práctico

2. Trabajo desarrollado

En lo que respecta al estudio de la seguridad sobre un sistema en K8s se ha seguido una metodología sustentada en los siguientes pasos:

1. Análisis del ámbito a estudiar
2. Estudio de las herramientas nativas de K8s
3. Estudio de las herramientas externas a utilizar
4. Explicación de vulnerabilidades
5. Comparativa y remediaciones

Así entonces se llevará a cabo de manera similar este esquema en cada uno de los flancos a estudiar, comprendidos éstos como un conjunto de características cuya configuración está relacionada, por lo que estas pueden ser gestionadas, observadas y controladas de manera agregada.

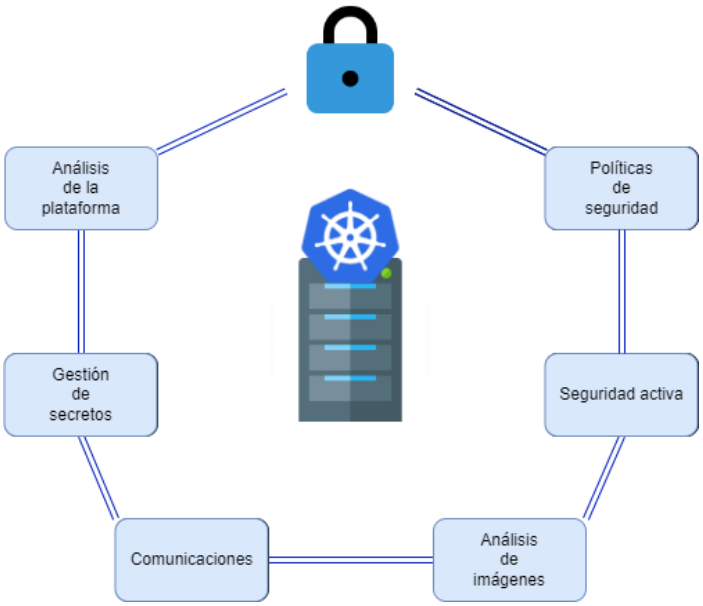


Figura 2. Resumen gráfico del proyecto

2.1 Análisis de la plataforma

El primer paso para trabajar de forma segura es la previa comprobación de que todo está configurado de manera correcta. Esta actividad es cuanto menos obligatoria y se ha de tener muy en cuenta, ya que en determinadas distribuciones es posible que exista cierta permisibilidad a la hora de realizar tareas concretas con el fin de que el usuario tenga facilitada la experiencia. Un ejemplo de esto suelen ser los privilegios definidos al acceder a directorios o archivos, en entornos de producción estos deben estar limitados al mínimo posible [3].
Aunque esta tarea se nombra y se ha de ejecutar al inicio, puede ser también comprobada de manera continua al introducir elementos nuevos, como nodos, pods o contenedores.

Para poder comprobar esto de forma nativa en K8s sería necesario revisar manualmente cada parámetro de interés y chequear si tienen los valores correctos, lo cual resultaría una tarea tediosa y compleja.

Este flanco inicial tiene la intención de hacer una **comprobación amplia** de la plataforma, en la cual se comprobarán aspectos genéricos y de fallo común. Para lograr este paso inicial se han planteado **kube-bench** y **kube-hunter**, ambas desarrolladas por la compañía aquasecurity, una de las más importantes empresas de seguridad cloud.

2.1.1 Kube-Bench

En primer lugar se ha utilizado kube-bench, reconocido software creado por aquasecurity, el cual se asegura de si Kubernetes ha sido implementado correctamente llevando a cabo las comprobaciones documentadas en CIS Kuberntes Benchmark , organización sin animo de lucro cuya misión es lograr un mundo digital seguro [4]. Cabe destacar que la configuración de kube-bench se realiza con archivos en formato YAML, esto implica directamente una buena base de actualización para que el desarrollo de las pruebas se complete y avance.

Kube-Bench es cuanto menos sencillo de llevar a cabo. Primero será necesario escoger su lugar de instalación, en este caso será el host que se conecta con minikube (el ordenador personal), podría ejecutarse dentro de un POD en algún nodo del cluster pero esto complica innecesariamente la labor, para observar las diferencias, dicho lugar de ejecución es usado con la herramienta Kube-Hunter.

En el Anexo I sección I se muestran algunos ejemplos del trabajo que se puede realizar con kube-bench, el explicar todos ellos en este documento resultaría inviable ya que se habla de cientos de casos a analizar, de igual modo pasa con muchos de los programas empleados durante este proyecto.

2.1.2 Kube-Hunter

Kube-Hunter es una herramienta open-source que encuentra debilidades en clusters de Kubernetes, esta *testea* un dominio o rango de direcciones probando diferentes configuraciones que puedan exponer al cluster.

Se encuentra disponible mediante contenedor, web, o ejecutando código en python.
Kube-Hunter es posible instalarlo en diferentes ubicaciones, en este se ha escogido un pod dentro del propio cluster, ya que esto además indica que tan expuesto quedaría el cluster si alguno de los pods quedará comprometido [5], puesto que te encuentras dentro del cluster observando todas las vulnerabilidades con cierto nivel de permisos. De igual forma se podría haber realizado antes con kube-bench, de este modo se prueban diferentes métodos de instalación.
Al ejecutar Kube-Hunter este muestra opciones sobre qué se quiere analizar:

```
Choose one of the options below:
1. Remote scanning      (scans one or more specific IPs or DNS names)
2. Interface scanning   (scans subnets on all local network interfaces)
```

3. IP range scanning (scans a given IP range)

Esto permite escanear una máquina remota, las interfaces de red de la máquina o un CIDR específico, respectivamente. Ya escogido el apartado a investigar, al empezar aparecen una serie de mensajes que explican las vulnerabilidades que porta el sistema, en caso de que estas existan.

Seguidamente se visualizan una serie de tablas indicando de manera desglosada y muy clara la vulnerabilidad, donde ocurre y en qué consiste esta. Aunque aquí Kube-Hunter, al contrario que Kube-Bench no resuelve dichas vulnerabilidades, emplea una plataforma web [6] con un pequeño buscador en el que al introducir la vulnerabilidad enseña a como solucionarlo.

Entendiendo más cuál es el potencial de Kube-Hunter se explicaran y solucionaran algunas de las fallas encontradas. Para la ejecución de Kube-Hunter se ha empleado la misma configuración cloud que con Kube-Bench, es decir la que viene por defecto al instalar minikube. Dichas pruebas serán mostradas en el Anexo I sección II .

Tras absorber las pruebas aportadas es deducible afirmar que ambas herramientas tienen una función similar (por no decir idéntica), tratando de encontrar distintos tipos de fallas.
Realmente no existen diferencias significativas entre ambas.
Los problemas mencionados anteriormente pueden ser resueltos de diferentes maneras:

- **Manualmente**: Tanto Kube-bench como Kube-hunter cuentan con una extensa documentación en la que explican detenidamente cada uno de los test que se pasan y cómo remediarlos. Tan solo sería necesario seguir los pasos.
- **Automáticamente**: Sería plausible programar scripts que realizarán la configuración segura mediante la ejecución de comandos.
- **Empleo de políticas**: Existe software mediante el cual se aplican ciertas políticas que pueden modificar o prevenir determinados comportamientos para lograr una seguridad en K8s. Estas serán explicadas a continuación.

2.2 Aplicación de políticas de seguridad

El manejo de las políticas de seguridad es algo fundamental para controlar y delimitar la actividad de usuarios y procesos en cualquier tipo de sistema. Este concepto se basa en redactar una serie de normas que son chequeadas antes de realizar una acción, teniendo en cuenta que si alguna de dichas acciones es impedida por una de las normas anteriores se tomarán medidas en relación a esta.

En este apartado se habla de dos importantes herramientas empleadas para aplicar una serie de políticas, según las mismas, en caso de que no se estén cumpliendo, se podrá actuar en consonancia según desee el administrador, todo esto hará del sistema un lugar mucho más seguro.
Es cierto que Kubernetes desarrolla de manera nativa políticas en el sistema, sin embargo el empleo de estas se encuentra un tanto limitado, lo cual se puede solucionar trabajando con aplicaciones externas como las que se verán a continuación.

Kyverno

En primer lugar Kyverno, es un motor de políticas diseñado **exclusivamente** para K8s. Este se usa mediante el comando *kubectl* y su descripción con YAML, por lo que un usuario semi acostumbrado a Kubernetes se encontrará con grandes facilidades a la hora de manejarlo ya que únicamente deberá conocer los recursos que desee modelar.
Sobre estos se aplican tres acciones básicas a partir de las que se rige todo el funcionamiento de Kyverno: Validar (allow), mutar (mutate) y generar (generate), lo que permite eficazmente afirmar si un recurso está definido adecuadamente, modificarlo en caso de que sea necesario o crear uno nuevo, respectivamente [7]. No obstante, además de esto, Kyverno ofrece un servicio de validación de imágenes, el cual es comentado más adelante.

Siguiendo con el modelo de explicación del apartado anterior, se explican a continuación una serie de políticas aplicables con Kyverno para comprobar las opciones que este ofrece y sus posibles usos. Dichas políticas se podrán comprobar en el Anexo I sección III

OPA Gatekeeper

En segundo lugar se presenta Open Policy Agent (OPA) Gatekeeper, se define como un motor de políticas de propósito general. Este permite validar o mutar(esta característica es relativamente nueva) solicitudes de diversa índole. Emplea un lenguaje de programación propio, Rego, utilizado para aplicar la lógica necesaria en el filtrado de las peticiones de acciones.

Se pueden comprobar las pruebas realizadas mediante OPA Gatekeeper en el Anexo I sección IV .

La comparativa entre Kyverno y OPA puede resultar muy útil ya que dependiendo de las características del proyecto que se desee, claramente se deberá escoger entre una herramienta u otra, a pesar de que ambas tengan un objetivo similar.

De primera instancia OPA no es exclusivo de Kubernetes, Kyverno si. Esto conlleva ventajas e inconvenientes dependiendo de las necesidades del usuario. En caso de que se trate de un proyecto abierto, o que puede escalar y comunicarse con distintos tipos de plataformas cloud, sin duda sería necesario escoger OPA, sin embargo, en caso de que dicha actividad fuera a llevarse a cabo únicamente en el entorno de K8s sería recomendable el empleo de Kyverno, ya que está diseñado exclusivamente para este sistema y no adaptado a él.

Seguidamente, Kyverno es un motor mucho más sencillo que OPA. Al igual que el punto anterior esto presenta ventajas e inconvenientes. Esto es debido a que para aplicar la operativa de Kyverno únicamente se emplean ficheros YAML que definen directamente la lógica que se va a aplicar, mientras que OPA lo realiza mediante su propio lenguaje, Rego. Esto hace que la curva de aprendizaje para el usuario sea mucho más extensa en OPA, sin embargo, OPA permite describir políticas complejas, lo que en caso de requerir esquemas muy específicos para el sistema, sería muy beneficioso. Nuevamente depende en este caso de las necesidades demandadas.

En cuanto a las capacidades de cada uno, Kyverno aparece mucho mejor posicionado, ya que este permite 3 tipos de actuación en base a las operaciones que se realicen: Permitir, añadir y mutar, todo ello sumado a un verificador y registrador de imágenes. OPA únicamente presenta la de permitir (o no) ciertas ejecuciones, si es cierto que han incluido la opción de mutar, pero esta se encuentra en una etapa incipiente.
Analizando otros aspectos, OPA lleva más tiempo en el mercado contando con una comunidad un tanto mayor, aunque Kyverno le sigue muy de cerca y con un crecimiento muy amplio.

Como se ha explicado, la elección entre un motor de políticas u otro radica en las necesidades que tenga el usuario, pero en este caso concreto, se trata de un cluster exclusivo de Kubernetes que no emplea políticas excesivamente sofisticadas. Por lo que particularmente escogería Kyverno. Las aptitudes de cada uno se recogen en la Tabla 1.

	Sencillez	**Descripción de políticas**	**Operaciones**	**Plataformas**
Kyverno	Alta	Limitada	Permitir, mutar, añadir, comprobar imagenes	Únicamente K8s
OPA	Baja	Alta	Permitir, mutar(beta)	Múltiples

Tabla 1. Comparación Kyverno VS. OPA

2.3 Seguridad activa

En cuanto a la seguridad activa es imprescindible hablar de **Falco**, desarrollada por la empresa SYSDIG y donada a la Cloud Native Foundation (CNF), centro de código abierto e independiente que aloja proyectos como Kubernetes y Prometheus para hacer que la nube nativa sea universal y sostenible[8].

Falco usa las *system calls* o llamadas al sistema para monitorizar y asegurar al propio sistema. Su actividad consiste, de manera resumida en:

- Análisis de las llamadas del sistema Linux desde el kernel en tiempo de ejecución
- Realizar “asserts” de las comunicaciones frente a un motor de reglas
- Alertar cuando una regla ha sido violada. [9]

Kubernetes ofrece de manera nativa opciones que permiten leer información sobre las actividades del sistema. El componente Kube-apiserver lleva a cabo la auditoría. Cada petición en cada fase de su ejecución genera un evento, que se pre-procesa según un cierto reglamento y se escribe en un backend [10].

En cuanto a las políticas, como se ha visto anteriormente, en Kubernetes se definen en formato YAML y dependiendo del evento ocurrido se pueden auditar o impedir. De manera similar Falco realiza este contraste con las conocidas *Falco rules*, en las que además distingue 3 tipos de elementos(Rules, Macros, Listas) que logran una gestión de las reglas

muy sencillo y práctico. Se puede observar el funcionamiento y diferencias de ambas en el Anexo II Sección I .

Con los elementos anteriores se realiza un contrapunto frente a los diferentes eventos que ocurren en el sistema, provenientes de las llamadas al sistema, los logs de kubernetes y actividad cloud heterogénea, observable en la Figura 3.

Figura 3.Funcionamiento general de Falco (link Sysdig - Falco)

Para entender un poco mejor su funcionamiento se contempla un ejemplo práctico, se puede pensar que teniendo nativamente a Kubernetes comprobando los distintos procesos y generando logs de ellos no sería del todo necesario emplear Falco, sin embargo la aportación que este hace a la seguridad y gestión de K8s es bastante elevada. Además de observar y analizar los heterogéneos eventos que se ejecutan en Kubernetes, es capaz de estudiar actividad proveniente de otros entornos, como puede ser AWS, Azure o Google Cloud, lo que presenta gran utilidad en despliegues multiplataforma.
Falco recopila toda esta información y con aquella que hace *match* frente a alguna de las Falco Rules podrá generar una alerta visible o actuar en consecuencia.
En cuanto a las alertas, Falco las puede mostrar por distintas salidas, estas pueden ser:

- **salida estándar**, como stdout, docker logs o kubectl logs; syslog;
- **archivos**, definidos a elección del usuario.
- **entradas a programa**, este punto resulta muy útil ya que dichas alertas pueden ser enviadas a cualquier programa, como por ejemplo un email, y trabajar directamente con ellas.
- **http[s] methods / gRPC clientes**, las cuales emplean formato JSON, lo que implica que las aplicaciones podrían utilizar de primera mano dichos datos como por ejemplo en dashboards, notificaciones u otras utilidades.

El empleo o no de estas salidas se escribe en el fichero /etc/falco/rules.d/falco.yaml, donde aparece una configuración genérica de la vía que actuará Falco, se observará su uso en Anexo II Sección II .

Se ha comprobado que Falco tiene grandes cualidades como motor de detección, se marcan unas reglas que el sistema ha de cumplir y en caso de que no sea así, se genera una alerta. Sin embargo, para asegurar realmente el sistema, es necesario responder frente a esos despropósitos. De esta manera en conjunción con otras aplicaciones Falco se comportará como un motor de respuesta, respondiendo cuando sea necesario si una o más

reglas no se cumplen. Para lograr esto se emplea también la herramienta Falco Sidekick, se define como un demonio simple que permite conectar Falco a tu ecosistema. Este toma los eventos de Falco y sus diferentes salidas colocando todo en un punto único[11]. Junto con esto es necesario un servicio que pueda actuar frente a los distintos eventos como pueden ser Kubeless, OpenFass o Argo. Mediante estos servicios se definen funciones que son capaces de desempeñar acciones concretas.

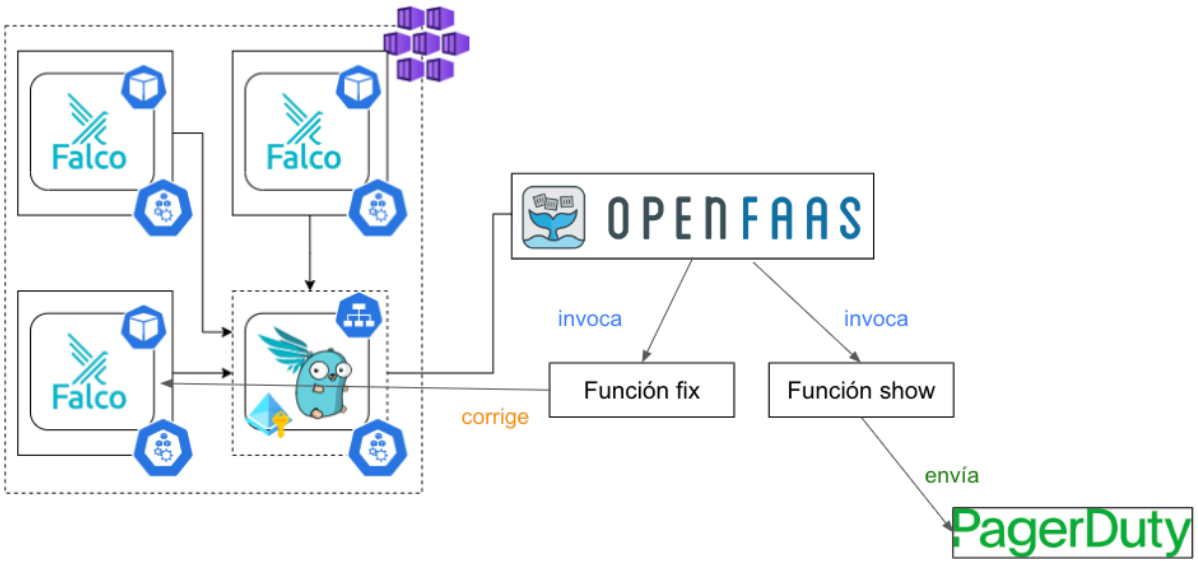


Figura 4. Ejemplo de funcionamiento Falco con remediación de incidencias

Se plantea un caso de uso en la Figura 4. Póngase el ejemplo de que en alguno de todos los nodos visibles en los que corre Falco hay establecida una Falco Rule para impedir ejecutar un SHELL dentro de un contenedor (por razones ya explicadas anteriormente). Si en alguno de dichos contenedores se ejecuta un SHELL, saltará una alerta la cual será enviada a OpenFaaS, invocando una función para solucionar dicho problema, por ejemplo se podría terminar el pod donde corre el contenedor para evitar que este continuará ejecutando comandos y una posible escalada de privilegios.
Dicha alerta se podría enviar a un sistema de incidencias como PagerDuty.

2.4 Seguridad en comunicaciones

El apartado relativo a las comunicaciones se entiende como la manera en la que unos componentes interactúan con otros (o con sí mismos). Este es un aspecto muy importante del desarrollo y puesta en funcionamiento de aplicaciones web. Por ello ha de estar configurado de manera muy precisa atendiendo a cada detalle.
Para entenderlo mejor se expondrá con un ejemplo visible en Anexo III Sección I.

Istio

De manera muy eficaz la herramienta Istio logra resolver muchas de las necesidades. Istio es una *servicemesh* de código abierto que se superpone de forma transparente a las aplicaciones distribuidas existentes [12]. Posee potentes funciones que logran eficientemente proteger, conectar y hacer monitoreo de servicios.

Las funciones de Istio básicamente se dividen en tres grandes grupos[12]:

- Gestión de tráfico
- Observabilidad
- Capacidad de seguridad

En cuanto a la gestión del tráfico, Istio permite, aplicando sus reglas de enrutamiento, controlar el tráfico y las llamadas a la API realizadas entre los diferentes servicios. Esto se entrelaza con las cuestiones de seguridad para lograr comunicaciones férreas.

Dividir una aplicación en microservicios sin duda puede traer grandes beneficios como puede ser la escalabilidad, tan importante en entornos cloud, sin embargo estos comprenden necesidades de seguridad.
En primer lugar es necesario encriptar el tráfico para evitar ataques de intermediarios así como cualquier tipo de *sniffing*, además de ello es necesario el empleo de distintos protocolos como TLS y mTLS (mutualTLS) que garanticen un control de accesos a cada servicio, junto con el uso de políticas que regulen el empleo (o no) de dichos protocolos. Por supuesto, siempre será necesario auditar cada acción que se realice para saber quien realizó determinadas actividades en un momento concreto.

Istio categoriza distintos niveles de seguridad para una protección modelable, son:

- **Secure by default** no es necesario introducir cambios en la aplicación o la estructura de la misma.
- **Defense in depth**, consiste en integrar el desarrollo con los sistemas de seguridad ya existentes.
- **Zero-trust network**, comprende el crear avanzadas soluciones en redes de dudosa seguridad.

Con el objetivo de comprender mejor el funcionamiento de la seguridad de Istio se explicará seguidamente su arquitectura vista desde un nivel alto

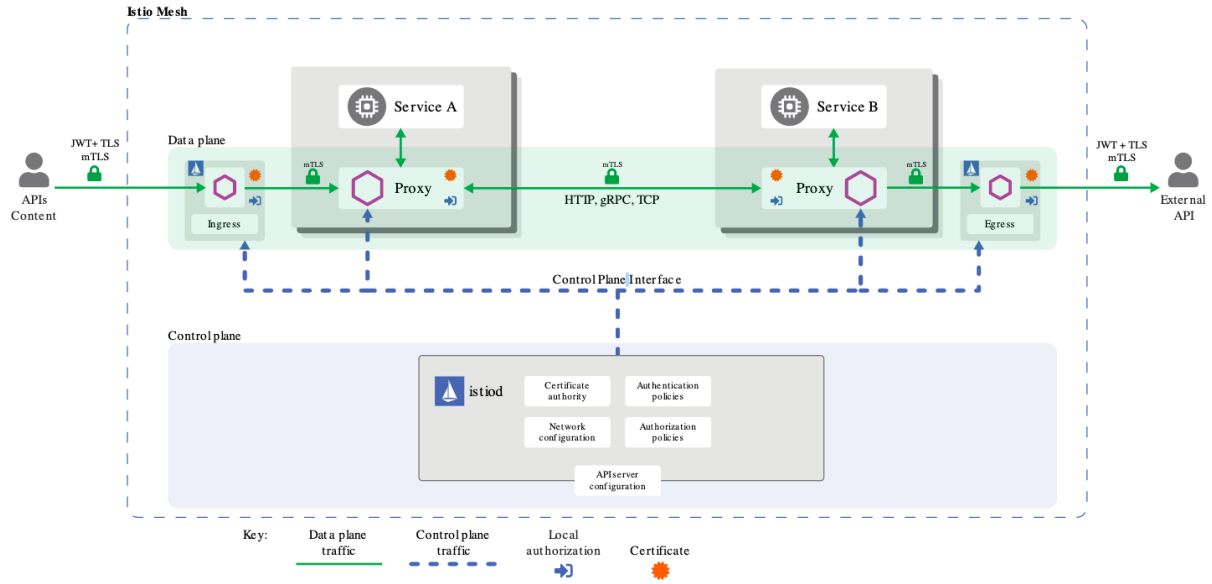


Figura 5. Arquitectura de isito link (Istio Arq)

En el diagrama visto en la Figura 5 es posible observar distintos componentes.

- Un CA (certificate authority) encargado de gestionar claves y certificados
- Un servidor API para la configuración, el cual distribuye políticas entre los proxies
- Proxies, llevando a cabo la comunicación entre cliente y servidor
- Extensiones que logran funciones extra como la auditoría.

En lo relativo a la gestión de identidades y el otorgar certificados, junto con cada proxy Envoy aparece un istio agent, los cuales se comunican con istiod para obtener los certificados y las claves pertinentes. El istio agent se ocupa de la caducidad del certificado.

Para la autenticación Istio plantea dos tipos. Autenticación de pares: esta es empleada entre servicios. Se realiza mediante el protocolo TLS (Transport Layer Security)
Esto puede ser aplicado de distintas maneras según las necesidades del usuario, concretamente mediante el empleo de 3 modos:

- **Permissive**: Los workloads permiten tráfico mTLS y texto sin formato.
- **Strict:** Únicamente se acepta mTLS
- **Disable**: mTLS se encuentra deshabilitado. Esto no se debe realizar a menos que el usuario aporte su solución personal en cuanto a lo que seguridad se refiere.

A su vez, esto propicia una gran flexibilidad a la hora de configurar las comunicaciones. Por ejemplo, si se tiene una organización ya existente y se desea migrar a Istio, mediante el modo permisivo se podrá ir logrando que los distintos workloads comiencen a emplear mTLS y aquellos que todavía no lo han podido instalar sigan enviado tráfico de la manera previa.

Es cierto que cualquiera puede pensar ¿realmente necesito Istio para emplear TLS? Obviamente la respuesta es no, pero la diferencia radica en que con el uso de Istio el empleo de este protocolo se realiza de forma automática empleando el patrón sidecar y los Envoy, mientras que si se quisiera realizar de manera manual sería necesario realizar muchos *certificate management*. Esta redirección de tráfico se podría constatar en tiempo real con el empleo de alguna de las herramientas de observabilidad siguientes, como pueden ser Graphana o Kiali.

Respecto a la observabilidad Istio proporciona una telemetría detallada que permite atender el comportamiento de los servicios, lo que facilita al usuario el poder mantener dichos servicios funcionando correctamente. Con esta funcionalidad se podrán leer métricas, hacer seguimiento distribuido y observar un completo acceso de registros.

Para poder entender mejor la capacidad que tiene Istio se comenta un caso en el que se han llevado a la práctica los conceptos explicados anteriormente, visible en el Anexo III Sección III . Se observa que Istio ofrece una gran operatividad con herramientas muy útiles para la administración de cualquier despliegue.

Kubernetes Network Policies

No hay que olvidar la opción nativa de Kubernetes, Kubernetes Network Policies (KNP), la cual ofrece una gran potencia a la hora de aplicar políticas de red. KNP permite un control del tráfico de IP/port entre pods, espacios de nombres y bloques de IP.

Por supuesto dicho control se realiza aplicando ficheros en formato YAML. Se pueden realizar múltiples acciones como denegar todo el tráfico de una fuente concreta, desviarlo hacia un puerto determinado o aceptar únicamente determinadas acciones. Se puede observar su funcionamiento de manera sencilla en el Anexo III Sección III

Tras un contundente análisis de ambas herramientas, se entiende que Istio es mucho más completo que KNP, ya que ofrece monitoreo, traza, aplicación de políticas, observabilidad, reparto de tráfico, mTLS simplificado… Sin embargo no implica que KNP no tenga escenario de uso, el empleo de esta es mucho más sencillo ya que de primera instancia no requiere instalación alguna y admite crear políticas decisivas para la seguridad del sistema. Para despliegues sencillos quizás no fuera necesario la instalación de Istio y sus *addons*, sin embargo, en cuanto aparezca un poco de complejidad se recomienda tajantemente su uso, ya que además de lograr una mayor comprensión de las estructuras existentes, ofrecerá al operador una visión amplia de todas las comunicaciones que están ocurriendo en su sistema.

2.5 Análisis estático de imágenes

Una **imagen** puede definirse como un paquete software ejecutable. Esta contiene los datos binarios de una aplicación y todas las dependencias software de la misma. Se consideran como una parte fundamental en la orquestación de K8s, ya que básicamente es lo que va a correr dentro de un contenedor. Por ello mismo, en estas no puede haber fallo alguno, ya que una brecha de seguridad en alguna imagen podría suponer el acceso a datos confidenciales o una escalada de privilegios.
Kubernetes de forma nativa no ofrece un medio, comandos o similar que permita analizar una imagen en busca de fallas, por ello se ha realizado el estudio de dos herramientas conocidas que han permitido obtener conclusiones sobre el estado de las imágenes, Grype y Trivy.

Trivy

Esta herramienta es de la conocida empresa Aqua, la cual provee de numerosas opciones que aportan seguridad a K8s, como la nombrada anteriormente kube-bench. Trivy es un escáner que permite analizar imágenes de contenedores, sistemas de ficheros e incluso repositorios de git con el objetivo de encontrar brechas de seguridad relativas a vulnerabilidades en la constitución de la imagen, configuraciones o secretos.

Para contemplar el ambiente de trabajo de Trivy se ha empleado uno de los ejemplos mostrados anteriormente, concretamente la imagen del POD de reviews usado para Istio, observable en el Anexo IV Sección IV. Cualquier imagen con un mínimo de complejidad bastaría para observar de manera general los aspectos que Trvy ofrece.

A pesar de que esta herramienta corre en una gran cantidad de sistemas operativos y busca en multitud de lenguajes específicos, no funciona con JAVA, algo a tener en cuenta ya que es uno de los lenguajes más usados de la actualidad. Por contra, está su competidor, Grype, el cual es explicado a continuación.

Grype

Se trata de un escáner para contenedores de imágenes y sistemas de archivos [13]. Este es compatible con multitud de SO. Una vez instalado Grype, su empleo es muy visible en el Anexo IV Sección II.

Para estudiar cuál de las dos se comporta mejor, se ha hecho una comparativa con 4 imágenes, analizadas con ambas herramientas. Dicho estudio se encuentra en el Anexo IV Sección III.
Por último se tendrá en cuenta la actividad que ambas herramientas tienen actualmente, se puede comprobar que Trivy tiene alrededor de 1200 forks y 12.900 stars en GitHub [14] mientras que Grype cuenta con 278 forks y 4000 starts [13].

Como conclusión en el análisis de imágenes, se escogería Trivy, la manera que tiene de arrojar los resultados más completa y clara, el apoyo de la comunidad y el pertenecer a una empresa que dedica gran parte de su operativa a la seguridad en K8s hacen que sea la herramienta a escoger. No ostenta, como es de suponer, en caso de utilizar proyectos con Java (lo cual puede ser bastante común), se deberá utilizar Grype u otro escaneador como Clair o similar.

2.6 Gestión de secretos

Se trata de una de las partes más importantes en lo que respecta a la seguridad de forma directa, ya que es uno de los mecanismos que permite limitar de manera intrínseca el acceso a determinados datos. Mediante esta forma se deben guardar credenciales, contraseñas, tokens, claves, certificados ssh y demás.
Para comprender su funcionamiento se propone un ejemplo de uso básico en el Anexo V Sección I

Respecto al uso de secretos existe un problema muy importante, **K8s no cifra** la información empleada para autenticación en los secretos, únicamente lo codifica en base64 para que “no sea visible al ojo humano”, lo que se puede revertir fácilmente con un comando que transcribe dicha cadena a texto de nuevo. Sin esfuerzo se entiende que esto no puede ser implementado cuando se estén tratando datos personales, contraseñas o ficheros clasificados. Para solucionar esto, se han desarrollado diferentes soluciones, de las cuales se ha llevado a cabo el estudio de Sealed Secrets y Hashicorp Vault.

SealedSecrets

De primera instancia **SealedSecrets** puede verse como la solución más simple. Su trabajo se basa en el empleo de kubeseal que encripta el secreto o Secret en un SealedSecret, el cual solo podrá ser desencriptado por el controlador, el cual se encuentra corriendo en el cluster objetivo.

Comenzando con la instalación del mismo, al terminar esta se crearán 2 llaves. La clave pública, la cual se utilizará para cifrar los secretos y enviarlos al target cluster. La clave privada, esta se emplea para descifrar el secreto cifrado anteriormente y poder usar dichas credenciales (ya únicamente en base64) para acceder a la aplicación que se desee. Este

proceso aparece bien resumido en la Figura 6. Los algoritmos de cifrado empleados son AES-256-GCM, RSA-OAEP con SHA-256 y x509.

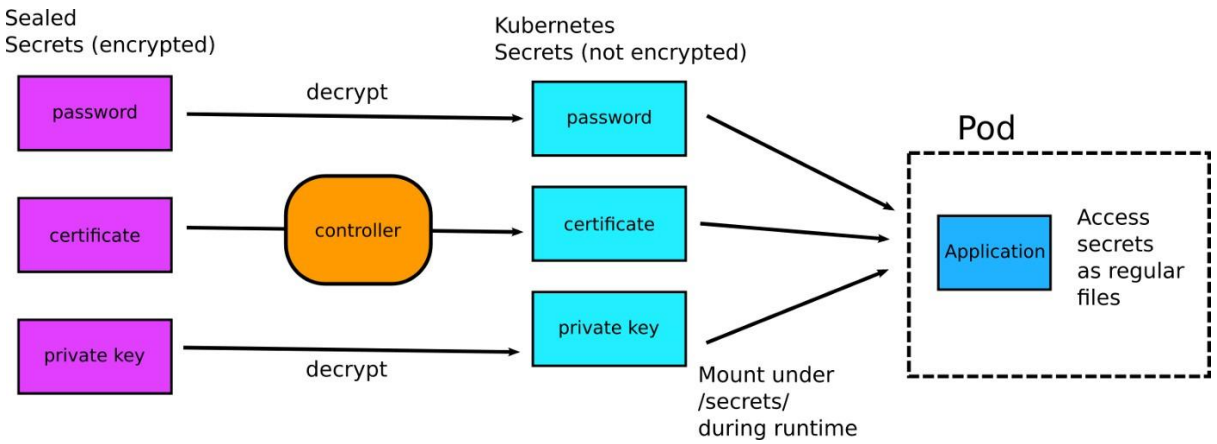


Figura 6. Funcionamiento Sealed Secrets (link Codefresh.io)

De esto se deduce una utilidad extra que aporta gran cantidad de facilidades: subir las credenciales a GitHub. Gracias al hecho de emplear Sealed Secrets se podrán subir dichos archivos al repositorio, y en el momento que el cluster desee emplearlo para acceder a una aplicación solo tendrá que hacer *pull* y desencriptar. Este caso presenta mayor utilidad aún con el empleo de *pipelines*.

Para hacerse una idea de lo sencillo que resulta emplear esta herramienta, teniendo un secreto existente, el único trabajo que habría que realizar para obtener un sealed secret seria el siguiente:

$ kubeseal <secretohackeable.json> secretonohackeable.json

Partiendo de un secreto creado anteriormente secretohackeable.json, se obtendrá en el mismo directorio secretonohackeable.json, el cual aparecerá cifrado, ilegible e indescriptible(en tiempo útil y sin poseer la clave privada).

Hashicorp Vault

Por otra parte se encuentra **Vault** de Hashicorp**,** este presenta una complicación un tanto mayor, además de tener que partir con un conocimiento y abstracción mayor sobre seguridad en lo que respecta a Kubeseal. Sin embargo, ofrece una gran variedad de recursos que facilitan y permiten la operatividad de un gran contexto de aplicaciones.

De manera general Vault consiste en un software que permite almacenar y gestionar secretos, todo ello realizando un control de acceso estricto sobre ellos.
Para entender más eficazmente el funcionamiento de Vault se ha analizado este a alto nivel en el Anexo V Sección II .

Tras examinar ambas herramientas se deduce que tanto SealedSecrets como Vault resuelven uno de los grandes problemas de Kubernetes, que es el acceso a contraseñas,

tokens o datos sensibles sin complicaciones. Ambos cifran la información logrando que esta sea prácticamente inaccesible. Sin embargo, muestran grandes diferencias.

Vault aporta un número de aplicaciones muy superior a SealedSecrets. Mediante su uso se puede autenticar usuarios, auditar accesos, definir políticas, conectar con servicios, emplear distintos motores de secretos y otras muchas, mientras que SealedSecrets únicamente permite encriptar los secretos con el uso de una clave pública y otra privada.

Si que es cierto que puede que para distintos casos no sea necesario todo el útil que expone Vault, y únicamente con el empleo de SealedSecrets sea suficiente. A pesar de ello, Vault es un claro ganador, cumple con la función realizada por SealedSecrets y muchas más, todo ello con una curva de aprendizaje no demasiado alta.

	Sencillez	**Operaciones**	**Plataformas**
Hashicorp Vault	Media	Autenticación, autorización, validación, encriptación de secretos, distintos SecretEngines, auditoría, secretos dinámicos, cifrado como servicio (...)	Múltiples
SealedSecrets	Alta	Encriptación de secretos	Únicamente K8s

Tabla 3. Comparación Hashicorp Vault VS. SealedSecrets

3. Lecciones aprendidas y conclusiones

3.1 Conocimientos adquiridos

En lo que concierne a la elaboración de este proyecto, era imperativo comprender en profundidad una gran cantidad de conceptos relacionados con los sistemas distribuidos y la seguridad, algunos de ellos genéricos y otros más profundos. Junto con esto, al intentar encontrar la mejor solución para cuestiones muy diversas como pueden ser las comunicaciones o el acceso autorizado a determinados datos, se han aprendido gran cantidad de tecnologías y programas diversos, por lo que se ha logrado abarcar una cantidad razonable de herramientas actuales del mercado, así como el conocer otras que también podrían ser de gran utilidad.

3.2 Ideas Futuras

Al tratarse el tema principal de la seguridad, es entendible que este proyecto podría prolongarse prácticamente en la medida que se desease, puesto que la cantidad de conceptos, capas y niveles que podrían ser estudiados son muy extensos. No obstante se plantean varios de ellos que podrían tener especial interés.

Se podría investigar sobre las posibilidades de encriptar etcd con distintos métodos. Etcd es un almacén que contiene todo el estado del cluster, incluyendo los secretos, por lo que el contenido de este debería estar encriptado siempre que sea posible. Existen distintos algoritmos y aplicaciones que podrían realizar esta tarea, decidir cuál sería el más conveniente para cada caso podria ser una buena continuación para este proyecto.

Además se podria incluir el análisis y comprobación de código, tanto estático como dinámico, como se ha enseñado en la asignatura de Seguridad Informática de 4º curso. Pueden existir multitud de vulnerabilidades escritas en el código, como desbordamientos, empleo de funciones no seguras o demás.

Junto con esto sería posible profundizar más en los aspectos ya comentados anteriormente, a pesar de que se ha hecho un análisis bastante completo en cada uno de los flancos, todavía quedan detalles que podrían requerir aplicaciones y contextos de seguridad específicos

3.3 Conclusiones

Se ha llevado a cabo el estudio e implementación de una defensa 360º en Kubernetes, evaluando la seguridad nativa que ofrece K8s y comparando la misma con la presentada por herramientas externas. Se considera que se han cumplido los objetivos de manera satisfactoria ya que ha sido posible dotar de una seguridad consistente a un sistema distribuido basado en Kubernetes por todos los flancos que este presenta; análisis general de la plataforma, aplicacion de politicas de seguridad, empleo de seguridad activa, comunicaciones, análisis de imágenes y gestión de secretos.

A pesar de haber comprendido cada una de estas secciones y haber podido desempeñar en ellas un trabajo que evite un funcionamiento erróneo se considera que la seguridad de estos sistemas abarca mucho más, pudiendo realizar gran cantidad de estudios, como se ha mencionado anteriormente.
No obstante la labor desempeñada resulta de gran utilidad para las distintas partes; el alumno propio, ya que se ha adquirido una abstracción concisa del funcionamiento de Kubernetes y plataformas cloud así como también de los aspectos más importantes en lo que a seguridad concierne; la empresa supervisora, debido a que NTT DATA lleva a cabo abundantes y amplios trabajos relacionados con sistemas distribuidos, microservicios, comunicación y seguridad en sí, el desarrollo y conclusiones de este TFG le permitirán llegar de manera breve a deducciones sobre la cantidad de tiempo dedicada a asegurar un sistema, que herramientas son necesarias aplicar y el por qué de dichos actos; cualquier persona o compañía interesada en plataformas cloud, por las mismas razones que para el caso concreto de NTT DATA, ya que se encontrará en el presente documento de manera clara y breve todo lo necesario para realizar una securización alta de su sistema o únicamente comprender los conceptos clave de la misma.

Con respecto a las dificultades encontradas, no ha existido un gran número de incidencias que hayan imposibilitado el desarrollo del trabajo, sin duda la parte de mayor envergadura fue el conseguir una comprensión competente sobre cómo funciona cada componente en los sistemas cloud, Kubernetes y el desarrollo de cada aplicación a emplear, sin embargo, leyendo gran cantidad de documentación, con paciencia y examinando cada proceso se ha podido realizar sin problemas.

En cuanto a Kubernetes, se trata de un sistema muy potente para la automatización de despliegues, escalado y configuración de contenedores, sin embargo, en lo que respecta a la seguridad deja mucho que desear, presenta graves fallas, tanto en su configuración inicial al levantar cualquier cluster como en lo que a comunicación respecta.

Al ser este un trabajo que compara resultados entre distintas herramientas es necesario llegar a conclusiones sobre cual es mejor que la otra. Sin embargo, desde un punto de vista personal y tras toda la documentación estudiada, se puede concluir que no es posible afirmar categóricamente que hay herramientas mejores que otras de una manera generalista en la mayoría de los casos, debido a que cada una tiene su contexto y espacio de actuación. Por supuesto hay aplicaciones mucho más completas que otras, ofreciendo más funcionalidades, abarcando más sistemas operativos o con una interfaz más simple, pero esto no quiere decir que sean mejor que la solución más sencilla para un caso concreto, ya que si esta se adapta al problema y lo hace empleando menos recursos será imperativo optar por la misma.
Se puede concluir que cada herramienta se adecua a su espacio de trabajo y a la forma en la que esta ha sido contemplada, pueden existir preferencias sobre una u otra, pero solo se definirá una como superior en el contexto de implementación pertinente.

4. Bibliografía

1. Definición de Kubernetes y razones de uso. Escrito por Kubernetes ©.
Fecha última modificación: 17/ 07 / 22
¿Qué es Kubernetes?

2. Explicación cuadrante mágico de Gartner para servicios Cloud. Escrito por Raj Bala, Bob Gill, Dennis Smith, Kevin Ji, David Wright.
Fecha última modificación: 21/ 07 / 21
Cuadrante Gartner

3. Ejemplos teoría de mínimo privilegio. Escrito por Cyberak®.
Principio del Mínimo Privilegio (PoLP)

4. Definición y finalidad del CIS. Escrito por Danilla Kirillov.
Fecha de última modificación: 27 / 01 / 22
Kubernetes cluster security assessment with kube-bench and kube-hunter – Flant blog

5. Kube-hunter, guía de instalación y modos de uso. Escrito por aquasecurity.
Fecha de última modificación: 04 / 07 / 22
GitHub - aquasecurity/kube-hunter: Hunt for security weaknesses in Kubernetes clusters

6. Explicación de vulnerabilidades kube-hunter. Escrito por aquasecurity
Fecha de última modificación: 04 / 07 / 22
Welcome to kube-hunter documentation

7. Definición de Kyverno. Escrito por Kyverno authors ©
Fecha de última modificación: 19 / 02 / 22
Kyverno

8. Web de la Cloud Native Computing Foundation. Escrito por The Linux Foundation®.
Fecha de última modificación: 02 / 11 / 22
Cloud Native Computing Foundation

9. Web oficial de Falco. Escrito por Sysdig®.
Fecha de última modificación: 08 / 08 / 22
https://falco.org/

10. Explicación sobre la monitorización en K8s. Escrito por Kubernetes ©
Fecha de última modificación: 2022
https://kubernetes.io/es/docs/tasks/debug-application-cluster/_print/

11. Documentación sobre Falco Sidekick. Escrito por Thomas Labarussias.
Fecha de última modificación: 02 / 11 / 2022
https://github.com/falcosecurity/falcosidekick

12. Web oficial de Istio. Escrito por Istio Authors.
Fecha de última modificación: 02 / 11 / 2022
https://istio.io/latest/about/service-mesh/

13. Escáner de imágenes Grype. Escrito por Anchor Inc.
Fecha de última modificación: 02 / 11 / 2022
https://github.com/anchore/grype

14. Escáner de imágenes Trivy. Escrito por aquasecurity.
Fecha de última modificación: 31 / 10 / 2022
https://github.com/aquasecurity/trivy

Anexo I

1. Pruebas realizadas con kube-bench

Para comenzar,, en la terminal del host se ejecuta:

$ kubectl apply -f job.yaml

Siendo job.yaml el archivo suministrado por el repositorio en GitHub de Kube-Bench que contiene todo lo necesario para pasar los test deseados.
(link kube-bench/job.yaml at main)

A continuación se obtienen los PODS existentes en el cluster para leer lo creado por kube-bench, y por último, con el nombre del pod obtenido se muestran los logs con el comando

$ kubectl logs kube-bench-dksgc

Lo que inmediatamente mostrará todas las pruebas realizadas y la categorización de su resultado. Similar a:

[INFO] 1 Control Plane Security Configuration
[INFO] 1.1 Control Plane Node Configuration Files
[PASS] 1.1.1 Ensure that the API server pod specification file permissions are set to 644 or more restrictive (Automated)
[PASS] 1.1.2 Ensure that the API server pod specification file ownership is set to root:root (Automated)

Así entonces cada mensaje se explica cómo <Categoría> <Identificador> <Descripción>
Siendo las categorías
- PASS: Cuando el test se ha superado correctamente
- WARN: Avisa de posibles inseguridades que son necesarias comprobar manualmente
- FAIL: Informa de test que no han sido superados correctamente
- INFO: Empleado para seleccionar los test llevados a cabo

Tras visualizar todos estos avisos se muestra en los logs las opciones para revisar y solucionar cada uno de los problemas encontrados, lo cual puede resultar francamente útil.
Kube-bench muestra una gran cantidad de pruebas realizadas, para lograr una idea de su funcionamiento se explicaran algunas de ellas y su proceso de remediación.

El primer mensaje que con el término FAIL al comienzo dice lo siguiente:
[FAIL] 1.1.11 Ensure that the etcd data directory permissions are set to 700 or more restrictive (Automated)

etcd es un almacén de valores clave de alta disponibilidad utilizado por las implementaciones de Kubernetes para el almacenamiento persistente de todos sus objetos de la API REST. Este directorio de datos debe estar protegido contra cualquier lectura o

escritura no autorizada. No debe ser legible ni escribible por ningún miembro del grupo ni por el mundo. (link etcd data directory permissions are set to 700 or more restrictive)

Para solucionarlo kube-bench dice
1.1.11 On the etcd server node, get the etcd data directory, passed as an argument --data-dir, from the command 'ps -ef | grep etcd'.
Run the below command (based on the etcd data directory found above). For example, chmod 700 /var/lib/etcd

Con el objetivo de solucionar el problema se accede al cluster de minikube ejecutando el siguiente comando

$ minikube ssh

Con el comando mencionado anteriormente por kube-bench, la ubicación del directorio pertinente:
/var/lib/minikube/etcd

Ejecutando con sudo el comando chmod previo, serán modificados sus permisos. Una vez realizado esto se comprueba los permisos de dicho directorio con *ls -l* verificando que son los deseados. Si se desea se puede volver a pasar kube-bench y comprobar que en esta ocasión la categoría del mensaje será *PASS*.

Finalmente se muestra un recuento de todo lo ocurrido, indicando la cantidad total de test corridos y el resultado que ha devuelto el sistema con estos.
Como se ha podido observar es relativamente fácil solucionar las vulnerabilidades que kube-bench muestra. Presenta una gran infraestructura con apoyo detrás que logra una buena experiencia al usuario.

Entre otros de los mensajes que aparecen es posible destacar

[FAIL] 1.1.19 Ensure that the Kubernetes PKI directory and file ownership is set to root:root (Automated)

Esto tiene vital importancia ya que Kubernetes emplea certificados PKI para la autenticación mediante TLS, muchos de ellos generados automáticamente y almacenados en /etc/kubernetes/pki/. La propiedad de todos los archivos y directorios debe estar establecida para root:root (link PKI certificates and requirements | Kubernetes, link Kubernetes PKI directory and file ownership is set to root:root). Para solucionar esto únicamente se ejecutará
$ chown -R root:root /etc/kubernetes/pki/

[FAIL] 1.2.19 Ensure that the --audit-log-path argument is set (Automated)
Para el correcto funcionamiento y la detección de Kubernetes es necesario saber lo que está ocurriendo en el sistema, por lo que es necesaria la existencia de una auditoría mínima. Si este parámetro no está habilitado seguía imposible desempeñar un seguimiento. Si se ejecuta el comando se comprobará el estado de la variable

$ ps -ef | grep kube-apiserver

Con el objetivo de solucionar este problema, en caso de que exista, se establecerá el parámetro –audit-log-path de /etc/kubernetes/manifests/kube-apiserver.yaml en una ruta y un archivo válidos.

2. Pruebas realizadas con Kube-Hunter

La primera de todas las vulnerabilidades tras ejecutarlo es observable en la figura 7
"CAP_NET_RAW Enabled" in Local to Pod (kube-hunter-lrzzz)

ID	LOCATION	MITRE CATEGORY	VULNERABILITY	DESCRIPTION	EVIDENCE
None	Local to Pod (kube-hunter-lrzzz)	Lateral Movement // ARP poisoning and IP spoofing	CAP_NET_RAW Enabled	CAP_NET_RAW is enabled by default for pods. If an attacker manages to compromise a pod, they could potentially take advantage of this capability to perform network attacks on other pods running on the same node	

Figura 7. Vulnerabilidad mostrada por Kube-Hunter

La información que obtiene de dicho fallo contiene las siguientes secciones:
- Localización: En este caso es el propio POD de Kube-Hunter
- Categoría: El daño que podría ocurrir si esta permanece
- Vulnerabilidad: Su nombre
- Descripción: En qué consiste
- Evidence: Las pruebas que lo refutan, en este caso no aparecen.

Esta vulnerabilidad consiste en que con el parámetro CAP_NET_RAW está activado, lo cual ocurre por defecto, un usuario malintencionado podría atacar otros pods dentro del mismo nodo ya que permite que los procesos falsifiquen cualquier tipo de paquete o se vinculen a cualquier dirección.
(link Mitigating CVE-2020-10749 in Kubernetes Environments | StackRox Community)

Para resolver este problema se emplea la herramienta Kyverno, que mediante la aplicación de políticas permite que en ciertos ambientes como PODS se cumplan unas determinadas reglas, lo que será explicado más tarde.

Así entonces ejecutando un archivo yaml (link Drop CAP_NET_RAW | Kyverno) con la previa instalación de Kyverno evitará que esta opción se active nuevamente.

Otra de las fallas de importancia es la relativa a los accesos de cuentas de servicio, observables en la figura 8.

Figura 8. Vulnerabilidad mostrada por Kube-Hunter

Cada pod tiene asociada una cuenta de servicio, la cual de manera preestablecida tiene acceso a la API de K8s, dicho acceso se puede realizar a partir de un token. Por lo tanto se comprende que si se tiene acceso a dicho token es posible acceder a la API, que es precisamente la información que ofrece la imagen anterior en el fallo KHV050.
Esto tendría terribles consecuencias ya que la API es la base de configuración de Kubernetes.
Para solucionar esto se recomienda tener una cuenta de servicio para cada workload y mantener el principio de mínimo privilegio, comentado anteriormente (link KHV050 - Read access to Pod service account token) .

3. Pruebas realizadas con Kyverno

De primer modo se ve el funcionamiento de una **política tipo Allow/Deny** (deny en este caso). El nombre de esta *policy* en Kyverno es allowed-label-changes, esta impide que se modifiquen etiquetas que no estén con la key *breakglass*. En algunas operaciones es necesario impedir la modificación de determinados recursos por cuestiones específicas, uno de ellos pueden ser las labels.

Seguidamente se ha creado un sencillo pod cuya tarea es comportarse como un servidor de tipo nginx, uno de los más empleados actualmente en el sector web.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
     name: nginxCambioImposible
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Una vez aplicado se pueden comprobar sus características a estudiar, en este caso lo que interesa es la etiqueta
$ kubectl get pod nginx --show-labels

```
NAME    READY   STATUS    RESTARTS   AGE   LABELS
nginx   1/1     Running   0          84s   name=nginx
```

Se observa que esta es nginx, como ahora no hay ninguna política establecida, se puede modificar a su gusto.

No obstante, una vez instaladas las políticas mediante el comando
$ kubectl apply -f allowed-label-changes.yaml
clusterpolicy.kyverno.io/allowed-label-changes created

Será imposible modificar ninguna de estas, intentándolo aparece un mensaje similar al siguiente indicando que existe una política que lo impide
$ kubectl apply -f pod-label.yaml

Error from server: error when applying patch:
{"metadata":{"annotations":{"kubectl.kubernetes.io/last-applied-configuration":"{\"apiVersion\":\"v1\",\"kind\":\"Pod\",\"metadata\":{\"annotations\":{},\"labels\":{\"name\":\"nginxCambioImposible\"},\"name\":\"nginx\",\"namespace\":\"default\"},\"spec\":{\"containers\":[{\"image\":\"nginx\",\"name\":\"nginx\",\"ports\":[{\"containerPort\":80}]}]}}\n"},"labels":{"name":"nginxCambioImposible"}}}
to:
Resource: "/v1, Resource=pods", GroupVersionKind: "/v1, Kind=Pod"
Name: "nginx", Namespace: "default"
for: "pod-label.yaml": admission webhook "validate.kyverno.svc-fail" denied the request:

resource Pod/default/nginx was blocked due to the following policies
allowed-label-changes:
  safe-label: The only label that may be removed or changed is `breakglass`.

En caso de querer revocar la política pertinente únicamente habrá que realizar el comando posterior
$ kubectl -f <policy-name>.yaml
Como se atiende, el implementar políticas con Kyverno es cuanto menos sencillo.

Para observar un ejemplo de política empleada en la **mutación** de recursos se va a comentar la llamada *mutate-large-termination-gps*. Esta muta todos los PODS entrantes para que tengan un periodo de gracia en segundos o tGPS como máximo de 50 segundos. Esto se hace para evitar que los nodos "se agoten", si existen numerosos PODS en un estado de terminación y ninguno de ellos sale puede acabar provocando un cluster inestable. No obstante el tGPS puede ser escogido por el usuario editando el fichero mutate-large-termination-gps.yaml.

Dentro del fichero mutate-large-termination-gps.yaml se encuentra

```
    match:
      resources:
        kinds:
          - Pod
    preconditions:
      all:
```

```
    - key: "{{request.object.spec.terminationGracePeriodSeconds || `0` }}"
      operator: GreaterThan
      value: 50   # maximum tGPS allowed by cluster admin
  mutate:
    patchStrategicMerge:
      spec:
        terminationGracePeriodSeconds: 50
```

Se observa que en la especificación de los pods, la calve/valor request.object.spec.terminationGracePeriodSeconds/<value>, establece como máximo 50 el periodo de gracia, y para aquellos que lo sobrepase, mutará dicho valor a 50. Cabe destacar que este fichero contiene más información necesaria para la aplicación de la política.

En lo relativo a la utilización de Kyverno, en lo concerniente a imágenes, una utilidad de gran importancia es la **verificación de imágenes**. Desde hace un tiempo es posible firmar imágenes con el objetivo de garantizar la seguridad a lo largo de toda la cadena de desarrollo y producción, asegurando la autoría de la imagen. Dichas firmas pueden ser comprobadas antes de ser introducidas en un clúster. En la siguiente política (link Verify Image | Kyverno) se verifica que exista una firma correcta proveniente de un directorio determinado, con el objetivo de comprobar si se ha realizado con su clave pública proporcionada. Es posible modificar el directorio y la clave para la utilización de unos personales.

```
rules:
  - name: verify-image
    match:
      any:
      - resources:
          kinds:
            - Pod
    verifyImages:
      - image: "ghcr.io/kyverno/test-verify-image:*"
        key: |-
          -----BEGIN PUBLIC KEY-----
          MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE8nXRh950IZbRj8Ra/N9sbqOPZrfM
          5/KAQN0/KjHcorm/J5yctVd7iEcnessRQjU917hmKO6JWVGHpDguIyakZA==
          -----END PUBLIC KEY-----
```

Para enseñar el funcionamiento de una política relacionada con el aspecto de “**generar**”, se va poner un caso de uso tan importante como los backups o copias de seguridad. Imagínese una empresa que tiene un único despliegue las facturas pendientes de todos sus clientes. Si este se pierde por algún motivo no podría recuperar la información y por tanto estaría expuesto a perder grandes cantidades de dinero por reclamar. Se podría establecer una política que creará por defecto una de estas copias de seguridad, por ejemplo al observar una determinada etiqueta (link Generate Gold Backup Policy | Kyverno). Aquí se

pueden realizar distintas configuraciones como la frecuencia con la que se realiza dicha copia de seguridad, durante cuánto tiempo se retiene etc.

4. Pruebas realizadas con OPA Gatekeeper

A diferencia de Kyverno, son necesarios dos archivos para aplicar OPA en su Kubernetes. En primer lugar es necesario definir el archivo *ConstraintTemplate*, que se ve con el siguiente ejemplo, proporcionado por OPA (link How to use Gatekeeper):

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8srequiredlabels
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredLabels
      validation:
        # Schema for the `parameters` field
        openAPIV3Schema:
          type: object
          properties:
            labels:
              type: array
              items:
                type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8srequiredlabels

        violation[{"msg": msg, "details": {"missing_labels": missing}}] {
          provided := {label | input.review.object.metadata.labels[label]}
          required := {label | label := input.parameters.labels[_]}
          missing := required - provided
          count(missing) > 0
          msg := sprintf("you must provide labels: %v", [missing])
        }
```

Al comienzo de este fichero se definen los parámetros que van a ser usados en las políticas, en *properties* aparece escrito *labels*, ya que esta política concreta revisará que se tengan unas etiquetas mínimas.
Seguidamente, entre más datos, aparece la descripción de la política implementada por Rego. En ella aparecen varios campos de interés:

- *Provided*: Son las etiquetas escritas en los recursos a analizar

- *Required*: Son las etiquetas mínimas que se espera que tengan los recursos a analizar
- *Missing*: Es la diferencia existente entre las etiquetas requeridas y las dadas.

A continuación se hace una comparación lógica que si se cumple se mostrará un mensaje junto con el impedimento de la acción que se quiera realizar.

Previo a aplicar esta política se debe informar a OPA Gatekeeper de que se desea aplicar una ConstraintTemplate, definiendo cuál de estas es, que se va a usar como comparación, y donde se va a realizar la búsqueda

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: ns-must-have-gk
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Namespace"]
  parameters:
    labels: ["gatekeeper"]
```

Así entonces continuando con el ejemplo anterior, se indica a Gatekeeper que se va a aplicar la política, K8sRequiredLabels logrando que en todos los espacios de nombre aparezca la etiqueta gatekeeper.

Anexo II

1. Comparación de políticas de K8s y Falco

La siguiente política de K8s (link Configure Minimum and Maximum CPU Constraints for a Namespace | Kubernetes) evitará crear pods cuyo uso de CPU se encuentre fuera de un rango determinado:

```
apiVersion: v1
kind: LimitRange
metadata:
 name: cpu-min-max-demo-lr
spec:
 limits:
 - max:
    cpu: "800m"
  min:
    cpu: "200m"
  type: Container
```

Respecto a las políticas nativas de Falco, están compuestas por los siguientes elementos

Rules: Situación en la cual se deberá generar una alerta acompañada de una descripción de la misma en la salida.
Ejemplo (link Rules | Falco):

```
rule: shell_in_container
        desc: notice shell activity within a container
        condition: evt.type = execve and evt.dir=< and container.id != host and proc.name =
        bash
        output: shell in a container (user=%user.name container_id=%container.id
        container_name=%container.name shell=%proc.name parent=%proc.pname
        cmdline=%proc.cmdline)
        priority: WARNING
```

Esta regla avisa de cuando se está ejecutando un BASH dentro de un contenedor. Esto tiene gran importancia debido a que mediante este se podrían ejecutar comandos que lograra una escalada de privilegios, obtener información a la que no se debería tener acceso etc

Macros: Son reglas o fragmentos de las mismas que pueden ser utilizadas dentro de otras reglas o macros. Estas se emplean cuando aparecen patrones comunes a detectar.

```
macro: open_write
  condition: (evt.type=open or evt.type=openat) and evt.is_open_write=true and
fd.typechar='f' and fd.num>=0
```

La macro anterior (link Default Macros | Falco) sirve para identificar cuando un archivo está abierto, lo que se podría emplear para numerosos casos, por ejemplo, para detectar si está abierto el fichero /etc/passwd que contiene información sobre las cuentas de usuarios del sistema.

Listas: Colecciones de elementos incluibles en reglas, macros u otras listas.
Por ejemplo, la siguiente lista sería una colección de los distintos binarios de shell (link Default Macros | Falco).

list: shell_binaries
items: [bash, csh, ksh, sh, tcsh, zsh, dash]

2. Ejemplo de ejecución Falco

Uno de los casos de interés sería para activar la salida por programa, para la cual sería necesario utilizar la siguiente conformación:

program_output:
  enabled: true
  keep_alive: true
  program: "<programa a emplear>"

Si keep_alive ha sido establecido a true el programa será iniciado una vez y escrito de manera continua, en caso contrario se reiniciará con cada mensaje.

Si se ejecutase Falco, con las salidas de stdout y programas activadas (por ejemplo un servicio web), tras esperar un breve inciso de tiempo aparecerán por terminal todos aquellos errores encontrados además de un resumen bastante indicativo:

$ falco
vents detected: 39
Rule counts by severity:
   ERROR: 22
   WARNING: 6
   NOTICE: 9
   DEBUG: 2
Triggered rules by rule name:
   Write below binary dir: 3
   Write below etc: 2
   Read sensitive file trusted after startup: 2
   Read sensitive file untrusted: 4
   Write below rpm database: 2
   DB program spawned process: 3
   Modify binary dirs: 11
   Mkdir binary dirs: 2
   Run shell untrusted: 2
   System procs network activity: 3
   Non sudo setuid: 3

Create files below dev: 2
Syscall event drop monitoring:
- event drop detected: 0 occurrences
- num times actions taken: 0

De la información arrojada se alcanza a distinguir información importante, como el total de infracciones encontradas categorizadas por niveles o los tipos de reglas que han sido ejecutadas.

La información pertinente obtenida en el programa se observa en la Figura 9.


```
{
  "took" : 11,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 6,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "falco",
        "_type" : "_doc",
        "_id" : "_NofOH0B3mCkJCNTxIu6",
        "_score" : 1.0,
        "_ignored" : [
          "output.keyword"
        ],
        "_source" : {
          "output" : "12:15:45.475098855: Warning Sensitive file opened for reading by non-trusted program
(user=root user_loginuid=-1 program=httpd command=httpd --loglevel info run
^syscall.ReadSensitiveFileUntrusted$ --sleep 6s file=/etc/shadow parent=event-generator gparent=containerd-shim
ggparent=systemd gggparent=<NA> container_id=59b596002e65 image=falcosecurity/event-generator)",
          "priority" : "Warning",
          "rule" : "Read sensitive file untrusted",
          "source" : "syscall",
          "tags" : [
            "filesystem",
            "mitre_credential_access",
            "mitre_discovery"
          ],
          "time" : "2021-11-19T12:15:45.475098855Z",
          "output_fields" : {
            "container.id" : "59b596002e65",
            "container.image.repository" : "falcosecurity/event-generator",
            "evt.time" : 1637324145475098855,
            "fd.name" : "/etc/shadow",
            "proc.aname[2]" : "containerd-shim",
            "proc.aname[3]" : "systemd",
            "proc.aname[4]" : null,
            "proc.cmdline" : "httpd --loglevel info run ^syscall.ReadSensitiveFileUntrusted$ --sleep 6s",
            "proc.name" : "httpd",
            "proc.pname" : "event-generator",
            "user.loginuid" : -1,
            "user.name" : "root"
          }
        }
      },
```


Figura 9. Información en formato JSON obtenida en el programa

Aquí aparecen también detalles de gran valor como la regla que ha detectado la falla, la prioridad de esta, de donde venia el fallo, cuando ha ocurrido etc.

Anexo III

1. Ejemplo de la importancia en comunicaciones

Imagínese un usuario que realiza una compra online, en el momento en el que se realiza el usuario únicamente ve un alerta confirmando su pedido, pero detrás de todo el proceso sus distintas peticiones habrán pasado por multitud de servicios diferentes comunicándose unos con otros, como pueden ser un webserver, un servicio de pagos, el carrito de compra, un gestor de inventario, la base de datos y otros muchos.
De añadido, lo habitual es que la empresa pertinente desee obtener mayores beneficios, añadiendo más funciones que le hagan vender más, cómo podría ser una sección de productos relacionados, lo cual hará introducir nuevos servicios que complican el trasfondo de la aplicación.
Cada uno de estos servicios requiere una configuración y una lógica de comunicaciones que permitan su correcto funcionamiento, así como el realizar estudios, obtención de métricas y otras practicidades.

2. Ejemplo de ejecución empleando Istio

Isito ofrece en su web un ejemplo muy claro de funcionamiento en el que guía paso a paso mientras enseña las distintas opciones aplicables al sistema (link Istio / Getting Started). En primer lugar es necesario instalar Istio e Ioctl (comando para interactuar con Istio) en el host, en este caso en particular será en local. Este proceso no resulta muy complicado.
Seguidamente se hará deployment de la aplicación de ejemplo Bookinfo (link Istio Bookinfo), al igual que con cualquier otro caso ejecutando

$ kubecetl apply -f samples/bookinfo/platform/kube/bookinfo.yaml

Comprobando como está distribuida la aplicación se observan sus PODS y servicios, los cuales serán explicados más adelante

$ kubectl get pods

```
NAME                          READY   STATUS    RESTARTS   AGE
details-v1-7f4669bdd9-l8lgf   2/2     Running   0          176m
productpage-v1-5586c4d4ff-mdkhq   2/2     Running   0          176m
ratings-v1-6cf6bc7c85-45648   2/2     Running   0          176m
reviews-v1-7598cc9867-s6sg6   2/2     Running   0          176m
reviews-v2-6bdd859457-m4pqt   2/2     Running   0          176m
reviews-v3-6c98f9d7d7-nfpt8   2/2     Running   0          176m
```

Figura 10. Pods de Bookinfo

$ kubectl get services

```
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
details       ClusterIP   10.96.128.22    <none>        9080/TCP   177m
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP    179m
productpage   ClusterIP   10.97.106.85    <none>        9080/TCP   177m
ratings       ClusterIP   10.107.88.160   <none>        9080/TCP   177m
reviews       ClusterIP   10.111.71.127   <none>        9080/TCP   177m
```

Figura 11. Servicios de Bookinfo

Para hacer accesible desde el exterior Bookinfo se instalará Istio Ingress Gateway, encargado de mapear cada ruta en el borde de la malla. Consiste en un load-balancer situado en el límite de la servicemesh recibiendo continuamente peticiones HTTP/TCP. Este trabaja de manera muy similar a Kubernetes Ingress, pero ofreciendo más personalización y flexibilidad. (Link Istio / Ingress Gateways). Para ello es necesario ejecutar
$kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml.

Por último, se abrirá un acceso para que Minikube envíe su tráfico al Istio Ingress Gateway mediante
$ minikube tunnel

Tras asegurarse de que se han asignado correctamente las direcciones IP y puertos, ya es posible trabajar en Bookinfo. En la Figura 12 se observa la que es la página de inicio de la aplicación.

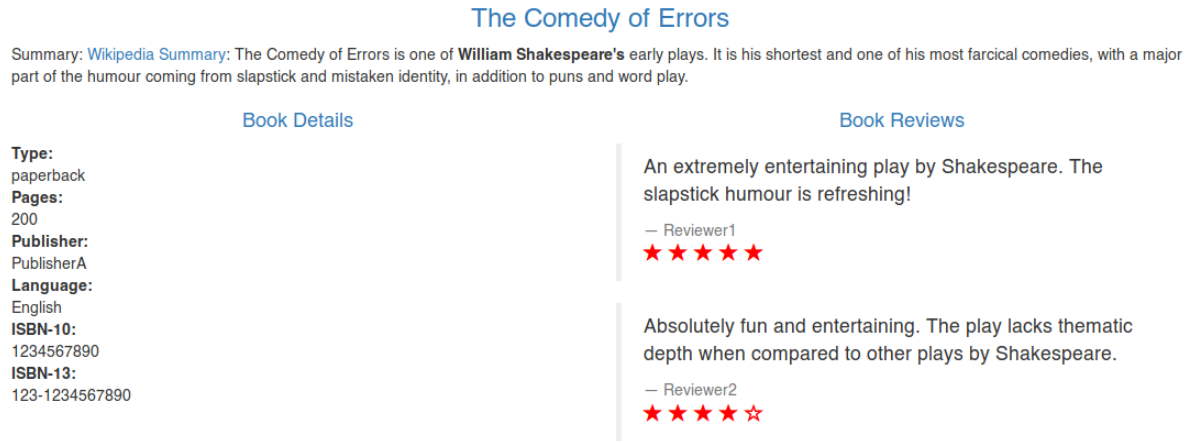


Figura 12. Página principal de Bookinfo

Para explotar toda la operativa que ofrece Istio se instalarán 4 *addons* o complementos Kiali, Prometheus, Grafana y Jaeger, mediante los cuales se llevará a cabo un seguimiento del despliegue logrando que el operador sea consciente de cómo está trabajando cada elemento.

En primer lugar, Kiali, es una consola de observabilidad con capacidades para validar y configurar parámetros en la malla de servicios. Kiali ayuda a comprender la estructura y el estado de la red de servicios al monitorear el flujo de tráfico para inferir la topología e informar errores (link Istio / Kiali). Empleando Grafana y Jaeger proporciona métricas detalladas y monitoreo de tráfico.

Así entonces en la siguiente Figura 13 se observa la estructura de Bookinfo

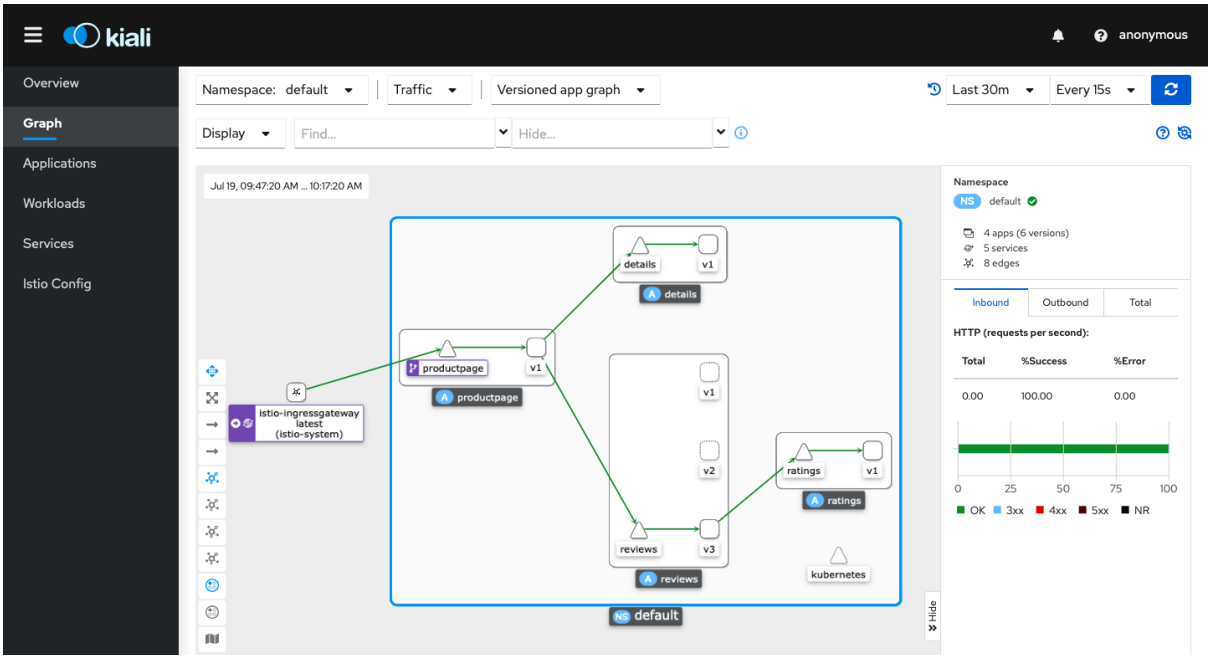


Figura 13. Despliegue de Bookinfo observado en Kiali

Contemplando detalladamente a la izquierda del esquema aparece el Istio Ingress Gateway como entrada hacia la aplicación. Este se conecta directamente con la página principal productpages, a través de la cual se accede al apartado de details y a los reviews. Este servicio tiene disponibles tres versiones v1, v2 y v3. Actualmente el tráfico pasa a través de la versión v3.

Aquí se comprobará una de las características que hacen de Istio una herramienta tan atractiva. Únicamente mediante la aplicación, como se ha visto anteriormente, de un fichero YAML es posible redistribuir el tráfico a gusto del operador, pudiendo enviar peticiones concretas a una versión u otra, esto tiene grandes ventajas ya que si, por ejemplo, una versión desarrollada comienza a dar problemas se puede volver rápidamente a una anterior sin causar una caída alargada del servicio. para probarlo, se desviará todo el tráfico de la v3 a la v1, a la vez que las peticiones a secciones concretas irán a v2 (link Istio / Virtual Service Istio / Virtual Service). Para ver el tráfico se realizan peticiones desde el host a la dirección deseada empleando curl:

$ for i in $(seq 1 100); do curl -s -o /dev/null "http://$GATEWAY_URL/productpage"; done

Observando el tráfico en el último minuto (figura 14) se observa como en la zona de reviews todas las peticiones pasan ahora a v1

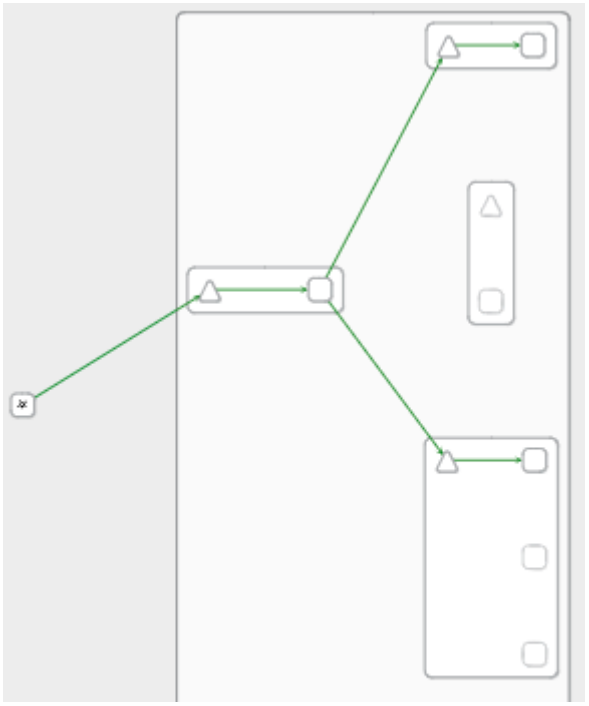


Figura 14. Tráfico de Bookinfo redirigido a v1

Aunque esto es de gran utilidad Istio va más lejos, ya que da la posibilidad de repartir dichas peticiones de manera ponderada. Para entender la funcionalidad de esto se propondrá un ejemplo.Imagínese un servicio de streaming que posee una versión inicial 1.0, para que el usuario tenga una mejor experiencia han desarrollado una versión 2.0, sin embargo, a pesar de haberla testeado correctamente no tienen la certeza de que se comporte a la perfección en producción, así que para ello han decidido presentarla y pasarle únicamente un 25% del tráfico real, mientras que el otro 75% lo mantienen en la versión 1.0. En el momento de que hayan comprobado en su totalidad que trabaja de manera óptima se migrará al 100%.

Además de este esquema en el dashboard de Kiali se pueden observar cada componente así como diversas métricas relacionadas con estos, en la Figura 15 se ven las peticiones dirigidas al servicio Reviews en los últimos 10 minutos cada 15 segundos.

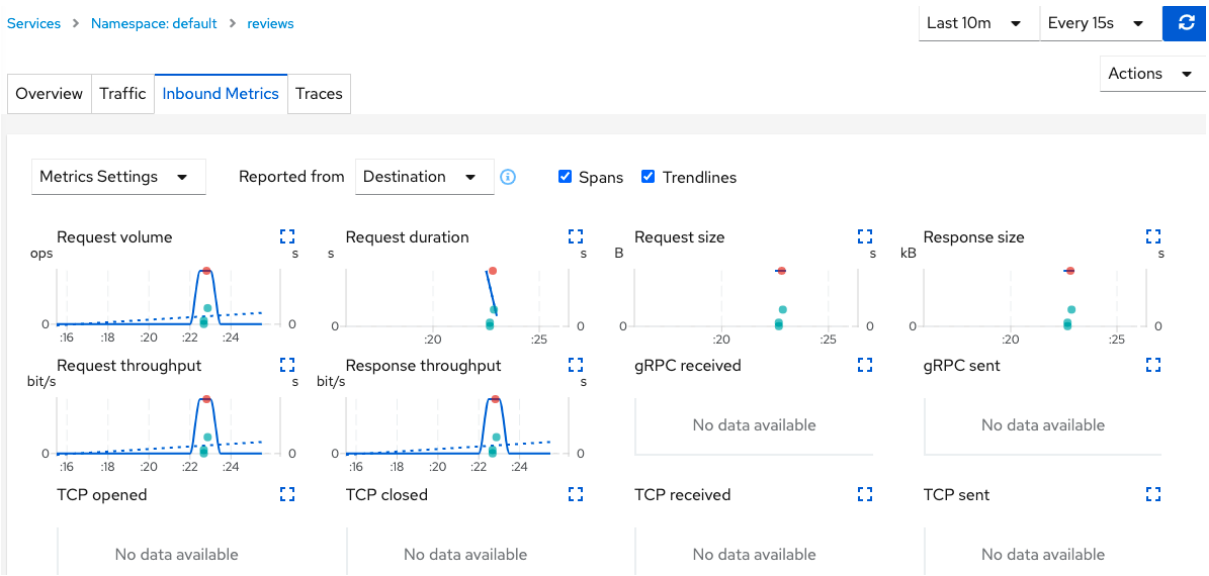


Figura 15. Graficas del servicio reviews

A pesar de estar integradas de cierta manera en Kiali, los dashboard de Prometheus y Graphite posibilitan el realizar acciones más específicas. Por ejemplo en Prometheus se pueden realizar *queries* complejas y observar las peticiones que se han realizado empleando un parámetro concreto, así como su sus gráficas, como se observa en la Figura 15, en la que se muestra una query realizada a un servicio concreto con una versión concreta. Se podría concretar mucho más como detallar tiempos, clientes etc.

Figura 15. Peticiones realizadas al servicio reviews con versión v3 mediante Prometheus

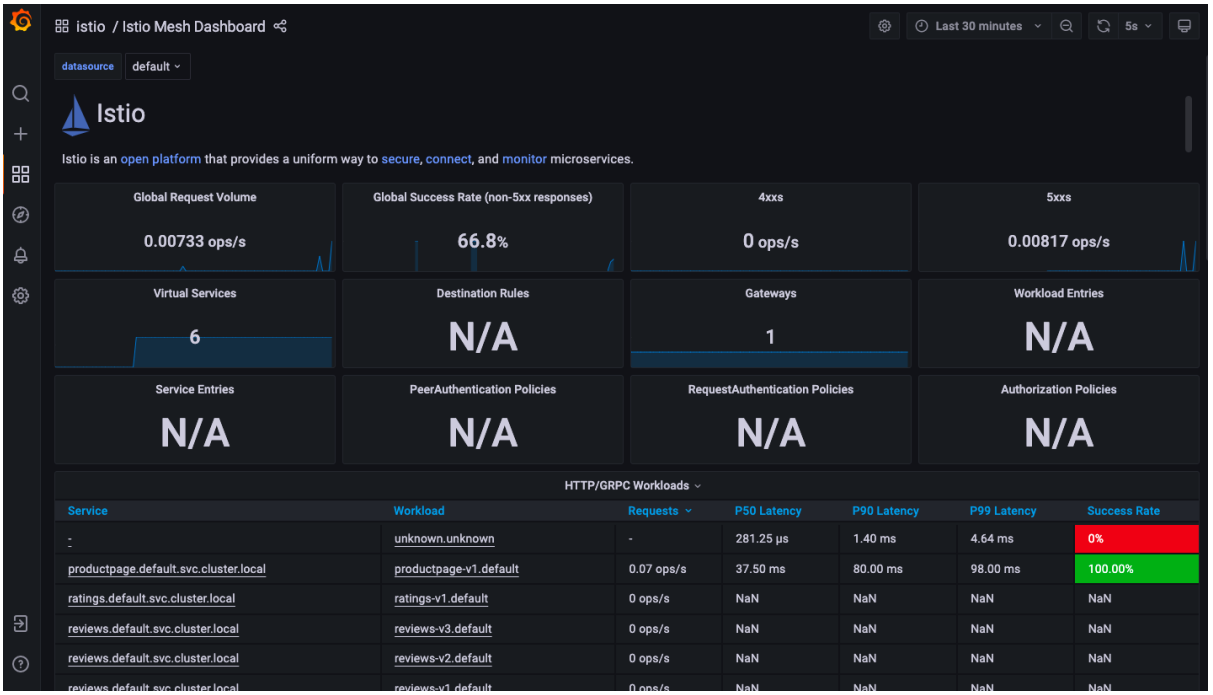


Figura 16. Dashboard de Graphana

Finalmente, otra cualidad importante que permite Istio es el seguir las peticiones a través de la red, también conocido como *tracing system*.

Para ello se emplea la herramienta Jaeger mediante la cual se puede seguir todo el recorrido de una request a través de los diferentes servicios. Este a su vez muestra información de interés como la profundidad de la petición, duración o comienzo y fin de la petición.
En la figura 17 aparece una petición hecha al servicio productpage. Explicando la operación de esta, como es normal comienza en el Istio Ingress Gateway para luego entrar en productpage.default. A su vez se realizan dos peticiones a details y reviews ya que la información de estas se encuentra integrada en productpage como se ha visto anteriormente..

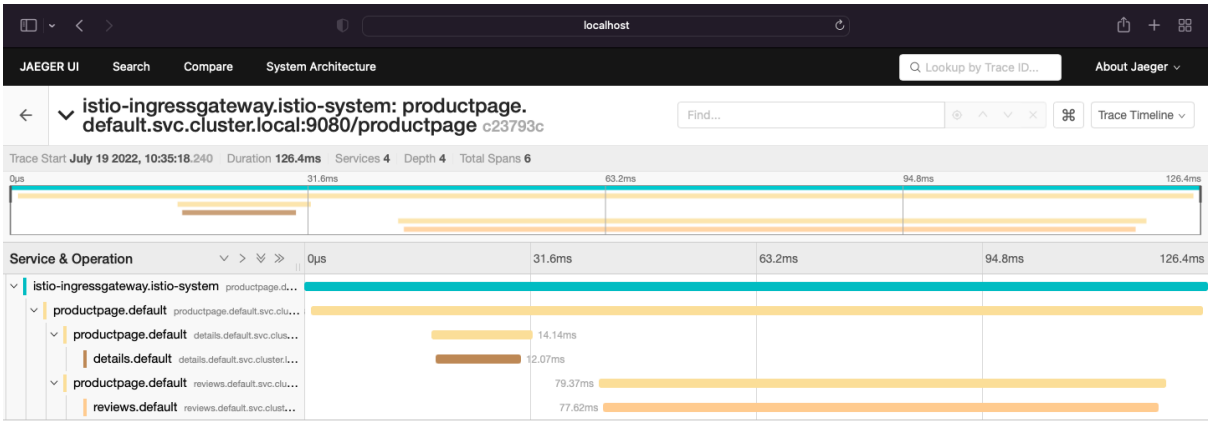


Figura 17. Seguimiento de petición a *productpage* en Jaeger

3. Aplicación de Kubernetes Network Policies

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: hollow-db-policy   #policy name
spec:
  podSelector:
    matchLabels:
      app: hollowdb  #pod applied to
  policyTypes:
  - Ingress #Ingress and/or Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: hollowapp #pod allowed
    ports:
    - protocol: TCP
      port: 3306 #port allowed
```

En este ejemplo, el cual es uno de los casos más comunes, se permite únicamente el tráfico a unos pods con MySQL desde otros que estén realizando la ejecución de una API. Es decir, únicamente los pods ejecutando la API, podrán acceder a la base de datos que explotan.

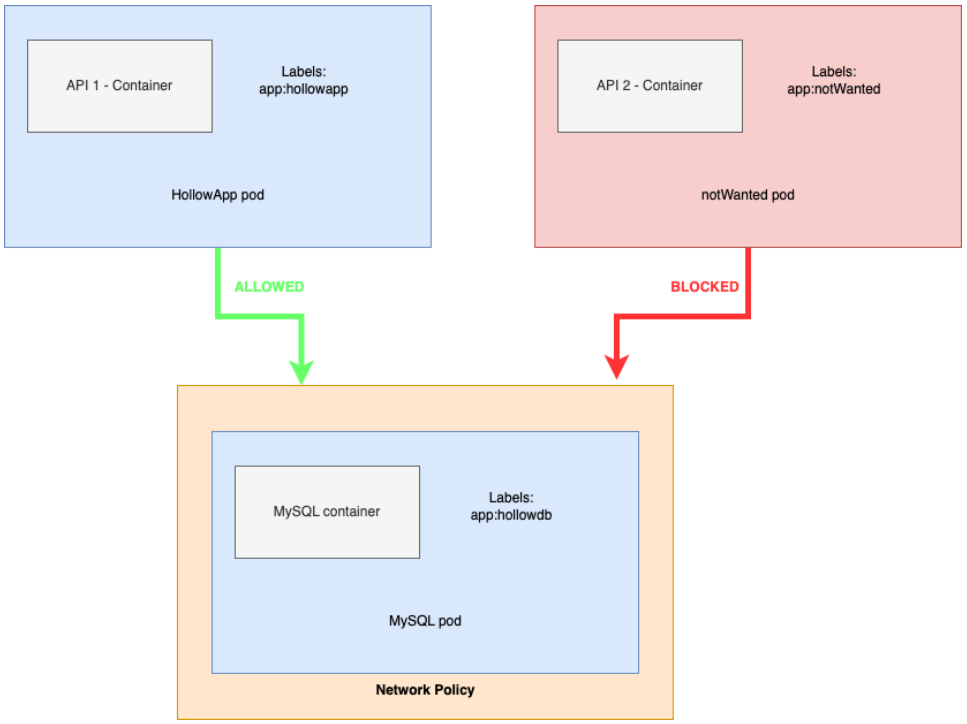


Figura 18. Bloqueo de uso de base de datos mediante el empleo de políticas

En la imagen anterior, se observa cómo se bloquea el tráfico proveniente del pod notWatned por no disponer de la etiqueta hollowapp, ya que se ha establecido en el yaml superior que solo estos podrán acceder a los pods de tipo hollowdb.

Anexo IV

1. Análisis de imagen con Trivy

En primer lugar el pod escogido es
reviews-v3-6c98f9d7d7-swmv4 y su imagen, situada en Docker
http://docker.io/istio/examples-bookinfo-reviews-v3:1.16.4.

Generalmente el uso de Trivy es cuanto menos sencillo, el comando a ejecutar sigue el siguiente patrón

$ trivy <target> [--security-checks <scanner1,scanner2>] TARGET_NAME

Dónde target es el recurso a analizar, por ejemplo si es una imagen se utilizará "image", --security-checks son especificaciones que se pueden usar si se desean para casos más concretos y finalmente TARGET_NAME el nombre del recurso, particularmente el nombre escrito anteriormente, por lo que quedaria asi el comienzo del análisis

$ trivy image http://docker.io/istio/examples-bookinfo-reviews-v3:1.16.4

Tras ejecutar el comando, y procesar la información, aparece un recuento de todas las vulnerabilidades encontradas y a continuación una tabla con toda la información útil, observable en la Figura 19.

Library	Vulnerability	Severity	Installed Version	Fixed Version	Title
apt	CVE-2020-27350	MEDIUM	1.8.2.1	1.8.2.2	apt: integer overflows and underflows while parsing .deb packages https://avd.aquasec.com/nvd/cve-2020-27350
	CVE-2011-3374	LOW			It was found that apt-key in apt, all versions, do not correctly... https://avd.aquasec.com/nvd/cve-2011-3374

Figura 19. Resultados obtenidos al aplicar Trivy

Como se puede observar, leyendo de izquierda a derecha, especifica el nombre de la librería que se trata, un código de vulnerabilidad (o varios) estándar relativo al Common Vulnerabilities and Exposures, la importancia de la brecha encontrada, las versiones, tanto la instalada como en la que aparece solucionada dicha falla (si existe). Finalmente un pequeño título o descripción junto con un enlace a la web de aqua security donde aportan más información sobre el error.

2. Análisis de imagen con Grype

Para comenzar el análisis de imágenes con grype, es necesario ejecutar

$ grype <image>

Siendo image la imagen que se desea analizar, en caso de existir más de una con el mismo nombre será necesario especificar su etiqueta para diferenciar(comprobar)
Cada uno de los paquetes escaneados que contengan la imagen especificada serán listados con los siguientes campos:

- Name: Nombre del paquete
- Installed: Version de la imagen
- Fixed-In: Version en la que la vulnerabilidad se ha arreglado
- Type: Tipo del paquete
- Vulnerability: El listado del CVE (Common Vulnerabilities and Exposures) de cada vulnerabilidad.
- Severity: La severidad de la vulnerabilidad

Además Grype ofrece especificaciones de filtrado para enfocar el escáner hacia donde desee el usuario.

Como se puede comprobar en la Figura 20 la información que aporta Grype es cuanto menos similar a la de Trivy, quizás un poco menos visual, pero a lo que respecta en contenido son muy similares

NAME	INSTALLED	FIXED-IN	TYPE	VULNERABILITY	SEVERITY
apt	1.8.2.1		deb	CVE-2011-3374	Negligible
apt	1.8.2.1	1.8.2.2	deb	CVE-2020-27350	Medium
bash	5.0-4		deb	CVE-2019-18276	Negligible

Figura 20. Resultados obtenidos tras ejecutar Grype

3. Comparativa Trivy vs Grype

Las variedad de imágenes escogidas son Ubuntu, Nginx, Redis y Alpine, todas ellas en su última versión, es decir, con etiqueta “latest”. A pesar de que había una gran cantidad de ellas para elegir, estas son unas de las más usadas en el ambiente de los contenedores. Así entonces los resultados se comprueban en la Tabla 2.

Herramienta	Imagen	Fallos
Trivy	Ubuntu	24
Grype	Ubuntu	24
Trivy	Nginx	147
Grype	Nginx	149
Trivy	Nginx: 1.17.2	334
Grype	Nging: 1.17.2	338
Trivy	Redis	80
Grype	Redis	81
Trivy	Alpine	2
Grype	Alpine	2

Tabla 2.Resultados en formato tabla

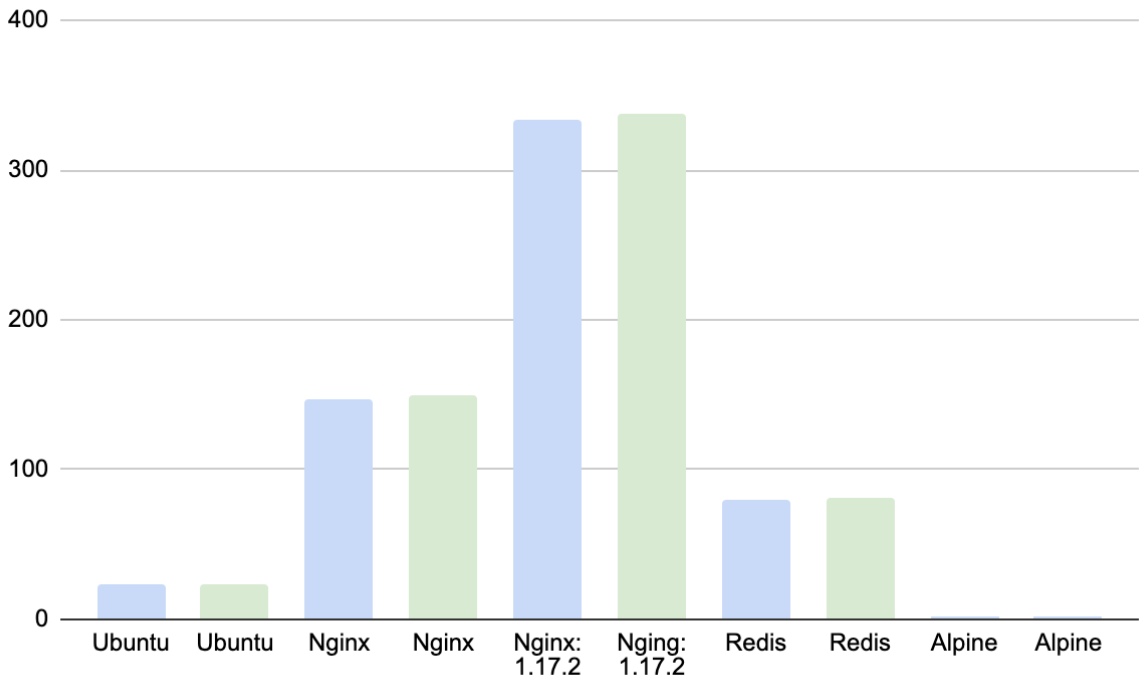


Figura 21. Resultados en formato gráfica

Examinando la gráfica anterior (Figura 21) se observa que los datos obtenidos por ambas herramientas son muy similares. Salvo en las distribuciones Nginx y Redis las vulnerabilidades encontradas han sido idénticas, e incluso en estas no ha diferido en más de dos unidades. Se puede pensar que se podía haber realizado una categorización de las

vulnerabilidades mayor, sin embargo la forma que tienen Trivy y Grype de clasificar las debilidades no es idéntica, por lo que podría haber llevado a confusiones el desempeño de otro modo.

Con el objetivo de observar la importancia de mantener una imagen actualizada que cubra todas las posibles vulnerabilidades que aparecen a medida que avanza el tiempo, se ha analizado la versión de una imagen Nginx anterior, concretamente nginx:1.17.2. Durante su examen, se han obtenido 324 y 328 vulnerabilidades por parte de Trivy y Grype respectivamente. La diferencia frente a las casi 150 extraídas de nginx:latest es cuanto menos notable, ya que de este nicho un atacante tendría el doble de puntos de acceso para poder acometer contra el sistema.

Puede sorprender al estudiar la gráfica Alpine el desigual bajo número de vulnerabilidades que tiene en comparación con las otras imágenes, esto es debido a que Alpine es una distribución de Linux sin apenas paquetes instalados pesando únicamente 5 Mb. Estas imágenes son empleadas en entornos de producción ya que la superficie de ataque resulta muy pequeña en relación, por ejemplo, a Ubuntu, que almacena gran cantidad de funcionalidades.

Anexo V

1. Ejemplo de la necesidad del empleo de secretos

Imagínese una imagen que se encuentra en un contenedor privado de docker. Si desde un cluster externo se intenta acceder y descargar dicha imagen dará error de acceso denegado, ya que dicho repositorio es privado y no se ha realizado ningún intento de autenticación. Por el contrario, si en la imagen se especifica que se puede acceder a esta mediante el uso de secretos, al intentar acceder especificando el secreto pertinente se podrá descargar la imagen y emplearla como se desee. Este mecanismo es empleado para no tener que tratar información crítica en múltiples ficheros, outputs o peticiones.

2. Análisis de arquitectura y funcionalidades Hashicorp VAULT

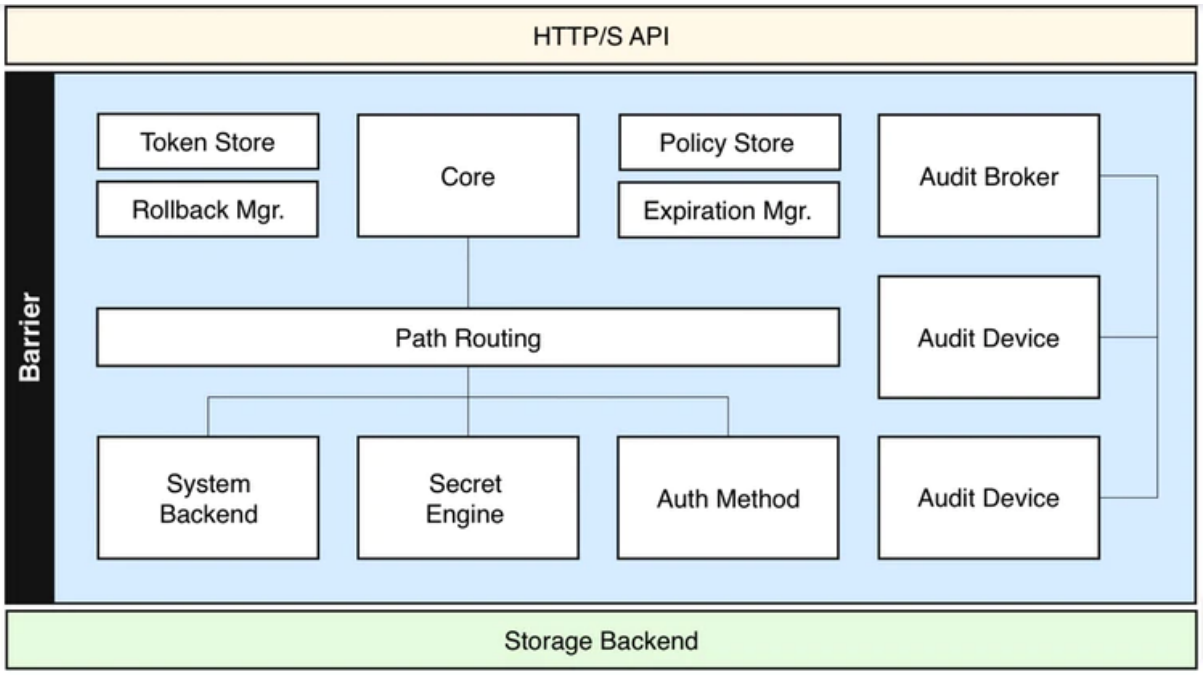


Figura 22. Vault Architecture by Hashicorp (link [The HashiCorp Vault Adoption Guide]( ))

Como se observa en la Figura 22 aparecen tres grandes grupos. Una API Rest sustentada por HTTP/S. Una barrera o Barrier que contiene prácticamente toda la operativa de Hashicorp Vault y un Storage Backend, lugar en el que se almacenan los secretos.

Cuando arranca Vault este aparece en estado *sealed* o sellado, desde este punto no es posible acceder a los secretos guardados, por lo que para configurarlo será necesario hacer un *unsealed*. Para modelar como va ser el funcionamiento de Vault hay que desbloquearlo, Vault emplea un esquema Shamir (link [Esquema de Shamir - Wikipedia, la enciclopedia libre]( ) el cual divide el que podría ser un solo acceso para un recurso en varios. Al iniciar Vault este crea varias llaves (5 por defecto), mediante las cuales, disponiendo de un número

concreto de ellas (3 por defecto), se podrá realizar la apertura de Vault y configurarlo según las necesidades pertinentes. Estas llaves no tienen que estar almacenadas en la misma ubicación ni se debe encargar de ello una sola persona, esta es la base del esquema de Shamir.

Así entonces, cualquier usuario que goce de los permisos adecuados podrá beneficiarse de las múltiples herramientas de Vault.

De manera general, todo el intento de interactuar con Vault pasa por el siguiente workflow de 4 etapas bien definidas:

1. Autenticar: Se comprueba si el cliente es quien dice ser
2. Validar: Se asegura la identidad del cliente con terceros confiables
3. Autorizar: Se compara la acción deseada por el cliente con las políticas existentes (dar acceso o no a endpoint de la API)
4. Acceso: Consiste en otorgar la entrada a los recursos requeridos, esto se realiza con un token formado por políticas y la identidad.

En cuanto a sus características, ya se puede observar en este punto que su operativa funciona de una forma mucho más amplia que SealedSecrets.

Al igual que SealedSecretes, Vault ofrece un **almacenamiento seguro de secretos** ya que los que son almacenados de forma persistente pasan primero por una encriptación empleando Cifrado AES 256-bit en GCM con 96-bit nonces, debido a que K8s únicamente los codifica en base64. Esto implica directamente que aunque una persona pudiera acceder al directorio donde se encuentran dichos secretos no podría acceder a su lectura y comprensión.

A pesar de que el principio base para evitar la lectura de secretos es muy similar con SealedSecrets, encriptando los campos necesarios, Vault tiene más características explotables, entre ellas destaca los Secrets Engines o **motores de secretos**. Este es el comienzo del empleo de secretos, tras configurar uno de estos como Microsoft Azure, AWS o Google Cloud Platform se podrá ya entonces guardar un secreto.

Los motores de secretos son componentes de Vault que almacenan, generan o cifran secretos. Algunos motores de secretos, como el motor de secretos de clave/valor, simplemente almacenan y leen datos. Otros motores de secretos se conectan a otros servicios y generan credenciales dinámicas bajo demanda. Otros motores de secretos proporcionan cifrado como servicio. (link Secrets Engines | Vault - HashiCorp Learn)

Si un usuario (o boot) desea realizar alguna acción con uno de estos motores, se deberá haber autenticado y disponer de las credenciales necesarias para emplear dicho motor.

Para comprobar si un usuario determinado posee las credenciales aptas para un recurso, Vault lo chequea contra las **políticas** de dicho recurso. Estas se crean en HCL, siendo también compatibles con JSON.

Por otra parte aparece el término de **secretos dinámicos**, un caso de uso de estos sería si por ejemplo un usuario desea acceder a una base de datos de forma temporal,Vault generará unas credenciales con los permisos pertinentes (un secreto). Después de esto, pasado un tiempo concreto especificado, Vault eliminará automáticamente dicho acceso.

Otra característica con una gran utilidad es el llamado **cifrado de datos al vuelo**, esto permite encriptar (y desencriptar) datos sin guardarlos. Esta propiedad permite ahorrar una gran cantidad de tiempo, por ejemplo, si un desarrollador desea cifrar las contraseñas y almacenarlas en su base SQL, logrará encriptar dichos datos sin tener que definir sus políticas de encriptación.
Todos los secretos cuentan con un tiempo de **expiración**, sin embargo el usuario puede pedir **renovarlo** a través de la API. (link [Introduction | Vault by HashiCorp])

Junto con todo ello Vault proporciona servicios de auditoría que almacenan un registro de cada solicitud y respuesta realizadas a la API

Anexo VI

Ejemplo práctico

Con el objetivo de producir una demostración práctica aplicable de todo lo contado anteriormente se ha llevado a cabo el levantamiento de un sistema cloud similar al de la Figura 23.

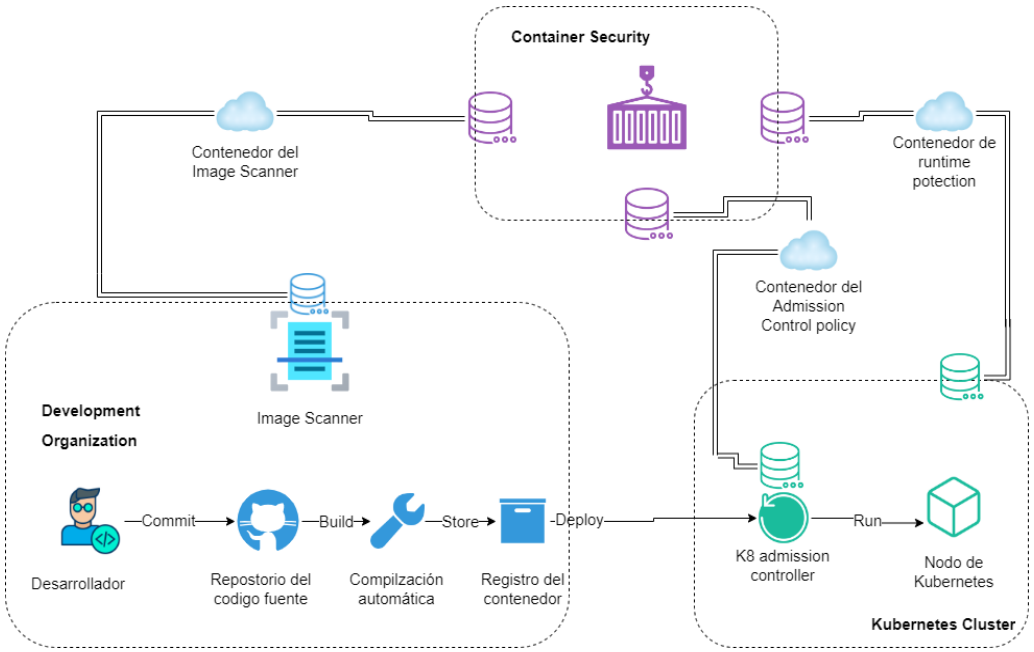


Figura 23. Arquitectura del ejemplo a probar

Explicando la Figura 23 se observan tres principales actores que interactúan entre ellos de manera segura.

En primer lugar se encuentra el llamado **Development Organization**, en este caso sería el ordenador personal, podría ser referido a cualquier ordenador de la empresa o, en caso de que se realizará el desarrollo con un equipo (como suele suceder) de varios.

Nada más comenzar se ha llevado a cabo un análisis de la plataforma utilizando Kube-Bench, el cual muestra los posibles riesgos preexistentes en el entorno a utilizar.
Así entonces se observa un flujo de trabajo en el cual, los desarrolladores escriben el código, hacen commit del mismo en un repositorio, se construye una imagen y se almacena en un registro, para finalmente hacer un despliegue en el cluster que se desee, por supuesto, después de haber escaneado la imagen.
Dependiendo del resultado de este escaneo, se decidirá si la imagen está preparada o no, para su despliegue. La decisión de esto puede ser subjetiva en cierta manera, ya que dependiendo del nivel de seguridad que se necesite se pueden abrir más o menos las restricciones a cumplir. En este caso, se consideraría que una imagen puede pasar si no contiene ningún tipo de vulnerabilidad.

La intención de todo esto es realizarlo mediante prácticas de *CI / CD*, integración y entrega continua, todo ello claramente de forma automática.

En todo este proceso, se debe realizar distintos *taggeos* que expliquen el estado en el que se encuentra la imagen a desplegar, por ejemplo:

- Al escribir la imagen: untagged
- Tras analizar con Trivy:
    - Si se puede desplegar: cleanImage
    - Si no se puede desplegar: dirtyImage

Ya en el **Kubernetes Cluster**, previo a correr la imagen mencionada anteriormente, esta es verificada por un Admission Controler, que verifica si sus características se atañen a las políticas y configuración deseada. En este caso, por lo comentado anteriormente, Kyverno seria la herramienta idónea.

En caso de que contradiga en algún aspecto las políticas establecidas, dependiendo de cual incumpla, podría ser rechazado el deployment o únicamente registrada la incidencia, por el contrario, si se han chequeado las políticas correctamente pasará a realizarse el despliegue del sistema.
Con el objetivo de lograr una mayor seguridad se puede realizar un firmado de imágenes, empleando por ejemplo Notary, que logren verificar la legitimidad de una imagen.

Para un mantenimiento del sistema adecuado, se monitorearán las actividades que ocurren en el mismo, empleando Falco, el cual realizará la función de Runtime Protection.

Sumado a esto, se emplea SealedSecrets, para permitir a los desarrolladores subir sus aportaciones con los secretos pertinentes a los repositorios, ya que estos se encuentran cifrados y no hay un riesgo alto de que puedan ser leídos, lo que facilita mucho la tarea.