MÁSTER UNIVERSITARIO EN CIBERSEGURIDAD Y PRIVACIDAD

TRABAJO DE FIN DE MÁSTER

Securización y monitorización inicial de un clúster de Kubernetes

Autor:
Rafael Antonio Marín Sánchez

Docentes:
Manuel Jesús Mendoza Flores
Víctor García Font

5 de Enero de 2023

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Securización y monitorización inicial de un clúster de Kubernetes*
Nombre del autor:	*Rafael Antonio Marín Sánchez*
Nombre del consultor/a:	*Manuel Jesús Mendoza Flores*
Nombre del PRA:	*Víctor García Font*
Fecha de entrega (mm/aaaa):	*01/2023*
Titulación o programa:	*Máster en Ciberseguridad y Privacidad*
Área del Trabajo Final:	*Seguridad Empresarial*
Idioma del trabajo:	*Castellano*
Palabras clave	*Kubernetes, Securización, Monitorización*

Resumen del Trabajo

La adopción de desarrollos ágiles, abandonando aplicaciones monolíticas y abrazando desarrollos basado en microservicios se han visto impulsados gracias a la creciente popularidad de herramientas de contenedores y orquestadores para administrar estos, como Kubernetes. Donde a diferencia de las máquinas virtuales, los contenedores tienen un menor impacto a nivel de consumo de recursos, ya que las primeras encapsulan totalmente el sistema operativo y recursos utilizados, frente a la compartición de recursos de los segundos. Esto ha propulsado un cambio en los enfoques de desarrollo, focalizándose en la creación de funcionalidades de forma "independiente", permitiendo así ser más competitivos en la respuesta a las exigencias del mercado y corrección de errores de aplicaciones.

El objetivo del trabajo es la realización de un estudio de las herramientas de orquestación de contenedores, concretamente Kubernetes, analizando el impacto en el mercado actual de la misma, el cambio de paradigma que supone y los nuevos retos de seguridad asociados. Para ello, tras definir todos los conceptos necesarios, se han presentado una serie de escenarios de buenas prácticas y herramientas de gestión de la seguridad.

Abstract

The adoption of agile development, abandoning monolithic applications and embracing microservice-based development has been driven by the growing

popularity of container and orchestration tools for managing them, such as Kubernetes. Unlike virtual machines, containers have a lower impact on resource consumption, as the former fully encapsulate the operating system and resources used, compared to the resource sharing of the latter. This has driven a shift in development approaches, focusing on the creation of "independent" functionalities, allowing for more competitiveness in responding to market demands and correcting application errors.

The aim of the work is to conduct a study of container orchestration tools, specifically Kubernetes, analyzing its impact on the current market, the paradigm shift it represents, and the new security challenges associated with it. To this end, after defining all necessary concepts, a series of best practice scenarios and security management tools have been presented.

Agradecimientos

“*Quiero expresar mi agradecimiento a mis amigos*
y familiares por su amor y apoyo incondicional
durante el proceso de investigación
y escritura del trabajo.

Agradecer a mí tutor por su apoyo
y guía durante el proceso de investigación
y escritura del trabajo.

Un agradecimiento especial
a ***mí pareja***
por facilitarme tiempo para
el proceso de desarrollo de la memoria,
asumiendo en ocasiones más carga
de tareas del día a día.

Espero que esta memoria
refleje la dedicación y el esfuerzo
de todos ustedes.

Con cariño,
Rafa.”

Índice

Lista de figuras

Lista de tablas

1. Introducción

1.1. Contexto y justificación del Trabajo

La adopción de desarrollos ágiles, abandonando aplicaciones monolíticas y abrazando desarrollos basado en microservicios se han visto impulsados gracias a la creciente popularidad de herramientas de contenedores como Docker[1]. A diferencia de las máquinas virtuales (**VM**), los contenedores tienen un menor impacto a nivel de consumo de recursos, ya que las primeras encapsulan totalmente el sistema operativo (**OS**) y recursos utilizados por cada **VM**, frente a la compartición de recursos y kernel del OS anfitrión de los segundos. Adicionalmente, los contenedores son más portables tanto para su creación como su destrucción, y por lo general, requieren un menor coste de mantenimiento.

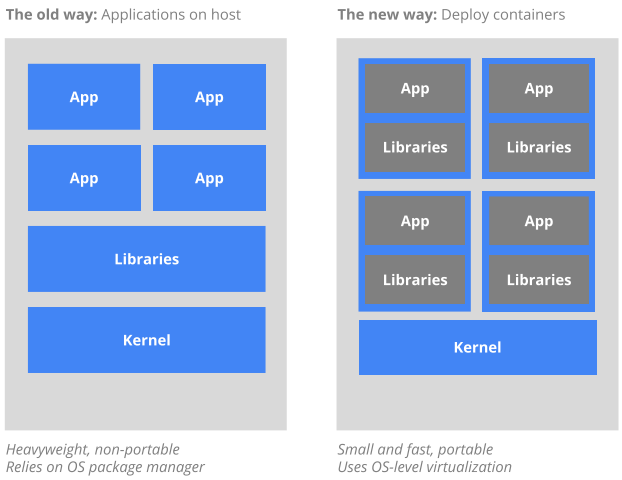

Figura 1 – Comparación de arquitecturas de aplicación
Ref. [5]

Esto ha propulsado un cambio en los enfoques de desarrollo, focalizándose en la creación de funcionalidades de forma “independiente”, permitiendo así ser más competitivos en la respuesta a las exigencias del mercado y corrección de errores de aplicaciones. Adicionalmente, la mejora de las infraestructuras de telecomunicaciones globales e incremento de la velocidad de conexión a internet en casi cualquier parte del planeta, han sido otro pilar importante en este cambio de paradigma. Este último punto es importante para entender cómo cada día más usuarios finales han cambiado la forma de consumir software. El software como un servicio es la tendencia de los últimos años (véase de ejemplo plataformas de streaming de video, música, etc). Ante estos nuevos retos y el

auge de la computación en la nube, nace el concepto de **SaaS** (Software as a Service), motivado por lo anteriormente comentado. [2]

Para entender como se ha llegado al estado actual de los contenedores tenemos que remontarnos a 2004. Google comienza a desarrollar el concepto de *'process container'* que posteriormente pasó a bautizarse como ***cgroups*** *(control groups).* Este concepto introdujo una interfaz unificada la cual daba la posibilidad de, para una agrupación de procesos, limitar y/o medir recursos de sistema usados (CPU, memoria, disco, etc), así como priorizar y controlar la ejecución de estos. Esta funcionalidad fue tan importante que en 2008 se integró al Kernel de Linux lanzando a su vez la primera versión de contenedores más parecida a la actual **LXC** (Linux Container).

En 2013, Docker aparece en la industria revolucionando nuevamente el concepto. Docker proporcionaba una interfaz amigable y fácil de usar, así como la posibilidad integración nativa en cualquier sistema operativo, lo cual permitía dejar atrás las limitaciones de desarrollos asociadas a los kernel de estos.

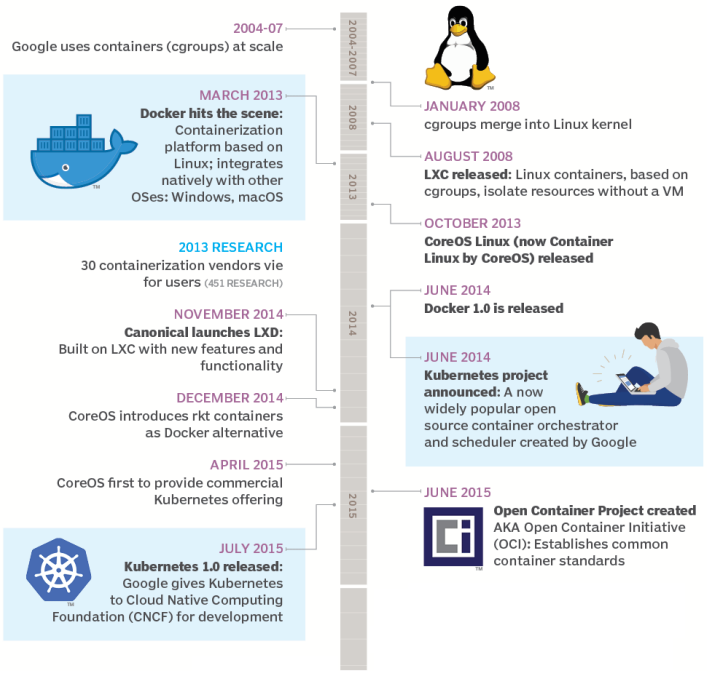

Figura 2 – Evolución de los contenedores
Ref: [26]

El uso de contenedores es ventajoso tanto en cuanto permite encapsular completamente una funcionalidad concreta de una aplicación y las dependencias

de esta, facilitando así el escalado tanto horizontal como vertical de la misma [3], aspecto fundamental en SaaS. Como desventaja, el crecimiento de una aplicación de este tipo implica la necesidad del manejo de un gran número de contenedores, así como afrontar las consecuencias de la naturaleza efímera de estos. Adicionalmente se presenta nuevos retos a nivel de red, persistencia de datos, vulnerabilidades, etc. Ante estas necesidades aparecen los primeros orquestadores, de los cuales el más popular es ***Kubernetes*** [4] [5]. Este proporciona utilidades para el control de despliegue, reinicio, escalado de contenedores, además de ayudar a toda la configuración de red que forma la infraestructura de la aplicación o el ecosistema de esta. ***Kubernetes***, y en general los orquestadores, también ofrecen una capa de seguridad a dicha infraestructura y aplicación.

1.2. Objetivos del Trabajo

Tras comentar la importancia actual de herramientas como Kubernetes en el desarrollo de software, surge igualmente la necesidad de analizar estos nuevos entornos desde el enfoque de la seguridad, es por esto por lo que el trabajo se centrará en el despliegue de un clúster de Kubernetes aplicando una configuración inicial siguiendo buenas prácticas e instalando una herramienta de detección de amenazas que avise en caso de intento de escalado de privilegios.

El trabajo se dividirá en varias etapas:

- **Estudio preliminar**
  Puesto que no se tiene experiencia en el uso de herramientas basadas en contenedores, motivación principal para abordar dicho tema, el trabajo constará de una gran carga de investigación de la tecnología, buenas prácticas, vulnerabilidades frecuentes, así como estudio de posibles soluciones para la detección de amenazas.
- **Diseño de la solución**
  Una vez concluida la parte de estudio, se comenzará a la creación de entorno de trabajo. Para esta etapa se valorará el desplegar el clúster en un entorno cloud, aprovechando el tier gratis de estudiante que poseen la mayoría de los principales proveedores.
- **Documentación**
  A medida que se avanza en las anteriores etapas y de forma paralela, se irá redactando la memoria del trabajo, la cual recogerá todo el análisis, desarrollo y conclusiones del tema propuesto.

1.3. Impacto en sostenibilidad, ético-social y de diversidad

Desde el punto de vista de la **CCEG** *(competencia de compromiso ético y global)*, podemos destacar varios aspectos del trabajo presentado:

- Sostenibilidad
  - ODS 9 - Industry, innovation and infrastructure
  - ODS 12 - Responsible consumption and production
  - ODS 13 - Climate action

El cambio de paradigma de implementación de la estructura que propone el uso de Kubernetes, se alinea con los puntos marcados, debido al mejor aprovechamiento de recursos que hace esta tecnología, esto a su vez implica un menor consumo de energía. Y la tecnología en sí misma es una innovación con respecto a la infraestructura monolítica tradicional.

- Comportamiento ético y responsabilidad social
  - ODS 8 - Decent work and economic growth

Este nuevo paradigma y el auge de los entornos cloud, permite que nazcan nuevos proyectos que aporten valor sin necesidad de realizar las inversiones iniciales que se requerían hace años.

- Diversidad
  - ODS 5 - Gender equality
  - ODS 10 - Reduced inequalities

Por lo ya comentado en los puntos anteriores, eso repercute directamente en a la reducción de desigualdades intrínsecas de la falta de capital. Además, en el sector IT se está experimentando una tendencia alcista de la representación femenina. [6]

1.4. Enfoque y método seguido

La motivación del trabajo es, en primera instancia adquirir conocimientos de tecnologías de contenedores y orquestación de estos desde el punto de vista de la seguridad informática. El enfoque que se le dará al trabajo será el detallar el proceso seguido para entender estas tecnologías y plasmarlo en una demo real.

1.5. Planificación del Trabajo

El desarrollo del trabajo tiene una fuerte limitación temporal marcado por la propia naturaleza de este. Tomando como referencia los diferentes tramos marcados por la propia universidad se ha realizado el siguiente diagrama de Gantt.

Figura 3 – Planificación de entregas de la UOC
Ref: Web del TFM de la UOC

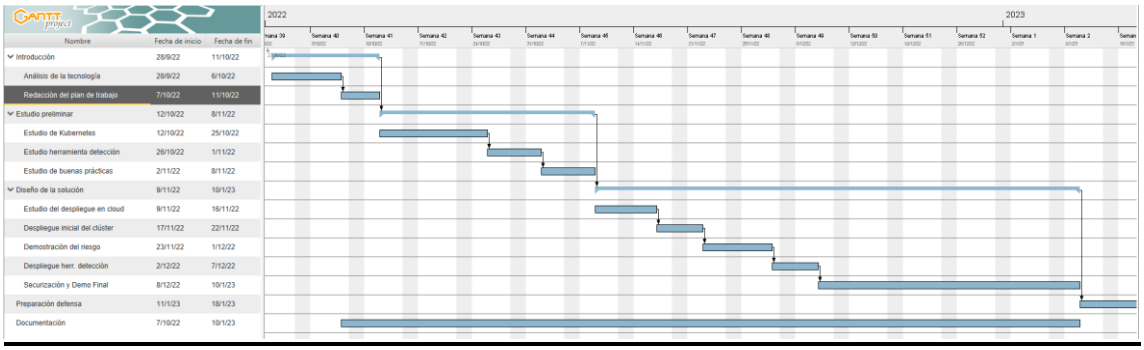

Figura 4 – Diagrama de Gantt del trabajo

- Introducción
  - Análisis de la tecnología
  - Redacción del plan de trabajo
- Estudio preliminar
  - Estudio de Kubernetes
  - Estudio herramienta detección
  - Estudio de buenas prácticas
- Diseño de la solución
  - Estudio del despliegue en cloud
  - Despliegue inicial del clúster
  - Demostración del riesgo
  - Despliegue herramienta de detección
  - Securización y demo final

2. Estudio Preliminar

En esta sección detallaremos todo estudio previo realizado sobre la tecnología y los diferentes puntos que pretendemos abordar en la demo final. En un primer lugar hablaremos sobre qué es un orquestador de contenedores y el estado de arte de estos, centrándonos en **Kubernetes**. Posteriormente, haremos una vista por los diferentes retos, a nivel de seguridad, a los que se enfrenta dicha tecnología y una serie de buenas prácticas para mitigarlos. Por último, exploraremos el conjunto de herramientas que ayudan a lidiar con estos desafíos, centrándonos en **Falco**, la herramienta elegida en el escenario práctico.

2.1. Ecosistema con contenedores

Sin profundizar demasiado en el concepto de contenedores, como hemos comentado en la introducción, estos nos permiten empaquetar y distribuir aplicaciones de software de manera que se puedan ejecutar de manera consistente en cualquier entorno. Los contenedores se basan en la tecnología de virtualización a nivel de sistema operativo y permiten a las aplicaciones ser ejecutadas en un entorno aislado del sistema operativo anfitrión. Esto significa que las aplicaciones contenidas pueden ser ejecutadas sin preocupaciones de dependencias o conflictos con otras aplicaciones o librerías instaladas en el sistema operativo anfitrión.

Algunos de los conceptos importantes son [48][49]:

- **Imagen:** es un archivo que contiene todo lo necesario para ejecutar una aplicación en un contenedor. Una imagen incluye el código de la aplicación, así como todas las dependencias y librerías necesarias para ejecutar la aplicación. También incluye la configuración del sistema operativo, como las variables de entorno y los archivos de configuración necesarios para ejecutar la aplicación.
- **Namespace**: mecanismo que permite, a nivel de sistema operativo, la creación entornos aislados en los que se pueden asignar distintos recursos y objetos, como procesos, archivos, sockets y otros de la máquina anfitriona, sin que estos sean visibles o accesibles desde otros entornos.
- **Container Runtime:** es un software que se encarga de ejecutar y administrar contenedores en un sistema operativo. Un *container runtime* proporciona una serie de funcionalidades para crear y gestionar contenedores, como crear y eliminar contenedores, iniciar y detener

contenedores, asignar recursos de sistema a los contenedores y gestionar la comunicación entre los contenedores y el sistema operativo anfitrión. Algunas implementaciones podrían ser ***containerd, Docker Engine*** y ***CRI-O***.

- **Container Runtime Interface (CRI):** es una interfaz diseñada para permitir a los sistemas de orquestación de contenedores, como *Kubernetes*, controlar y comunicarse con diferentes implementaciones de *container runtime*. ***CRI*** especifica cómo se deben iniciar, detener y gestionar los contenedores, y es utilizada por herramientas de orquestación de contenedores como *Kubernetes* para interactuar con el sistema operativo anfitrión y gestionar los contenedores en un clúster. Esto permite elegir la implementación de ***CRI*** que mejor se adapte a las necesidades y preferencias, sin tener que preocuparse por la compatibilidad con el sistema de orquestación de contenedores.
- **Open Container Initiative (OCI):** es una organización sin fines de lucro formada en 2015 para promover la adopción y el desarrollo de estándares abiertos para contenedores de software. ***OCI*** se centra en estandarizar la forma en que los contenedores se construyen, distribuyen y ejecutan, con el objetivo de facilitar la interoperabilidad entre diferentes sistemas de contenedores y herramientas de contenedores. ***OCI*** ha desarrollado dos estándares principales ampliamente utilizados en la industria:
    - **Estándar de Imagen de Contenedor**, especifica cómo deben construirse y distribuirse las imágenes de contenedor
    - **Estándar de Runtime Container**, especifica cómo deben ejecutarse y gestionarse los contenedores.

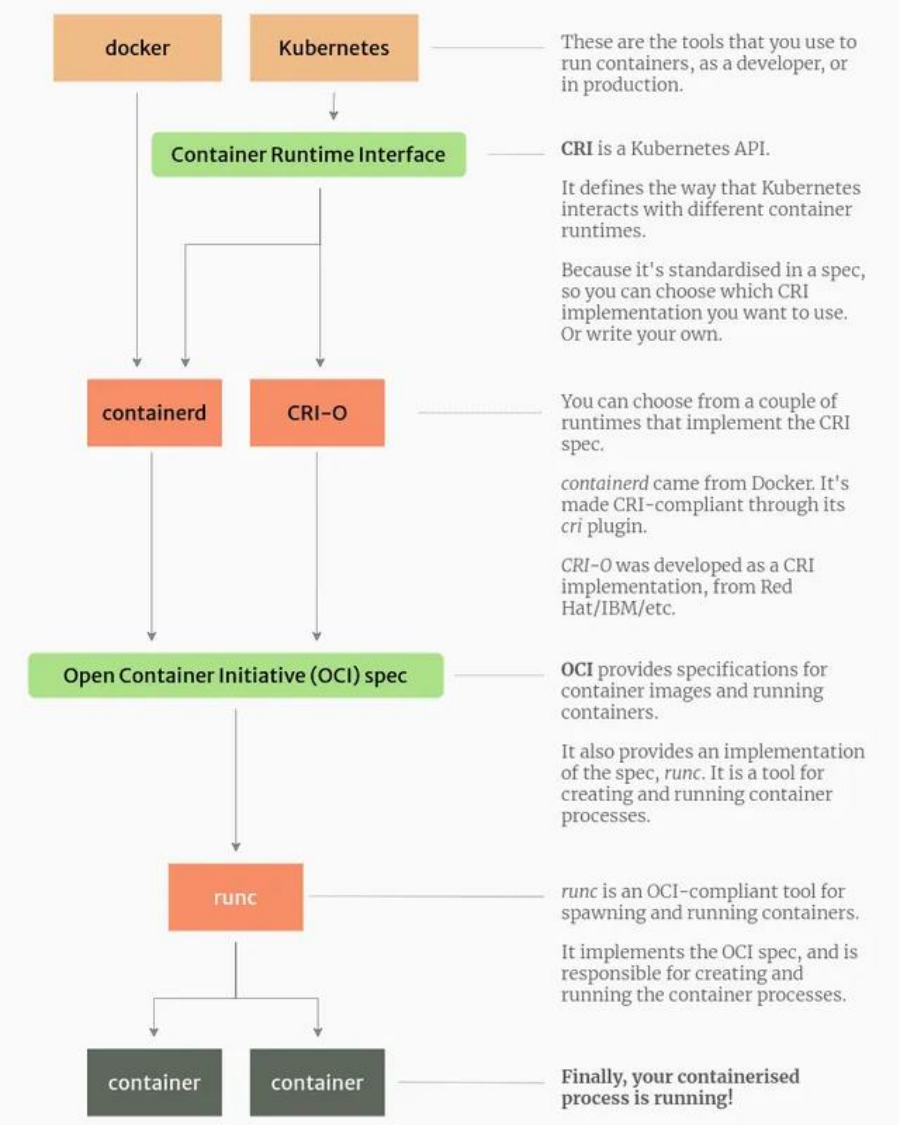

Figura 5 – Infografía del ecosistema de contenedores
Ref:[48]

2.2. Orquestadores de contenedores

La orquestación de contenedores se podría definir como la automatización del control del ciclo de vida de los contenedores: adquisición, construcción, entrega, despliegue, inicio y mantenimiento. [27]

- **Adquisición**: selección y obtención de la imagen base que formará el contenedor.
- **Construcción**: empaquetado de la aplicación en sus dependencias en una imagen.
- **Entrega**: subida a producción de la imagen con la aplicación, tras el sometimiento a diferentes pruebas.

- **Despliegue**: instanciar la nueva imagen y mantenerla actualizada.
- **Inicio**: definición de la lógica de funcionamiento (políticas escalado, métricas, etc).
- **Mantenimiento**: ya sea online (monitorización asociada) u off-line (debugging de problemas)

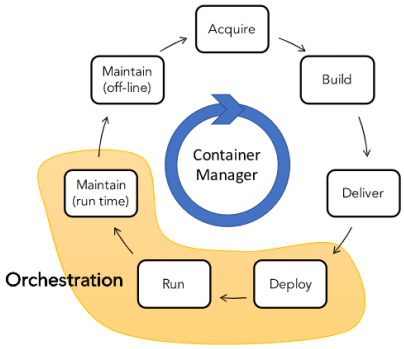

Figura 6 – Ciclo de vida de un contenedor
Ref. [27]

Los orquestadores más importantes del mercado son:

Herramienta	Características principales
Docker Swarm [28]	• Pequeños desarrollos → pocas funcionalidades • Soportado en cualquier sistema operativo • Ligero y fácil de usar • Balanceo de carga automático • Clústeres nativos para Docker • Agrupación, listado, control en nodos • Consola integrada • Dispone de repositorio de imágenes propio • Creada por *Docker Enterprise*
Kubernetes	• Gran comunidad Open-Source • Compleja instalación y curva de aprendizaje • Soportado en cualquier sistema operativo • Soportado en cualquier proveedor de nube

	• No dispone de repositorio de imágenes propio • Creada por *Google*
OpenShift [29]	• Solo soportado en distribuciones propietarias • Políticas de seguridad muy estrictas • Interfaz amigable y fácil de usar • Plantillas poco flexibles • Dispone de repositorio de imágenes propio • Usa Kubernetes como • Creada por *RedHat*

Tabla 1 – Comparativa entre Orquestadores

Siendo **Kubernetes** el estándar de facto. Tanto es así que todos los proveedores principales proveedores de nube tienen su propia implementación de este.

Proveedor	Servicio
Amazon	EKS
Microsoft	AKS
Google	GKE
Alibaba	ACK
IBM	IBM Cloud Kubernetes Service

Tabla 2 – Implementaciones de Kubernetes en nube pública

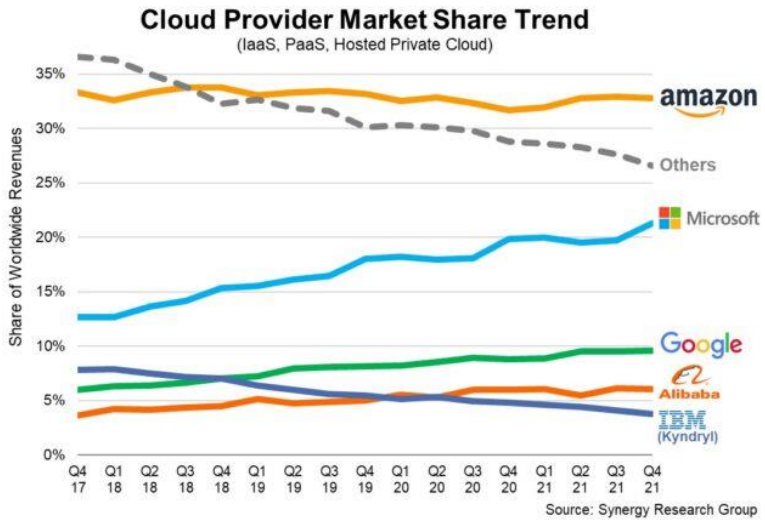

Figura 7 – Principales proveedores de nube pública
Ref:[47]

2.2.1. Kubernetes

Kubernetes es una palabra de origen griego que significa "*timonel*" o "*piloto*". La palabra "*kubernēs*" proviene de "*kubernáo*", que significa "*dirigir con habilidad*" o "*gobernar*". Esta palabra se utiliza en el contexto de Kubernetes para hacer referencia a la capacidad de la plataforma para orquestar y dirigir el funcionamiento de aplicaciones en contenedores. La palabra "*Kubernetes*" se escribe con "*K*" para hacer referencia a la palabra griega original y para diferenciarlo de otros proyectos de software que utilizan palabras similares, pero con ortografía diferente. [7]

Con la llegada al mercado de Docker en el año 2013, existen nuevas necesidades las cuales son las motivaciones para el desarrollo de Kubernetes. Las primeras funcionalidades de Kubernetes lanzadas en 2014 fueron:

- Despliegue de muchas instancias de una misma aplicación
- Balanceo de carga y rutado de tráfico hacía estas múltiples instancias
- Monitoreo básico de las instancias y autoreparación para asegurar el buen funcionamiento de estas
- Agrupación de varias máquinas donde poder distribuir el trabajo
- Necesidad de que la herramienta fuera *open source*, siendo donado en este mismo año a **CNCF** *(Cloud Native Computing Foundation)* bajo la licencia *Apache 2.0*

Aunque no fue hasta 2015 y gracias a la ayuda del equipo de **OpenShift** (*RedHat*) cuando empieza a adquirir importancia en el mercado. **Kubernetes** tiene como propósito reducir la complejidad de construcción nuevos desarrollos.

Un clúster de Kubernetes puede ser entendido como un conjunto de recursos de cómputo que proporcionan un mecanismo de abstracción para la ejecución de servicios basados en contenedores. Kubernetes ofrece un marco base común, el cual, sin necesidad de mucho aprendizaje previo, permite a desarrolladores desplegar un aplicaciones complejas, distribuidas y escalables. [7]

Las principales características de Kubernetes, en la actualidad serían:

- Alta eficiencia en la utilización de recursos
- Abstracción de la infraestructura para centrar en el esfuerzo en la aplicación

- Agilidad en la creación de nuevas funcionalidades de aplicaciones, capaces de ser encapsuladas en contenedores en vez de máquinas virtuales
- Facilidad para integración con herramientas de **CI/CD** *(Integración continua y despliegue continuo)*, permitiendo aplicar el potencial que estas nos bridan (test automáticos, etc)
- Portabilidad entre diferentes entornos y proveedores de nubes (casi en su totalidad centrada en entornos UNIX)
- Consistencia de entornos de desarrollo (ya sea en local, preproducción o producción)
- Facilidad de monitorización tanto de la infraestructura como de las aplicaciones que esta alberga

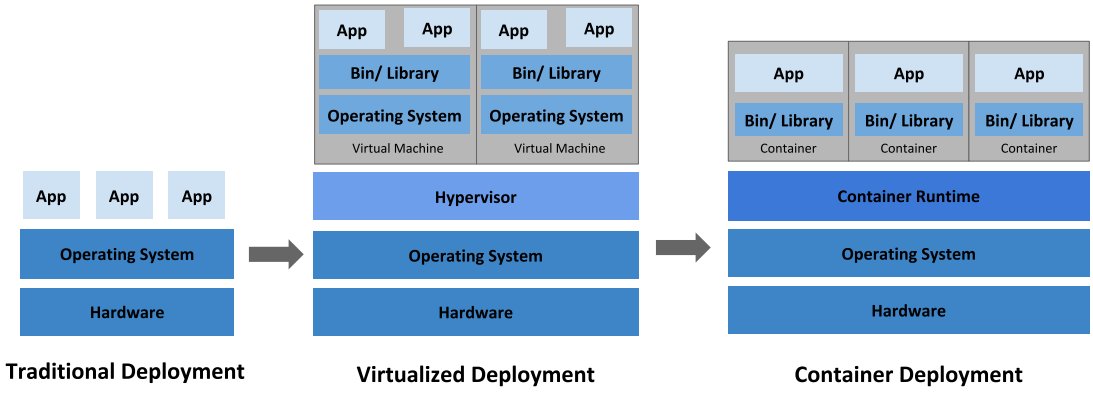

Figura 8 – Evolución infraestructura
Ref. [5]

Un clúster de Kubernetes está compuesto por varios componentes que trabajan juntos para proporcionar un sistema de orquestación de contenedores distribuido y con alta disponibilidad. Los componentes principales de un clúster de Kubernetes son:

- **Control plane:** Agrupan el conjunto de componentes que toman decisiones de forma global en el clúster y responden ante los diferentes eventos que suceden en este, así como gestionar el estado del clúster y asegurar que se cumplan las políticas y objetivos especificados.
    - **API Server**: responsable de la gestión de la API de Kubernetes proporcionando un punto de entrada para todas las solicitudes hacia esta, validándolas y autorizando estas. Además, de aplicar los cambios solicitados en el sistema de Kubernetes. La API de Kubernetes se basa en REST y proporciona un conjunto de

operaciones para realizar tareas como crear y administrar contenedores, desplegar aplicaciones, administrar el almacenamiento, verificar cualquier estado o configuración del clúster o de cualquiera de sus componentes, etc.

- **Etcd**: sistema de almacenamiento distribuido diseñado para ser rápido y seguro. Se utiliza como una base de datos de *clave-valor* para almacenar información sobre los nodos del clúster, los contenedores que se están ejecutando en cada nodo y los servicios que se están ejecutando en el clúster del clúster proporcionando consistencia, tolerancia a fallos y alta disponibilidad.
- **Scheduler**: componente del sistema de Kubernetes que se encarga de asignar pods a los nodos del clúster. Este toma en cuenta una serie de factores, como la capacidad de los nodos, la demanda de recursos de los pods y la configuración especificada por el usuario, para decidir dónde colocar cada pod. Es responsable de asegurar que los pods se ejecuten de manera equilibrada en el clúster y de evitar la sobrecarga de los nodos, así de mover pods de un nodo a otro si se produce un cambio en la carga de trabajo o si existe problema en un nodo en particular.
- **Controller-manager**: proceso de control de que se encarga de realizar las labores necesarias para que cada objeto definido en el clúster tenga el estado definido en su creación *(desired status)* analizando el estado actual de este *(current state)*, consultando la información contenida en ***etcd***.

- **Nodos**: conjunto de máquinas virtuales o físicas sobre las cuales corren los contenedores y herramientas de orquestación.
  - **Pod:** es la unidad desplegable más pequeña en Kubernetes, abstrayéndonos del concepto de contenedor y la tecnología de contenerización. Un pod puede estar formado por uno o varios contenedores los cuales comparten contexto (almacenamiento y recursos de red). Los pods son dinámicos y pueden ser creados, eliminados y replicados según sea necesario para satisfacer la demanda de la aplicación. El ***scheduler*** de Kubernetes se encarga de asignar estos pods a los nodos del clúster y de ajustar la distribución de estos según cambie las demandas de la aplicación.
  - **Kubelet:** agente de Kubernetes desplegado en cada nodo del clúster encargado de gestionar los contenedores en el nodo. Se comunica con la API de Kubernetes para informar sobre el estado del nodo y los contenedores dentro de este. Recibe instrucciones

sobre qué contenedores deben ejecutarse en el nodo y cómo deben configurarse. Además, es el encargado de iniciar y detener, asignar recursos de sistema a los contenedores y gestionar la comunicación entre estos y el sistema operativo anfitrión.

- **Container runtime**: como ya hemos comentado en el punto 2.1 es el software responsable de que los contenedores se puedan ejecutar.
- **Kube-proxy:** componente de Kubernetes desplegado en cada nodo del clúster, es el encargado de enrutar el tráfico entre los contenedores y servicios adecuados en el clúster, así como de mantener una vista actualizada del estado del clúster. Además, proporciona funcionalidades de balanceo de carga y protección contra ataques de denegación de servicio.

- **Addons:** Herramientas externas que haciendo uso de los recursos del clúster proporcionan funcionalidades extra a este.
  - **DNS**: permite el uso de nombre en lugar de IP para identificar servicios, pods, nodos y el propio clúster.
  - **Dashboard**: implementa una interfaz gráfica de propósito general para el clúster.
  - **Monitoring**: permite establecer mediciones en base-tiempo de cualquier objecto del clúster.
  - **Logs**: permite conocer que está pasando dentro de una aplicación, pods, nodo, etc. Son especialmente útiles en tareas de debbuging de problemas.

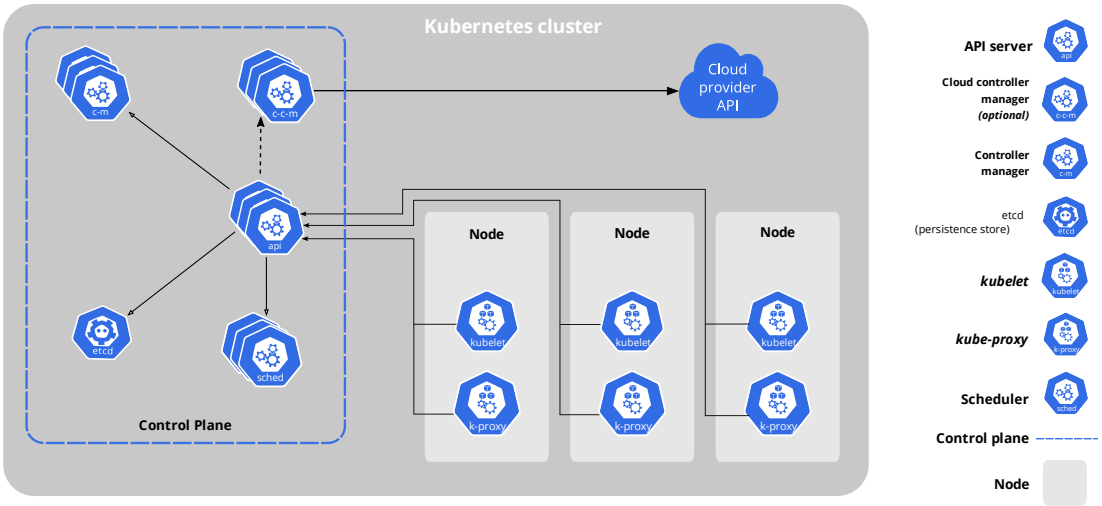

Figura 9 – Componentes clúster Kubernetes
Ref. [5]

Adicionalmente, cabe mencionar algunos componentes y características adicionales disponibles en Kubernetes:

- **Ingress**: componente de Kubernetes que se encarga de gestionar el tráfico de entrada al clúster. Permite a los usuarios configurar reglas de enrutamiento para dirigir el tráfico entrante a los servicios y contenedores adecuados desde el exterior.

- **ConfigMap y secretos**: componentes de Kubernetes que permiten a los usuarios almacenar y gestionar la configuración y los secretos de la aplicación de manera segura. Los configMap se utilizan para almacenar configuración no sensible, mientras que los secretos se utilizan para almacenar información sensible, como contraseñas y tokens de acceso.

- **DaemonSet**: componente de Kubernetes encargado de asegurar de que una copia de un pod se ejecute en cada nodo del clúster. Se utilizan a menudo para implementar tareas de segundo plano que deben ejecutarse en todos los nodos del clúster, como la recopilación de métricas, la monitorización o el registro de eventos.

- **StatefulSet**: componente de Kubernetes encargado de gestionar la ejecución y el escalado de aplicaciones que requieren estado, como bases de datos o aplicaciones de cola. Los *StatefulSet* proporcionan funcionalidades adicionales para garantizar la disponibilidad y la consistencia del estado en el clúster a través de un conjunto de recursos persistentes, como volúmenes de almacenamiento.

- **Horizontal Pod Autoscaler (HPA)**: componente de Kubernetes que se encarga de ajustar automáticamente el número de réplicas de un pod en función de la carga de trabajo. Esto permite ajustar el tamaño del clúster dinámicamente para satisfacer las necesidades por demanda.

Por último, desde el punto de vista de la arquitectura:

- **Nodos trabajadores** *(workers)* los cuales contienen el *container runtime, kublet y kube proxy* ya comentados.
- **Nodos maestros** *(master)* los cuales forman el *control plane* anteriormente comentado.

2.3. Retos de seguridad

La interconexión global y el, cada vez más extendido, uso de servicios de nube pública han llevado a la delegación de cierta responsabilidad en el tratamiento y almacenamiento de información valiosa para empresas y ciberdelincuentes, estos últimos cada vez más capacitados y con amplios conocimientos tecnológicos. Esto ha llevado a que la seguridad sea una prioridad cada vez mayor, siendo tanto que gobiernos y entidades públicas llevan años con el foco puesto en ella, véase el caso de Europa y la **GDPR** (*General Data Protection Regulation*).

Desde el punto de vista de la seguridad, los desarrollos basados en contenedores deben afrontar las vulnerabilidades propias de las aplicaciones y los nuevos vectores de ataque propias de la infraestructura.

Para esto, enfoques tradicionales basados en **WAF** *(Web Application Firewalls),* **DAST** *(Dynamic Application Security Testing)* o **SAST** *(Static Application Security Testing),* no son suficientes en entornos tan cambiantes, incluso van en contra de la agilidad propia de este nuevo paradigma. Es por esto que, con un enfoque centrado en la flexibilidad, nacen conceptos como **RASP** *(Runtime Application Self Protection)* e **IAST** *(Interactive Application Security Testing)*.

IAST es un método de prueba de la seguridad de una aplicación en tiempo de ejecución. A diferencia de otras formas de prueba, como el análisis estático o la prueba de penetración, **IAST** se produce mientras la aplicación se está ejecutando y se está utilizando activamente, proporcionando a los desarrolladores la oportunidad de abordar los problemas antes de que puedan ser explotados por atacantes. **IAST** implica el uso de herramientas especializadas que se integran en la aplicación y monitorean su comportamiento mientras se está utilizando. Estas herramientas analizan la entrada y la salida de la aplicación, así como sus operaciones internas, para identificar cualquier vulnerabilidad o debilidad de seguridad. También pueden identificar problemas con la configuración o implementación de la aplicación que podrían dar lugar a riesgos de seguridad. [30]

Existen diferentes soluciones disponibles **IAST** en el mercado, algunos ejemplos podrían ser:

- **Veracode**: plataforma de pruebas de seguridad en la nube que ofrece tanto análisis estático como capacidades de IAST. [64]

- **AppScan**: herramienta desarrollada por IBM que se puede utilizar tanto para el análisis estático como para el IAST. Analiza el código fuente de una aplicación, así como su comportamiento en tiempo de ejecución, para identificar vulnerabilidades de seguridad. [65]
- **Fortify WebInspect**: herramienta desarrollada específicamente diseñada para probar aplicaciones web. Se puede utilizar tanto para el análisis estático como para el IAST y es capaz de identificar una amplia gama de vulnerabilidades de seguridad. [66]

	Análisis Completo	Falsos positivos bajos	Análisis del código	Integración en el ciclo de vida
SAST	**Si**	No	**Si**	**Si**
DAST	No	**Si**	No	No
IAST	No	**Si**	**Si**	**Si**

Tabla 3 – Comparativa entre estrategias de análisis de vulnerabilidades

RASP tiene el enfoque de protección interno de la aplicación, a diferencia del enfoque perimetral de **WAF**, llegando incluso a ofrecer protección como fallos en la lógica de negocio. **RASP** actúa inyectando sensores para controlar las entradas/salidas de la aplicación, así como sus estados intermedios, monitorizando el comportamiento de la aplicación mientras se está utilizando e identificando cualquier actividad sospechosa o malintencionada. Además, se puede configurar para tomar medidas evitando que la actividad ocurra o para mitigar su impacto, como bloquear la entrada maliciosa o cerrar la aplicación. Otro punto positivo es que suelen ser herramientas fáciles de mantener y configurar, además de poder usarse en conjunto con el **WAF** tradicional.[31]

La principal diferencia entre *Interactive Application Security Testing* (**IAST**) y *Runtime Application Self-Protection* (**RASP**) es en el momento en que se realiza la protección de la aplicación. **IAST** se utiliza para identificar y corregir problemas de seguridad durante el desarrollo de una aplicación, mientras que **RASP** se utiliza para proteger a la aplicación de amenazas y vulnerabilidades de seguridad mientras se está utilizando

Con una visión centrada en la tecnología elegida, **Kubernetes**, la **OWASP** *(Open Web Application Security Project, organización sin ánimo de lucro centrada en*

mejorar la seguridad del software) nos proporciona un listado de los riesgos más comunes que se producen en las infraestructuras basadas en Kubernetes.[32]

- Exceso de privilegios y fallos en configuraciones iniciales
- Imágenes vulnerables por uso de dependencias de terceros vulneradas
- Ausencias de políticas restrictivas de seguridad
- Ausencias de monitorización y logado de acciones
- Mecanismos de autenticación poco sólidos
- Dependencias de desactualizadas
- Ausencia de segmentación de red
- Pobre protección para el manejo de secretos

Desde el punto de vista del atacante, **MITRE ATT&CK®** nos ofrece un marco base de conocimiento de las tácticas y técnicas más comunes en ciberataques. Estas casuísticas se ven claramente relacionadas con los riesgos comunes expuesto por la **OWASP** y como estas con explotadas por los atacantes.[33]

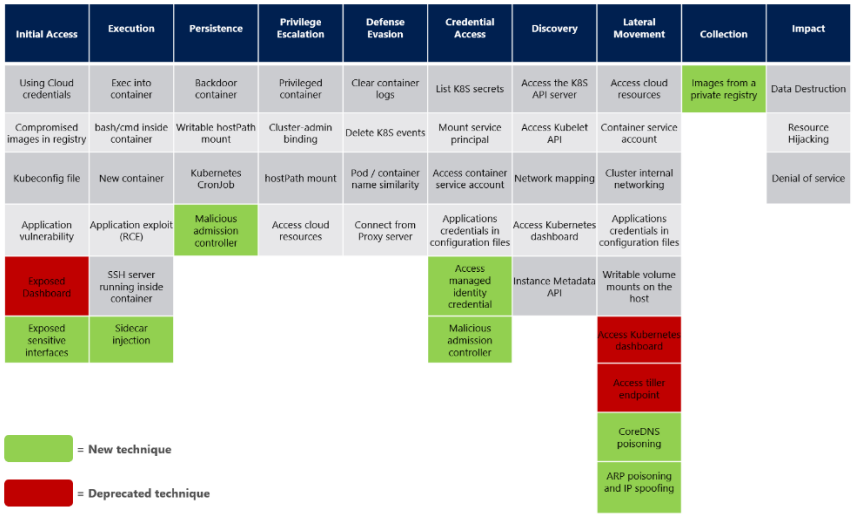

Figura 10 - MITRE ATT&CK® en Kubernetes
Ref:[33]

Realizando una agrupación por tácticas de ataque tendríamos:

- Acceso Inicial
  - **Robo de credenciales cloud**, permitiendo tener acceso a la capa de mantenimiento del clúster.
  - **Despliegue de imagen vulnerable**, el hecho de utilizar por parte del usuario imágenes que contengan vulnerabilidades permite a un atacante habilitar puertas traseras para realizar futuras intrusiones.

  - **Acceso al fichero Kubeconfig**, donde se alojan credenciales e información relevante de un clúster
  - **Despliegue de aplicación vulnerable**, el hecho de desplegar aplicaciones con vulnerabilidades permite a un atacante habilitar puertas traseras para realizar futuras intrusiones.
  - **Exposición de interfaces sensibles,** acceso a servicios sensibles.
- Ejecución código
  - **Creación de nuevos contenedores**, un atacante con permisos para desplegar contenedores podría crear nuevos recursos para ejecución de código malicioso.
  - **Despliegue de aplicaciones vulnerables**, similar al punto anterior, el hecho de desplegar aplicaciones con vulnerabilidades permite a un atacante habilitar puertas traseras para realizar futuras intrusiones o ejecución de código remoto.
  - **Acceso SSH desde un contenedor**, tras vulnerar alguna forma de autenticación válida, un atacante puede podría ganar acceso a los contenedores mediante el protocolo SSH.
- Persistencia
  - **Acceso a volúmenes**, tener persistencia en el host del clúster habilitando poder crear planificadores para ejecutar código malicioso, etc.
  - **Acceso al planificador**, pudiendo planificar la ejecución de código malicioso.
- Escalado de privilegios
  - **Acceso a contenedor privilegiado**, habilita a un atacante tener acceso total al host del clúster y a los recursos de este. Esta técnica seria equivalente a la posibilidad del atacante de desplegar contenedores privilegiados.
  - **Acceso del hostPath**, como ya hemos comentado en el punto anterior, tener acceso a los volúmenes del clúster permite a un atacante ganar acceso al host del clúster y los recursos de este.
- Ocultación de actividad
  - **Borrado de logs y de eventos,** práctica habitual por los atacantes para evitar dejar rastro de su actividad.
  - **Pods/contenedores con nombres parecidos**, cuando un atacante tiene la posibilidad de desplegar recursos dentro del clúster, el hecho de que asigne nombre a estos recursos similares del de las aplicaciones desplegadas por el usuario dificulta la detección de estos recursos maliciosos.

- **Uso de servidor proxy**, es habitual que los atacantes usen agentes externos para enmascarar su conexión, como agentes proxy.

- Robo de credenciales
  - **Listado de secretos**, atacantes con los privilegios suficientes pueden llegar a listar todos los secretos almacenados en el clúster realizando solicitudes a la API.
  - **Acceso a Service Account**, las cuentas de servicio son montadas en todos los pods creados en el clúster para poder comunicar son la API del mismo, el hecho de que un atacante tome el control de esta permite usar y/o tener acceso cualquier recurso del clúster.
  - **Consulta de credenciales en ficheros de configuración**, es común ver contraseñas almacenadas como variables de entorno o en ficheros de configuración. Cualquier atacante con acceso a la API o a estos ficheros de configuración podría llegar a consultarlas.
- Descubrimiento
  - **Acceso a la API**, como ya hemos comentado, cualquier un atacante con acceso a la API pude llegar a listar los componentes del clúster y el estado de estos.
  - **Acceso a agente de Kubernetes**, similar al caso anterior limitando visibilidad al nodo en cuestión. Este agente no requiere autenticación y permite a cualquier atacante con acceso a la red del nodo o del host consultarlo.
  - **Mapeo de la red**, por defecto no existe aislamiento de comunicación de red entre los recursos de un clúster, cualquier nodo o coi .Cualquier atacante con acceso a la red podría listar todos los recursos y aplicaciones con el fin de encontrar vulnerabilidades que explotar.
  - **Acceso al dashboard**, atacante con acceso a algún recurso del clúster, podrían usar la red de este para comunicarse con este servicio para listar información de clúster.
  - **Acceso a la metadata del clúster**, atacantes con acceso a un contenedor puede llegar a consultas de obtención de información relativa al host o recursos de este.
- Movimiento lateral
  - **Compartición de recursos**, en entornos basados contenedores existe la posibilidad de ganar acceso a otros contenedores u otros recursos y/o servicios del clúster desencadenando varias de los vectores comentados en los puntos anteriores, partiendo de un contenedor con configuraciones poco restrictivas. Dichos

movimientos pueden ser desencadenados por **definición de roles pocos restrictivos, secretos almacenados en ficheros de configuración, acceso a cuentas de servicio**, entre otros.

- Impacto
    - En cualquiera de los vectores de ataques comentados las consecuencias pueden ser devastadoras. Estas pueden ir desde **denegaciones de servicio, destrucción de información** o **abusar de los recursos del clúster** (véase, crear ingentes recursos dentro de un proveedor de nube para hacer perder dinero a la entidad comprometida)

Enumerando algunas de vulnerabilidades registradas: [34]

- **CVE-2020-8551:** En las versiones de Kubernetes 1.15.0-1.15.9, 1.16.0-1.16.6, y 1.17.0-1.17.2, a través de la API de kubelet se puede desplegar un DoS (Denegación de servicio).
- **CVE-2017-1002101:** En versiones Kubernetes 1.3.x, 1.4.x, 1.5.x, 1.6.x y versiones anteriores a 1.7.14, 1.8.9 y 1.9.4, contenedores (incluso pods no privilegiados) con cualquier tipo de volúmenes montados con *subPath* pueden acceder a ficheros/directorios fuera del volumen, incluido el árbol de directorios del host.

2.3.1. Buenas prácticas en Kubernetes

Tras lo visto en el punto anterior, en este apartado nos centraremos en enumerar una serie de consideraciones, a modo de buenas prácticas, para tener en cuenta en un despliegue inicial de un clúster de Kubernetes con el fin de proteger la infraestructura y mitigar posibles vulnerabilidades.

- Uso de **RBAC** *(Role-based Access Control)*: mecanismo de control de acceso que se utiliza en Kubernetes para limitar el acceso a los recursos del clúster. En Kubernetes, el RBAC se utiliza para definir roles y asignarlos a usuarios o grupos de usuarios. Cada rol tiene un conjunto de permisos que determinan qué acciones pueden realizar los usuarios asociados al rol. Por ejemplo, se podría definir un rol "desarrollador" con permisos para crear y eliminar contenedores, y asignar ese rol a todos los desarrolladores del equipo. De esta manera, los desarrolladores tendrían acceso a los recursos necesarios para realizar sus tareas, pero no podrían realizar acciones que no estén autorizadas, como modificar la configuración del clúster o eliminar recursos críticos.

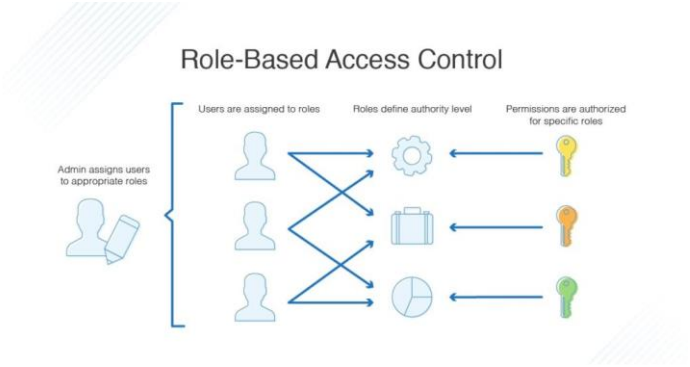

Figura 11 – Role-Based Access Control
Ref. [21]

- Uso de **namespaces**: como ya hemos comentado en el punto 2.1, son una forma de dividir los recursos de un clúster. Cada namespace es una entidad aislada que contiene un conjunto de recursos de *Kubernetes*, como pods, servicios y volúmenes, permitiendo su gestión y organización de forma independiente. El uso de namespaces en *Kubernetes* es recomendable por varias razones:
    - **Organización**: los namespaces permiten organizar los recursos del clúster en espacios de nombres lógicos, lo que facilita su gestión y el seguimiento de los cambios en el tiempo.
    - **Aislamiento**: los namespaces proporcionan un nivel adicional de aislamiento entre los diferentes equipos o proyectos que utilizan el clúster. Esto permite a los usuarios trabajar de manera aislada sin interferir con los recursos de otros equipos.
    - **Control de acceso**: los namespaces también se pueden utilizar para controlar el acceso a los recursos del clúster. Por ejemplo, se pueden asignar roles y permisos de acceso a cada namespace de manera individual, lo que permite a los administradores controlar quién tiene acceso a qué recursos.
    - **Escalabilidad**: los namespaces también son útiles para escalar el clúster de manera más eficiente. Por ejemplo, se pueden crear varios namespaces para diferentes equipos o proyectos y asignar a cada uno un número diferente de recursos, lo que permite ajustar el tamaño del clúster de manera más precisa a las necesidades de cada equipo o proyecto.

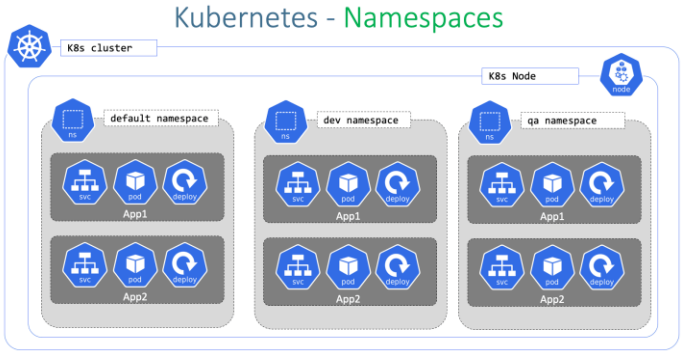

Figura 12 – Kubernetes Namespaces
Ref. [22]

- Uso del **limitador de recursos**: nos habilita la posibilidad de concretar los recursos necesarios y/o límites para un pod. Desde el punto de vista del desarrollo nos permite controlar que un pod no capitalice todos los recursos de nuestro clúster por cualquier motivo o que un atacante no explote esa posible vía, ya sea aprovechando nuestros recursos para tareas como la minería criptográfica o intentando tener pérdidas por facturación monstruosa si nuestro clúster estuviera en nube pública.

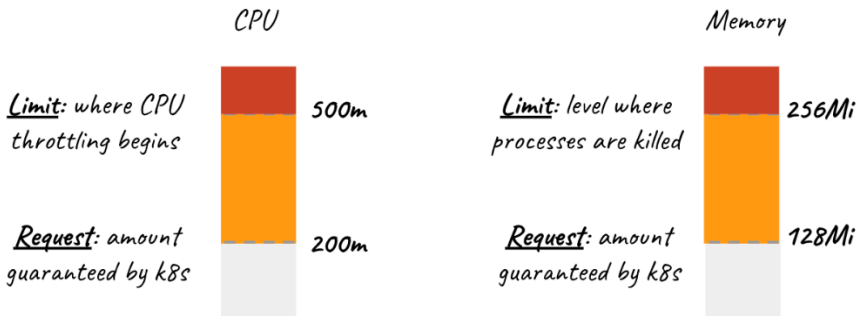

Figura 13 – Limite de recursos en Kubernetes
Ref. [24]

- Mantener *Kubernetes* **actualizado**: como cualquier software, pero más aún en software de código abierto, es importante estar al tanto de los parches y nuevas versiones que se van publicando. Es muy común que estos parches suelan incluir soluciones a fallos de seguridad o vías potenciales de ataque detectadas.
- Aplicar **políticas de red**: por defecto, todos los contenedores pueden interactuar entre sí, además del ya comentado uso de namespaces, unas políticas de red restrictivas pueden protegernos antes agentes maliciosos.

- Uso de etiquetas [16][17]: aparte de proporcionarnos facilidades para organizar nuestros recursos y el seguimiento de nuestra metadata, facilitando el análisis de intrusiones.
- Uso de **secretos**: *Kubernetes* nos proporciona una funcionalidad para almacenar información sensible en formato de *clave-valor*, donde dicho valor está encriptado, evitando así tener que codificar dicha información en el código de la aplicación o recursos usados.

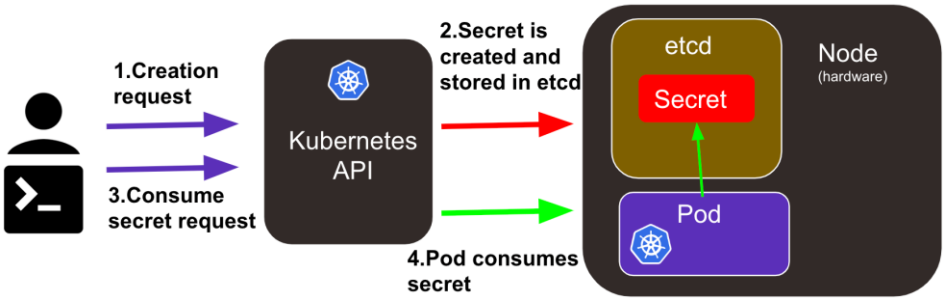

Figura 14 – Secretos en Kubernetes
Ref. [23]

Adicionalmente, se pueden tener en cuenta una serie de consideraciones extra no relacionadas con la configuración de clúster propiamente. [18]

- **Escaneo de imágenes**: es muy importante que las imágenes que usemos en nuestros contenedores estén libres de vulnerabilidades, así como seleccionar las que contengan el software mínimo necesario para nuestra tarea. Existen herramientas para el análisis de estas en todas las fases del flujo de CI/CD como podría ser *Docker Scan* [19]. Dicha herramienta nos proporciona un análisis de las vulnerabilidades y exposiciones comunes (CVE).
- **Herramientas de detección de amenazas**: existen múltiples soluciones para este punto, entre ellas la mencionada en el punto 2.3, pero estas se pueden agrupar en dos grandes grupos *prevención y detección* de intrusiones.
- **Service Mesh:** lo cual proporciona un marco base para el despliegue de aplicaciones basadas en microservicios aportando observabilidad, manejo de las comunicaciones y seguridad de forma transparente (sin necesidad de configuración extra, solo desplegar el servicio) a estos, gracias a la encriptación del tráfico de red.[20]

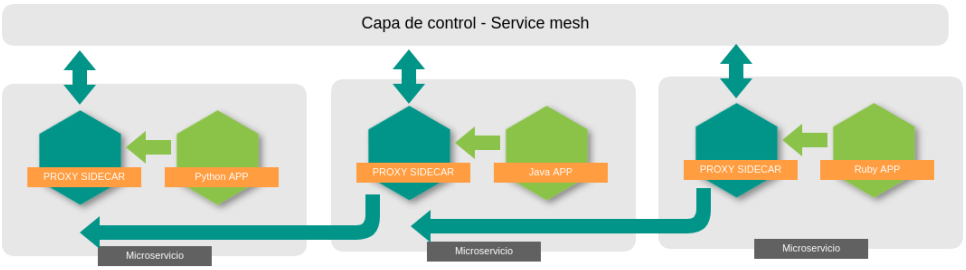

Figura 15 – Ejemplo Service Mesh
Ref. [25]

2.3.2. Service Meshes

Service Mesh nace como una solución al cambio de paradigma que supone las aplicaciones basadas en microservicios, donde estos tienen la necesidad de comunicarse entre sí a través de la red y donde cada "porción" de la aplicación puede estar en un sistema distinto (sistemas distribuidos).

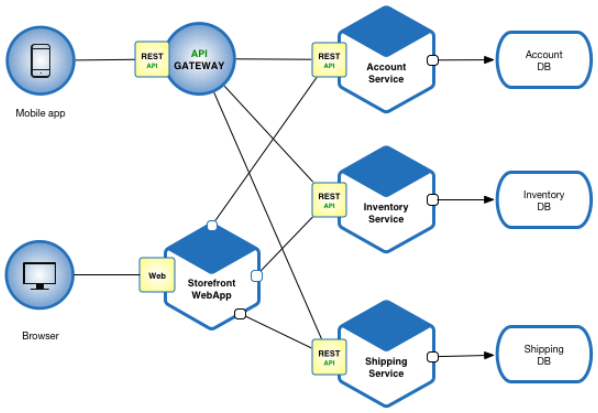

Figura 16 – Arquitectura Microservicios
Ref.[40]

La comunicación entre dos microservicios no se realiza de forma de directa, sino que existe una pareja de proxis intermedios que se encargan del intercambio de dicho tráfico, dotando de todos los datos necesarios para que se efectúe este y encriptándolo sin necesidad de que el desarrollador tenga que implementarlo en la aplicación. Adicionalmente, proporciona funcionalidades de observabilidad (obtención de métricas, logs, trazas, etc), políticas de red, manejo de certificados y enrutamiento del tráfico entre otras.

En el mercado existen diferentes implementaciones de **Service Mesh**, como pueden ser Linkerd, Consul o Istio. Para ejemplificar una implementación de

Service Mesh hemos decido optar por ***Istio***, una de las más populares en el momento de la realización del estudio.

Aplicación	Forks	Marcado como favorito (star)
Linkerd	1100	9200
Consul	4200	25800
Istio	6900	32100

Tabla 4 – Comparativa con datos de Github
Ref:[50][51][52]

Istio es una aplicación desarrollada originalmente por el esfuerzo conjunto de Google, IBM y Lyft, en la actualidad es mantenida por la comunidad. **Istio** proviene del griego y significa *"navegar"*. Algunas de las funcionalidades más destacables de Istio son: [39]

- **Istio Ingress Gateway:** componente que hace de puerta de entrada o proxy del clúster en su conjunto proporcionando un mayor control del tráfico que llega a este y realizar un rutado óptimo de este.
- **Gestión de certificados:** posibilidad de actuar como **CA** (autoridad de certificación), asegurando una comunicación segura basada en **TLS** (Transport Layer Security).
- **Descubrimiento de servicios automático:** posibilidad de detectar de forma automática los *endpoints* de cada microservicio sin necesidad de que el desarrollador deba configurarlo, asignado así el proxy correspondiente.
- **Obtención de métricas:** posibilidad de analizar y obtener métricas del tráfico que circula a través de los proxies desplegados

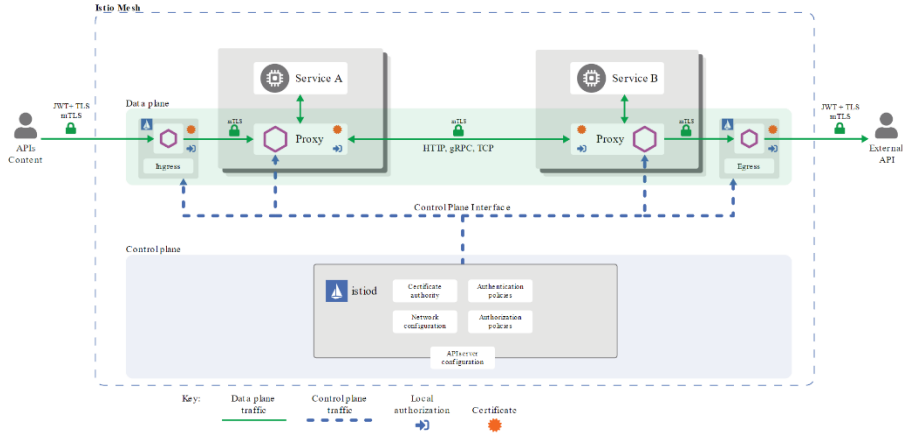

Figura 17 – Arquitectura de seguridad de Istio
Ref.[39]

2.4. Herramientas de terceros en Kubernetes

Existen numerosas herramientas hoy en día que extienden las funcionalidades de Kubernetes. Centrándonos en la seguridad, enumeraremos algunas de las más populares en el momento de la realización del estudio:

- **Popeye**: permite detectar problemas de rendimiento y recursos en el clúster, incluyendo el uso excesivo de memoria y CPU, el agotamiento de los recursos del sistema, la falta de balanceo de carga y los errores de configuración. Proporciona informes y recomendaciones para ayudar a identificar y solucionar problemas de rendimiento en el clúster., pero no realiza ninguna corrección.[36]
- **Kyverno**: herramienta de gestión de políticas para clústeres de Kubernetes. Permite definir, aplicar y validar políticas para los recursos de Kubernetes de manera sencilla y eficiente, así como establecer políticas para garantizar estándares de seguridad y cumplimiento, mejorar la calidad del código y asegurar la integridad y consistencia de los recursos del clúster. Kyverno se ejecuta como una aplicación en el clúster y proporciona una interfaz de línea de comandos y una API para la gestión y aplicación políticas a los recursos de Kubernetes. Adicionalmente, actúa como *Admission Controller* controlando y/o restringiendo cualquier petición que se realice a la API de Kubernetes, pudiendo sincronizar dicha configuración entre namespaces.[37]

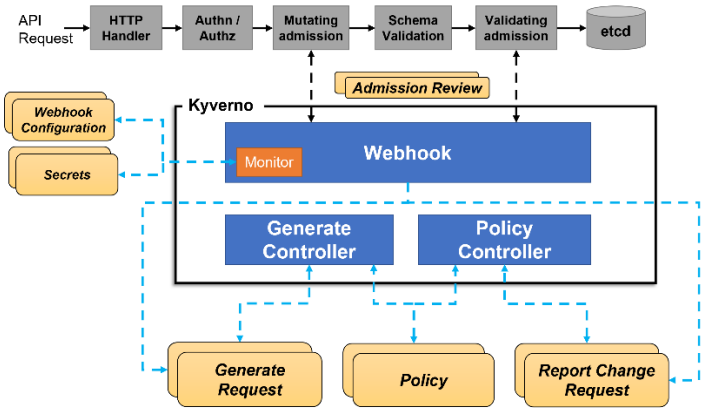

Figura 18 – Funcionamiento de Kyverno
Ref. [37]

- **Calico**: proporciona una solución de securización de red basada en políticas. Además, realiza control de tráfico y encriptación de este dentro del clúster generando alertas y reportes. Se basa en la tecnología de enrutamiento de red de alta velocidad y bajo consumo de recursos llamada **BGP** (Border Gateway Protocol). ***BGP*** es un protocolo de enrutamiento de *capa 3*, el cual funciona evaluando el costo de cada posible ruta entre dos redes y eligiendo la ruta más corta o económica. Utiliza BGP para distribuir información de enrutamiento entre los nodos del clúster de Kubernetes y para asegurar que el tráfico se dirija a través del clúster de manera eficiente y segura [38]
- **Istio**: como ya hemos comentado en el punto 2.3.2, es una herramienta que proporciona una serie de características clave para las aplicaciones basadas en microservicios, como el enrutamiento de tráfico, la gestión de la conectividad y la seguridad, mediante un conjunto de componentes de software que se instalan en el clúster de Kubernetes y se integran con las aplicaciones.
- **Falco**: herramienta de monitoreo y detección de amenazas en tiempo real usada para detectar y alertar sobre comportamientos anómalos o sospechosos en un clúster de Kubernetes.

Para el desarrollo del trabajo hemos elegido la herramienta **Falco** para la realización del escenario práctico.

2.4.1. Falco

Como hemos comentado en la introducción a *Kubernetes* en el punto 2.2.1, existen multitud de herramientas que añaden funcionalidades a la infraestructura

desplegada con esta tecnología. En este apartado se ha elegido centrar el foco en una herramienta de monitoreo y detección de amenazas en tiempo real, concretamente **Falco**.

Falco es una herramienta desarrollada en 2016 por la empresa Sysdig, siendo el primer proyecto de seguridad en adherirse a la **CNCF** *(Cloud Native Computing Foundation)* [10][11]. Con el fin de detectar anomalías, **Falco** nos permite realizar comprobaciones de cualquier petición que se realice a la API de Kubernetes o realice cualquier servicio, así como cualquier llamada de sistema al OS anfitrión.

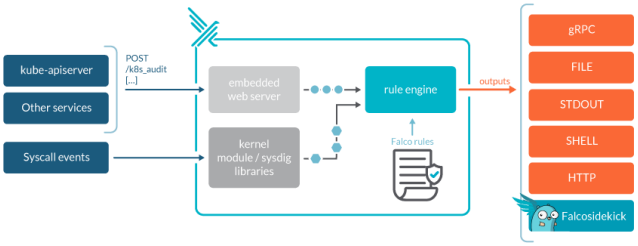

Figura 19 – Funcionamiento de Falco
Ref. [11]

Falco es compuesto por varios componentes principales:

- **Ejecutable:** es la herramienta CLI *(Commad-Line Interface)* con que interactuamos directamente
- **Configuración:** conjunto de archivos que definen el conjunto de reglas y el funcionamiento de las alertas
- **Drivers:** software necesario para que Falco reciba información del sistema (normalmente *libscap* y *libsinsp*)
- **Plugins:** software que extiende las funcionalidades de Falco

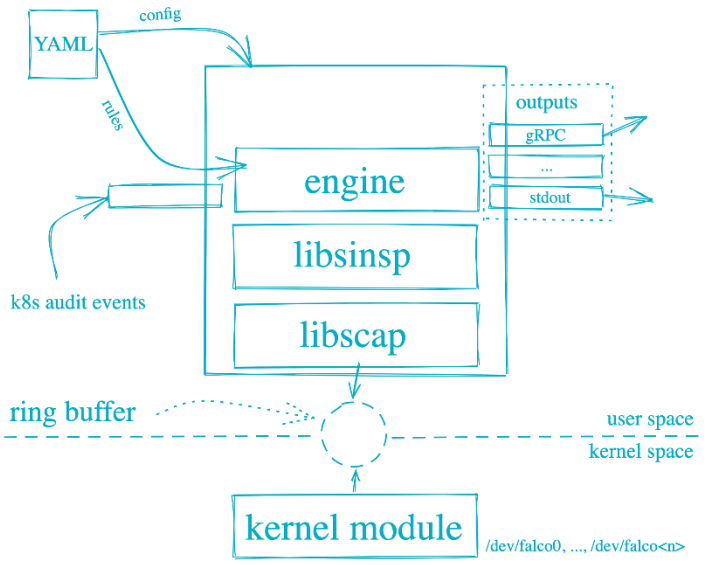

Figura 20 – Arquitectura de Falco
Ref. [10]

Falco tiene una gran flexibilidad permitiéndonos definir una serie de chequeos a modo de reglas presentadas en ficheros YAML, así como como la generación de alertas ante la detección de alguna anomalía. Por defecto, realiza una serie de comprobaciones que cubre muchos de las vulnerabilidades más comunes ya comentadas en la sección de OWASP y *MITRE ATT&CK®:*

- Escalado de privilegios
- Cambios en los namespace (*agrupación de objetos aislados dentro de un clúster*)
- Creación de enlaces simbólicos
- Cambio de privilegios en ficheros
- Lectura/escritura en directorios críticos (etc, usr/bin, …)
- Ejecución de binarios

En caso de que se detecte alguna anomalía **Falco** generaría una alerta, la cual podemos configurar para que sea publicada en:

- Salida estándar
- Un fichero
- Syslog
- API o HTTP endpoint

Para el escenario que propondremos en el siguiente capítulo optaremos por publicarlo por la salida estándar para facilitar la demostración.

3. Resultados

En esta sección detallaremos todo lo relativo a la puesta en prácticas de algunas de las buenas prácticas comentadas en el punto 2.2.1 en un clúster de nueva creación y la ejemplificación de despliegue y prueba de la herramienta *Falco*.

Tras la valoración de las diferentes posibilidades de escenario base se ha optado por un clúster local, concretamente desplegado con **KiND** [42] por su flexibilidad y simpleza, descartando así soluciones en nuble pública, las cuales agregarían una capa extra de estudio al trabajo. Otra herramienta para el uso local de *Kubernetes* sería **minikube**.

Minikube y *KiND* son herramientas utilizadas para ejecutar entornos de *Kubernetes* localmente. Aunque ambas herramientas tienen el mismo propósito, hay algunas diferencias notables entre ellas:

- **Estructura**: *minikube* ejecuta un único nodo *Kubernetes* en una máquina virtual local, mientras que *KiND* crea un clúster completo de *Kubernetes* con varios nodos en contenedores Docker. Esto significa que *KiND* es más adecuado para probar la funcionalidad del clúster y el escalado de aplicaciones, mientras que *minikube* es más adecuado para pruebas más simples.
- **Instalación**: *minikube* requiere la instalación de una máquina virtual y de *Kubernetes* en su sistema local, mientras que *KiND* se instala como una solución de contenedores y se ejecuta directamente sobre Docker. Esto significa que *KiND* es más fácil de instalar y configurar que *minikube*.
- **Versión de Kubernetes**: ambos soportan diferentes versiones de *Kubernetes*, lo que puede ser un factor para considerar al elegir qué herramienta utilizar. Por ejemplo, *minikube* soporta una gama más amplia de versiones de *Kubernetes*, mientras que *KiND* se centra en las versiones más recientes.

3.1. Despliegue de clúster

Como máquina anfitriona encargada de alojar el clúster se usará un equipo personal con Windows 10 como sistema operativo. Para la instalación de *KiND* [42] haremos uso del gestor de paquetes *Chocolatey.* **Chocolatey** es una herramienta *open-source* que permite la instalación de binarios y dependencias de forma automática y sencilla. Dichas dependencias están respaldadas por la empresa *VirusTotal* encargada de verificar la seguridad de estas.

1. Instalación de Chocolatey [41]

En una consola privilegiada y siguiendo la documentación oficial, ejecutamos los siguientes comandos donde permitimos, ante un escenario de políticas de lanzamiento de script restrictivas, la ejecución de script remotos.

```
PS C:\Users\drave> Get-ExecutionPolicy
Restricted
PS C:\Users\drave> Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol
= [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('h
ttps://community.chocolatey.org/install.ps1'))
Forcing web requests to allow TLS v1.2 (Required for requests to Chocolatey.org)
Getting latest version of the Chocolatey package for download.
Not using proxy.
Getting Chocolatey from https://community.chocolatey.org/api/v2/package/chocolatey/1.2.0.
Downloading https://community.chocolatey.org/api/v2/package/chocolatey/1.2.0 to C:\Users\drave\AppData\Local\Temp\chocol
atey\chocoInstall\chocolatey.zip
Not using proxy.
Extracting C:\Users\drave\AppData\Local\Temp\chocolatey\chocoInstall\chocolatey.zip to C:\Users\drave\AppData\Local\Temp
\chocolatey\chocoInstall
Installing Chocolatey on the local machine
  Setting ChocolateyInstall to 'C:\ProgramData\chocolatey'
PS C:\Users\drave> choco --version
1.2.0
```

Figura 21 – Instalación de Chocolatey

La versión de *Chocolatey* empleada para el total desarrollo de la demo fue la **1.2.0**.

2. Instalación de KiND [42]

En una consola privilegiada y siguiendo la documentación oficial, lanzamos el comando `choco install kind`, con el fin de instalar todas las dependencias necesarias incluido el motor de Docker y Docker-Desktop. A su vez, Docker hace uso del **WSL** (Windows Subsystem Linux), integración nativa de las syscall de Linux en el Kernel de Windows que permite ejecutar un entorno Linux sin la necesidad de la creación de máquinas virtuales extra.

```
PS C:\Users\drave> choco install kind
Chocolatey v1.2.0
Installing the following packages:
kind
By installing, you accept licenses for the packages.
Progress: Downloading docker-desktop 4.15.0... 100%
Progress: Downloading dotnetfx 4.8.0.20220524... 100%
Progress: Downloading chocolatey-dotnetfx.extension 1.0.1... 100%
Progress: Downloading KB2919355 1.0.20160915... 100%
Progress: Downloading KB2919442 1.0.20160915... 100%
Progress: Downloading kind 0.17.0... 100%

Chocolatey installed 6/6 packages.
 See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).

Installed:
 - kb2919355 v1.0.20160915
 - dotnetfx v4.8.0.20220524
 - kb2919442 v1.0.20160915
 - chocolatey-dotnetfx.extension v1.0.1
 - kind v0.17.0
```

Figura 22 – Instalación de KiND y sus dependencias

Se realiza la instalación de *KiND* en su versión **0.17.0**, la empleada en la totalidad del desarrollo de la demo.

3. Instalación de Kubectl [53]

Con el fin de poder interactuar mediante consola con el clúster desplegado necesitamos tener instalado ***kubectl***, la consola de comando nativa de *Kubernetes*. Siguiendo la documentación oficial de *Kubernetes*:

Figura 23 – Instalación de Kubectl

Se realiza la instalación de *kubetcl* en su versión cli **1.25.4** y server **1.25.3**, la empleada en la totalidad del desarrollo de la demo.

4. Instalación de Helm [43]

Con el fin de poder facilitar la instalación de recursos en *Kubernetes* instalaremos el gestor de paquetes más popular en *Kubernetes*, **Helm**. *Helm*, es una herramienta *open-source* apoyada por la comunidad y respalda por la **CNCF**.

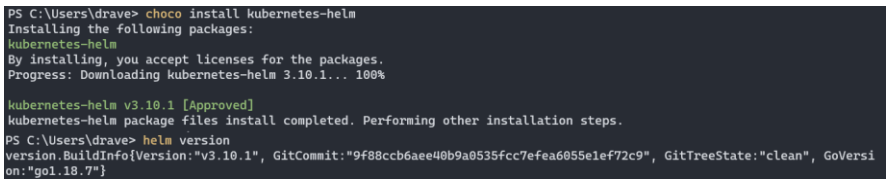

Figura 24 – Instalación de Helm

Se realiza la instalación de *Helm* en su versión **3.10.1**, la empleada en la totalidad del desarrollo de la demo.

5. Despliegue del clúster

Con el fin de ilustrar muchas de la mayoría de los errores de configuración y vulnerabilidades más comunes se ha optado por desplegar el clúster que nos propone el proyecto **Kubernetes Goat** [44]. **Kubernetes Goat** es un proyecto que permite desplegar un entorno de *Kubernetes* con los errores de configuraciones, errores de seguridad y vulnerabilidades de entornos reales más comunes, creado por *madhuakula* [45],.

El proyecto nos proporciona una serie de comandos y configuración en formato *yaml* para su despliegue inicial. [Anexo 1]. Una vez descargados los recursos y siguiendo la guía de instalación, lanzamos los siguientes comandos:

```
cd kubernetes-goat\platforms\kind-setup
```

```
kind create cluster --config kind-cluster-setup.yaml --name kubernetes-goat-cluster
```

```
PS G:\My Drive\UOC\TFM\kubernetes-goat\platforms\kind-setup> kind create cluster --config kind-cluster-setup.yaml --name
 kubernetes-goat-cluster
Creating cluster "kubernetes-goat-cluster" ...
 ✓ Ensuring node image (kindest/node:v1.25.3)
 ✓ Preparing nodes
 ✓ Writing configuration
 ✓ Starting control-plane
 ✓ Installing CNI
 ✓ Installing StorageClass
Set kubectl context to "kind-kubernetes-goat-cluster"
You can now use your cluster with:

kubectl cluster-info --context kind-kubernetes-goat-cluster

Not sure what to do next?  Check out https://kind.sigs.k8s.io/docs/user/quick-start/
PS G:\My Drive\UOC\TFM\kubernetes-goat\platforms\kind-setup> kubectl cluster-info --context kind-kubernetes-goat-cluster

Kubernetes control plane is running at https://127.0.0.1:50901
CoreDNS is running at https://127.0.0.1:50901/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

Figura 25 – Despliegue de Kubernetes Goat

6. Comprobaciones y configuración final

Una vez desplegado el entorno, realizamos una serie de comprobaciones para si todo está funcionando correctamente.

```
PS G:\My Drive\UOC\TFM\kubernetes-goat> kubectl get pods
NAME                                        READY   STATUS      RESTARTS   AGE
batch-check-job-nd29c                       0/1     Completed   0          24m
build-code-deployment-57974868c5-8qsmp      1/1     Running     0          23m
health-check-deployment-77c7b6686-7sd5n     1/1     Running     0          23m
hidden-in-layers-ncbd8                      1/1     Running     0          23m
internal-proxy-deployment-599f567fb4-zn6g4   2/2     Running     0          23m
kubernetes-goat-home-deployment-74cd49bf79-n8zk2   1/1     Running     0          23m
metadata-db-86579bcb65-xd8wg                1/1     Running     0          24m
poor-registry-deployment-cc975f599-r4rzl    1/1     Running     0          23m
system-monitor-deployment-756598dbd-7bb6l   1/1     Running     0          23m
PS G:\My Drive\UOC\TFM\kubernetes-goat> kubectl get services
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
build-code-service            ClusterIP   10.96.171.184   <none>        3000/TCP         24m
health-check-service          ClusterIP   10.96.147.91    <none>        80/TCP           23m
internal-proxy-api-service    ClusterIP   10.96.57.158    <none>        3000/TCP         23m
internal-proxy-info-app-service   NodePort    10.96.152.114   <none>        5000:30003/TCP   23m
kubernetes                    ClusterIP   10.96.0.1       <none>        443/TCP          38m
kubernetes-goat-home-service  ClusterIP   10.96.206.216   <none>        80/TCP           23m
metadata-db                   ClusterIP   10.96.125.58    <none>        80/TCP           24m
poor-registry-service         ClusterIP   10.96.235.18    <none>        5000/TCP         23m
system-monitor-service        ClusterIP   10.96.33.74     <none>        8080/TCP         23m
```

Figura 26 – Comprobaciones tras el despliegue

Finalmente, con la idea de que la aplicación sea accesible desde fuera de clúster debemos realizar una *redirección de puertos*. El script que se nos proporcionaba

para esta tarea se ha traducido a powershell *access-kubernetes-goat.ps1* ya que se encontraba en bash. [Anexo 2]

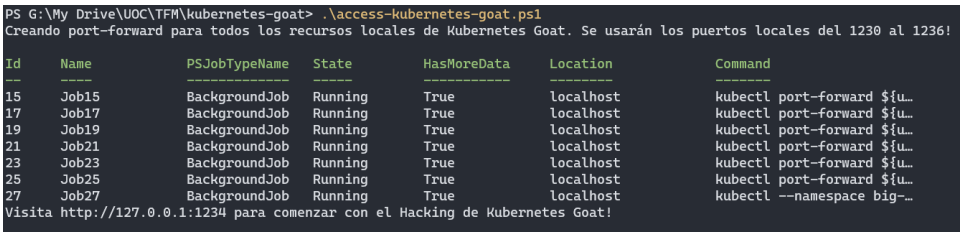

Figura 27 – Redireccionamientos de puertos

Las especificaciones del escenario son:

- **Máquina anfitriona**:
  - **Sistema operativo**:
  - **CPU**: Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz x 8
  - **RAM**: 16GB

3.2. Aplicación de buenas prácticas

En este apartado nos centraremos en ilustrar algunas de las amenazas y riegos que podemos encontrar al desplegar un clúster comentadas en el punto 2.3.1 y la corrección de estas.

3.2.1. Mala configuración de RBAC

En el mundo real es común ver que los desarrolladores y equipos de DevOps suelen proporcionar privilegios adicionales de los necesarios. Esto proporciona a los atacantes más control y privilegios de los que se pretendía. **Goat Kubernetes** nos propone un escenario donde se despliega un pod llamado hunger-check el cual, en su configuración, define un rol privilegiado ***(secret-reader)*** asociado al namespace ***(big-monolith)*** que lo alberga.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: big-monolith
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: big-monolith
  name: secret-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["*"] # all the resources
  verbs: ["get", "watch", "list"]
```

Figura 28 – Definición namespace y role

Para realizar el despliegue del escenario ejecutamos el comando `kubectl apply -f scenarios\hunger-check\deployment.yaml`.

```
PS G:\My Drive\UOC\TFM\kubernetes-goat> k get pods -n big-monolith
NAME                                     READY   STATUS    RESTARTS   AGE
hunger-check-deployment-57896c4d6c-m2rzj   1/1     Running   0          22m
```

Figura 29 – Pod escenario de RBAC mal configurados

Una vez desplegado y dentro del contenedor con una consola con bash mediante el comando `kubectl -n big-monolith exec -it hunger-check-deployment-57896c4d6c-m2rzj -- bash`, comenzamos el ataque.

Por defecto, *Kubernetes* guarda la información de tokens y cuentas de servicio en la ruta ***/var/run/secrets/kubernetes.io/serviceaccount/*** de cada container para poder comunicarse con los servicios del clúster. Listado los ficheros de la ruta encontramos la información de necesaria para efectuar consultas a la API de *Kubernetes*.

```
root@hunger-check-deployment-57896c4d6c-m2rzj:/# ls -la /var/run/secrets/kubernetes.io/serviceaccount/
total 4
drwxrwxrwt 3 root root  140 Jan  1 19:51 .
drwxr-xr-x 3 root root 4096 Jan  1 19:51 ..
drwxr-xr-x 2 root root  100 Jan  1 19:51 ..2023_01_01_19_51_33.2344692059
lrwxrwxrwx 1 root root   32 Jan  1 19:51 ..data -> ..2023_01_01_19_51_33.2344692059
lrwxrwxrwx 1 root root   13 Jan  1 19:51 ca.crt -> ..data/ca.crt
lrwxrwxrwx 1 root root   16 Jan  1 19:51 namespace -> ..data/namespace
lrwxrwxrwx 1 root root   12 Jan  1 19:51 token -> ..data/token
```

Figura 30 – Ficheros asociados a la cuenta de servicio

Definiendo una serie de variables de entorno para facilitarnos la tarea de descubrimiento, realizamos la consulta de los secretos asociados al namespace ***big-monolith*.**

```
$> export APISERVER=https://${KUBERNETES_SERVICE_HOST}
$> export SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount
$> export NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)
$> export TOKEN=$(cat ${SERVICEACCOUNT}/token)
$> export CACERT=${SERVICEACCOUNT}/ca.crt
```

Por defecto, *Kubernetes* define una serie de variable de entorno en cada contenedor. Sabiendo esto y con la información proporcionada por los ficheros del directorio enumeramos el **namespace**, **token** y el **certificado de autoridad** para firmar las peticiones contra la API.

```
PS C:\Users\drave> k -n big-monolith exec hunger-check-deployment-57896c4d6c-m2rzj -- printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=hunger-check-deployment-57896c4d6c-m2rzj
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
HOME=/root
```

Figura 31 – Variables de entorno por defecto en contenedor de k8s

Realizando la siguiente petición mediante **curl** conseguimos listar todos los secretos almacenados asociados al namespace ***big-monolith***:

```
$> curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X
GET ${APISERVER}/api/v1/namespaces/${NAMESPACE}/secrets
```


```
        "manager": "kubectl-client-side-apply",
        "operation": "Update",
        "apiVersion": "v1",
        "time": "2023-01-01T19:51:33Z",
        "fieldsType": "FieldsV1",
        "fieldsV1": {
          "f:data": {
            ".": {},
            "f:k8swebhookapikey": {}
          },
          "f:metadata": {
            "f:annotations": {
              ".": {},
              "f:kubectl.kubernetes.io/last-applied-configuration": {}
            }
          },
          "f:type": {}
        }
      }
    ]
  },
  "data": {
    "k8swebhookapikey": "azhzLWdvYXQtZGZjZjYzMDUzOTU1M2VjZjk1ODZmZGZkYTE5NjhmZWM="
  },
  "type": "Opaque"
 }
]
```


Figura 32 – Respuesta API

donde con la ayuda del comando **base64** con la bandera **-d** *(descodificar)* podemos obtener el texto plano el secreto en cuestión.

```
root@hunger-check-deployment-57896c4d6c-m2rzj:/# $(echo "azhzLWdvYXQtZGZjZjYzMDUzOTU1M2VjZjk1ODZmZGZkYTE5NjhmZWM=" | base64 -d)
bash: k8s-goat-dfcf630539553ecf9586fdfda1968fec: command not found
```

Figura 33 – Secreto descodificado

El problema de este escenario ha sido la definición del **rol privilegiado** en el despliegue del pod. Una posible solución sería la definición y asociación de un rol con los privilegios mínimos necesarios para la aplicación que alberga. Por ejemplo, limitando las acciones permitidas a la inspección de los pods del namespace.

```yaml
kind: Namespace
metadata:
  name: sec-big-monolith
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: sec-big-monolith
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Figura 34 – Escenario securizado

Tras realizar el despliegue del escenario securizado e intentando realizar la consulta de secretos anterior, siguiendo los pasos anteriores, nos encontramos que el recurso es inaccesible en este caso.

```
root@hunger-check-resolve-deployment-59c9849897-d62vc:/var/run/secrets/kubernetes.io/serviceaccount# curl --cacert ${CACERT} --
header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api/v1/namespaces/${NAMESPACE}/secrets
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "secrets is forbidden: User \"system:serviceaccount:sec-big-monolith:sec-big-monolith-sa\" cannot list resource \"
secrets\" in API group \"\" in the namespace \"sec-big-monolith\"",
  "reason": "Forbidden",
  "details": {
    "kind": "secrets"
  },
  "code": 403
}
```

Figura 35 – Error de acceso a los secretos

En este ejemplo se ha visto la importancia comentada en el punto 2.3.1 de asociar a cada pod los mínimos privilegios posibles y de realizar una buena gestión de permisos dentro del clúster.

3.2.2. Limitación de recursos

Limitar los recursos en los despliegues de *Kubernetes* es importante porque le permite asegurar que cada aplicación tenga suficientes recursos para ejecutarse de manera efectiva. Además, como ya hemos comentado en el punto 2.3.1, al limitar los recursos, puede evitar que una aplicación consuma demasiados recursos y cause problemas de rendimiento o inestabilidad en el sistema. **Goat Kubernetes** nos propone un nuevo escenario con el pod del ejercicio anterior donde se producirá una capitalización de recursos usando el comando ***stress-ng*,** el cual nos permite generar una carga elevada en el sistema al ejecutar varias tareas simultáneamente, lo que permite evaluar cómo el sistema maneja la sobrecarga y si hay algún problema de rendimiento o inestabilidad. Dicho pod será desplegado en el namespace ***(big-monolith)***.

Para realizar el despliegue del escenario ejecutamos el comando `kubectl apply -f scenarios\hunger-check\deployment.yaml`.

```
PS G:\My Drive\UOC\TFM\kubernetes-goat> k -n big-monolith get pods
NAME                                     READY   STATUS    RESTARTS   AGE
hunger-check-deployment-57896c4d6c-67zrj   1/1     Running   0          37s
```

Figura 36 – Pods para escenario de limitación de recursos

Una vez desplegado, revisamos el consumo de recursos del pods con el comando `kubectl --namespace big-monolith top pod hunger-check-deployment-57896c4d6c-67zrj`. En la figura 37 podemos ver la comparativa de recursos, donde la primera línea representa el consumo el reposo frente al consumo en el lanzamiento del comando `stress-ng --vm 2 --vm-bytes 2G --timeout 30s`.

```
PS C:\Users\drave> kubectl --namespace big-monolith top pod hunger-check-deployment-57896c4d6c-67zrj
NAME                                     CPU(cores)   MEMORY(bytes)
hunger-check-deployment-57896c4d6c-67zrj   1m           3Mi
PS C:\Users\drave> kubectl --namespace big-monolith top pod hunger-check-deployment-57896c4d6c-67zrj
NAME                                     CPU(cores)   MEMORY(bytes)
hunger-check-deployment-57896c4d6c-67zrj   306m         2059Mi
```

Figura 37 – Comparativa uso de recursos

La solución propuesta sería la de limitar a nivel de despliegue los recursos asignados a cada contenedor donde el campo **request** representa los recursos mínimos necesarios y el campo **limits** los recursos máximos que puede usar.

```yaml
containers:
- name: hunger-check-resolve
  image: madhuakula/k8s-goat-hunger-check
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
    limits:
      memory: "128Mi"
      cpu: "500m"
```

Figura 38 – Escenario limitado

3.2.3. Escaneo de imágenes

El problema de la minería criptográfica en contenedores de *Kubernetes* es algo realmente relevante [57], es por esto que existe un riesgo en el uso de imágenes de repositorios públicos. **Goat Kubernetes** nos propone un nuevo escenario para ejemplificar esto, donde al realizar el despliega un **Job** (usado para ejecutar

tareas puntuales y asegurar que se completen correctamente) llamado **batch-check-job**.

Para realizar el despliegue del escenario ejecutamos el comando `kubectl apply -f scenarios\batch-check\job.yaml`.

```
PS G:\My Drive\UOC\TFM\kubernetes-goat> k -n sec-big-monolith get jobs
NAME              COMPLETIONS   DURATION   AGE
batch-check-job   1/1           9s         2m21s
PS G:\My Drive\UOC\TFM\kubernetes-goat> k -n sec-big-monolith describe job batch-check-job
Name:             batch-check-job
Namespace:        sec-big-monolith
Selector:         controller-uid=38cfd159-cc44-432c-ad7c-45ca38193971
Labels:           controller-uid=38cfd159-cc44-432c-ad7c-45ca38193971
                  job-name=batch-check-job
Annotations:      batch.kubernetes.io/job-tracking:
Parallelism:      1
Completions:      1
Completion Mode:  NonIndexed
Start Time:       Wed, 04 Jan 2023 21:05:09 +0100
Completed At:     Wed, 04 Jan 2023 21:05:18 +0100
Duration:         9s
Pods Statuses:    0 Active (0 Ready) / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=38cfd159-cc44-432c-ad7c-45ca38193971
           job-name=batch-check-job
  Containers:
   batch-check:
    Image:        madhuakula/k8s-goat-batch-check
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
```

Figura 39 – Descripción del job

La figura 39 nos muestra información detallada del job ejecutado, donde podemos remarcar el uso de la imagen ***madhuakula/k8s-goat-batch-check***. Con la ayuda del comando `docker history madhuakula/k8s-goat-batch-check --no-trunc` podemos inspeccionar el dockerfile de la imagen y es aquí donde detectamos que una capa de la imagen se realiza un ***curl*** a un recurso externo, el cual podría ser la ejecución de un script para lanzar un proceso de minería criptográfica.

```
PS G:\My Drive\UOC\TFM\kubernetes-goat> docker history madhuakula/k8s-goat-batch-check --no-trunc
IMAGE                                                                     CREATED        CREATED BY


          SIZE      COMMENT
sha256:cb43bcb572b74468336c6854282c538e9ac7f2efc294aa3e49ce34fab7a275c7   7 months ago   CMD ["ps" "auxx"]


          0B        buildkit.dockerfile.v0
<missing>                                                                 7 months ago   RUN /bin/sh -c apk add --no-cac
he htop curl ca-certificates    && echo "curl -sSL https://madhuakula.com/kubernetes-goat/k8s-goat-a5e0a28fa75bf42912394
3abedb065d1 && echo 'id' | sh " > /usr/bin/system-startup     && chmod +x /usr/bin/system-startup     && rm -rf /tmp/* #
 buildkit   2.96MB    buildkit.dockerfile.v0
<missing>                                                                 7 months ago   LABEL MAINTAINER=Madhu Akula IN
FO=Kubernetes Goat

          0B        buildkit.dockerfile.v0
<missing>                                                                 9 months ago   /bin/sh -c #(nop)  CMD ["/bin/s
h"]

          0B
<missing>                                                                 9 months ago   /bin/sh -c #(nop) ADD file:5d67
3d25da3a14ce1f6cf66e4c7fd4f4b85a3759a9d93efb3fd9ff852b5b56e4 in /

          5.57MB
```

Figura 40 – Capas de la imagen de docker

Una solución podría ser usar la herramienta de incorporada en Docker-desktop **Docker-scan** y analizar la imagen antes de ser usada. Esta herramienta nos

hace un reporte de las vulnerabilidades que pueda contener la imagen, así como la gravedad de estas y su referencia CVE.

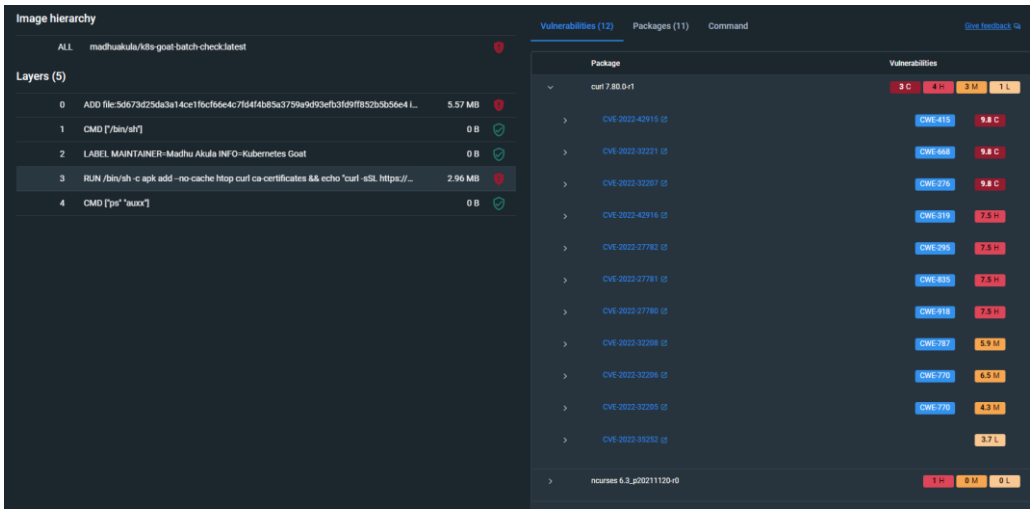

Figura 41 – Reporte Docker-Scan

3.3. Ejemplo con la herramienta Falco

En este apartado nos centraremos en el desarrollo del escenario de despliegue de la herramienta de detección de amenazas, **Falco**, en el clúster de *Kubernetes* y la exploración del fichero **/etc/shadow** de un contenedor del clúster con el fin de provocar un evento en la salida de *Falco*.

En un primer lugar el escenario propuesto se apoyaba en un clúster desplegado con la herramienta *KiND* usando como sistema operativo anfitrión **WSL**. Desafortunadamente *Falco* tiene como dependencias una serie de cabeceras del kernel de Linux, las cuales no se encontraban disponibles para versión del *WSL* de la máquina anfitriona.

Por lo anteriormente comentado se decidió, para este apartado, modificar el escenario base de trabajo realizando un despliegue del clúster con *KiND* sobre **Ubuntu 20.04.1 LTS**, siguiendo pasos similares a los comentados en el punto 3.1.

Las especificaciones del escenario son:

- **Máquina virtual**:
  - **Sistema operativo**: GNU/Linux Ubuntu 20.04.1 LTS
  - **CPU**: Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz x 4
  - **RAM**: 8GB

Seguimos los siguientes pasos para la configuración del nuevo entorno:

1. Instalación de Docker

Se realizó la instalación directamente desde los repositorios que nos propone la documentación de Docker, con los siguientes comandos [54]:

```
$> sudo mkdir -p /etc/apt/keyrings
$> curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
$> sudo apt-get update
$> sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

La versión de Docker empleada para el total desarrollo de la demo fue la ***20.10.22 build 3a2c30b***.

```
uoc@Ubuntu:~$ docker --version
Docker version 20.10.22, build 3a2c30b
```

Figura 42 – Versión Docker en entorno Ubuntu

2. Instalación de KiND

La instalación de *KiND* difiere un poco a la empleada en el escenario original. Siguiendo la documentación oficial y ejecutando los siguientes comandos [42]:

```
$> curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.17.0/kind-linux-amd64
$> chmod +x ./kind
$> sudo mv ./kind /usr/local/bin/kind
```

Se realiza la instalación de *KiND* en su versión ***0.17.0***, la empleada en la totalidad del desarrollo de la demo.

```
uoc@Ubuntu:~$ kind --version
kind version 0.17.0
```

Figura 43 – Versión KiND en entorno Ubuntu

3. Instalación de Kubectl

La instalación de **kubectl** difiere un poco a la empleada en el escenario original. Siguiendo la documentación oficial y ejecutando los siguientes comandos [53]:

```
$> curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
$> sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Se realiza la instalación de kubetcl en su versión cli ***1.26.0*** y server ***1.25.3***, la empleada en la totalidad del desarrollo de la demo.

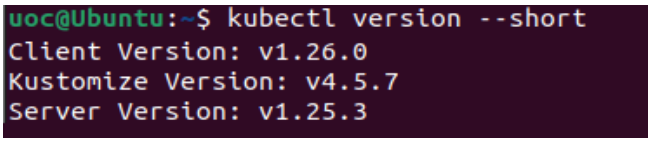

Figura 44 – Versión kubectl en entorno Ubuntu

4. Instalación de Helm

La instalación de *Helm* difiere un poco a la empleada en el escenario original. Siguiendo la documentación oficial y ejecutando los siguientes comandos [43]:

```
$> curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
$> chmod 700 get_helm.sh
$> ./get_helm.shsudo install -o root -g root -m 0755 kubectl
/usr/local/bin/kubectl
```

Se realiza la instalación de *Helm* en su versión ***3.10.3***, la empleada en la totalidad del desarrollo de la demo.

Figura 45 – Version Helm en entorno Ubuntu

5. Instalación herramienta Falco

La instalación de *Falco*, siguiendo la documentación oficial, se llevó a cabo ejecutando los siguientes comandos [55]:

```
$> curl -s https://falco.org/repo/falcosecurity-3672BA8F.asc | apt-key
add -
```

```
$> echo "deb https://download.falco.org/packages/deb stable main" | tee
-a /etc/apt/sources.list.d/falcosecurity.list
$> apt-get update -y
$> apt-get -y install linux-headers-$(uname -r)
$> apt-get install -y falco
```

Se realiza la instalación de *Falco* en su versión ***0.33.1***, la empleada en la totalidad del desarrollo de la demo.

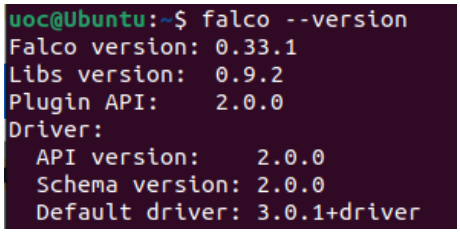

Figura 46 – Versión Falco en entorno Ubuntu

Esto nos permitió tener todas las dependencias necesarias en el OS anfitrión. Para la integración de *Falco* en el clúster nos apoyamos en la herramienta *Helm* lanzando los siguientes comandos:

```
$> helm repo add falcosecurity https://falcosecurity.github.io/charts
$> helm repo update
$> helm install falco --set tty=true falcosecurity/falco --namespace
falco --create-namespace
```

Hay que destacar el uso de la bandera *`--set tty=true`* en la integración de falco en el clúster, la cual nos permite nos habilita la posibilidad de tener un streaming en tiempo real de la salida de la herramienta, en lugar de tener que consultar de forma manual el buffer de salida de esta.

Tras el despliegue con *Helm*, se instancia un pod de falco en cada nodo del clúster. En el escenario usado solo existe un nodo, definido así en la YAML de configuración del proyecto **Kubernetes Goat**.

Figura 47 – Listado de pods de falco

Con el fin de acceder a las aleras que falco reporta en la salida estándar del contenedor, vinculamos la salida estándar de este con la de nuestra consola mediante el siguiente comando y gracias a la bandera configurada en el despliegue con *Helm*, `kubectl logs -n falco -f falco-5cz5h --all-containers`.

Para ejemplificar el reporte de falco, realizamos el despliegue de un pod llamada *hacker-container* mediante el comando `kubectl run -it hacker-container --image=madhuakula/hacker-container - bash`, el cual además de desplegar el clúster invoca una consola interactiva (bandera **-it**) con bash. Una vez dentro de contenedor accedemos al vemos por defecto hemos entrado como **root** y consultado, por ejemplo, el fichero **/etc/shadow** con **cat** provocamos una alerta en la herramienta *Falco*.

En la figura 34, podemos ver ejemplificado lo comentado en el párrafo anterior donde el recuadro rojo delimita el reporte de falco por la salida estándar, el recuadro verde delimita el comando efectuado dentro del contenedor y el recuadro amarilla el cual remarca el nombre del pod/contenedor donde se ha efectuado dicho comando.

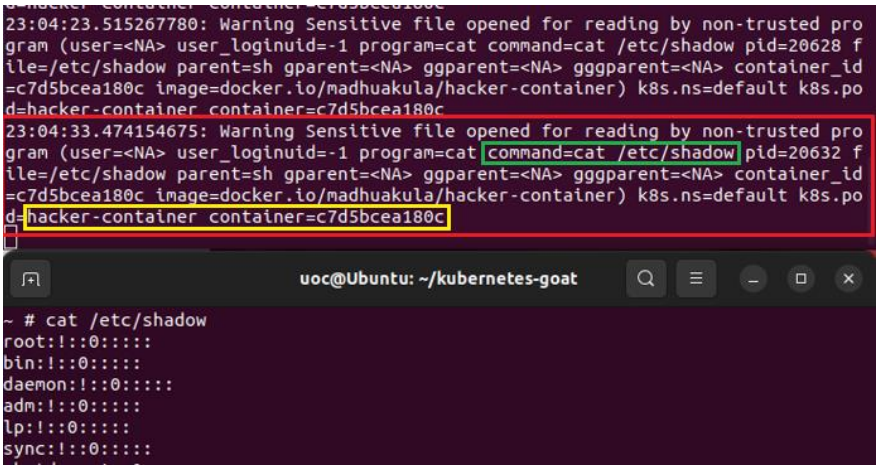

Figura 48 – Evento anómalo publicado por Falco

La alerta producida por *Falco* podría ser enviada a cualquier aplicación de gestión de eventos y ser presentada. Por ejemplo, podría ser tratada por la herramienta **Prometheus** y presentada por la herramienta **Grafana**.

4. Conclusiones y trabajos futuros

El objetivo del trabajo era la realización de un estudio de las herramientas de orquestación de contenedores, concretamente *Kubernetes*, analizando el impacto en el mercado actual de la misma, el cambio de paradigma que supone y los nuevos retos de seguridad asociados.

4.1. Conclusiones

A lo largo de la memoria se han ido presentado diferentes aspectos: arquitecturas distribuidas basadas en contenedores, descripción de *Kubernetes*, retos de seguridad asociados a *Kubernetes*, errores y vectores de ataque comunes en estos entornos, buenas prácticas para su mitigación y herramientas de terceros enfocadas en la seguridad. Finalmente, se ha presentado una serie de escenarios donde ejemplificar el impacto de algunas de los vectores de ataque comunes y el uso de una de las herramientas mencionadas en el punto 2.4.

La planificación y las entregas se han podido cumplir según lo previsto. Esto ha facilitado la obtención del resultado esperado al inicio del desarrollo, cumpliendo así los objetivos marcados.

La primera y más clara conclusión es que el “mundo” de los contenedores es lo suficientemente complejo y extenso como para comprender un estudio en sí mismo. En la memoria se ha intentado dar pinceladas de la tecnología con el fin de llegar a la realización de los escenarios prácticos y esto se refleja en la carga teórica de la misma sirviendo para solventar la falta de experiencia en esta tecnología, la cual la motivación principal para la elección del tema.

Enumerando las conclusiones sobre la seguridad en *Kubernetes*:

- *Kubernetes* proporciona una gran cantidad de características de seguridad integradas, como autenticación y autorización, encriptación de tráfico de red y control de acceso a recursos.
- Es importante configurar adecuadamente la seguridad en *Kubernetes*, ya que el sistema se basa en la confianza en las identidades y las autorizaciones de los usuarios y los componentes del sistema.
- Es recomendable utilizar un enfoque de "defensa en profundidad" para proteger contra posibles vulnerabilidades en *Kubernetes*. Esto implica

implementar varias medidas de seguridad en diferentes capas del sistema.

- Es esencial mantener actualizado el sistema *Kubernetes* para aprovechar las últimas correcciones de vulnerabilidades y mejoras de seguridad.
- Es importante monitorear de cerca el clúster de *Kubernetes* y detectar posibles problemas de seguridad a medida que surjan.

En general, *Kubernetes* ofrece una buena seguridad integrada, pero requiere una configuración y un mantenimiento adecuados para garantizar la protección adecuada de las aplicaciones y los datos.

4.2. Líneas de mejora

Una de las cuestiones que se descartaron durante el desarrollo de la memoria fue el despliegue del escenario práctico en un entorno de nube pública. Esta decisión se tomó debido a la falta de experiencia en estos entornos, el grado extra de complejidad que agregaba y la limitación temporal propia de este trabajo. Es por esto, que uno de los puntos de mejora de este sería su integración y despliegue en un entorno de nube pública, donde se podría realizar una comparativa de las diferentes soluciones que ofrecen los principales proveedores y las diferencias de desarrollo en cada uno de estos.

Adicionalmente, se podría extender el escenario de la herramienta **Falco** realizando una integración de las alertas generadas con herramientas como la herramienta gestión de eventos y visualización como se comentó en el párrafo final del punto 3.3.

5. Glosario

SaaS: Software as a Service
CCEG: Competencia de compromiso ético y global
CI/CD: Integración continua y despliegue continuo
K8s: Kubernetes
CNCF: Cloud Native Computing Foundation
CLI: Commad-Line Interface
GDPR: General Data Protection Regulation
RBAC: Role-based Access Control
Namespace: agrupación de objetos aislados dentro de un clúster, permitiendo que organizar/dividir los recursos dentro de este.
CVE: Common Vulnerabilities and Exposures
OS: Sistema Operativo
LXC: Linux Container
Cgroups: Control groups, funcionalidad del kernel de Linux que permite limitar y aislar recursos del sistema (CPU, memoria, disco, red, etc) para un conjunto de procesos.
VM: virtual machine, máquina virtual
CNCF: Cloud Native Computing Foundation
WAF: Web Application Firewalls
RASP: Runtime Application Self Protection
IAST: Interactive Application Security Testing
WAF: Web Application Firewalls
DAST: Dynamic Application Security Testing
SAST: Static Application Security Testing
OWASP: Open Web Application Security Project, organización sin ánimo de lucro centrada en mejorar la seguridad del software.
MITRE ATT&CK®: marco base de conocimiento de las tácticas y técnicas más comunes en ciberataques.
CA: Certificate authority, autoridad de certificación
TLS: Transport Layer Security
WSL: Windows Subsystem Linux, integración nativa de las *syscall* de Linux en el Kernel de Windows permitiendo ejecutar un entorno Linux sin la necesidad de la creación de máquinas virtuales extra.

6. Bibliografía

[1] "Message from the WoC 2016 Workshop Chairs," in 2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW), Berlin, Germany, 2016 pp. xix-xix. doi: 10.1109/IC2EW.2016.65
url: https://doi.ieeecomputersociety.org/10.1109/IC2EW.2016.65

[2] OpenSaaS and the future of government IT innovation. (n.d.). Opensource.com. Consultado el 9 de Octubre de 2022 en https://opensource.com/government/14/1/opensaas-and-government-innovation

[3] Atlassian. (n.d.). Microservices vs. monolithic architecture. Consultado el 9 de Octubre de 2022 en https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[4] What is container orchestration? (2022, September 28). VMware. Consultado el 9 de Octubre de 2022 en https://www.vmware.com/topics/glossary/content/container-orchestration.html

[5] ¿Qué es Kubernetes? (n.d.). Kubernetes. Consultado el 9 de Octubre de 2022 en https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

[6] Women in the ICT sector. (n.d.-a). European Institute for Gender Equality. Consultado el 9 de Octubre de 2022 https://eige.europa.eu/publications/work-life-balance/eu-policies-on-work-life-balance/women-in-ict

[7] The history of Kubernetes & the community behind It. (2018, July 20). Kubernetes. Consultado el 20 de Octubre de 2022 en https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/

[8] Kubernetes Components. (n.d.). Kubernetes. Consultado el 20 de Octubre de 2022 en https://kubernetes.io/docs/concepts/overview/components/

[9] Cluster Architecture. (n.d.). Kubernetes. Consultado el 21 de Octubre de 2022 en https://kubernetes.io/docs/concepts/architecture/

[10] Timar, G. (2015). The falcon project. Wings Epress. Consultado el 29 de Octubre de 2022 en https://falco.org/docs/getting-started/

[11] Cerrada, V. J. (2021, November 15). Getting started with runtime security and Falco. Sysdig. Consultado el 3 de Diciembre de 2022 en https://sysdig.com/blog/intro-runtime-security-falco/

[12] Sysdig. (2022, January 24). Sysdig. Consultado el 29 de Octubre de 2022 en https://sysdig.com/

[13] Kubernetes runtime security. (2022, January 10). ARMO. Consultado el 30 de Octubre de 2022 en https://www.armosec.io/glossary/kubernetes-runtime-security/
[14] Creane, B., & Gupta, A. (2021). Kubernetes security and observability: A holistic approach to securing containers and cloud native applications. O'Reilly Media.

[15] Best practices. (n.d.). Kubernetes. Consultado el 30 de Octubre de 2022 en https://kubernetes.io/docs/setup/best-practices/

[16] Recommended Labels. (n.d.). Kubernetes. Consultado el 7 de Noviembre de 2022 en https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/

[17] Sadek, J. (2021, November 2). Label standard and best practices for Kubernetes security. Tigera. Consultado el 7 de Noviembre de 2022 en https://www.tigera.io/blog/label-standard-and-best-practices-for-kubernetes-security/

[18] Kubernetes security: 8 best practices to secure your cluster. (2021, July 19). Tigera. Consultado el 1 de Noviembre de 2022 en https://www.tigera.io/learn/guides/kubernetes-security/

[19] Best practices for scanning images. (2022, November 7). Docker Documentation. Consultado el 7 de Noviembre de 2022 en https://docs.docker.com/develop/scan-images/

[20] The Istio service mesh. (n.d.). Istio. Consultado el 7 de Noviembre de 2022 en https://istio.io/latest/about/service-mesh/

[21] RBAC vs. ABAC access control: What's the difference? - DNSstuff. (2019, October 31). Software Reviews, Opinions, and Tips - DNSstuff. Consultado el 7 de Noviembre de 2022 en https://www.dnsstuff.com/rbac-vs-abac-access-control

[22] Daida, K. R. (n.d.). Azure AKS Kubernetes Namespaces Introduction - Azure Kubernetes Service. Stacksimplify.com. Consultado el 7 de Noviembre de 2022 en https://stacksimplify.com/azure-aks/azure-kubernetes-service-namespaces-imperative/

[23] Lilette, E. (n.d.). Kubernetes secrets: how to create & set up k8s secrets. Padok.Fr. Consultado el 7 de Noviembre de 2022 en https://www.padok.fr/en/blog/kubernetes-secrets

[24] Blog, K. (2019, September 25). A practical guide to setting Kubernetes requests and limits. Kubecost.com. Consultado el 7 de Noviembre de 2022 en https://blog.kubecost.com/blog/requests-and-limits/

[25] Hablando de microservicios...¿Qué es Service Mesh? (n.d.). Com.mx. Consultado el 7 de Noviembre de 2022 en https://galvarado.com.mx/post/hablando-de-microservicios-que-es-service-mesh/

[26] Mell, E. (2020, April 2). The evolution of containers: Docker, Kubernetes and the future. SearchITOperations; TechTarget. Consultado el 24 de Noviembre de 2022 en https://www.techtarget.com/searchitoperations/feature/Dive-into-the-decades-long-history-of-container-technology

[27] Casalicchio, & Iannucci, S. (2020). The state-of-the-art in container technologies: Application, orchestration and security. Concurrency and Computation, 32(17). Consultado el 25 de Noviembre de 2022 en https://doi.org/10.1002/cpe.5668

[28] Powell, R. (2021, October 13). Docker Swarm vs Kubernetes: how to choose a container orchestration tool. CircleCI. Consultado el 25 de Noviembre de 2022 en https://circleci.com/blog/docker-swarm-vs-kubernetes/?utm_source=google&utm_medium=sem&utm_campaign=sem-google-dg--emea-en-dsa-maxConv-auth-nb&utm_term=g_-_c__dsa_&utm_content=&gclid=CjwKCAiA7IGcBhA8EiwAFfUDsdKFKnSl4JzmeMQTnWRcFMrADoQCsuKt40Y-O617MvTfUgLoZ6_XUxoCouYQAvD_BwE

[29] Simplilearn. (2019, October 21). Openshift Vs. Kubernetes : What is the Difference? [2022 Edition]. Simplilearn.com; Simplilearn. Consultado el 25 de Noviembre de 2022 en https://www.simplilearn.com/kubernetes-vs-openshift-article

[30] Murray, A. (2020, July 16). IAST: Interactive application security testing. Mend. Consultado el 28 de Noviembre de 2022 en https://www.mend.io/resources/blog/iast-interactive-application-security-testing/

[31] Mix, C. (2021, May 18). ¿Qué Es Rasp? CronUp Ciberseguridad. Consultado el 28 de Noviembre de 2022 en https://www.cronup.com/que-es-rasp/

[32] OWASP kubernetes Top Ten. (n.d.). Owasp.org. Consultado el 28 de Noviembre de 2022 en https://owasp.org/www-project-kubernetes-top-ten/

[33] Foster, M. (n.d.). 2021 Kubernetes threat matrix updates: Things you should know. Redhat.com. Consultado el 28 de Noviembre de 2022 en https://cloud.redhat.com/blog/2021-kubernetes-threat-matrix-updates-things-you-should-know

[34] Official CVE Feed. (n.d.). Kubernetes. Consultado el 5 de Diciembre de 2022 en https://kubernetes.io/docs/reference/issues-security/official-cve-feed/

[35] ⎈ Falco - Runtime security monitoring & detection. (n.d.). Madhuakula.com. Consultado el 29 de Octubre de 2022 en https://madhuakula.com/kubernetes-goat/docs/scenarios/scenario-18/

[36] Popeye - A Kubernetes Cluster Sanitizer. (2011). Books LLC, Wiki Series. Consultado el 29 de Octubre de 2022 en https://popeyecli.io/

[37] kyverno. (2014). John Wiley & Sons, Ltd. Consultado el 30 de Octubre de 2022 en https://kyverno.io/docs/introduction/
[38] Calico. (n.d.). Tigera.Io. Consultado el 30 de Octubre de 2022 en https://projectcalico.docs.tigera.io/about/about-calico

[39] Istio. (n.d.). Istio. Consultado el 2 de Diciembre de 2022 en https://istio.io/

[40] Microservices Pattern: Microservice Architecture pattern. (n.d.). Microservices.Io; Chris Richardson. Consultado el 30 de Octubre de 2022 en https://microservices.io/patterns/microservices.html

[41] Installing Chocolatey. (n.d.). Consultado el 5 de Diciembre de 2022 en https://chocolatey.org/install

[42] Quick start. (n.d.). K8s.Io. Consultado el 5 de Diciembre de 2022 en https://kind.sigs.k8s.io/docs/user/quick-start/

[43] Helm. (n.d.). Helm.Sh. Consultado el 5 de Diciembre de 2022 en https://helm.sh/docs/

[44] KiND - Kubernetes IN Docker. (n.d.). Madhuakula.com. Consultado el 5 de Diciembre de 2022 en https://madhuakula.com/kubernetes-goat/docs/how-to-run/kind

[45] Madhu Akula. (n.d.). Sans.org. Consultado el 6 de Diciembre de 2022 en https://www.sans.org/profiles/madhu-akula/

[46] Kubernetes adoption, security, and market trends report 2022. (n.d.). Redhat.com. Consultado el 15 de Diciembre de 2022 en https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview

[47] Bishop, T. (2022, May 18). Microsoft changes cloud practices in Europe, acknowledges missteps in rivalry with Amazon. GeekWire. Consultado el 15 de Diciembre de 2022 en https://www.geekwire.com/2022/microsoft-changes-cloud-practices-in-europe-acknowledges-missteps-in-competition-with-amazon/

[48] Donohue, T. (2020, December 3). The differences between Docker, containerd, CRI-O and runc. Tutorial Works. Consultado el 16 de Diciembre de 2022 en https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/

[49] OCI Certified. (n.d.). Opencontainers.org. Consultado el 16 de Diciembre de 2022 en https://opencontainers.org/community/certified/

[50] linkerd2: Ultralight, security-first service mesh for Kubernetes. Main repo for Linkerd 2.x. (n.d.). Consultado el 18 de Diciembre de 2022 en https://github.com/linkerd/linkerd2

[51] istio: Connect, secure, control, and observe services. (n.d.). Consultado el 18 de Diciembre de 2022 en https://github.com/istio/istio

[52] consul: Consul is a distributed, highly available, and data center aware solution to connect and configure applications across dynamic, distributed infrastructure. (n.d.). Consultado el 18 de Diciembre de 2022 en https://github.com/hashicorp/consul

[53] Install and Set Up kubectl on Windows. (n.d.). Kubernetes. Consultado el 26 de Diciembre de 2022 en https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/

[54] Install Docker Engine on Ubuntu. (2022, December 29). Docker Documentation. Consultado el 29 de Diciembre de 2022 en https://docs.docker.com/engine/install/ubuntu/

[55] Install. (n.d.). Falco. Consultado el 29 de Diciembre de 2022 en https://falco.org/docs/getting-started/installation/

[56] Kubernetes API OVERVIEW. (n.d.). Kubernetes.Io. Consultado el 1 de Enero de 2023 en https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.20/

[57] Sasson, A. (2021, March 26). 20 million miners: Finding malicious cryptojacking images in docker hub. Unit 42. Consultado el 3 de Enero de 2023 en https://unit42.paloaltonetworks.com/malicious-cryptojacking-images/

[58] Container escape to the host system. (n.d.). Madhuakula.com. Consultado el 3 de Enero de 2023 https://madhuakula.com/kubernetes-goat/docs/scenarios/scenario-4/

[59] Roper, J. (n.d.). 15 Kubernetes best practices every developer should know. Spacelift. Consultado el 7 de Noviembre de 2022 en https://spacelift.io/blog/kubernetes-best-practices

[60] Gaona, C. [UC-yt0UFauVHfh1rfI1IPI4w]. (2021, July 7). Seguridad en Kubernetes - UTN. Youtube. https://www.youtube.com/watch?v=Z1PzO8eT_qs

[61] CCN [CCNCERT-CCN]. (2019, December 26). Seguridad en contenedores y Kubernetes (David Castillo, Fortinet). Youtube. https://www.youtube.com/watch?v=ZKKBXWhGQbw

[62] Nerd, P. [PeladoNerd]. (2022, March 15). Seguridad en Contenedores con Tim Allclair (Google/Kubernetes). Youtube. https://www.youtube.com/watch?v=g9MAcBZQXjM

[63] Nana, T. W. [TechWorldwithNana]. (2020, November 6). Kubernetes Tutorial for beginners [FULL COURSE in 4 hours]. Youtube. https://www.youtube.com/watch?v=X48VuDVv0do

[64] What is IAST? Interactive Application Security Testing. (n.d.). Veracode. Consultado el 5 de Enero de 2023 https://www.veracode.com/security/interactive-application-security-testing-iast

[65] HCLSoftware. (n.d.). Hcltechsw.com. Consultado el 5 de Enero de 2023 https://www.hcltechsw.com/appscan/features

[66] Cankurt, S. (2022, May 24). Fortify WebInspect - application security toolset. AppSec Santa. Consultado el 5 de Enero de 2023 https://www.appsecsanta.com/fortify-webinspect

7. Anexos

7.1. Anexo 1 – Configuración del despliegue del clúster

Fichero: kind-cluster-setup.yaml

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
    extraMounts:
      - hostPath: /var/run/docker.sock
        containerPath: /var/run/docker.sock
```

Fichero: setup-kubernetes-goat.ps1

```
kubectl apply -f scenarios/insecure-rbac/setup.yaml
kubectl apply -f scenarios/batch-check/job.yaml
kubectl apply -f scenarios/build-code/deployment.yaml
kubectl apply -f scenarios/cache-store/deployment.yaml
kubectl apply -f scenarios/health-check/deployment.yaml
kubectl apply -f scenarios/hunger-check/deployment.yaml
kubectl apply -f scenarios/internal-proxy/deployment.yaml
kubectl apply -f scenarios/kubernetes-goat-home/deployment.yaml
kubectl apply -f scenarios/poor-registry/deployment.yaml
kubectl apply -f scenarios/system-monitor/deployment.yaml
kubectl apply -f scenarios/hidden-in-layers/deployment.yaml
```

7.2. Anexo 2 - Redireccionamientos de puertos

Fichero: setup-kubernetes-goat.ps1

```
Write-Host 'Creando port-forward para todos los recursos locales de
Kubernetes Goat. Se usarán los puertos locales del 1230 al 1236!'

# Exposing Sensitive keys in code bases Scenario
$POD_NAME = kubectl get pods --namespace default -l "app=build-code" -o
jsonpath="{.items[0].metadata.name}"
kubectl port-forward $POD_NAME --address 127.0.0.1 1230:3000 &

# Exposing DIND (docker-in-docker) exploitation Scenario
$POD_NAME = kubectl get pods --namespace default -l "app=health-check" -
o jsonpath="{.items[0].metadata.name}"
kubectl port-forward $POD_NAME --address 127.0.0.1 1231:80 &

# Exposing SSRF in K8S world Scenario
$POD_NAME = kubectl get pods --namespace default -l "app=internal-proxy"
-o jsonpath="{.items[0].metadata.name}"
kubectl port-forward $POD_NAME --address 127.0.0.1 1232:3000 &

# Exposing Container escape to access host system Scenario
$POD_NAME = kubectl get pods --namespace default -l "app=system-monitor"
-o jsonpath="{.items[0].metadata.name}"
kubectl port-forward $POD_NAME --address 127.0.0.1 1233:8080 &

# Exposing Kubernetes Goat Home
$POD_NAME = kubectl get pods --namespace default -l "app=kubernetes-goat-
home" -o jsonpath="{.items[0].metadata.name}"
kubectl port-forward $POD_NAME --address 127.0.0.1 1234:80 &

# Exposing Attacking private registry Scenario
$POD_NAME = kubectl get pods --namespace default -l "app=poor-registry" -
o jsonpath="{.items[0].metadata.name}"
kubectl port-forward $POD_NAME --address 127.0.0.1 1235:5000 &

# Exposing DoS resources Scenario
$POD_NAME = kubectl get pods --namespace big-monolith -l "app=hunger-
check" -o jsonpath="{.items[0].metadata.name}"
kubectl --namespace big-monolith port-forward $POD_NAME --address
127.0.0.1 1236:8080 &

Write-Host "Visita http://127.0.0.1:1234 para comenzar con el Hacking de
Kubernetes Goat!"
```