Máster Interuniversitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones (MISTIC)

Trabajo de Fin de Máster

Análisis de seguridad de arquitecturas basadas en Kubernetes

Autor:
Manuel Simón Nóvoa

Docentes:
Juan Carlos Fernández Jara
Víctor García Font

1 de junio de 2021

Agradecimientos

À minha avoa Carme, a "Mucha Maravilhas".

Índice

1. Introducción **14**
1.1. Contexto y justificación del trabajo . . . 14
1.2. Objetivos del Trabajo . . . 15
1.3. Enfoque y método seguido . . . 16
1.4. Medios materiales necesarios . . . 16

2. Planificación **18**
2.1. Metodología de trabajo . . . 18
2.1.1. Diferenciación general de las metodologías de trabajo . . . 18
2.1.2. Justificación de la metodología de trabajo escogida . . . 18
2.2. Planificación del trabajo . . . 18
2.3. Estimación temporal . . . 19
2.4. Gestión de la configuración . . . 21
2.4.1. Identificación de los elementos de la configuración . . . 21
2.4.2. Control de cambios . . . 22
2.4.3. Control de versiones . . . 22
2.5. Gestión de riesgos . . . 22
2.5.1. Metodología a seguir para realizar la gestión de riesgos . . . 22
2.5.2. Identificación de amenazas . . . 25
2.5.3. Identificación de activos . . . 27
2.5.4. Análisis de riesgos . . . 27
2.5.5. Seguimiento de los riesgos . . . 32

3. Conceptos básicos **34**
3.1. Máquinas virtuales . . . 34
3.2. Tecnologías de contenerización . . . 34
3.3. Docker Hub . . . 35
3.4. Kubernetes . . . 36
3.4.1. Componentes principales . . . 37
3.4.2. Arquitectura de Kubernetes . . . 39

4. Especificación de requisitos **42**
4.1. Limitaciones . . . 42
4.2. Listado de requisitos no funcionales . . . 42

5. Instalación y configuración del laboratorio de pruebas **48**
5.1. Arquitectura y especificaciones . . . 48
5.2. Instalación y configuración básica del SO en las máquinas virtuales . . . 49
5.3. Instalación de Kubernetes . . . 52
5.3.1. Instalación y configuración de nodos *worker* . . . 52
5.3.2. Instalación y configuración de nodo *master* . . . 53

6. Análisis de vulnerabilidades **56**
6.1. Análisis estático sobre imágenes de contenedores . . . 56
6.1.1. Docker Scan . . . 56
6.2. Análisis estático sobre ficheros de configuración YAML . . . 61
6.3. Limitación de recursos . . . 62
6.3.1. Limitación de memoria RAM . . . 63
6.3.2. Limitación de CPU . . . 64
6.3.3. Limitación de red . . . 66
6.4. Permisos de ejecución dentro de los contenedores . . . 66
6.4.1. Implicaciones a nivel de seguridad dentro del contenedor . . . 66
6.4.2. Implicaciones a nivel de seguridad en la máquina anfitriona . . . 67
6.5. Comparativa de aislamiento entre virtualizaciones: máquinas virtuales VS contenedores . . . 69
6.6. Escritura en disco . . . 72
6.7. Comprobación de la existencia de un mecanismo de validación mediante firmas digitales en Kubernetes . . . 72
6.8. Herencia de vulnerabilidades entre imágenes . . . 74
6.9. Comprobación de los espacios de trabajo existentes por defecto . . . 75
6.10. Comprobación de la interconexión por red de contenedores en diferentes nodos . . . 76

7. Securización **80**
7.1. Análisis estático sobre imágenes de contenedores . . . 80
7.1.1. Posible ejemplo de securización . . . 81
7.1.2. Procedimiento estandarizado . . . 83
7.2. Limitación de recursos . . . 84
7.2.1. Limitación de memoria . . . 84
7.2.2. Limitación de CPU . . . 86
7.3. Limitación de los permisos de ejecución dentro de los contenedores . . . 87
7.3.1. Comprobación de la securización: implicaciones dentro del contenedor 87
7.3.2. Comprobación de la securización: implicaciones en la máquina anfitriona . . . 88
7.4. Compromiso entre portabilidad y seguridad . . . 90
7.5. Sistemas de solo lectura . . . 92
7.6. Validación de imágenes mediante firma digital . . . 92
7.6.1. Docker Content Trust . . . 93
7.6.2. Connaisseur . . . 93
7.7. Herencia de vulnerabilidades entre imágenes . . . 97
7.8. Diferenciación de espacios . . . 98
7.9. Configuración y uso del componente *Secret* . . . 99
7.10. Análisis estático sobre ficheros de configuración YAML . . . 102
7.11. *Service Mesh* . . . 103
7.11.1. Introducción teórica a *Service Mesh* . . . 103
7.11.2. Introducción teórica a Istio . . . 104
7.11.3. Instalación y configuración del *Service Mesh* Istio en el clúster . . . 107
7.11.4. Funcionalidades “extendidas” de Istio . . . 109

8. Conclusiones **114**
8.1. Clasificación final de los análisis de seguridad realizados 114

9. Trabajo futuro **118**
9.1. Limitación de red 118
9.2. Securización del componente `etcd` 119
9.3. Securización de la red externa 120
9.4. Control de Acceso Basado en Roles 120
9.5. Mejora continua de los ficheros de configuración YAML 122
9.6. Modelo centralizado: MAC 122
9.7. Otras referencias de consulta 123

A. Anexos **126**
A.1. Comandos para la instalación y configuración de Kubernetes 126
A.1.1. Nodos *worker* 126
A.1.2. Nodo *master* 126
A.2. Fichero YAML para el despliegue de la imagen Nginx 1.20.0 127
A.3. Código C de un programa altamente consumidor de memoria 128
A.4. Fichero YAML para el despliegue de un pod con un contenedor Ubuntu . . 129
A.5. Fichero YAML para el despliegue de un *deployment* de Ubuntu con 4 réplicas 130
A.6. Fichero JSON para la creación de nuevos espacios de nombres 131
A.7. Fichero JSON para la creación de nuevos espacios de nombres 132
A.8. Fichero YAML para el despliegue de un *deployment* de Ubuntu securizado 133

Índice de figuras

1. Metodología Scrum . . . 19
2. Entregas indicadas por la universidad . . . 19
3. Estructura de Desglose del Trabajo . . . 20
4. Diagrama de la arquitectura y componentes de Kubernetes . . . 41
5. Diagrama de la arquitectura del laboratorio . . . 49
6. Instalación del SO en las diferentes máquinas virtuales . . . 50
7. Ejemplo de configuración de red en la máquina "Worker1" . . . 51
8. Ejemplo de configuración del fichero `/etc/hosts` en la máquina "Worker1" 51
9. Instalación de Docker en un nodo *worker* . . . 52
10. Activación del servicio de Docker en un nodo *worker* . . . 52
11. Añadimos clave y repositorio de Kubernetes en un nodo *worker* . . . 53
12. Instalación de los paquetes necesarios en un nodo *worker* . . . 53
13. Incorporación del nodo *worker* en los nodos administrados por el nodo *master* 53
14. Inicialización del nodo *master* . . . 54
15. Resultado de la inicialización . . . 54
16. Ejecución de los comandos post-inicialización . . . 54
17. Comprobación de la detección de los nodos *worker* . . . 54
18. Imagen de Nginx en Docker Hub. *Tag* 1.20.0. . . . 58
19. Descarga de la imagen objetivo y ejecución de análisis estático . . . 58
20. Fin del resultado del análisis estático de una imagen . . . 59
21. Ejemplo de listado de dependencias de software con vulnerabilidades en imagen Nginx . . . 60
22. Imagen Nginx 1.20.0 corriendo sobre contenedor en el nodo *worker* 2 . . . 61
23. Análisis estático de fichero YAML para el despliegue de Nginx . . . 62
24. Mediciones del uso de la memoria RAM antes de la prueba . . . 64
25. Mediciones del uso de la memoria RAM después de la prueba . . . 64
26. Mediciones del uso de CPU antes de la prueba . . . 65
27. Mediciones del uso de CPU después de la prueba . . . 65
28. Mediciones del uso de CPU en la máquina anfitriona después de la prueba 65
29. Prueba de uso del total del ancho de banda . . . 66
30. Obtención de permisos de superusuario dentro del contenedor . . . 67
31. Acceso al fichero `/etc/hosts` de la máquina anfitriona desde dentro de un contenedor . . . 69
32. Comprobación, desde la máquina anfitriona, de que el fichero `/etc/hosts` fue modificado . . . 69
33. Laboratorio modificado . . . 70
34. Mediciones del uso de la memoria RAM de la máquina anfitriona antes de la prueba . . . 71
35. Mediciones del uso de la memoria RAM de la máquina anfitriona después de la prueba . . . 71
36. Máquina virtual "Worker1" abortada en VirtualBox . . . 72
37. Escritura en disco de forma indiscriminada . . . 72
38. Despliegue de una imagen con origen Google Cloud sin validar su firma . . 73
39. Ejemplificación de herencia de vulnerabilidades entre imágenes dependientes 74

40. Comprobación del espacio de nombres de las ejecuciones corriendo sobre el clúster . . . 75
41. Representación gráfica de la prueba de interconexión de contenedores en red 76
42. Comprobación del despliegue y obtención del nombre de los *pods* . . . 77
43. *Pod* Nginx en nodo *worker* 1 . . . 77
44. *Pod* Ubuntu en nodo *worker* 2 . . . 77
45. Obtención de la IP interna del contenedor Nginx en el nodo *worker* 1 . . . 78
46. Prueba de establecimiento de conexión . . . 78
47. Imagen de Nginx en Docker Hub. *Tag* 1.11.0. . . . 80
48. Fin del resultado del análisis estático de la imagen Nginx:1.11.0 . . . 81
49. Creación de imagen local basada en el contenedor en ejecución actualizado 82
50. *Push* de la imagen hacia un repositorio en el Docker Hub . . . 82
51. Ejecución de análisis estático sobre la imagen actualizada . . . 83
52. Fin del resultado del análisis estático de la imagen actualizada . . . 83
53. Mediciones del uso de la memoria RAM antes de la prueba (correcciones aplicadas) . . . 85
54. Mediciones del uso de la memoria RAM después de la prueba (correcciones aplicadas) . . . 85
55. Mediciones del uso de CPU antes de la prueba (correcciones aplicadas) . . . 86
56. Mediciones del uso de CPU después de la prueba (correcciones aplicadas) . . . 86
57. Comprobación de iniciación del contenedor sin permisos `root` . . . 88
58. Comprobación de UID y GID dentro del contenedor . . . 88
59. Intento fallido de ejecución de comando privilegiado . . . 88
60. Creación de nuevo usuario `dockermap` y sus respectivos `subuid` y `subgid` . . . 89
61. Permiso denegado al intentar sobreescribir un fichero protegido . . . 90
62. Propuesta de combinación de ambas tecnologías de virtualización . . . 91
63. Comprobación del sistema de solo lectura . . . 92
64. Relación entre las claves usadas por "Docker Content Trust" . . . 94
65. Resumen gráfico del funcionamiento de Connaisseur . . . 95
66. Descarga de imagen Debian . . . 96
67. Operación `commit` sobre el contenedor en funcionamiento . . . 96
68. Subida al repositorio personal del *tag* no firmado . . . 96
69. Configuración y primer uso de "Docker Content Trust" . . . 97
70. Recuperación de la parte pública de la firma creada . . . 97
71. Proceso de instalación de Connaisseur . . . 97
72. Connaisseur rechaza imagen sin firma digital . . . 98
73. Connaisseur acepta imagen con firma digital . . . 98
74. Espacios de nombre creados por defecto en un clúster Kubernetes . . . 98
75. Creación y comprobación de la existencia de nuevos espacios . . . 99
76. Asociación de los nuevos espacios de nombre a un contexto . . . 99
77. Cambio al contexto de desarrollo, asociado al "espacio1" . . . 99
78. Demostración del aislamiento entre *namespaces* dentro del mismo clúster . . . 100
79. Obtención en base-64 de la información privada . . . 100
80. Almacenamiento del *secret* en el clúster Kubernetes . . . 101
81. Comprobación del correcto uso de un *secret* desde dentro de un contenedor 102
82. Análisis estático sobre fichero YAML mejorado . . . 102

83. Resumen visual de *Service Mesh* 105
84. Implementación de *Service Mesh* de Istio 106
85. Representación gráfica del funcionamiento de Istio 106
86. Descarga y descompresión de la última versión de Istio 107
87. Incorporación al `PATH` de `istioctl` 108
88. Proceso de instalación de Istio 108
89. Comprobación de que Istio ya está corriendo en el clúster 108
90. Comprobación de despligue sin sidecar 108
91. Configuración del autoinyectado automático de *proxies* 109
92. Despliegue de las integraciones de Istio 110
93. Obtención de las direcciones de acceso de las nuevas integraciones 110
94. Ejemplo de uso de Kiali, parte 1 111
95. Ejemplo de uso de Kiali, parte 2 111
96. Ejemplo de uso de Kiali, parte 3 112
97. Ejemplo de uso de Kiali, parte 4 112
98. Ejemplo de uso de Kiali, parte 5 113
99. Representación gráfica de las 4C de seguridad en *Cloud Native* 115
100. Propuesta de securización de la red externa 121

Glosario

CVE *Common Vulnerabilities and Exposures* (Vulnerabilidades y Exposiciones Comunes). 56

DHCP *Dynamic Host Configuration Protocol* (Protocolo de Configuración Dinámica de Host). 49

DNS *Domain Name System* (Sistema de Nombres de Dominio). 50

DoS *Denial of Service* (Denegación de Servicio). 63, 65, 84, 85, 87

ECTS *European Credit Transfer and Accumulation System* (Sistema Europeo de Transferencia y Acumulación de Créditos). 21

EDT Estructura de Desglose del Trabajo. 19, 21, 32, 43

GID *Group identifier* (Identificador de grupo). 9, 87–90

HTTP *Hypertext Transfer Protocol* (Protocolo de Transferencia de Hipertexto). 103

JSON *JavaScript Object Notation* (Notación de Objeto de JavaScript). 6, 98, 131, 132

MAC *Mandatory Access Control* (Control de Acceso Obligatorio). 6, 122, 123

OOM *Out-Of-Memory* (Memoria Insuficiente). 63, 64

QoS *Quality of Service* (Calidad de Servicio). 115, 119

RBAC *Role-Based Access Control* (Control de Acceso Basado en Roles). 120

SaaS *Software as a Service* (Software como Servicio). 14

SO Sistema Operativo. 4, 8, 44, 49, 50, 81

TLS *Transport Layer Security* (Seguridad de la Capa de Transporte). 46, 105, 109, 115, 116, 119

UID *User identifier* (Identificador de usuario). 9, 67, 68, 87–90

XML *eXtensible Markup Language* (Lenguaje de Marcado Extensible). 21

1. Introducción

1.1. Contexto y justificación del trabajo

En los últimos años, se ha vivido una gran transformación en el software, el cual comenzó a dejar de ser visto como un "producto" monolítico, indivisible, que se ofrecía para ser instalado directamente en la máquina del usuario, para pasar a ser subdividido en numerosos microservicios: pequeñas piezas de software más fácilmente mantenibles que interactúan entre sí [4]. Esta nueva perspectiva no solo cambiaría la manera de entender cómo diseñar y programar el software, sino que también transformaría la forma en la que el usuario utiliza o consume dicho software. Con esta evolución hacia los microservicios y gracias a las conexiones de Internet cada vez más veloces, el software se está convirtiendo cada día más y más en un servicio que el usuario consume, en lugar de tenerlo instalado y configurado en su propia máquina. Es decir, estamos viviendo una gran evolución hacia el SaaS (*Software as a Service*), donde los usuarios ya no precisan más de realizar instalaciones y configuraciones locales, sino que una conexión a Internet es suficiente para conseguir acceso a un software cada vez más sofisticado.

Como consecuencia de la evolución hacia este modelo, se ha desencadenado la aparición de numerosas tecnologías en lo referente al mundo de computación en la nube. Por ejemplo, al dejar de tener el software instalado en la máquina del usuario, en el modelo SaaS es preciso que los microservicios creados puedan desplegarse de tal forma que lleguen a servir millones de transacciones por segundo, desde el servicio *cloud* ofrecido. Así pues, bajo la necesidad de albergar gran cantidad de aplicaciones pequeñas e independientes, que interactúan entre sí, y aprovechando los máximos recursos disponibles en una máquina, prolifera el uso de las tecnologías de contenerización, al resultar éstas el escenario ideal para contener estas pequeñas piezas unitarias de software. Aunque la posibilidad de crear múltiples entornos virtuales replicables dentro de una misma máquina no resulta ninguna novedad, pues ya existen numerosas herramientas para la gestión de máquinas virtuales, como, por ejemplo, Vagrant, los contenedores suponen una forma mucho más ágil y eficiente de crear, procesar y destruir dichos entornos, manteniendo igualmente un buen nivel de independencia entre los microservicios que correrán dentro de estos contenedores, frente a las máquinas virtuales, que resultan menos livianas y portables [4][1]. En resumen: los entornos basados en contenedores consiguen un buen nivel de aislamiento, combinado con una pérdida de rendimiento mínima y una alta velocidad de iniciación. Si a estas características le sumamos una arquitectura basada en microservicios, donde el software puede ser descompuesto en pequeñas piezas unitarias; cada una de estas piezas unitarias puede fallar, escalar, ser mantenida o reutilizada de una forma totalmente independiente.

A pesar de que los contenedores pueden suponer, a priori, una herramienta ideal para el mundo *cloud* y su subdivisión del software en microservicios, existe otra problemática todavía no especificada: la dificultad de cómo crear, gestionar, monitorizar, destruir, estos pequeños entornos independientes. Al trabajar con pocos contenedores, una posibilidad puede ser realizar esta gestión mediante *scripts* de despliegue; sin embargo, la descomposición del software en microservicios puede llevar a que un software esté conformado por cientos o miles de piezas unitarias, que residirán en contenedores independientes. Llegados

a este punto, la posibilidad de controlar los numerosos entornos haciendo uso de *scripts* de despliegue se vuelve mucho más compleja y, en respuesta a esta dificultad, surgen las herramientas de orquestación, como puede ser Kubernetes, capaz de administrar un alto número de contenedores, incluso si éstos están distribuidos en diferentes entornos [4][43].

Gracias a herramientas como Kubernetes la gestión de contenedores consigue una amplia mejora, ofreciendo una respuesta eficaz a algunos de los principios de los que depende el software descompuesto en microservicios: asegurar la alta disponibilidad, capacidad de escalabilidad y la posibilidad de recuperación en caso de desastre.

1.2. Objetivos del Trabajo

Comprendida la necesidad de los servicios de orquestación para gestionar entornos de trabajo basados en microservicios, compuestos, potencialmente, a su vez, de cientos o miles de contenedores, debe surgir igualmente la necesidad de comprobación de que dichos entornos resultan aceptables desde el punto de vista de la seguridad informática. Es por ello que el presente trabajo se centrará en realizar un análisis de seguridad de arquitecturas basadas en Kubernetes. Para ello, se abordarán los siguientes objetivos:

1. **Estudio preliminar**

   Puesto que se desconocen, en un primer momento, la utilización de la herramienta de orquestación Kubernetes, así como las implicaciones de seguridad que este tipo de entornos pueden suscitar, el trabajo a realizar incluirá una importante parte de estudio, en la que se investigará sobre el uso de esta herramienta, posibles vulnerabilidades y la susceptibilidad del entorno ante las mismas.

2. **Análisis**

   Concluido el estudio preliminar y comprendido el funcionamiento de la herramienta Kubernetes, será necesario crear y configurar un laboratorio, un entorno de pruebas, en el que poder trabajar con la herramienta de orquestación. Además, se realizará un análisis sobre las vulnerabilidades previamente estudiadas, verificando los avances teóricos también de una forma práctica, apoyándonos para esto en el laboratorio creado.

3. **Detección y corrección o prevención**

   Basándonos en los estudios y análisis realizados, en este punto existirán una serie de vulnerabilidades identificadas. Partiendo de este punto, se profundizará en aquellas vulnerabilidades más destacables, tratando de entender sus implicaciones en el marco en el que se desarrolla este proyecto. Además, basándonos en los avances obtenidos, se desarrollará una propuesta de securización, en la cual se pondrán en práctica las propuestas de mejora basadas en los estudios previos. Tras aplicar estas mejoras en el entorno de pruebas, se comprobará la eficacia de la securización realizada.

4. **Documentación**

   Acompañando al desempeño del trabajo indicado en los puntos anteriores, se realizará, de manera paralela, una memoria en la que se verá reflejado un estudio de

seguridad del entorno descrito. Dicha memoria abarcará todos los pasos indicados, incluyendo el estudio de las tecnologías de contenerización junto con la herramienta de orquestación Kubernetes, un análisis de las posibles problemáticas de seguridad existentes en entornos formados por estas tecnologías, la explicación de las pruebas prácticas llevadas a cabo y, finalmente, las recomendaciones indicadas para obtener un entorno seguro, basadas en todos los estudios, tanto teóricos como prácticos, realizados anteriormente, así como una serie de conclusiones.

1.3. Enfoque y método seguido

Respecto al enfoque y método a seguir, se utilizará una metodología ágil como puede ser Scrum [9]. Se eligió este tipo de marco debido a que la naturaleza del trabajo impide la ejecución de un ciclo de vida tradicional: la existencia de un alto nivel de incertidumbre en el que, a partir de los estudios realizados en la fase de análisis, se analizarán los requisitos necesarios para poder utilizar contenedores y el servicio de orquestación Kubernetes de forma segura. Para esto, probablemente sean necesarias múltiples iteraciones, hasta alcanzar el nivel de seguridad deseado. Por lo tanto, una metodología ágil como Scrum resulta más adecuada en un proyecto lleno de incertidumbres y posibles cambios. Al tratarse de un trabajo mayormente individual, a pesar de contar con la ayuda y guía del tutor, será necesario realizar ciertas adaptaciones al marco, detalladas en la sección 2.1.2.

1.4. Medios materiales necesarios

- Ordenador de trabajo con posibilidad de desplegar múltiples máquinas virtuales, sobre las que ejecutar varios contenedores y trabajar con la tecnología de orquestación Kubernetes.

2. Planificación

2.1. Metodología de trabajo

2.1.1. Diferenciación general de las metodologías de trabajo

El primer paso para poder obtener una buena planificación pasa por la elección de una metodología de trabajo adecuada al proyecto. La importancia de esta selección viene dada porque la elección errónea de una metodología de trabajo podría llevar a incumplir fechas de entrega o, en el peor de los casos, la cancelación de la totalidad del proyecto. Es importante entender que la metodología adecuada dependerá del tipo de proyecto a realizar, ya que ciertas metodologías se adaptan mejor a ciertos escenarios de trabajo. Podemos dividir las metodologías en dos grandes bloques:

- **Metodologías clásicas:** basadas en una planificación temporal con gran detalle desde el inicio del proyecto que no debe ser modificada, existiendo igualmente una serie de requisitos inalterables. Son metodologías "pesadas" y muy poco tolerantes a escenarios cambiantes o con altos niveles de incertidumbre.
- **Metodologías ágiles:** basadas en una continua revisión de los avances y en un constante contacto entre la parte encargada del desarrollo y el cliente. En este tipo de metodologías, se asume que el proyecto sufrirá modificaciones en su planificación inicial y requisitos. Este enfoque las convierte en metodologías mucho más flexibles y tolerantes a escenarios cambiantes o con altos niveles de incertidumbre.

2.1.2. Justificación de la metodología de trabajo escogida

Teniendo en cuenta las diferencias citadas en la sección anterior, podemos concluir que la naturaleza de este proyecto impide la ejecución de una metodología tradicional, tal y como fue analizado en la sección 1.3. La metodología escogida fue Scrum, una de las metodologías ágiles más populares, ya que es muy probable que se añadan nuevas tareas a la lista de tareas a realizar a medida que el proyecto avance y se vaya dando respuesta a ciertos términos que se desconocen en el comienzo del mismo, lo que complica la realización temprana de una planificación detallada y exacta. No obstante, es necesario incluir algunas modificaciones: el autor del proyecto asumirá el papel de todos los miembros del equipo de trabajo, asumiendo así las diferentes responsabilidades y cargos en función de las necesidades. El rol de *Scrum Master* será asumido por los tutores del proyecto, a los que se les hará llegar los avances del proyecto mediante las entregas calendarizadas, así como pequeñas modificaciones intermedias. El rol de *Product Owner* también será asumido por el autor del proyecto, al tratarse del máximo interesado en el correcto desarrollo del mismo y en la obtención de resultados.

Un resumen visual de la metodología de trabajo escogida puede observarse en la figura 1.

2.2. Planificación del trabajo

Aún existiendo inconvenientes para una realización temporal exhaustiva, sí que resulta posible, incluso desde el comienzo del proyecto, la división del mismo en diferentes mó-

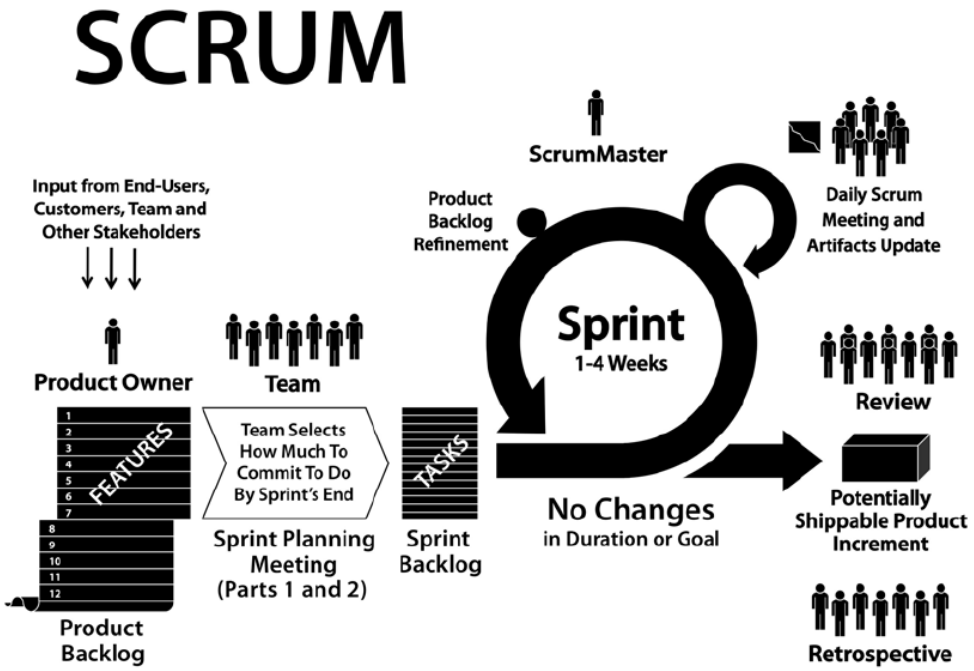


Figura 1: Metodología Scrum

Fuente: [9]

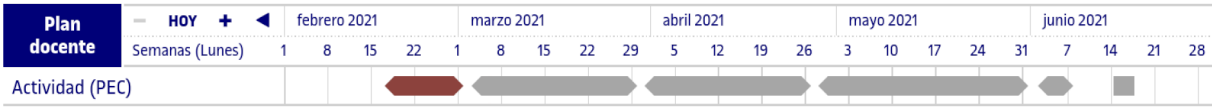


Figura 2: Entregas indicadas por la universidad

Fuente: Campus virtual de la Universitat Oberta de Cayalunya

dulos. De esta forma, para planificar la ejecución del siguiente trabajo, se ha desarrollado una EDT (Estructura de Desglose del Trabajo), mostrada en la figura 3, que, atendiendo al PMBOK [12], otorga una visión global desde lo más general a lo más específico del alcance del proyecto, mostrando de una forma jerárquica el trabajo que es necesario llevar a cabo para su realización. Así, en la EDT quedarán detallados los objetivos que deben ser alcanzados a lo largo de las diferentes etapas por las que pasará el proyecto. Concretamente, los 6 apartados que componen el nivel 2 (color anaranjado) de la EDT se corresponden directamente con las entregas indicadas por la universidad para el correcto desarrollo del estudio, las cuales se muestran en la figura 2.

2.3. Estimación temporal

A pesar de conocer la planificación inicial del proyecto y las dificultades a las que se enfrenta, en lo referente a la realización de una estructuración temporal, debemos tener en cuenta que se debe atener, igualmente, a unas grandes limitaciones de tiempo. Concretamente, la fecha de comienzo de este proyecto se trata del 17 de febrero de 2021, con una fecha de fin del 8 de junio de 2021, para presentar todos los materiales que conformarán

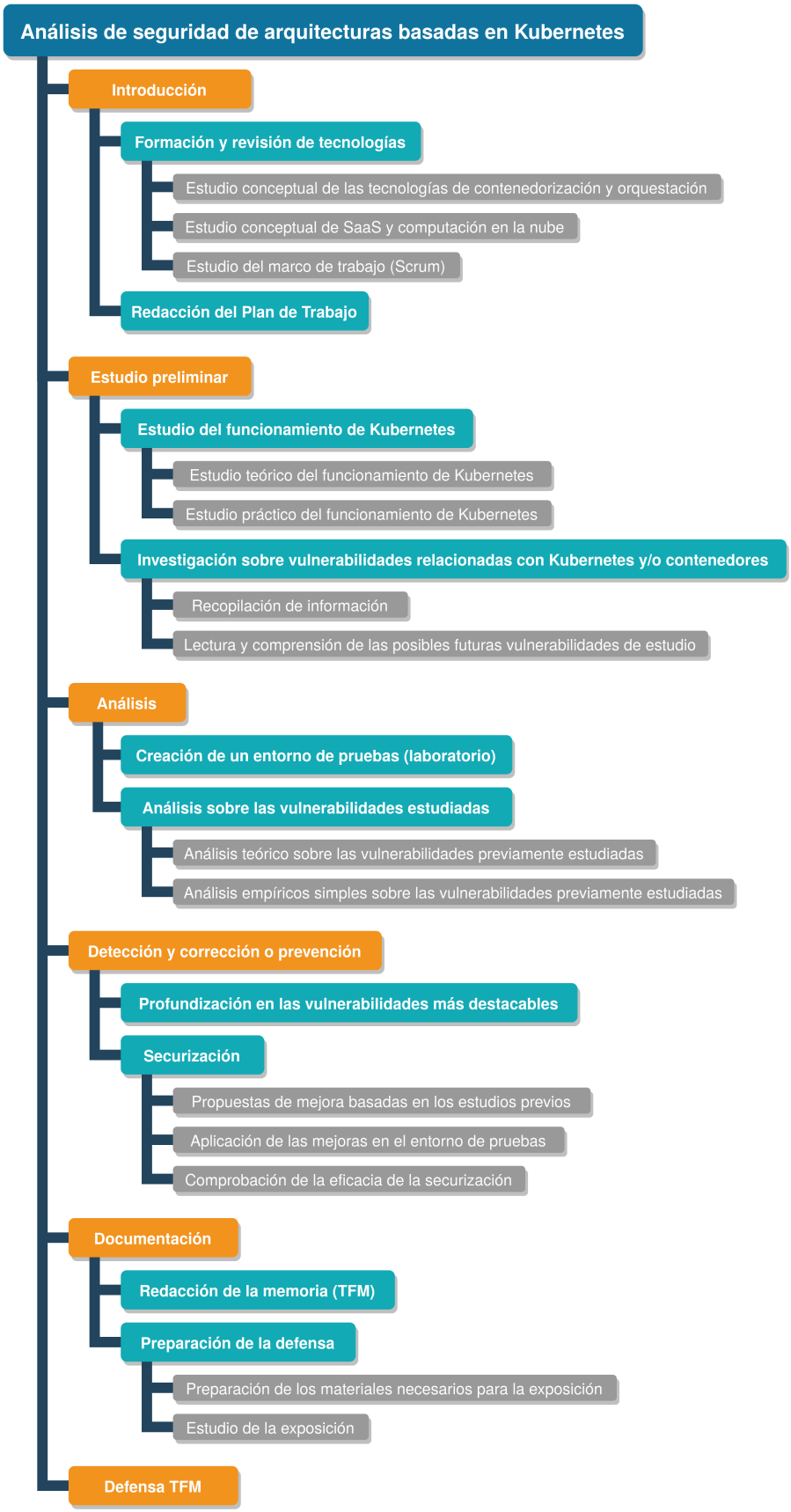


Figura 3: Estructura de Desglose del Trabajo

su defensa. De esta forma, el proyecto debe ser realizado en su totalidad en, aproximadamente, 15 semanas.

Especificando un poco más el reparto temporal, el presente trabajo tiene asociada una dedicación de 9 créditos ECTS. Atendiendo a la Guía de uso del ECTS [10], un crédito ECTS equivale entre 25 y 30 horas de trabajo. Por tanto, el presente proyecto debe tener una dedicación temporal de aproximadamente 270 horas, a repartir entre un total de 15 semanas, dando lugar a una dedicación aproximada de 18 horas semanales.

Además, debido al hecho de tener que obedecer a la planificación de entregables dada por la universidad, es posible indicar el número de semanas de trabajo que tendrá asociadas cada uno de los objetivos de nivel 2 indicados en la EDT, excluyendo su propia defensa:

- **Introducción:** 2 semanas.
- **Estudio preliminar:** 4 semanas.
- **Análisis:** 4 semanas.
- **Detección y corrección o prevención:** 5 semanas.
- **Documentación:**
  - **Redacción de la memoria:** 15 semanas, siendo realizada de forma paralela con el resto de objetivos.
  - **Preparación de la defensa:** 1 semana, a realizar después de la entrega de la memoria y, por tanto, no incluida en las 15 semanas de trabajo previamente indicadas.

2.4. Gestión de la configuración

2.4.1. Identificación de los elementos de la configuración

Se entiende por elemento de la configuración cada uno de los elementos del proyecto que se vea afectado por la gestión de la configuración. Los elementos identificados son:

- **Memoria del proyecto:** la presente memoria. Constituye un soporte para representar todos los avances realizados en el proyecto.
- **Diagramas y figuras:** diagramas elaborados con la herramienta en línea Diagrams.net[1], para la explicación visual de ciertos elementos del proyecto. Además de tratar las figuras como ficheros de imagen, también se incluye su representación XML, para la posible aplicación de futuras modificaciones.

[1] `https://app.diagrams.net/`

2.4.2. Control de cambios

La gestión de la configuración se limitará al control de versiones de los diferentes elementos de la configuración. El control de cambios será llevado a cabo solamente por una persona, el autor del proyecto.

2.4.3. Control de versiones

Para realizar el control de versiones se utilizará el sistema de control de versiones Git, gracias al cual será posible tener una trazabilidad de los cambios en los elementos de la configuración. Además, los datos no serán almacenados de manera local, sino que se hará uso de la plataforma GitHub[2], almacenando los ficheros y sus cambios de forma remota. De esta forma, la plataforma cumplirá con un doble cometido: por un lado permitir realizar el control de versiones y, por otro, funcionar como un servicio de copia de seguridad de los elementos de la configuración.

En lo que concierne a la organización de este control de versiones, en la plataforma GitHub será creado un repositorio privado en el que un único editor, el autor del proyecto, trabajará sobre la rama "*main*". A pesar de que fueron detectados varios elementos de la configuración, todos ellos llevarán su gestión de cambios bajo el mismo repositorio y de manera conjunta. Para realizar los cambios y su control, cada uno de ellos estará acompañado por un mensaje explicativo. Una modificación en el control de versiones no tiene por qué estar necesariamente asociada con la finalización de un entregable o tarea, sino que cualquier avance que sea considerado de relevancia, será guardado para evitar riesgos asociados con la pérdida de información en el equipo de trabajo local.

El control de cambios de la memoria requiere un poco más de profundización, al ser el principal activo del proyecto. La memoria del proyecto será redactada empleando el sistema de preparación de documentos LaTeX, lo que permitirá su división en diferentes ficheros TeX. Dichos ficheros, al estar compuestos por texto simple, resultan perfectos para un correcto control de cambios con un sistema como Git, en el que se podrán observar las modificaciones del texto en los diferentes ficheros, sin que la modificación de una sección suponga la creación de una nueva versión en la totalidad para la memoria.

La memoria será desarrollada empleando la plataforma en línea Overleaf[3], y para facilitar las tareas de sincronización con el repositorio remoto, se hará uso del software GitKraken[4], una interfaz multiplataforma para trabajar con Git.

2.5. Gestión de riesgos

2.5.1. Metodología a seguir para realizar la gestión de riesgos

La gestión de riesgos supone otro de los puntos principales en la planificación de un proyecto, puesto que es necesario detectar los riesgos asociados al mismo para evitar

[2] `https://github.com/`
[3] `https://www.overleaf.com`
[4] `https://www.gitkraken.com/`

incumplir la planificación del trabajo o la estimación temporal. Así como su identificación, también es importante la aplicación de medidas para su prevención, miniaturización o eliminación. De esta forma, para realizar la gestión de riesgos, se seguirán los siguientes pasos:

1. **Identificación de los riesgos:** detección de aquellas amenazas que puedan poner en peligro la viabilidad y desarrollo del proyecto.
2. **Análisis y clasificación:** una vez detectadas las amenazas existentes, serán evaluadas individualmente para determinar si suponen un riesgo para el proyecto. Cada riesgo tendrá asociada una probabilidad e impacto, gracias a los cuales se podrá realizar una planificación frente a éstos. En el caso de que exista un impacto y/o probabilidad muy bajos, serán descartados del plan de respuesta.
3. **Planificación:** una vez clasificados los riesgos, serán explicadas las medidas de prevención, miniaturización o contingencia a aplicar.
4. **Seguimiento:** a lo largo de la vida del proyecto, se comprobará la evolución y posible materialización de los riesgos, con el objetivo de dar respuesta en el menor tiempo posible.

Antes de realizar la gestión de riesgos, es necesario aclarar una serie de métricas, en las que se basará esta gestión:

Medidores de los riesgos

- **Probabilidad** de que se materialice un riesgo:
  - **Baja:** menos de 25 %.
  - **Media:** entre 25 % y 75 %.
  - **Alta:** más de 75 %.
- **Impacto** en tiempo, esfuerzo o coste en caso de que se materialice un riesgo:
  - **Insignificante:** la manifestación de un riesgo tendrá una repercusión mínima en el desarrollo del proyecto.
  - **Tolerable:** la manifestación del riesgo provocará retrasos en la realización de tareas asociadas al proyecto, pero será posible cumplir con los plazos de entrega establecidos.
  - **Serio:** la manifestación del riesgo provocará que alguna entrega intermedia del proyecto no se pueda realizar dentro del plazo indicado, suponiendo una posterior reestructuración de la planificación del trabajo y/o de la estimación temporal, siendo posible que ciertas tareas tengan que ser descartadas.
  - **Catastrófico:** la manifestación del riesgo provocará que no sea posible realizar la entrega final del proyecto.

Tabla 1: Valoración de la probabilidad

Probabilidad	**Valor numérico asociado**
Baja	0.2
Media	0.5
Alta	0.8

Tabla 2: Valoración del impacto

Impacto	**Valor numérico asociado**
Insignificante	0.1
Tolerable	0.4
Serio	0.6
Catastrófico	0.9

- **Nivel de exposición:** se compone de la combinación de los anteriores valores de probabilidad e impacto, convirtiéndolos así en una medida tangible. Para ello, es necesario asociar los anteriores medidores con valores numéricos concretos. La valoración de la probabilidad se puede observar en la tabla 1, y la valoración del impacto, en la tabla 2. Una vez establecidos unos valores numéricos para la probabilidad y el impacto, es posible obtener los valores que reflejen el nivel de exposición. Para obtener esta métrica, se realizará el producto de los valores de probabilidad e impacto, cuyos resultados pueden observarse en la tabla 3. Atendiendo a los valores obtenidos para el nivel de exposición al riesgo, se definen los siguientes tipos de exposición:
  - **Baja:** exposición entre 0 y 0.15.
  - **Media:** exposición entre 0.16 y 0.4.
  - **Alta:** exposición mayor que 0.4.

Tipos de riesgo :

Para realizar la clasificación de los riesgos, se seguirá la categorización de Sommerville [13], en la que se definen tres tipos de riesgos:

- **De proyecto:** afectan a la programación temporal o a los recursos del proyecto.
- **De producto:** afectan a la calidad o desempeño del producto.
- **De negocio:** afectan a la organización responsable del desarrollo del proyecto.

Tabla 3: Nivel de exposición al riesgo

		Impacto			
		Insignificante (0.1)	Tolerable (0.4)	Serio (0.6)	Catastrófico (0.9)
Probabilidad	Baja (0.2)	0.02	0.08	0.12	0.18
	Media (0.5)	0.05	0.2	0.3	0.45
	Alta (0.8)	0.08	0.32	0.48	0.72

Tipos de estrategias a seguir :

Una vez sean identificados los posibles riesgos que puedan afectar, potencialmente, al proyecto, es necesario realizar un análisis de cada uno de ellos, para realizar una categorización en función de su relevancia. Las estrategias a seguir para combatir un potencial riesgo, serán las siguientes:

- **Prevención:** se perseguirá reducir la probabilidad de que un riesgo llegue a materializarse.
- **Miniaturización:** se perseguirá reducir el impacto producido por la materialización de un potencial riesgo.
- **Contingencia:** se aplicará un plan de contingencia en el que estarán indicadas las acciones a realizar en caso de que se materialice un potencial riesgo.
- **Aceptación:** en este caso, no existirá una estrategia contra el riesgo y se asumirá el impacto asociado al mismo, así como sus consecuencias.
- **Transferencia:** la gestión del riesgo se traspasará a otra entidad, que lo deberá gestionar en nuestro lugar.

2.5.2. Identificación de amenazas

Tabla 4: Amenaza A1

Identificador	A1
Nombre	Pérdida del repositorio local.
Descripción	El repositorio del proyecto está almacenado de forma local en la máquina del autor de dicho proyecto. Por algún motivo, como pérdida del equipo o corrupción del disco duro, se pierden los datos correspondientes a la última versión de los elementos de la configuración.

Tabla 5: Amenaza A2

Identificador	A2
Nombre	Atraso en la realización de uno de los objetivos.
Descripción	Alguno de los objetivos indicados en la sección 2.2 no es entregado dentro de los plazos indicados en la sección 2.3, debido a improvistos en el desarrollo del proyecto.

Tabla 6: Amenaza A3

Identificador	A3
Nombre	Elementos de la configuración no identificados.
Descripción	Algunos de los elementos de la configuración son ignorados a la hora de realizar su identificación, por lo que no serán gestionados por el proceso de gestión de la configuración definido en la sección 2.4

Tabla 7: Amenaza A4

Identificador	A4
Nombre	Identificación de elementos de la configuración innecesarios.
Descripción	Son detectados elementos de la configuración que finalmente no resultan relevantes para la ejecución del proyecto, lo que provoca atrasos a la hora de realizar la gestión de la configuración del proyecto.

Tabla 8: Amenaza A5

Identificador	A5
Nombre	Definición de un proceso de gestión de la configuración ineficaz y/o incorrecto.
Descripción	A la hora de ejecutar el proceso de gestión de la configuración, se observan errores o una cantidad de documentación necesaria que dificultan la ejecución de los objetivos precisos para la ejecución del proyecto.

Tabla 9: Amenaza A6

Identificador	A6
Nombre	Entrega de una versión incorrecta.
Descripción	A la hora de realizar alguna de las entregas correspondientes al proyecto, tanto entregas intermedias como la final, indicadas en la sección 2.2, se produce una confusión en la versión a entregar.

Tabla 10: Amenaza A7

Identificador	A7
Nombre	Pérdida de cambios de especial relevancia entre diferentes versiones de la gestión de la configuración.
Descripción	Las actualizaciones de versiones realizadas en la gestión de la configuración están demasiado espaciadas en el tiempo, lo que provoca que, en caso de querer volver a alguna versión anterior del proyecto, se pierdan cambios que puedan ser considerados de relevancia.

Tabla 11: Amenaza A8

Identificador	A8
Nombre	Equipo de trabajo indisponible.
Descripción	El equipo de trabajo del autor del proyecto deja de estar disponible debido a algún acontecimiento inesperado, como un robo o un daño hardware, impidiendo continuar la realización del proyecto.

Tabla 12: Amenaza A9

Identificador	A9
Nombre	Corrupción de los ficheros que conforman el proyecto.
Descripción	Debido a algún fallo software o a una mala manipulación de los ficheros, alguno de los elementos de la configuración indicados en la sección 2.4.1 resulta corrupto, siendo imposible acceder a la información que contiene. Esta amenaza tiene dos posibilidades: ▪ Corrupción en el equipo local del autor del proyecto. ▪ Corrupción en el sistema remoto indicado en la sección 2.4.3.

Tabla 13: Amenaza A10

Identificador	A10
Nombre	Caída de la plataforma en línea para la edición de la memoria.
Descripción	La plataforma en línea usada para realizar la edición de la memoria del proyecto, indicada en la sección 2.4.3, deja de estar disponible, momentánea o definitivamente, durante el periodo de tiempo en el que se desarrolla el proyecto, impidiendo continuar con su edición.

2.5.3. Identificación de activos

ACT1: Equipo de trabajo.

ACT2: Elementos de la configuración.

ACT3: Repositorio.

ACT4: Proceso de gestión de la configuración.

2.5.4. Análisis de riesgos

Los riesgos constituyen la probabilidad de que las amenazas, definidas en la sección 2.5.2, actúen sobre los activos, definidos en la sección 2.5.3, causando daños o pérdidas. Por lo tanto, cada uno de los riesgos especificados en esta sección estará directamente relacionado con cada una de las amenazas presentadas previamente. Esta correlación estará indicada por el número existente en el identificador del riesgo, que coincidirá siempre con el número existente en los identificadores de las amenazas.

Tabla 14: Riesgo R1

Identificador	R1	**Tipo de riesgo**	De proyecto
Probabilidad	Baja	**Impacto**	Insignificante
Exposición	Baja	**Activos afectados**	ACT1, ACT2, ACT3
Tratamiento	**Aceptación:** entendemos que los últimos datos del proyecto, para los cuales solo existiría una copia local, serían irrecuperables. No obstante, gracias al proceso de gestión de la configuración, donde se define un repositorio remoto que, además, actúa como copia de seguridad, podríamos recuperar los datos de la última versión del proyecto. Puesto que las actualizaciones entre versiones se realizan con una frecuencia elevada, solamente habría que repetir los últimos cambios, suponiendo unas pocas horas extra de trabajo.		
Indicadores	El sistema no permite acceder a los últimos cambios en disco realizados en el proyecto.		

Tabla 15: Riesgo R2

Identificador	R2	**Tipo de riesgo**	De proyecto
Probabilidad	Media	**Impacto**	Serio
Exposición	Media	**Activos afectados**	ACT2
Tratamiento	▪ **Prevención:** realizar una buena planificación. ▪ **Miniaturización:** descartar tareas con poca relevancia para el desarrollo final del proyecto.		
Indicadores	El plazo especificado para alguna de las entregas que componen los objetivos supera su planificación inicial por un plazo superior a tres días.		

Tabla 16: Riesgo R3

Identificador	R3	**Tipo de riesgo**	De producto
Probabilidad	Media	**Impacto**	Tolerable
Exposición	Media	**Activos afectados**	ACT2, ACT4
Tratamiento	**Prevención:** identificación adecuada de los elementos de la configuración y realización de una gestión de la configuración simple y eficiente. **Miniaturización:** en caso de detectar el problema, realizar las modificaciones pertinentes en la gestión de la configuración para añadir los nuevos elementos de la configuración no identificados. Ya que el gestor de la configuración se trata, igualmente, del autor del proyecto, realizar pequeños cambios justificados en la gestión de la configuración inicial no debería constituir un problema.	
Indicadores	Conforme el proyecto avanza, se observa que ciertos elementos que forman parte del proyecto no están siendo contemplados en la gestión de la configuración, al no haber sido identificados como elementos de la configuración.		

Tabla 17: Riesgo R4

Identificador	R4	**Tipo de riesgo**	De producto
Probabilidad	Media	**Impacto**	Tolerable
Exposición	Media	**Activos afectados**	ACT2, ACT4
Tratamiento	**Prevención:** identificación adecuada de los elementos de la configuración y realización de una gestión de la configuración simple y eficiente. **Miniaturización:** en caso de detectar el problema, realizar las modificaciones pertinentes en la gestión de la configuración para excluir los elementos innecesarios. Ya que el gestor de la configuración se trata, igualmente, del autor del proyecto, realizar pequeños cambios justificados en la gestión de la configuración inicial no debería constituir un problema.	
Indicadores	Conforme el proyecto avanza, se observa que ciertos elementos que realmente no son precisos para el desarrollo del proyecto, están siendo contemplados en la gestión de la configuración, por lo que su control supone una pérdida innecesaria de recursos.		

Tabla 18: Riesgo R5

Identificador	R5	**Tipo de riesgo**	De producto
Probabilidad	Media	**Impacto**	Tolerable
Exposición	Media	**Activos afectados**	ACT2, ACT4
Tratamiento	**Prevención:** realización de una gestión de la configuración eficaz y eficiente. **Miniaturización:** en caso de detectar el problema, realizar las modificaciones pertinentes en la gestión de la configuración para que ésta se adapte correctamente al proyecto. Ya que el gestor de la configuración se trata, igualmente, del autor del proyecto, realizar pequeños cambios justificados en la gestión de la configuración inicial no debería constituir un problema.		
Indicadores	Conforme el proyecto avanza, se observan serios problemas para llevar a cabo una correcta gestión de la configuración (por ejemplo, dificultades o imposibilidad para volver a una versión anterior del proyecto).		

Tabla 19: Riesgo R6

Identificador	R6	**Tipo de riesgo**	De producto
Probabilidad	Baja	**Impacto**	Catastrófico
Exposición	Media	**Activos afectados**	ACT2
Tratamiento	**Prevención:** revisión, al menos en dos ocasiones, de que la versión entregada concuerda con las últimas modificaciones existentes en el proyecto.		
Indicadores	La versión del proyecto entregada en el campus virtual no coincide con la última versión disponible del proyecto.		

Tabla 20: Riesgo R7

Identificador	R7	**Tipo de riesgo**	De proyecto
Probabilidad	Media	**Impacto**	Tolerable
Exposición	Media	**Activos afectados**	ACT2, ACT4
Tratamiento	**Prevención:** establecer una gestión de la configuración ágil, que permita la creación de nuevas versiones del proyecto sin la realización de grandes modificaciones.		
Indicadores	Al querer retroceder a una versión anterior del proyecto, empleando la gestión de la configuración, se pierden muchos datos entre las versiones.		

Tabla 21: Riesgo R8

Identificador	R8	**Tipo de riesgo**	De negocio
Probabilidad	Baja	**Impacto**	Catastrófico
Exposición	Media	**Activos afectados**	ACT1
Tratamiento	**Prevención:** posibilidad de acceder a un equipo de trabajo de repuesto para poder continuar el proyecto en un periodo de tiempo inferior a 12 horas.		
Indicadores	No es posible trabajar con el equipo de trabajo, o hacerlo de manera adecuada para el correcto desarrollo del proyecto.		

Tabla 22: Riesgo R9

Identificador	R9	**Tipo de riesgo**	De proyecto
Probabilidad	Baja	**Impacto**	Catastrófico
Exposición	Media	**Activos afectados**	ACT2
Tratamiento	▪ **En el equipo local → Contingencia:** Al contar con una copia de seguridad, gracias al repositorio remoto empleado en la gestión de la configuración, en caso de que se corrompieran los ficheros locales, se intentaría acceder a dicha copia de seguridad. ▪ **En el sistema remoto → Transferencia:** Entendemos que la plataforma empleada para almacenar el repositorio remoto del proyecto posee medidas propias para evitar este tipo de situaciones.		
Indicadores	Al acceder a alguno de los ficheros que compone el proyecto, este no se puede leer correctamente.		

Tabla 23: Riesgo R10

Identificador	R10	**Tipo de riesgo**	De negocio
Probabilidad	Baja	**Impacto**	Catastrófico
Exposición	Media	**Activos afectados**	ACT2
Tratamiento	**Contingencia:** al estar definidos como elementos de la configuración todos los ficheros TEXque componen la memoria del proyecto, estarían almacenados de forma local y también en el repositorio remoto. Por tanto, el plan de contingencia pasaría por subir todos los documentos a otra plataforma en línea de edición de texto LATEX, o emplear un editor local en la propia máquina de trabajo.		
Indicadores	La plataforma en línea para la edición de la memoria, indicada en la sección 2.4.3 no está disponible, temporal o definitivamente.		

2.5.5. Seguimiento de los riesgos

Realizados los pasos de identificación y análisis, en el que se incluye la planificación, solamente queda realizar un seguimiento de los riesgos a lo largo de la vida del proyecto, comprobando su materialización en función de los indicadores detallados y aplicar el seguimiento correspondiente, ambos descritos en la sección 2.5.4.

Riesgos materializados :

- **Riesgo R2**, recogido en la tabla 15, directamente asociado con la **amenaza A2**, recogida en la tabla 5. Este riesgo tenía una clasificación con probabilidad media, impacto serio y, por tanto, una exposición final media. Como fue especificado anteriormente, la manifestación de un riesgo con impacto serio, provocaría que alguna entrega intermedia del proyecto no se pudiera realizar dentro del plazo entregado, suponiendo una posterior reestructuración de la planificación del trabajo. Cumpliéndose las estimaciones realizadas con la gestión de riesgos realizada, la tercera entrega parcial del proyecto, correspondiente a la fase de análisis, reflejada en la EDT representada en la figura 3, tuvo que ser postergada hasta el 9 de mayo de 2021 (fecha de entrega original: 27 de abril de 2021).

  Concretizando un poco más sobre dicha materialización, ésta vino provocada por una situación personal complicada para el autor y único realizador del proyecto, lo que obligó a pausar la realización de este trabajo durante un par de semanas a lo largo de la fase de análisis. Para poder paliar dicho riesgo, fue preciso realizar una mayor carga de trabajo en la fase de detección y corrección o prevención, al disponer de un tiempo menor para su realización. Además, también fue aplicado el tratamiento de miniaturización, teniendo que descartar la securización de uno los requisitos especificados en la sección 4.2, pasando dicho estudio a la sección de trabajo futuro.

El riesgo R2 fue el único materializado a lo largo de la vida de desarrollo del proyecto.

3. Conceptos básicos

En la sección 1, fueron introducidas una serie de tecnologías con las que trabajaremos para la realización del presente proyecto. En esta sección se desarrollarán un poco más los principales conceptos que envuelven a dichas tecnologías.

3.1. Máquinas virtuales

Debemos tener en cuenta que la necesidad de crear entornos aislados y portátiles no se trata de ninguna novedad, sino que las máquinas virtuales ya introdujeron previamente esta idea, con la posibilidad de incluir en su interior todas las dependencias software precisas: librerías, código y datos, con la posibilidad de replicar el mismo entorno en cualquier lugar.

Las máquinas virtuales ofrecen la posibilidad de crear entornos completamente aislados, ya que en su interior poseen un sistema operativo completo, incluyendo una administración propia de la memoria y de controladores para dispositivos virtuales. Es precisamente esta necesidad de replicar ciertos recursos, en lugar de compartirlos con la máquina anfitriona, la que convierte al modelo de virtualización mediante máquinas virtuales en un modelo menos eficiente y más pesado. Precisamente, su modelo de funcionamiento acarreará una serie de ventajas y desventajas. Por un lado, además de resultar ambientes menos ágiles y portátiles, debido al uso que las máquinas virtuales hacen del hipervisor, se añade un nivel de complejidad en el cual resulta complicado que los entornos contenidos virtualizados puedan llegar a hacer uso de recursos propios de la máquina anfitriona, como pueden ser aceleradores hardware o unidades de procesamiento gráfico para computación paralela [5], recursos ampliamente empleados en determinados entornos.

No obstante, desde el punto de vista de la seguridad informática, es precisamente este nivel "extra" de abstracción el que permite que sea considerado seguro otorgar privilegios de superusuario en el interior de las máquinas virtuales. Es justamente este uso que las máquinas virtuales hacen del hipervisor el que impide que una máquina virtual consiga ejecutar instrucciones que puedan poner en riesgo la integridad de la máquina que la contiene. [11]

3.2. Tecnologías de contenerización

Con la incorporación de funciones de virtualización ligeras para el *kernel* de GNU/Linux, por ejemplo, con la implantación de los espacios de nombre (*namespaces*), fue posible implementar una nueva forma de virtualización: la virtualización a nivel de sistema operativo, también conocida como contenerización.

La contenerización se trata de un método de virtualización que permite la ejecución de múltiples entornos bajo una misma máquina anfitriona común, consiguiendo establecer claras separaciones entre los diferentes procesos. Como hemos visto, estas ventajas también pueden ser ofrecidas por otras tecnologías de virtualización, como pueden ser las máquinas virtuales; no obstante, la utilización de contenedores conlleva otra serie de

mejoras, como puede ser la compartición del *kernel* con la máquina anfitriona, empleando solamente una serie de capas ligeras o librerías sobre el sistema operativo. El uso de estas librerías permitirá traducir o redireccionar las llamadas al sistema operativo, de tal forma que se puedan ejecutar directamente sobre el hardware compartido, haciendo uso, para ello, del *kernel* de la propia máquina anfitriona [6]. Como resultado, la virtualización a nivel de sistema operativo supone una pérdida de rendimiento mucho menor en comparación con otras tecnologías, pudiendo llegar a considerarse una pérdida despreciable. También debemos tener en cuenta que cuando trabajamos con máquinas virtuales estamos creando sistemas con unos límites muy definidos, en términos de CPU, memoria y demás recursos que, en muchos casos, suponen una limitación inferior y superior de los recursos de forma exclusiva. Este modelo implicará que, probablemente, los recursos asignados a las máquinas virtuales no vayan a ser empleados en su totalidad, o, al menos, la mayor parte del tiempo. Sin embargo, el uso de contenedores permite unos límites más laxos que se adaptan en función a los recursos disponibles, contando con la ventaja de poder establecer unos límites máximos en la mayor parte de los casos, dependiendo de la tecnología de contenerización empleada.

Gracias a esta serie de características, los contenedores suponen una solución más ligera, rápida y fácil de escalar que el uso de máquinas virtuales, permitiendo un funcionamiento más flexible [11]. Por el contrario, desde el punto de vista de la seguridad informática, es precisamente esta compartición de recursos, que permite agilizar el despliegue y funcionamiento de los contenedores, la que supone un mayor riesgo, en caso de que alguno de los contenedores o la propia tecnología de contenerización se vean comprometidos.

A lo largo de este estudio, trataremos de trabajar, en la medida de lo posible, con la tecnología de contenerización Docker, al tratarse actualmente del estándar de facto a la hora de trabajar con virtualización a nivel de sistema operativo, de forma que obtengamos unas pruebas lo más estandarizadas posible.

Finalmente, es importante diferenciar un par de conceptos básicos a la hora de trabajar con tecnologías de contenerización:

- **Imagen:** una imagen se trata de un paquete ejecutable que incluye todo lo necesario para correr una aplicación o similar (código, librerías, variables de entorno, etc.).
- **Contenedor:** un contenedor, por el contrario, se trata de la instancia en tiempo de ejecución de una imagen, es decir, en lo que se transforma una imagen en memoria cuando es ejecutada. [23]

3.3. Docker Hub

Asociada a la tecnología de contenerización Docker, aquélla seleccionada para ser usada a lo largo de la vida del proyecto, existe un portal de almacenamiento de imágenes de contenedores denominado Docker Hub[5], en el que se recogen imágenes de contenedores de

[5] `https://hub.docker.com/`

diferentes fuentes, así como diversas informaciones útiles, como el nombre del autor, si se trata de una imagen oficial o no, etc., constituyéndose así como un recurso centralizado para el descubrimiento, distribución, gestión de cambios y automatización del flujo de trabajo en toda la línea de desarrollo. [24]

Este servicio permite buscar entre multitud de imágenes, a través de las cuales podemos obtener sistemas totalmente configurados para su despliegue y funcionamiento. Otra característica que debemos tener en cuenta es que las nuevas imágenes no tienen por qué ser hechas desde cero, sino que es posible extender imágenes ya existentes, creando así una relación padre-hijo entre las diferentes imágenes. [7]

3.4. Kubernetes

Comprendidas las ventajas ofrecidas por las tecnologías de virtualización a nivel de sistema operativo y atendiendo a la evolución cara los microservicios, como vimos en la sección de introducción, es posible entender la actual proliferación del uso de las tecnologías de contenerización, al resultar éstas el escenario ideal para contener estas pequeñas piezas unitarias de software.

No obstante, pese a ser posible ejecutar estos microservicios en diferentes contenedores independientes, debemos tener en cuenta que también existe la problemática de crear, gestionar, monitorizar, destruir, estos pequeños entornos independientes, puesto que las tecnologías de contenerización no resuelven este problema por sí mismas. Por tanto, la necesidad de gestionar entornos conformados por gran cantidad de microservicios, cada uno de ellos almacenado en un contenedor propio e independiente, surgen como respuesta las herramientas de orquestación: herramientas capaces de gestionar grandes cantidades de contenedores, incluso si estos se encuentran distribuidos en diferentes máquinas o entornos. Una de las herramientas de orquestación más populares se trata de Kubernetes. Los principales objetivos de Kubernetes y, en general, de todas las herramientas de orquestación, son [43]:

- Gestionar numerosos (cientos o miles) contenedores.
- Gestionar contenedores dispersos en entornos múltiples y heterogéneos.
- Proveer alta disponibilidad, ofreciendo los mecanismos necesarios para garantizar que los microservicios contenidos en los diferentes contenedores estén siempre disponibles.
- Escalabilidad, ofreciendo los mecanismos necesarios para replicar los servicios ofrecidos por un contenedor en múltiples instancias, de forma que se consigue un alto rendimiento.
- Recuperación ante desastre, existiendo un mecanismo que consiga almacenar los datos y recuperando el servicio siempre en su último estado.
- Balance de la carga de trabajo: si el tráfico para un determinado contenedor fuese alto, es posible balancear la carga y distribuir el tráfico de red para ofrecer un servicio estable.

- Orquestación del almacenamiento, permitiendo gestionar un sistema de almacenamiento entre diferentes opciones, como almacenamientos locales o en la nube.
- Gestión de ficheros de configuración, existiendo la posibilidad de gestionar datos privados, tales como credenciales, *tokens*, claves SSH, etc.

3.4.1. Componentes principales

Para poder cumplir con los objetivos especificados anteriormente, Kubernetes hará uso de una serie de componentes que definen su comportamiento. Los principales componentes de Kubernetes son:

Nodos (*nodes*) :

Se tratan de las máquinas sobre las que correrán los contenedores y la herramienta de orquestación. Es decir, las diferentes máquinas físicas o virtuales.

Pod :

Un *pod* se trata de la unidad de gestión más pequeña de Kubernetes, la cual supone una abstracción sobre un contenedor. Para formarse, pueden crear un contenedor basado en una imagen, de las miles de disponibles en servicios repositorios como, por ejemplo, el ya mencionado DockerHub. Normalmente, un *pod* contiene una sola aplicación (o microservicio) contenerizada, pero no se trata realmente de una restricción estricta. Por ejemplo, es común que se agrupen diversas aplicaciones dentro de un mismo *pod* en caso de que exista un enlace muy fuerte entre ellas.

Puesto que los *pods* se tratan de una abstracción sobre un contenedor, también sirven para abstraernos de la propia tecnología de contenerización, haciendo que no sea necesario trabajar con ellas directamente. Por ejemplo, en el caso de querer utilizar contenedores Docker, gracias a Kubernetes nunca sería necesario interactuar con esta tecnología, sino que simplemente interactuaríamos con la capa de la herramienta de orquestación.

Para comunicarse entre sí, los *pods* emplean sus direcciones IP, por lo que debemos tener en cuenta que cada *pod* tendrá su propia dirección. Esto es, para interactuar, se hará uso de la dirección IP asignada al *pod*, y no al contenedor. No obstante, estas direcciones no están pensadas para ser expuestas públicamente, de cara a un uso final en producción, puesto que los *pods* son tratados dinámicamente y, por tanto, sus direcciones IP son temporales y efímeras.

Servicio (*service*) :

Para solucionar el problema de obtención de una dirección IP estática asignada a cada *pod*, existen los denominados *services*, que se encargarán de comunicarse con los *pods* y servir como una capa de abstracción a nivel de IP, de tal forma que, aunque los *pods* varíen internamente su dirección IP, por ejemplo, porque un *pod* cae y es substituido por otro,

se mantendrá siempre la misma dirección IP, dada por el servicio. Podemos diferenciar dos tipos de servicios:

- **Internos:** la dirección IP dada no sería visible desde el exterior. Normalmente, son empleadas para establecer direcciones IP de bases de datos que son consultadas por aplicaciones internas, pero que no queremos que dichas bases de datos sean directamente accesibles desde el exterior, por ejemplo.
- **Externos:** se trata precisamente del caso contrario, de direcciones IP que queremos que sean accesibles desde el exterior, como, por ejemplo, aplicaciones web.

Ingress :

Puesto que lo más común es que el usuario final no quiera acceder a los servicios ofrecidos por la aplicación final desde una dirección IP, sino que querrá acceder a través de un dominio, para ello existe el componente *ingress*. Así, teniendo en cuenta los componentes vistos hasta el momento, si un usuario quiere acceder a una aplicación o microservicio, la petición seguiría el siguiente recorrido: Petición → *Ingress* → *Service* → Dirección IP interna → *Pod* → Contenedor → Aplicación o microservicio contenerizado.

ConfigMap :

Para el caso de aquellas aplicaciones que precisen de ficheros de configuración, donde indicar, por ejemplo, direcciones IP, puertos, etc., esta serie de propiedades serán especificadas en el *ConfigMap*.

Secret :

Tiene un propósito casi idéntico al *ConfigMap*, pero para almacenar aquellos datos que son considerados privados, como nombres de usuario o contraseñas, información que nunca debería estar almacenada en el *ConfigMap*, puesto que sería almacenada en texto plano, mientras que en el *Secret* es almacenada cifrada.

Debemos tener en cuenta que el uso del *Secret* no está habilitado por defecto, sino que debe ser configurado para poder ser usado.

Volúmenes (*volumes*) :

Los *volumes* no se tratan sino del propio almacenamiento del que harán uso los *pods* y, por tanto, los contenedores y las aplicaciones que contengan. Básicamente, anexa un dispositivo de almacenamiento, que puede ser local, remoto, etc., a un *pod*.

Kubernetes no se encargará explícitamente de realizar la gestión de los datos y su persistencia, por lo que es responsabilidad del administrador de sistemas realizar tal administración: por ejemplo, será responsable de organizar y realizar copias de seguridad y la posibilidad de restaurarlas en caso de desastre.

Deployments :

Vimos anteriormente que uno de los objetivos de Kubernetes es poder proveer alta disponibilidad, ofreciendo mecanismos para asegurar que los servicios contenidos en los contenedores están siempre disponibles. Para poder ofrecer esta alta disponibilidad es necesario replicar los diferentes *pods*, preferiblemente entre varias máquinas, de tal forma que si alguno de ellos cae, otra de las réplicas pueda dar respuesta, ofreciendo el mismo servicio. Básicamente, los *deployments* permitirán redirigir el tráfico a *pods* disponibles para prestar el servicio requerido. Por tanto, en la práctica, no trabajaremos directamente con *pods*, sino con *deployments*, funcionando así como un nuevo nivel de abstracción sobre los *pods*. Así, el usuario final realizaría una petición que seguiría el siguiente recorrido: Petición → *Ingress* → *Service* → *Deployment* → Dirección IP interna → *Pod* → Contenedor → Aplicación o microservicio contenerizado.

Sin embargo, debemos tener en cuenta que el uso de *deployments* está pensado para aquellas aplicaciones o microservicios que no dependan de una consistencia en los datos almacenados. Es decir, la respuesta dada debe ser la misma independientemente de los datos internos que las aplicaciones procesen. Por ejemplo, un microservicio que ofrezca la generación de contraseñas aleatorias: sería posible distribuir este microservicio entre varios contenedores en varias máquinas diferentes. Pero otras aplicaciones, como bases de datos, no son aptas para ser utilizadas directamente con *deployments*, porque no sería posible mantener una consistencia de los datos almacenados en los diferentes *pods*.

StatefulSet :

Para poder solucionar el problema de tratar con datos persistentes entre diferentes *pods*, surgen los *StatefulSets*. Su propósito es, precisamente, conseguir lecturas y escrituras de manera sincronizada entre diferentes *pods*, de tal forma que se eviten las inconsistencias entre las diferentes instancias. No obstante, no resultan fáciles de controlar, por lo que una práctica común es, simplemente, dejar aquellos servicios con necesidad de persistencia de datos, como pueden ser las bases de datos, fuera del clúster gestionado por Kubernetes.

3.4.2. Arquitectura de Kubernetes

Ahora que conocemos qué es Kubernetes, así como sus principales componentes, podemos profundizar un poco en cómo está estructurado, en su arquitectura.

Cuando tenemos una infraestructura basada en Kubernetes, debemos considerar que existirán dos tipos de nodos: los nodos maestros (*master*) y los nodos esclavos o trabajadores (*slave* o *worker*). Los nodos *master* y los nodos *worker* tendrán diferentes componentes en su interior, ya que no poseen el mismo cometido.

Nodos trabajadores :

Su funcionamiento depende de tres procesos diferentes, que deben ser instalados en cada uno de los nodos trabajadores:

- ***Container runtime*:** el software responsable de correr los contenedores. Es decir, la propia tecnología de contenerización, como puede ser Docker o Containerd.
- **Kublet:** es el proceso encargado de calendarizar las ejecuciones de los *pods* y los contenedores, encargándose por tanto de interactuar tanto como con la propia máquina (el nodo), como con los contenedores que contiene.
- **Kube *proxy*:** es el encargado de realizar las distribuciones de las peticiones entre los diferentes elementos, balanceando la carga hacia los *pods* disponibles con una menor latencia, de tal forma que tiempo de respuesta sea óptimo.

Nodos *master* :

Sobre ellos recae la responsabilidad de gestionar toda la red de contenedores que ofrecen el servicio distribuido en microservicios, por lo que desempeñan el papel más importante. A pesar de esto, no se tratarán de los nodos que asumirán la mayor carga de trabajo, por lo que no es necesario que tengan demasiados recursos. Los procesos implicados en su funcionamiento y, por tanto, en el funcionamiento del clúster, son:

- ***API Server*:** se trata del componente con el que realmente interactuaremos, empleando para ello algún cliente. Se trata de una especie de puerta de enlace y sistema de autenticación, al ser el encargado de recibir las peticiones, validarlas y redirigirlas a algún otro proceso interno. Desde el punto de vista de la seguridad esto simplifica las cosas, ya que solamente existirá un punto de entrada al clúster.
- ***Scheduler*:** este proceso posee la inteligencia para saber en qué nodo asignar los *pods* o cualquier otro componente.
- ***Controller Manager*:** es el encargado de detectar la caída de los nodos, con el fin de conseguir reprogramarlos y volver a levantarlos lo antes posible.
- `etcd`**:** puede ser interpretado como el "cerebro" del clúster, ya que todos los cambios que sucedan en el clúster, tales como detección de caídas, reprogramaciones, etc., serán almacenadas en la *Key Value Store*, que está gestionada por `etcd`. Por lo tanto, todos los datos relativos a los estados de los diferentes componentes del clúster, están almacenados y gestionados por este proceso.

Una representación gráfica de los conceptos de arquitectura y componentes vistos en esta sección puede ser consultada en la figura 4.

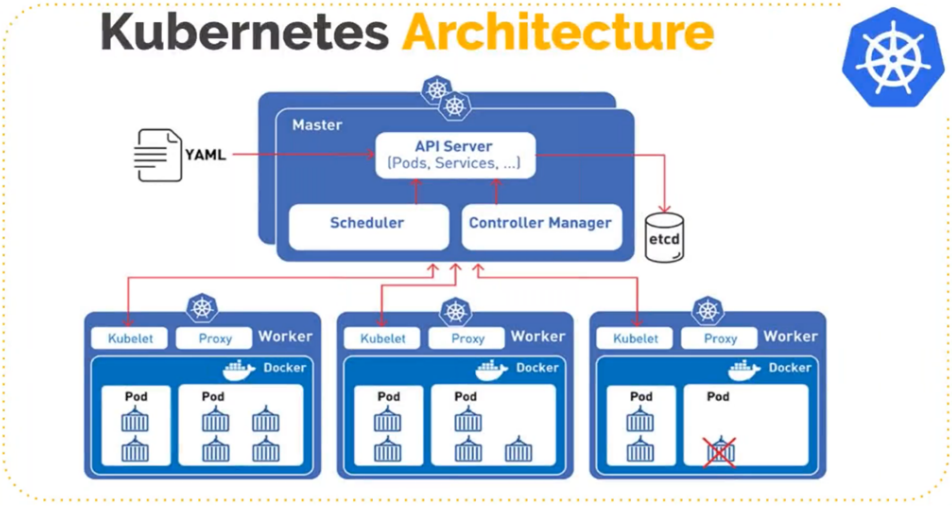


Figura 4: Diagrama de la arquitectura y componentes de Kubernetes

Fuente: [26]

4. Especificación de requisitos

Realizada una planificación del proyecto y su desarrollo, así como explicados los conceptos teóricos con los que trabajaremos, especificaremos una serie de requisitos que definirán la evolución de dicho proyecto.

Debemos tener en cuenta que este proyecto no se trata de un desarrollo software, por lo que muchas de las categorizaciones habituales de requisitos en este tipo de proyectos no podrán tener lugar. Por ejemplo, al no tratarse de un desarrollo software *per se*, no existirán especificaciones referentes a: casos de uso, actores, requisitos funcionales, matriz de trazabilidad, etc. Por tanto, la especificación de requisitos quedará limitada a la definición de los requisitos no funcionales.

4.1. Limitaciones

Además de los requisitos, también existirán una serie de limitaciones que definirán, igualmente, el desarrollo del proyecto. Las limitaciones existentes para este estudio son las siguientes:

- **Creación del laboratorio de pruebas en un ambiente local:** aunque muchos portales de computación en la nube ofrecen la posibilidad de instalar y configurar un laboratorio de Kubernetes al estilo *cloud*, dichos entornos en la nube no ofrecerán, probablemente, tanta flexibilidad a la hora de realizar modificaciones en las configuraciones de Kubernetes o las tecnologías de contenerización implicadas. Por tanto, para asegurar la mayor flexibilidad posible en las configuraciones que fuere preciso realizar, se realizará la instalación y configuración de un laboratorio de pruebas local.
- **Creación del laboratorio de pruebas en una única máquina física, mediante la ayuda de máquinas virtuales:** aunque un laboratorio de trabajo repartido en diferentes máquinas físicas podría resultar interesante para el estudio que nos ocupa, existe la limitación de no poseer suficientes máquinas disponibles para realizar dicho despliegue. Por tanto, el laboratorio de pruebas tendrá que ser creado dentro de una sola máquina.
- **Uso de software libre o gratuito:** puesto que el presente proyecto no cuenta con ningún tipo de financiación, resulta importante reducir los gastos empleando software libre o, como mínimo, gratuito.
- **Limitación temporal:** el presente proyecto presenta una fuerte limitación temporal, al deber seguir los plazos de entrega especificados por la Universitat Oberta de Catalunya para la correcta superación de la asignatura asociada al Trabajo de Final de Máster. Dichas limitaciones temporales están especificadas en la sección 2.3.

4.2. Listado de requisitos no funcionales

Los requisitos no funcionales son aquellos que, si bien indican una serie de requerimientos para alcanzar la realización del proyecto, no pueden ser considerados como

funcionalidades. Para la creación del listado de requisitos no funcionales se tuvieron en cuenta los estudios realizados a lo largo de la fase de estudio preliminar, establecida en la EDT representada en la figura 3. Por tanto, el listado de requisitos se basa en los conocimientos extraídos por el autor del proyecto en sus primeras fases de desarrollo, pero siempre tomando como referencia otros estudios de vulnerabilidades realizados por investigadores especializados en la securización de entornos virtualizados. [3][15][27]

Tabla 24: Requisito no funcional RNF1

ID	RNF1
Nombre	Análisis estático sobre imágenes de contenedores
Descripción	Comprobación de la existencia (o inexistencia) de vulnerabilidades en imágenes de contenedores descargadas desde fuentes bien conocidas y consideradas "de confianza". Aplicación de correcciones, en caso de que aplique.

Tabla 25: Requisito no funcional RNF2

ID	RNF2
Nombre	Análisis estático sobre ficheros de configuración YAML
Descripción	Los análisis estáticos no tienen por qué limitarse a la localización de vulnerabilidades en el software, sino que también pueden ser de ayuda para alertar de malas configuraciones o malas prácticas en los despliegues. Para ello, se comprobará el fichero de configuración YAML en el que se indica la información relativa a un despliegue. Aplicación de correcciones, en caso de que aplique.

Tabla 26: Requisito no funcional RNF3

ID	RNF3
Nombre	Limitación de recursos: memoria RAM
Descripción	Comprobar si existe la posibilidad de usar recursos de memoria en las máquinas que corren los contenedores de una forma excesiva. Es decir, que una aplicación pueda llegar a consumir tanta memoria que impida el funcionamiento normal, parcial o totalmente, de las otras aplicaciones ubicadas en otros contenedores, pero dentro de la misma máquina. Aplicación de correcciones, en caso de que aplique.

Tabla 27: Requisito no funcional RNF4

ID	RNF4
Nombre	Limitación de recursos: CPU
Descripción	Comprobar si existe la posibilidad de usar recursos de CPU en las máquinas que corren los contenedores de una forma excesiva. Es decir, que una aplicación pueda llegar a consumir tanto CPU que impida el funcionamiento normal, parcial o totalmente, de las otras aplicaciones ubicadas en otros contenedores, pero dentro de la misma máquina. Aplicación de correcciones, en caso de que aplique.

Tabla 28: Requisito no funcional RNF5

ID	RNF5
Nombre	Limitación de recursos: red
Descripción	Comprobar si existe la posibilidad de usar recursos de red, como el ancho de banda, en las máquinas que corren los contenedores de una forma excesiva. Es decir, que una aplicación pueda llegar a acaparar el ancho de banda, impidiendo el funcionamiento normal, parcial o totalmente, de las otras aplicaciones ubicadas en otros contenedores, pero dentro de la misma máquina. Aplicación de correcciones, en caso de que aplique.

Tabla 29: Requisito no funcional RNF6

ID	RNF6
Nombre	Limitación de los permisos de ejecución dentro de los contenedores
Descripción	Comprobar cuáles son los permisos existentes por defecto cuando invocamos la ejecución de un nuevo contenedor a través de la herramienta de orquestación Kubernetes. En caso de que existan permisos de superusuario (`root`), razonar sobre su necesidad y aplicar correcciones, en caso de que aplique.

Tabla 30: Requisito no funcional RNF7

ID	RNF7
Nombre	Comparativa de aislamiento entre virtualizaciones: máquinas virtuales VS contenedores
Descripción	Desde el comienzo de este proyecto, se han expuesto las diferencias, ventajas e inconvenientes existentes entre los contenedores y las máquinas virtuales. Si bien la virtualización a nivel de SO ofrece una metodología y despliegues más ágiles, presentan un aislamiento que podría ser considerado "menor", en comparación con la virtualización de hardware. Debemos comprobar estas diferencias y aplicar correcciones, en caso de que aplique.

Tabla 31: Requisito no funcional RNF8

ID	RNF8
Nombre	Estudio de sistemas de solo lectura
Descripción	En ciertas ocasiones, podemos querer desplegar cierto tipo de aplicaciones para las que la escritura en disco es algo totalmente prescindible. Es decir, una vez la aplicación esté correctamente desplegada y configurada, ésta solamente precisará realizar lecturas en disco, pero no más escrituras. En este tipo de situación, permitir que cualquier aplicación pueda escribir en disco de forma indiscriminada puede constituir una vulnerabilidad innecesaria. Por ejemplo, en caso de un mal funcionamiento o ataque, podrían sobreescribir ficheros de configuración importantes. Por tanto, debemos comprobar si es posible realizar escrituras indiscriminadas en la configuración por defecto y aplicar correcciones, en caso de que aplique.

Tabla 32: Requisito no funcional RNF9

ID	RNF9
Nombre	Validación de imágenes mediante firma digital
Descripción	Explorar la posibilidad de establecer un mecanismo para asegurar que el contenido de una imagen a desplegar en un clúster Kubernetes no fue modificada por una tercera persona y que los datos contenidos por la misma son idénticos a los dados por su desarrollador, contando para ello de la ayuda de las firmas digitales.

Tabla 33: Requisito no funcional RNF10

ID	RNF10
Nombre	Herencia de vulnerabilidades entre imágenes
Descripción	Comprobación de la posible herencia de vulnerabilidades entre imágenes dependientes.

Tabla 34: Requisito no funcional RNF11

ID	RNF11
Nombre	Diferenciación de espacios para diferentes aplicaciones y/o equipos
Descripción	En un entorno contenerizado grande, es común contar con diferentes equipos ejecutando diferentes aplicaciones. Dichos equipos y/o aplicaciones no tienen por qué estar en contacto, sino que este hecho es más bien algo indeseable, pues podrían ocurrir conflictos entre ellos. Se comprobará la política por defecto y se aplicarán correcciones para la separación de espacios, en caso de que aplique.

Tabla 35: Requisito no funcional RNF12

ID	RNF12
Nombre	Limitación de la comunicación de contenedores en diferentes nodos
Descripción	Comprobar si existe la posibilidad de establecer comunicación entre diferentes contenedores desplegados bajo diferentes nodos del clúster. Una interconexión total de todos los contenedores podría ampliar la superficie de ataque, en caso de un potencial ataque de red. Aplicación de limitaciones, en caso de que aplique.

Tabla 36: Requisito no funcional RNF13

ID	RNF13
Nombre	Configuración y uso del componente *Secret*
Descripción	Al definir los componentes principales de Kubernetes, fue mencionado el componente *Secret*, pensado para almacenar información considerada privada de manera cifrada, siendo así uno de los componentes pensados para aumentar la seguridad de un clúster Kubernetes. Configurar el uso de dicho componente.

Tabla 37: Requisito no funcional RNF14

ID	RNF14
Nombre	Establecimiento de conexiones de red seguras
Descripción	La configuración por defecto de Kubernetes no incorpora una interconexión de red segura entre los diferentes componentes de un clúster, siendo responsabilidad del administrador de sistemas la tarea de securizar todas estas conexiones. Habilitar comunicaciones con TLS supondría la capacidad de prevenir el esnifado de tráfico indebido o la posibilidad de verificar la identidad de cada componente [15]. Configuración de conexiones seguras mediante el uso de TLS.

Tabla 38: Requisito no funcional RNF15

ID	RNF15
Nombre	Monitorización del clúster
Descripción	Aunque no se corresponda estrictamente con un securización, el hecho de monitorizar constantemente los componentes que conforman un clúster, puede ser de gran ayuda para la detección de ataques o comportamientos extraños provocados por software malintencionado. Instalación de un software de monitorización.

5. Instalación y configuración del laboratorio de pruebas

5.1. Arquitectura y especificaciones

Atendiendo a las limitaciones especificadas en la sección 4.1, el laboratorio de pruebas será instalado en una única máquina física, el ordenador portátil personal del autor de este proyecto. Todos los nodos que compondrán el laboratorio de pruebas se instalarán dentro de máquinas virtuales, cada uno en una máquina independiente, incluyendo el nodo *master* y los diferentes nodos *worker*. Para poder crear las diferentes máquinas virtuales implicadas, se empleará la tecnología de virtualización a nivel hardware, VirtualBox[6]. Dentro de cada una de las máquinas virtuales, será instalada la tecnología de virtualización a nivel de sistema operativo, Docker; así como el orquestador de contenedores, Kubernetes. Para establecer la conexión en red, de tal forma que las diferentes máquinas y los diferentes contenedores ubicados en tales máquinas puedan comunicarse entre sí (especialmente importante la comunicación entre el nodo *master* y el resto de nodos *worker*), se establecerá una conexión de red en puente (*bridged networking*). Este tipo de red permitirá conectar cada máquina virtual a la red mediante el adaptador de red usado en la máquina anfitriona, consiguiendo en cada una de las máquinas una dirección independiente que la identificará, permitiendo que las diferentes máquinas puedan distinguirse y comunicarse. Además, puesto que las máquinas virtuales se comunicarán directamente con la tarjeta de red instalada en la máquina anfitriona e intercambiará paquetes de red directamente con ella, se evitará la pila de red del sistema operativo de la máquina anfitriona [41]. En la figura 5 se puede apreciar la arquitectura que tendrá el laboratorio.

Las especificaciones de las máquinas que conformarán el laboratorio son las siguientes:

- **Máquina anfitriona:**
  - **Sistema operativo:** GNU/Linux Ubuntu 18.04.5 LTS.
  - **CPU:** Intel Core i7-9750H CPU @ 2.60GHz × 12
  - **RAM:** 16GB
- **Máquina virtual "Master":**
  - **Sistema operativo:** GNU/Linux Ubuntu 20.04.2 LTS.
  - **CPU:** Intel Core i7-9750H CPU @ 2.60GHz × 2
  - **RAM:** 2GB
- **Máquinas virtuales "Worker" × 3:**
  - **Sistema operativo:** GNU/Linux Ubuntu 20.04.2 LTS.
  - **CPU:** Intel Core i7-9750H CPU @ 2.60GHz × 1
  - **RAM:** 2GB

[6] `https://www.virtualbox.org/`

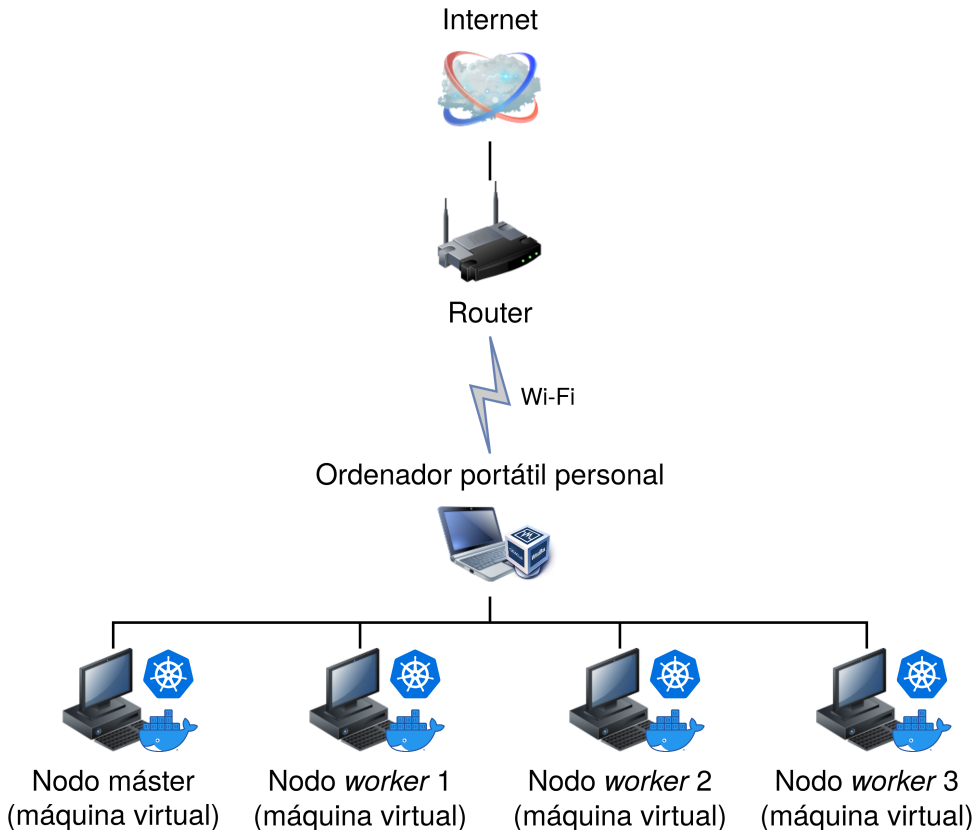


Figura 5: Diagrama de la arquitectura del laboratorio

5.2. Instalación y configuración básica del SO en las máquinas virtuales

Para realizar los procesos de instalación y configuración iniciales se empleó la herramienta gráfica de instalación proporcionada por el propio sistema operativo Ubuntu 20.04.2 LTS, en el que se siguió un procedimiento idéntico para todas las máquinas virtuales, como se puede observar en la figura 6.

Para poder obtener una configuración de red estable y que la dirección IP no varíe en cada ejecución debido al uso del protocolo DHCP, se modificó la configuración de red por defecto y se estableció una configuración manual según lo indicado en la figura 7, adaptando cada dirección a la máquina involucrada. De esta forma, la configuración de red queda resumida en la tabla 39.

Finalmente, puesto que las máquinas del laboratorio se han configurado con unas direcciones IP estáticas, se modificaron los ficheros `/etc/hosts` de cada máquina, de tal forma que se asignó un alias de red a cada una de las máquinas implicadas, de tal forma que resulte más sencillo para un humano realizar cualquier comunicación de red entre ellas. Un ejemplo de esta configuración puede observarse en la figura 8.

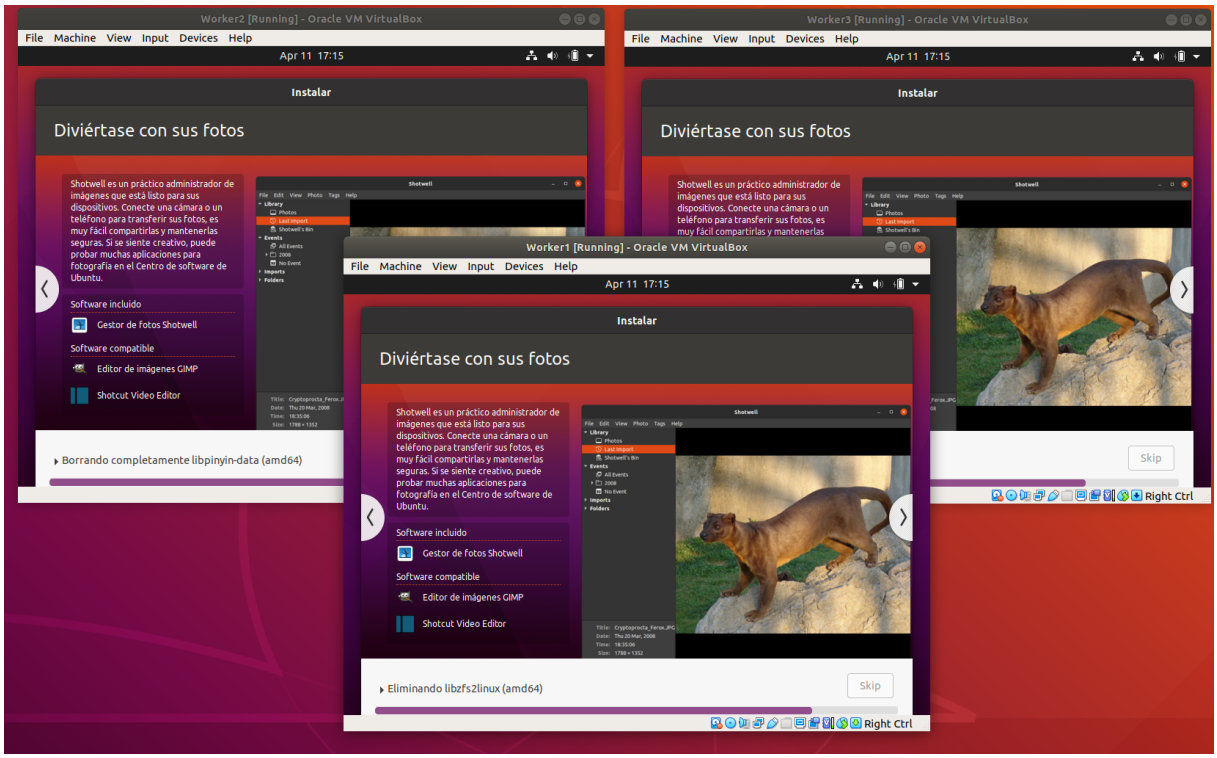


Figura 6: Instalación del SO en las diferentes máquinas virtuales

Tabla 39: Configuraciones de red del laboratorio

Puerta de enlace	192.168.0.1
Máscara de red	255.255.255.0
DNS	8.8.8.8 y 8.8.4.4
Dirección IP máquina “Master”	192.168.0.100
Dirección IP máquina “Worker1”	192.168.0.101
Dirección IP máquina “Worker2”	192.168.0.102
Dirección IP máquina “Worker3”	192.168.0.103

Figura 7: Ejemplo de configuración de red en la máquina "Worker1"

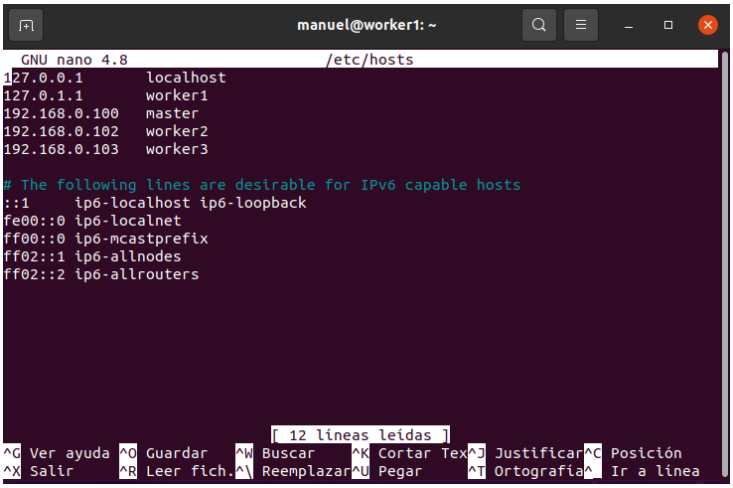


Figura 8: Ejemplo de configuración del fichero `/etc/hosts` en la máquina "Worker1"

5.3. Instalación de Kubernetes

Una vez instalado el sistema operativo base sobre todas las máquinas virtuales que conformarán el laboratorio de pruebas, realizaremos la instalación de la tecnología de contenerización, Docker; así como del orquestador, Kubernetes.

5.3.1. Instalación y configuración de nodos *worker*

Se muestra a continuación un ejemplo de instalación y configuración de Kubernetes en un nodo *worker*, cuyo proceso será replicado en todos los nodos *worker* usados en el laboratorio:

1. Instalación de Docker[7], desde los propios repositorios de Ubuntu:

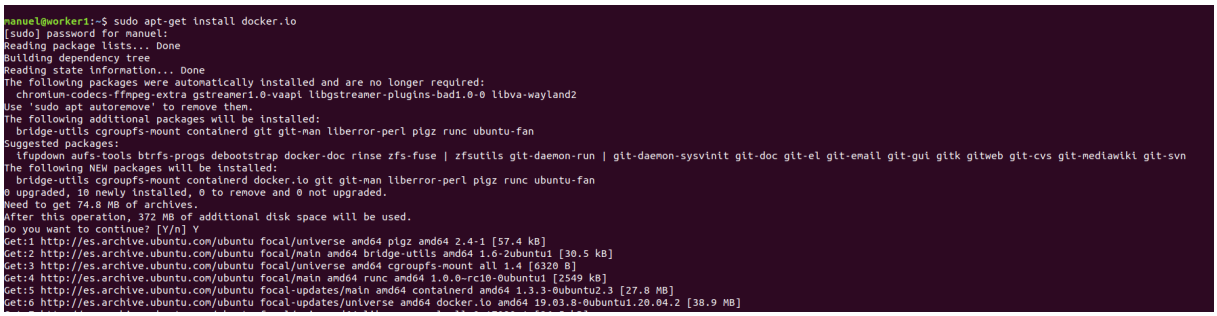


Figura 9: Instalación de Docker en un nodo *worker*

2. Activación del servicio de Docker y comprobación de que está corriendo correctamente:

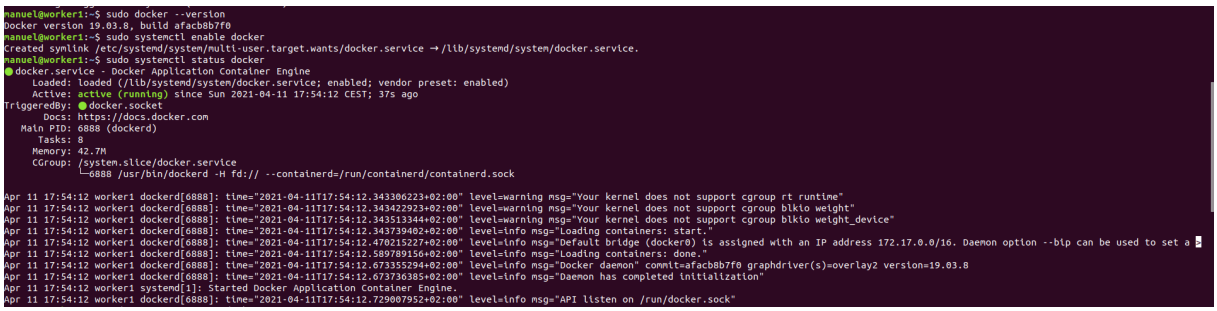


Figura 10: Activación del servicio de Docker en un nodo *worker*

3. Añadimos la clave de Kubernetes, así como el repositorio asociado:

[7]Nota: a lo largo de todo el proceso de realización del proyecto, se ha mantenido la misma versión de Docker, Docker 19.03.8, en todos los nodos que conforman el clúster, con el fin de poder obtener una misma base para todas las pruebas, así como para poder asegurar la futura reprodubilidad de las mismas, en caso de querer replicar este laboratorio.

Figura 11: Añadimos clave y repositorio de Kubernetes en un nodo *worker*

4. Instalación de los paquetes necesarios para el funcionamiento de Kubernetes[8] en un nodo *worker*, haciendo referencia a algunos de los componentes indicados en la sección 3.4.1.

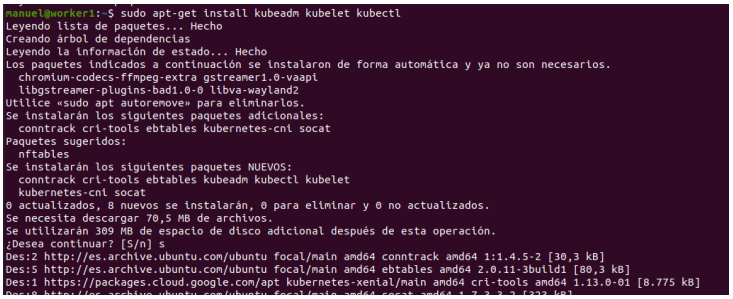


Figura 12: Instalación de los paquetes necesarios en un nodo *worker*

5. Añadimos el nodo *worker* a la lista de nodos administrados por el nodo *master*.[9]

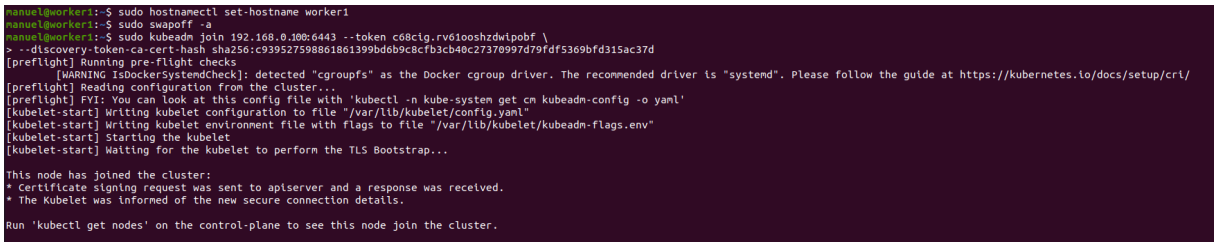


Figura 13: Incorporación del nodo *worker* en los nodos administrados por el nodo *master*

5.3.2. Instalación y configuración de nodo *master*

Los primeros pasos de instalación y configuración del nodo *master* son idénticos a los pasos dados en los nodos *worker*. Concretamente, los cuatro primeros pasos de instalación

[8]Nota: a lo largo de todo el proceso de realización del proyecto, se ha mantenido la misma versión de Kubernetes, Kubernetes 1.21.0, con el fin de poder obtener una misma base para todas las pruebas, así como para poder asegurar la futura reprodubilidad de las mismas, en caso de querer replicar este laboratorio.

[9]Para esto, debemos tener Kubernetes instalado y configurado en el nodo *master* previamente. Copiaremos y pegaremos el comando arrojado por el nodo *master* tras acabar la instalación de Kubernetes.

y configuración serán exactamente idénticos. A partir del quinto punto, la instalación y configuración en un nodo *master* seguirá los siguientes pasos:

5. Inicialización del nodo *master*. Una vez terminado el proceso, nos devolverá el comando a ejecutar en cada uno de los nodos *worker* para que sean administrados por él.

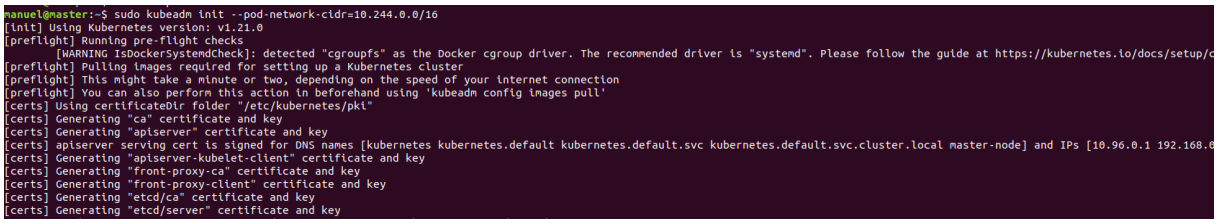


Figura 14: Inicialización del nodo *master*

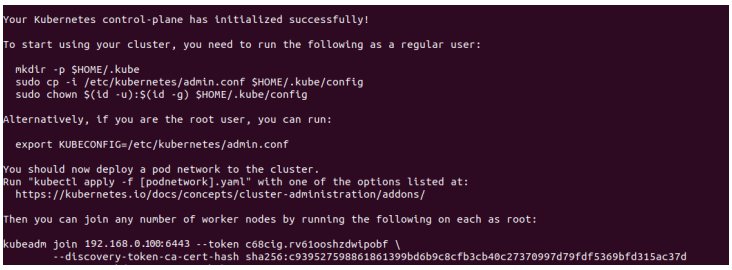


Figura 15: Resultado de la inicialización

6. Ejecución de los comandos indicados tras la inicialización para poder comenzar a usar el clúster, creación de una red *pod* y comprobación del estado de la red.

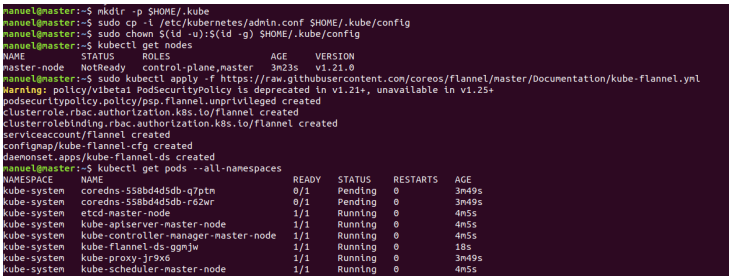


Figura 16: Ejecución de los comandos post-inicialización

7. Una vez todos los nodos *worker* se han añadido, comprobamos que dichos nodos son debidamente detectados desde el nodo *master*.

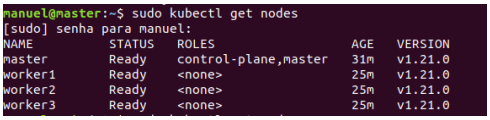


Figura 17: Comprobación de la detección de los nodos *worker*

Realizados los pasos anteriores, tendremos Kubernetes debidamente instalado y configurado en los diferentes nodos que conformarán el laboratorio de pruebas. Una recopilación de los comandos utilizados en este proceso puede ser consultada en el anexo A.1.

6. Análisis de vulnerabilidades

Disponiendo del laboratorio de pruebas descrito a lo largo de la sección 5, en esta sección se analizarán algunas de las vulnerabilidades a las que un entorno de trabajo "estándar" con Kubernetes podría ser vulnerable. Para realizar estos análisis, se seguirá como guía la lista de requisitos no funcionales establecidos en la sección 4.2, que a su vez toman su base en el estudio previo realizado por el autor del proyecto. Por tanto, el objetivo de esta sección será presentar una serie de amenazas que podrían materializarse en vulnerabilidades que tengan un impacto sobre la seguridad del entorno basado en Kubernetes, describirlas desde un punto de vista teórico y, finalmente, comprobar su materialización en el laboratorio de pruebas.

6.1. Análisis estático sobre imágenes de contenedores

Tomando como base el requisito no funcional RNF1, establecido en la tabla 24, este primer estudio comprobará la existencia (o inexistencia) de vulnerabilidades en imágenes de contenedores descargadas desde fuentes bien conocidas y consideradas "de confianza". Cuando trabajamos con un sistema basado en contenedores, debemos tener presente que, junto con las ventajas que estos ofrecen, como el despliegue de entornos más ágiles, también estamos introduciendo nuevos elementos a nuestro sistema, que podrían llegar a convertirse en nuevos posibles vectores de ataque. La implicación de nuevos sistemas operativos y librerías en el conjunto del sistema pueden llevar a la aparición de nuevas vulnerabilidades, que pueden ser más difícilmente controlables que las asociadas al sistema de la propia máquina anfitriona.

6.1.1. Docker Scan

Para realizar este estudio, se hará uso de la funcionalidad "Docker Scan", disponible como parte del software que incorpora la tecnología de contedorización Docker. Dicha funcionalidad permite el escaneo de vulnerabilidades en imágenes Docker albergadas en el Docker Hub, permitiéndoles a los desarrollares tomar las acciones que consideren pertinentes para solucionar los problemas identificados tras el escaneo. "Docker Scan" realiza el análisis tomando como base el motor Snyk[10], por lo que no se trata de una funcionalidad totalmente gestionada por Docker, sino que es preciso poseer una cuenta en la plataforma de Snyk y configurarla en la máquina local.

Los resultados los análisis contendrán información como: la lista de vulnerabilidades y exposiciones comunes (CVE), las fuentes, los paquetes, las bibliotecas del sistema operativo, las versiones en las que se introdujeron y una versión fija recomendada (si está disponible) para remediar los CVE descubiertos. [42]

Prerrequisitos :

1. Instalación del *plugin* "Docker Scan", puesto que no es un componente de Docker incluido en la instalación base.

[10] `https://snyk.io/`

a) Incorporación de los repositorios oficiales de Docker en el listado de repositorios aceptados por la máquina en la que realicemos la instalación:

Listing 1: Incorporación de los repositorios de Docker

```
echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-
    ↪ keyring.gpg] https://download.docker.com/linux/ubuntu $(
    ↪ lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/
    ↪ docker.list > /dev/null
```

b) Actualización de las fuentes e instalación del *plugin*:

Listing 2: Actualización de las fuentes e instalación del *plugin*

```
sudo apt-get update
sudo apt-get install docker-scan-plugin
```

c) Incorporación del *token* personal para realizar el inicio de sesión en el servicio de Snyk. Puesto que el análisis estático de vulnerabilidades no es un servicio prestado directamente por Kubernetes o Docker, sino que se realiza a través de una herramienta externa que es posible integrar, es preciso poseer una cuenta en dicho servicio e iniciar sesión mediante nuestro *token* de acceso personal.

Listing 3: Incorporación del *token* personal de Snyk

```
Docker Scan --login --token <<TOKEN PERSONAL>>
```

Análisis de una imagen de estudio :

Para comprobar la existencia de vulnerabilidades en una imagen mediante un análisis estático, primeramente seleccionaremos una imagen popular, ampliamente usada y proveniente de una fuente oficial de confianza. La imagen seleccionada se trata del servidor de aplicaciones web Nginx. Puesto que el *tag* que hace referencia a la última versión disponible (*latest*) puede variar según la fecha, se seleccionará uno de los últimos *tags* disponibles, con el objetivo de garantizar la posible futura reproducibilidad del experimento. La versión seleccionada se trata de la 1.20.0[11]. Como se puede observar en la figura 18, dicha imagen está categorizada como una de las imágenes oficiales de Docker, además de tratarse del *build* oficial de los desarrolladores de Nginx, por lo que, sin mayor conocimiento, podríamos llegar a asumir que se trata de una imagen totalmente segura y libre de vulnerabilidades.

Una vez seleccionado el objetivo de análisis, descargamos la imagen deseada en el nodo *master* mediante el comando `pull` y procedemos a realizar un análisis estático de vulnerabilidades con la herramienta "Docker Scan", tal y como se muestra en la figura 19. Como se puede observar, el análisis estático arroja numerosas vulnerabilidades conocidas. Aunque el comienzo del análisis empieza con algunas vulnerabilidades con baja severidad, también son arrojados resultados más preocupantes donde se localizan numerosas vulnerabilidades con una severidad alta, como se puede apreciar en la figura 20. Tras su ejecución, podemos observar como el resultado del análisis fue la localización de 91 vulnerabilidades.

[11] `https://hub.docker.com/layers/nginx/library/nginx/1.20.0/images/sha256-c49b6dd667686f63b4dd3ec1736714543e50e1a2680ca3f1221fa1d3cd3b501d`

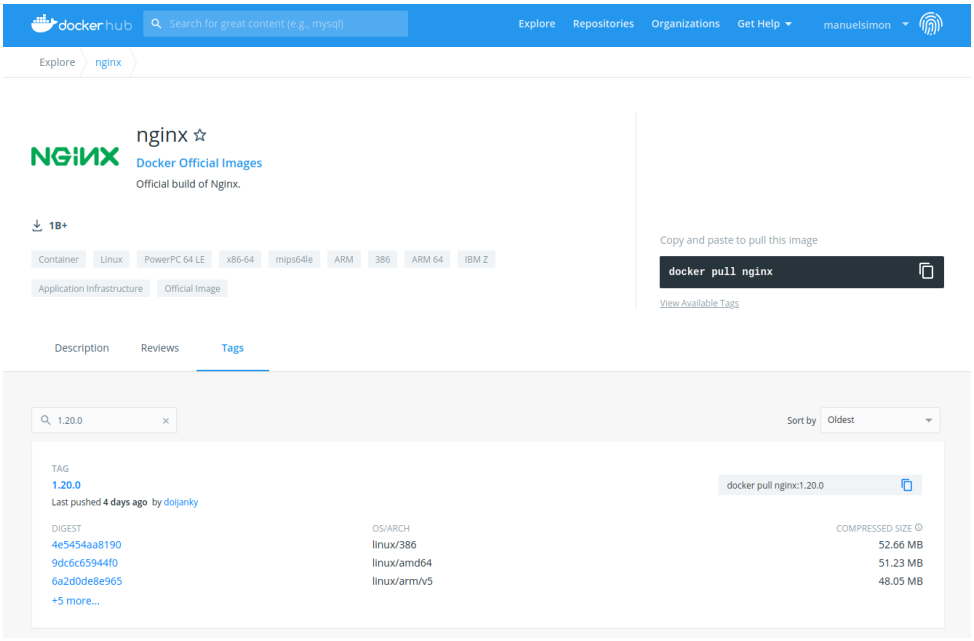


Figura 18: Imagen de Nginx en Docker Hub. *Tag* 1.20.0.

```
manuel@master:~$ sudo docker pull nginx:1.20.0
1.20.0: Pulling from library/nginx
f7ec5a41d630: Pull complete
d064bcebeb06: Pull complete
975c0a9d7b94: Pull complete
1384c783a2a3: Pull complete
a0f1673d45e1: Pull complete
fc05b7dad2e3: Pull complete
Digest: sha256:c1e6c074ac4f7a543d5a55379ba7bb0d331961b9deca1bc5c5902ef0a354a63d
Status: Downloaded newer image for nginx:1.20.0
docker.io/library/nginx:1.20.0
manuel@master:~$ sudo docker scan nginx:1.20.0
manuel@master:~$ sudo docker scan nginx:1.20.0

Testing nginx:1.20.0...

✗ Low severity vulnerability found in tiff/libtiff5
  Description: Out-of-Bounds
  Info: https://snyk.io/vuln/SNYK-DEBIAN10-TIFF-1079067
  Introduced through: nginx-module-image-filter@1.20.0-1~buster
  From: nginx-module-image-filter@1.20.0-1~buster > libgd2/libgd3@2.2.5-5.2 > tiff/libtiff5@4.1.0+git191117-2~deb10u2

✗ Low severity vulnerability found in tiff/libtiff5
  Description: Out-of-Bounds
  Info: https://snyk.io/vuln/SNYK-DEBIAN10-TIFF-1079073
  Introduced through: nginx-module-image-filter@1.20.0-1~buster
  From: nginx-module-image-filter@1.20.0-1~buster > libgd2/libgd3@2.2.5-5.2 > tiff/libtiff5@4.1.0+git191117-2~deb10u2

✗ Low severity vulnerability found in tiff/libtiff5
  Description: Use After Free
  Info: https://snyk.io/vuln/SNYK-DEBIAN10-TIFF-405264
  Introduced through: nginx-module-image-filter@1.20.0-1~buster
  From: nginx-module-image-filter@1.20.0-1~buster > libgd2/libgd3@2.2.5-5.2 > tiff/libtiff5@4.1.0+git191117-2~deb10u2

✗ Low severity vulnerability found in tiff/libtiff5
  Description: Divide By Zero
  Info: https://snyk.io/vuln/SNYK-DEBIAN10-TIFF-405344
  Introduced through: nginx-module-image-filter@1.20.0-1~buster
  From: nginx-module-image-filter@1.20.0-1~buster > libgd2/libgd3@2.2.5-5.2 > tiff/libtiff5@4.1.0+git191117-2~deb10u2

✗ Low severity vulnerability found in tiff/libtiff5
  Description: NULL Pointer Dereference
  Info: https://snyk.io/vuln/SNYK-DEBIAN10-TIFF-405387
  Introduced through: nginx-module-image-filter@1.20.0-1~buster
  From: nginx-module-image-filter@1.20.0-1~buster > libgd2/libgd3@2.2.5-5.2 > tiff/libtiff5@4.1.0+git191117-2~deb10u2
```

Figura 19: Descarga de la imagen objetivo y ejecución de análisis estático

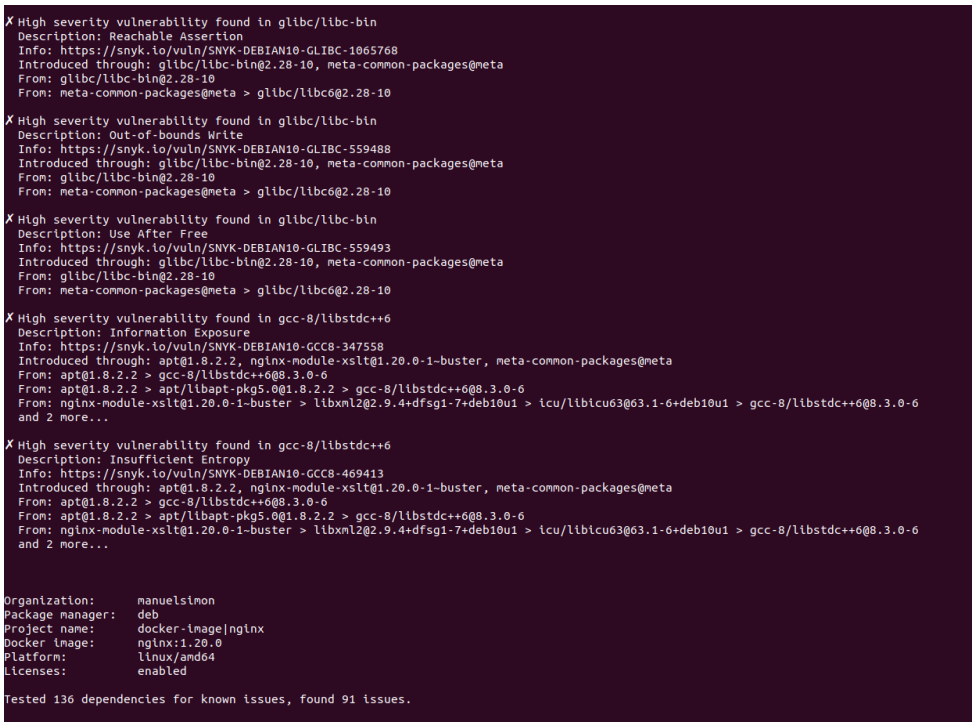


Figura 20: Fin del resultado del análisis estático de una imagen

Otra de las funcionalidades ofrecidas por esta herramienta de análisis estático de vulnerabilidades, es la posibilidad de, no solo localizar dichas vulnerabilidades, sino también obtener un listado del software que las provoca, así como sus dependencias. Un ejemplo de dicho árbol de software afectado y dependencias puede observarse en la figura 21.

Despliegue de imagen con vulnerabilidades en el laboratorio de pruebas :

Una vez localizada una imagen con vulnerabilidades, procederemos a desplegar dicha imagen en el clúster de pruebas, para observar si es posible realizar el despliegue sin ningún tipo de problemas y, al mismo tiempo, confirmar o desmentir la existencia de algún aviso sobre dichas vulnerabilidades a la hora de emplear Kubernetes. Para realizar el despliegue se hará uso del fichero de configuración YAML disponible en el anexo A.2. En dicho fichero, se indica la imagen a desplegar y su versión (1.20.0), el número de réplicas (2) y el puerto que usará la aplicación dentro del contenedor (80). Realizamos el despliegue indicando el fichero YAML con las configuraciones mediante el comando `kubectl apply -f nginx.yaml`. Hecho esto, nos aseguramos de que el despliegue se realizó correctamente, comprobando para ello la lista de *pods*. Identificado alguno de los *pods* en los que está corriendo la aplicación, obtenemos información sobre el mismo con el comando `kubectl describe pod «POD ID»`. A partir de la información obtenida, podemos comprobar que la aplicación contenerizada está siendo ejecutada en el nodo *worker* 2, desde donde podemos acceder a la misma mediante un navegador web, indicando la IP asociada al servidor Nginx. Este proceso puede observarse en la figura 22.

Como conclusión, el experimento realizado en esta sección ha permitido confirmar la

```
manuel@master:~$ sudo docker scan --dependency-tree nginx:1.20.0
docker-image|nginx @ 1.20.0
  ├ apt @ 1.8.2.2
  │  ├ adduser @ 3.118
  │  ├ apt/libapt-pkg5.0 @ 1.8.2.2
  │  │  ├ gcc-8/libstdc++6 @ 8.3.0-6
  │  │  ├ libzstd/libzstd1 @ 1.3.8+dfsg-3+deb10u2
  │  │  ├ lz4/liblz4-1 @ 1.8.3-1
  │  │  ├ systemd/libsystemd0 @ 241-7~deb10u7
  │  │  │  ├ libgcrypt20 @ 1.8.4-5
  │  │  │  └ lz4/liblz4-1 @ 1.8.3-1
  │  │  └ systemd/libudev1 @ 241-7~deb10u7
  │  ├ debian-archive-keyring @ 2019.1+deb10u1
  │  ├ gcc-8/libstdc++6 @ 8.3.0-6
  │  ├ gnupg2/gpgv @ 2.2.12-1+deb10u1
  │  │  ├ libgcrypt20 @ 1.8.4-5
  │  │  └ libgpg-error/libgpg-error0 @ 1.35-1
  │  ├ gnutls28/libgnutls30 @ 3.6.7-4+deb10u6
  │  └ libseccomp/libseccomp2 @ 2.3.3-4
  ├ apt/libapt-pkg5.0 @ 1.8.2.2
  ├ base-files @ 10.3+deb10u9
  │  └ mawk/mawk @ 1.3.3-17+b3
  ├ base-passwd @ 3.5.46
  │  └ cdebconf/libdebconfclient0 @ 0.249
  ├ bash @ 5.0-4
  │  ├ base-files @ 10.3+deb10u9
  │  ├ debianutils @ 4.8.6.1
  │  └ ncurses/libtinfo6 @ 6.1+20181013-2+deb10u2
  ├ ca-certificates @ 20200601~deb10u2
  │  └ openssl @ 1.1.1d-0+deb10u6
  │     └ openssl/libssl1.1 @ 1.1.1d-0+deb10u6
  ├ cdebconf/libdebconfclient0 @ 0.249
  ├ curl @ 7.64.0-4+deb10u2
  │  └ curl/libcurl4 @ 7.64.0-4+deb10u2
  │     ├ e2fsprogs/libcom-err2 @ 1.44.5-1+deb10u3
  │     ├ krb5/libgssapi-krb5-2 @ 1.17-3+deb10u1
  │     │  ├ e2fsprogs/libcom-err2 @ 1.44.5-1+deb10u3
  │     │  ├ keyutils/libkeyutils1 @ 1.6-6
  │     │  ├ krb5/libk5crypto3 @ 1.17-3+deb10u1
  │     │  │  ├ keyutils/libkeyutils1 @ 1.6-6
  │     │  │  └ krb5/libkrb5support0 @ 1.17-3+deb10u1
  │     │  │     └ keyutils/libkeyutils1 @ 1.6-6
  │     │  ├ krb5/libkrb5-3 @ 1.17-3+deb10u1
  │     │  │  ├ e2fsprogs/libcom-err2 @ 1.44.5-1+deb10u3
  │     │  │  ├ keyutils/libkeyutils1 @ 1.6-6
  │     │  │  ├ krb5/libk5crypto3 @ 1.17-3+deb10u1
  │     │  │  ├ krb5/libkrb5support0 @ 1.17-3+deb10u1
  │     │  │  └ openssl/libssl1.1 @ 1.1.1d-0+deb10u6
  │     │  └ krb5/libkrb5support0 @ 1.17-3+deb10u1
  │     ├ krb5/libk5crypto3 @ 1.17-3+deb10u1
  │     ├ krb5/libkrb5-3 @ 1.17-3+deb10u1
  │     ├ libidn2/libidn2-0 @ 2.0.5-1+deb10u1
  │     │  └ libunistring/libunistring2 @ 0.9.10-1
  │     ├ libpsl/libpsl5 @ 0.20.2-2
  │     │  ├ libidn2/libidn2-0 @ 2.0.5-1+deb10u1
  │     │  └ libunistring/libunistring2 @ 0.9.10-1
  │     ├ libssh2/libssh2-1 @ 1.8.0-2.1
  │     │  └ libgcrypt20 @ 1.8.4-5
  │     │     └ libgpg-error/libgpg-error0 @ 1.35-1
  │     ├ nghttp2/libnghttp2-14 @ 1.36.0-2+deb10u1
  │     ├ openldap/libldap-2.4-2 @ 2.4.47+dfsg-3+deb10u6
  │     │  ├ cyrus-sasl2/libsasl2-2 @ 2.1.27+dfsg-1+deb10u1
  │     │  │  └ cyrus-sasl2/libsasl2-modules-db @ 2.1.27+dfsg-1+deb10u1
  │     │  │     └ db5.3/libdb5.3 @ 5.3.28+dfsg1-0.5
  │     │  ├ gnutls28/libgnutls30 @ 3.6.7-4+deb10u6
  │     │  │  ├ gmp/libgmp10 @ 2:6.1.2+dfsg-4
  │     │  │  ├ libidn2/libidn2-0 @ 2.0.5-1+deb10u1
  │     │  │  ├ libtasn1-6 @ 4.13-3
  │     │  │  ├ libunistring/libunistring2 @ 0.9.10-1
  │     │  │  ├ nettle/libhogweed4 @ 3.4.1-1
  │     │  │  │  ├ gmp/libgmp10 @ 2:6.1.2+dfsg-4
  │     │  │  │  └ nettle/libnettle6 @ 3.4.1-1
  │     │  │  ├ nettle/libnettle6 @ 3.4.1-1
  │     │  │  └ p11-kit/libp11-kit0 @ 0.23.15-2+deb10u1
  │     │  │     └ libffi/libffi6 @ 3.2.1-9
  │     │  └ openldap/libldap-common @ 2.4.47+dfsg-3+deb10u6
  │     ├ openssl/libssl1.1 @ 1.1.1d-0+deb10u6
  │     └ rtmpdump/librtmp1 @ 2.4+20151223.gitfa8646d.1-2
  │        ├ gmp/libgmp10 @ 2:6.1.2+dfsg-4
  │        ├ gnutls28/libgnutls30 @ 3.6.7-4+deb10u6
  │        ├ nettle/libhogweed4 @ 3.4.1-1
  │        └ nettle/libnettle6 @ 3.4.1-1
  ├ dash @ 0.5.10.2-5
  │  └ debianutils @ 4.8.6.1
  ├ debian-archive-keyring @ 2019.1+deb10u1
  ├ debianutils @ 4.8.6.1
  ├ diffutils @ 1:3.7-3
  ├ e2fsprogs @ 1.44.5-1+deb10u3
  │  ├ e2fsprogs/libcom-err2 @ 1.44.5-1+deb10u3
  │  ├ e2fsprogs/libext2fs2 @ 1.44.5-1+deb10u3
  │  ├ e2fsprogs/libss2 @ 1.44.5-1+deb10u3
  │  │  └ e2fsprogs/libcom-err2 @ 1.44.5-1+deb10u3
  │  ├ util-linux/libblkid1 @ 2.33.1-0.1
  │  │  └ util-linux/libuuid1 @ 2.33.1-0.1
  │  └ util-linux/libuuid1 @ 2.33.1-0.1
  ├ e2fsprogs/libext2fs2 @ 1.44.5-1+deb10u3
  ├ e2fsprogs/libss2 @ 1.44.5-1+deb10u3
  ├ findutils @ 4.6.0+git+20190209-2
  ├ gettext/gettext-base @ 0.19.8.1-9
  ├ glibc/libc-bin @ 2.28-10
  ├ gnupg2/gpgv @ 2.2.12-1+deb10u1
```

Figura 21: Ejemplo de listado de dependencias de software con vulnerabilidades en imagen Nginx

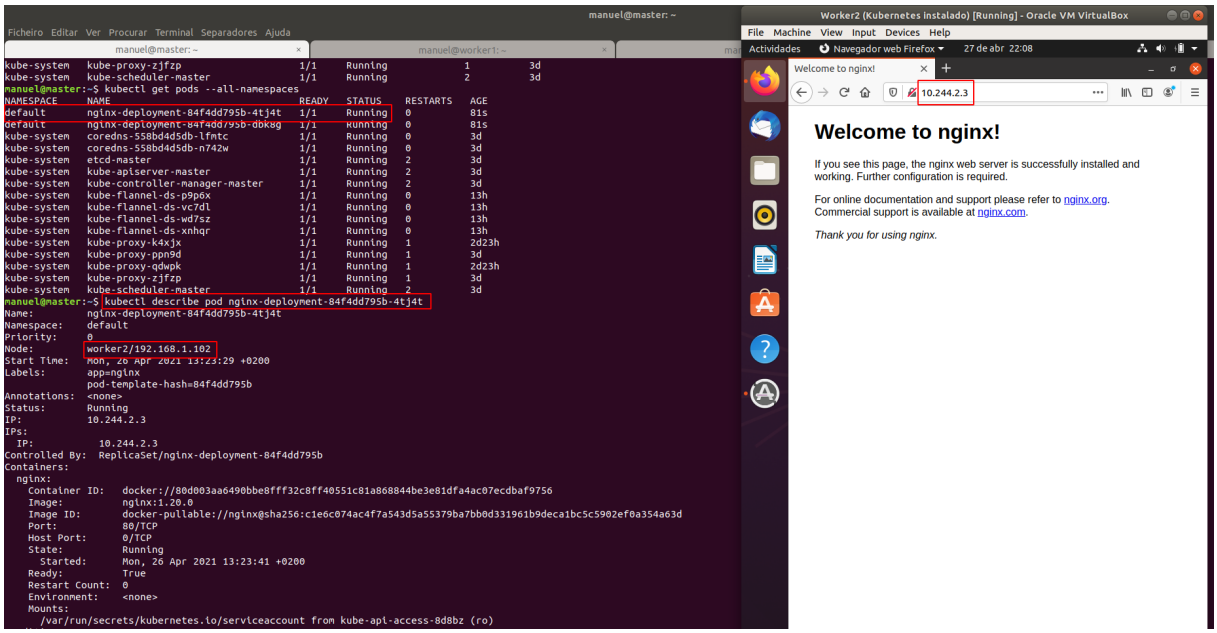


Figura 22: Imagen Nginx 1.20.0 corriendo sobre contenedor en el nodo *worker* 2

existencia de vulnerabilidades conocidas en imágenes de fuentes consideradas oficiales y "de confianza". No obstante, la existencia de dichas vulnerabilidades no implican la existencia de cualquier tipo de advertencia en el propio repositorio o la hora de realizar un despliegue, sino que es preciso tomar medidas propias para localizar dichas vulnerabilidades.

6.2. Análisis estático sobre ficheros de configuración YAML

En la sección 6.1, vimos como es posible realizar un análisis estático sobre la imagen de un contenedor para localizar vulnerabilidades conocidas en el software contenido en dicha imagen. No obstante, como ya fue indicado en el requisito no funcional RNF2, establecido en la tabla 25, este tipo de análisis no tienen por qué limitarse al estudio de software, sino que también pueden ser de gran ayuda para localizar malas configuraciones o malas prácticas en los ficheros YAML, que servirán para indicar el despliegue a realizar en un clúster Kubernetes. De esta forma, si conseguimos realizar un análisis estático sobre un fichero YAML antes de realizar el despliegue, tendremos la oportunidad de detectar estos errores y solventarlos, de considerarlo necesario.

Para realizar este primer estudio práctico de análisis, se hará uso de la herramienta Kubesec[12], ideada para realizar análisis de riesgos de seguridad en diferentes recursos de Kubernetes [15]. El primer paso será instalar Kubesec en nuestro sistema, por ejemplo, descargando la imagen correspondiente desde el Docker Hub en el nodo maestro del laboratorio de pruebas, con el comando "`sudo docker pull kubesec/kubesec:v2`". Una vez poseamos la imagen correspondiente, podemos desplegarla en un contenedor y ejecutar el análisis mediante el uso de una *shell* interactiva. Para realizar el estudio se utilizará el fichero YAML localizado en el anexo A.2, un fichero de configuración muy simple em-

[12] `https://kubesec.io/`

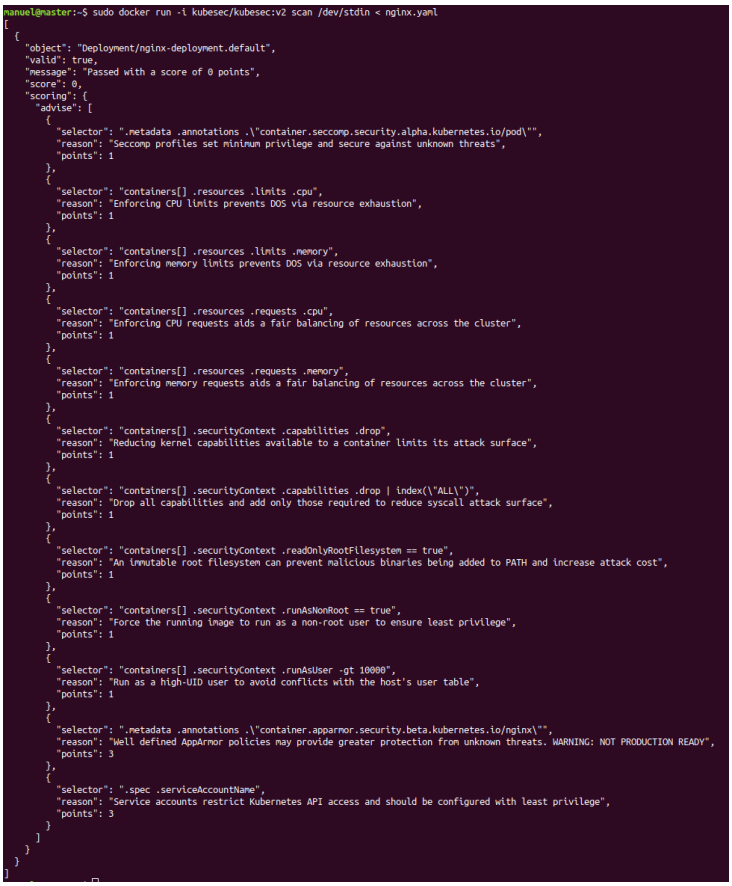


Figura 23: Análisis estático de fichero YAML para el despliegue de Nginx

pleado con anterioridad a lo largo de la sección 6.1. El resultado de este análisis puede observarse en la figura 23.

Como podemos observar, el resultado arrojado por el análisis estático nos indica que existen diferentes puntos débiles en el fichero de configuración YAML, como la inexistencia de limitación de recursos o la posibilidad de ejecutar los contenedores con privilegios de superusuario (`root`), por ejemplo. Por tanto, este tipo de análisis resultan un recurso muy interesante a la hora de compaginarlos con otro tipo de securizaciones en el clúster, de tal forma que se pueda comprobar de una manera rápida y sencilla la existencia o inexistencia de vulnerabilidades en la configuración del despliegue, de tal forma que puedan ser corregidas antes de que la propia aplicación sea desplegada.

6.3. Limitación de recursos

La limitación de recursos físicos, como pueden ser la memoria, la CPU o el uso de red, se trata de un aspecto crítico en lo que concierne a la seguridad de un sistema basado en contenedores, ya que un contenedor malicioso o, simplemente, mal configurado, podría hacer un uso excesivo de los recursos de los que se dispone, llegando a ocupar la práctica totalidad de los mismos y perjudicando de esta forma al resto de contenedores alojados en la misma máquina, o incluso al comportamiento de la propia máquina anfitriona, pudiendo llegar a producirse, en el peor de los casos, situaciones o ataques de denegación de servicio

(DoS) [6].

6.3.1. Limitación de memoria RAM

En esta primera subsección del estudio sobre la limitación de recursos en un ambiente de contenedores orquestado por la tecnología Kubernetes, se explorará la posibilidad de que un determinado contenedor llegue a emplear los recursos de memoria de manera excesiva, hasta ocupar la práctica totalidad de la memoria disponible en la máquina anfitriona sobre la que correrá dicho contenedor, realizando así el estudio del requisito no funcional RNF3, reflejado en la tabla 26. Para ello, desde el nodo *master* se solicitará la ejecución de una imagen Debian[13], que se asignará a alguno de los nodos trabajadores; posteriormente estableceremos una consola de comandos interactiva sobre el nuevo contenedor y ejecutaremos un pequeño programa de prueba que tratará de comprobar si es posible, por defecto, consumir la totalidad de la memoria de la máquina anfitriona.

Por tanto, para la ejecución de esta prueba se hará uso de un pequeño programa escrito en C, basado en la realización de reservas de memoria en un bucle infinito, sin liberar la memoria reservada, y cuyo código fuente puede ser consultado en el anexo A.3. Como observación, el código, en una primera instancia, no contaba con la llamada a la función `sleep()`, no obstante, se observó que de esta forma, no era posible completar el experimento, ya que el proceso asociado era matado inmediatamente por la protección "*out-of-memory*". Este comportamiento tenía lugar, ya que, de forma predeterminada, si se produce un error de memoria insuficiente (OOM), el *kernel* se encargará de matar los procesos involucrados, aunque estuvieran en un contenedor. Por tanto, tras esta primera prueba de concepto, podemos confirmar que, por defecto, ya existe un pequeño sistema de control de memoria ante los casos más graves. Atendiendo a la documentación oficial de Docker, dicho comportamiento puede ser modificado, usando para ello la opción `-oom-kill-disable`. Sin embargo, también se indica que dicha opción solo debe ser activada sobre contenedores en los que se haya aplicado alguna política de limitación de memoria, ya que sino la máquina anfitriona podría quedarse sin memoria y el *kernel* podría necesitar matar los procesos del sistema anfitrión para liberar memoria y provocando un mal funcionamiento no solo de las aplicaciones contenerizadas, sino de aquellas corriendo sobre la propia máquina anfitriona [34], confirmando la tesis previamente establecida.

A pesar de este hecho, todavía resulta interesante el estudio de aplicaciones contenerizadas que hagan un uso exhaustivo de la memoria, ya que el consumo excesivo no tiene por qué realizarse de forma súbita. Fue por esto que se añadió la función `sleep()` al código, pues de esta forma se producirán pequeñas pausas entre cada reserva de memoria, que impedirán que el proceso sea matado tan rápidamente y, por tanto, permita la continuación de este estudio.

Así pues, para realizar este estudio de memoria, se tomaron medidas del uso de la memoria RAM antes de la ejecución, reflejadas en la figura 24, y después de la ejecución del programa de consumo exhaustivo, reflejadas en la figura 25.

[13] `https://hub.docker.com/_/debian`

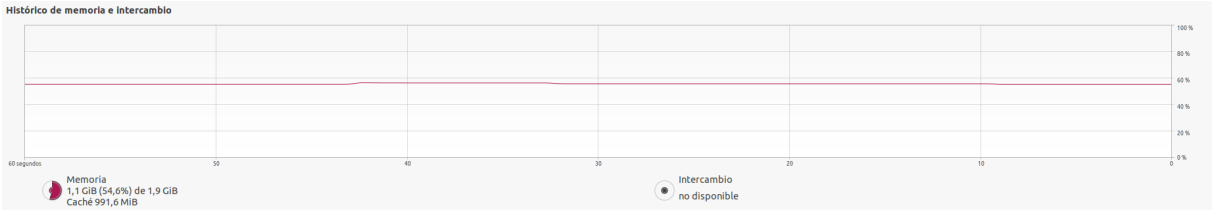


Figura 24: Mediciones del uso de la memoria RAM antes de la prueba

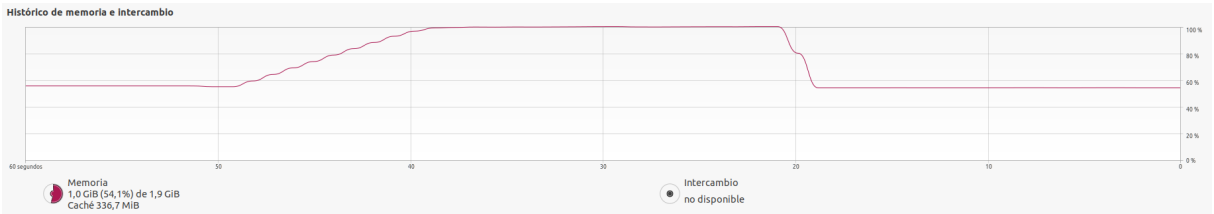


Figura 25: Mediciones del uso de la memoria RAM después de la prueba

Tal y como se observa en las figuras, en un momento inicial el uso de la memoria RAM es constante, pero una vez el programa entra en ejecución, dicha memoria se va consumiendo "paulatinamente", hasta llegar un punto en el que la memoria está totalmente ocupada. Durante el tiempo que la aplicación consiguió consumir la memoria en su totalidad, la máquina anfitriona dejó de dar respuesta, consumándose así la situación de denegación de servicio perseguida. Es decir, durante unos instantes, todos los contenedores albergados en la máquina "Worker1", dejaron de dar servicio, así como todas las aplicaciones no contenerizadas que pudieran estar corriendo sobre él. Pasado un tiempo, podemos ver como la protección OOM entra en acción y mata el proceso asociado. De todas formas, esta prueba es suficiente para mostrar la problemática existente en caso de no limitar el uso de memoria en un entorno basado en contenedores y orquestado por Kubernetes.

6.3.2. Limitación de CPU

Esta segunda parte del estudio de limitación de recursos supone una continuidad inmediata de la sección 6.3.1. En este caso, en lugar de forzar un uso excesivo de memoria, se provocará el uso excesivo de CPU, pero para ello se empleará el mismo enfoque e incluso el mismo programa de pruebas, ya que, al estar basado en un bucle infinito, también hará un alto uso de CPU. Igualmente, el objetivo a perseguir es el mismo: tratar que un contenedor consuma tanta CPU que deje sin servicio el nodo en el que da servicio, dentro de una red de contenedores gestionada por Kubernetes.

En esta ocasión, podemos observar la situación antes de la prueba en la figura 26, mientras que los resultados después de la prueba pueden consultarse en la figura 27.

Podemos observar como el programa lleva a un uso excesivo de los recursos CPU de la máquina, dejándola en un estado sin respuesta. No obstante, ya que las mediciones fueron realizados dentro de una máquina virtual, puede parecer que los picos máximos no llegan a consumir el 100 % de la CPU. Puesto que las mediciones del uso de la CPU dentro de

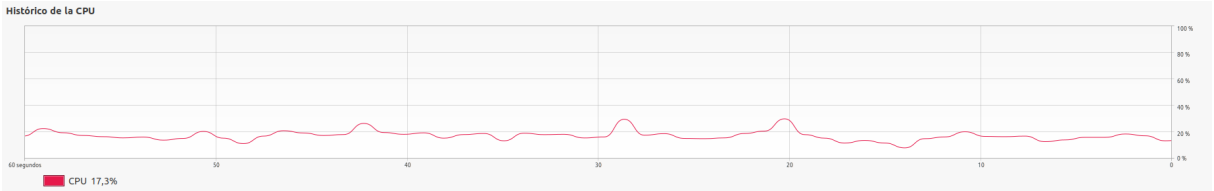


Figura 26: Mediciones del uso de CPU antes de la prueba

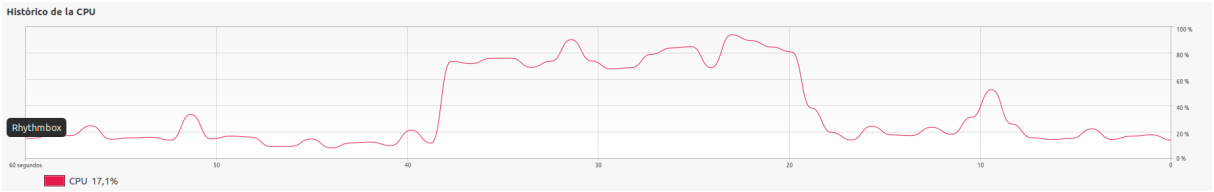


Figura 27: Mediciones del uso de CPU después de la prueba

la máquina virtual pueden no estar reflejando la realidad de un modo del todo correcto, para esta prueba en particular también se comprobaron las mediciones en la máquina anfitriona, sobre la que corren las máquinas virtuales, ya que los recursos de CPU son compartidos entre la máquina anfitriona y las máquinas virtuales. De esta forma, podemos obtener una aproximación más realista de la situación en la figura 28. En este caso, podemos ver como, realmente, una de las CPUs está alcanzando el 100 % de su capacidad durante un periodo prolongado de tiempo, correspondiente a la ejecución del software con un bucle infinito. Dicha CPU se tratará de la CPU compartida con la máquina virtual sobre la que corre el nodo *worker* 1.

Como hemos podido ver tras las pruebas realizadas en las secciones 6.3.1 y 6.3.2, la política por defecto de Kubernetes consiste en permitir una demanda de recursos ilimitada. Por tanto, si un usuario malintencionado quisiera realizar un ataque de denegación de servicio, podría comenzarlo desde cualquier *pod* dentro del clúster, que realizara una alta demanda de recursos. En esta situación, si ninguna protección lo impide, el componente `kube-scheduler` se encargaría de crear un nuevo *pod* con una nueva instancia del contenedor. Este proceso podría repetirse indefinidamente, consumiendo más y más recursos a medida que el `kube-scheduler` se encarga de crear nuevas instancias en el clúster, hasta llegar al punto en el que se consumirían la totalidad de recursos de memoria y CPU, completándose así el ataque de DoS, al no poder proporcionar los recursos necesarios a otros procesos. [3]

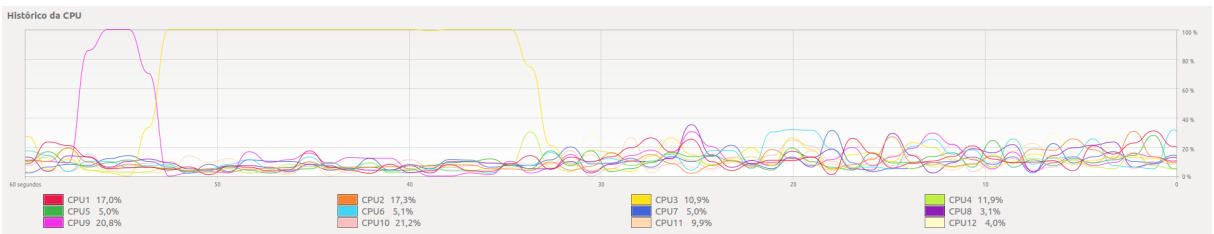


Figura 28: Mediciones del uso de CPU en la máquina anfitriona después de la prueba

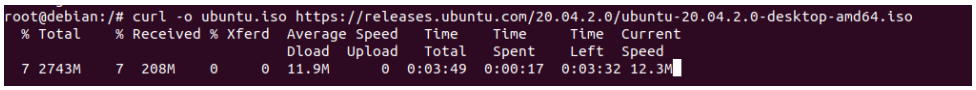


Figura 29: Prueba de uso del total del ancho de banda

6.3.3. Limitación de red

La última prueba que realizaremos sobre la limitación de recursos hace referencia a los recursos de red, correspondiéndose con el estudio del requisito no funcional RNF5, reflejado en la tabla 28. Es decir, si existe algún límite en, por ejemplo, la velocidad de descarga o el tiempo en el que podemos hacer uso del máximo de ancho de banda disponible. En esta ocasión, no será necesario el uso de ningún programa o *script*, sino que bastará con la ejecución del comando `curl`, para bajar algún fichero desde una fuente externa proveniente de Internet que consiga ocupar la totalidad del ancho de banda. El fichero seleccionado para la prueba fue la imagen de Ubuntu 20.04.2.0 Desktop para sistemas de 64 bits[14]. En la figura 29 se puede observar como, a través de la ejecución de este comando, el ancho de banda disponible de forma compartida para este contenedor, el resto de contenedores corriendo sobre la misma máquina, otros contenedores corriendo sobre otras máquinas e inclusive la máquina anfitriona, se ve ocupada por una única instancia. Resaltar que el caso de tener un ancho de banda compartido por tantas máquinas viene dado por el hecho de disponer de un laboratorio de pruebas en un único ordenador, sobre el que corren todos los recursos, por lo que no existe más solución que todos ellos compartan la red. No obstante, el hecho de que diferentes *pods* o contenedores tengan que compartir espacio en una misma máquina, lo cual sería el ambiente más habitual, ya podría suponer un problema de limitación de recursos entre estos contenedores, impidiendo, potencialmente, la correcta ejecución de alguno de ellos.

6.4. Permisos de ejecución dentro de los contenedores

A lo largo de esta sección, comprobaremos el tipo de permisos que un usuario puede obtener dentro de un contenedor, en caso de que no se apliquen limitaciones. Debemos tener en cuenta que los permisos existentes dentro de un contenedor tienen un papel fundamental en su ejecución interna, pero también se deben tener en cuenta como una potencial amenaza para la seguridad del sistema anfitrión.

6.4.1. Implicaciones a nivel de seguridad dentro del contenedor

En caso de que un ataque tuviera éxito y consiguiera tomar el control del contenedor, resulta obvio que no supone el mismo nivel de riesgo que dicho control se ejerza desde un usuario con privilegios limitados que con privilegios de superusuario. En caso de que los permisos sean a nivel de usuario con privilegios limitados, probablemente el ataque se verá muy acotado, afectando a un número reducido de aplicaciones a las que el usuario tenga acceso. No obstante, si un atacante consigue tomar el control de un contenedor bajo el usuario `root`, el alcance del ataque podría llegar a ser mucho mayor. Por ejemplo,

[14]Disponible en: `https://releases.ubuntu.com/20.04.2.0/ubuntu-20.04.2.0-desktop-amd64.iso`

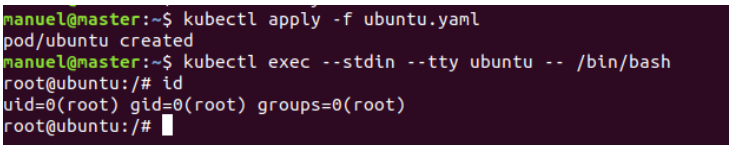


Figura 30: Obtención de permisos de superusuario dentro del contenedor

tendría acceso a todos los recursos del contenedor, pudiendo detener todas las aplicaciones corriendo sobre él e incluso llegar a interferir sobre otros contenedores, por ejemplo, interfiriendo en la red.

Para comprobar la posibilidad de acceder como superusuario dentro de un contenedor, se aplicará el fichero de configuración YAML ubicado en el anexo A.4, a partir del cual se creará un *pod* sobre el que correrá un contenedor con la última versión disponible de Ubuntu. Indicar que en dicho fichero YAML no se especifica ninguna configuración para el establecimiento de permisos de ejecución, por lo que una vez entremos en una consola de comandos interactiva, conseguiremos entrar con los permisos habilitados por defecto. En la figura 30 podemos observar como, por defecto, conseguimos entrar en el contenedor configurado por el orquestador Kubernetes bajo el usuario `root` (UID = 0).

6.4.2. Implicaciones a nivel de seguridad en la máquina anfitriona

En caso de que un atacante llegara a tomar el control de un contenedor, el ataque podría ir más allá y, en el peor de los casos, "romper" el aislamiento ofrecido por la virtualización a nivel de sistema operativo. Aunque las probabilidades de que se materialice tal riesgo son ínfimas, no debemos obviarlo, puesto que las consecuencias podrían ser catastróficas. En tal circunstancia, el atacante llegaría a tener acceso al sistema de la máquina anfitriona, pero, en caso de que el ataque se efectúe como superusuario desde el contenedor, ¿conseguiría el acceso en la máquina anfitriona también como superusuario?

Para dar respuesta a esta pregunta, debemos tener en cuenta que Docker relega algunas de sus funcionalidades de seguridad directamente en las características inherentes del *kernel* de GNU/Linux. Por tanto, este límite de privilegios entre los contenedores y la máquina anfitriona ya fue diseñado y resuelto. Los controles que efectúa el *kernel* incluyen un aislamiento de procesos mediante espacios de nombre (*namespaces*[15]), que permiten aislar usuarios, procesos, redes o dispositivos; así como también la administración de recursos mediante *cgroups*[16]. Ambos, espacios de nombre y *cgroups* colaboran para ofrecer este aislamiento, y fueron combinados en la versión 2.6.24 del *kernel* de GNU/Linux. Así pues, los contenedores se ejecutarán siempre en un entorno realmente aislado y controlado en la máquina anfitriona. Teniendo noción de este funcionamiento, cabe destacar que Docker posee opciones avanzadas, como el *flag* `-priveleged`, que permitiría elevar las capacidades de un contenedor y eliminar así las limitaciones impuestas por los *cgroups*, dejando al sistema en un estado totalmente vulnerable, por lo que su uso resulta absolutamente desaconsejado.

[15] `http://man7.org/linux/man-pages/man7/namespaces.7.html`
[16] `http://man7.org/linux/man-pages/man7/cgroups.7.html`

Desde el punto de vista práctico, para el caso que nos ocupa actualmente, cabe destacar que hasta la versión 1.10 de Docker, no existía un espacio de nombres para el usuario. Es decir, si un proceso o usuario malintencionado consiguiera, de alguna forma, "salir" del contenedor, éste obtendría un acceso a la máquina anfitriona con exactamente los mismos privilegios que tenía dentro del contenedor. De esta forma, si existían permisos de superusuario (UID = 0), estos permisos también se mantendrían en el exterior, produciéndose así un ataque de escalada de privilegios, en el que un usuario no autorizado obtendría privilegios sobre un sistema al que no debería tener acceso. A partir de la versión 1.10 de Docker, esta vulnerabilidad fue corregida, gracias a la adopción de los espacios de nombre para los usuarios. Por tanto, desde dicha versión, cada proceso posee su propio conjunto de identificadores para usuarios y grupos. De esta forma, si un proceso dentro de un contenedor tiene asociado un UID = 0, correspondiente a permisos de superusuario, en la máquina anfitriona dicho UID estaría mapeado con un identificador totalmente distinto, perteneciente a un usuario no privilegiado. Este mapeo evita que procesos ejecutados dentro de un contenedor puedan obtener permisos de superusuario fuera del contenedor. [2][7]

Realizar una prueba en la que se consiga imitar la amenaza previamente descrita, en la que se "rompería" la barrera de la contedorización, resultaría sumamente complejo y fuera del alcance de este proyecto. Sin embargo, sí existen otras formas más sencillas de comprobar si un contenedor está siendo limitado por la aplicación de espacios de usuario o no. A pesar de que la protección de espacios de usuario está disponible desde la versión 1.10 de Docker, esta es una característica que, atendiendo a la documentación oficial, no está habilitada por defecto, siendo labor del administrador asegurar su correcta configuración [25]. En caso de comprobar este hecho, estaríamos ante una falla importante de seguridad en la configuración por defecto de una arquitectura basada en Kubernetes trabajando con contenedores Docker. La prueba que realizaremos para comprobar la existencia o inexistencia de una protección ofrecida por los espacios de nombres es muy simple; si observamos nuevamente el fichero de configuración YAML asociado al despliegue de un *pod* con un contenedor Ubuntu, disponible en el anexo A.4, veremos como se ha habilitado también la compartición de un espacio del disco. Concretamente, el contenedor tendrá un punto de montaje, ubicado en el directorio `/root-test/`, que se conectará directamente con el directorio `/etc/` de la máquina anfitriona. Dentro del directorio `/hosts/` hay diferentes ficheros de configuración que no deberían ser editables por un usuario sin permisos de superusuario, como podría ser el popular archivo `/etc/hosts`, un fichero donde es posible asociar diferentes direcciones IP con nombres de servidores, un fichero de configuración ampliamente usado y cuya modificación o eliminación podría ser catastrófica para el correcto funcionamiento de un servidor en red.

De este modo, puesto que se ha establecido un punto de montaje dentro del contenedor donde el usuario `root`, usuario con el que se accede por defecto, tal y como vimos en la sección 6.4.1 tiene acceso al fichero `/etc/hosts` de la máquina anfitriona, comprobaremos si es posible modificar este archivo libremente o no. Recordemos que, en caso de que los espacios de nombre estuvieran activados, el usuario efectivo dentro de la máquina anfitriona no tendría UID = 0 y, por tanto, no podría modificar tal fichero bajo ninguna circunstancia. No obstante, puesto que la protección ofrecida por los espacios de nombre está deshabilitada por defecto, el usuario `root` existente dentro del contenedor también

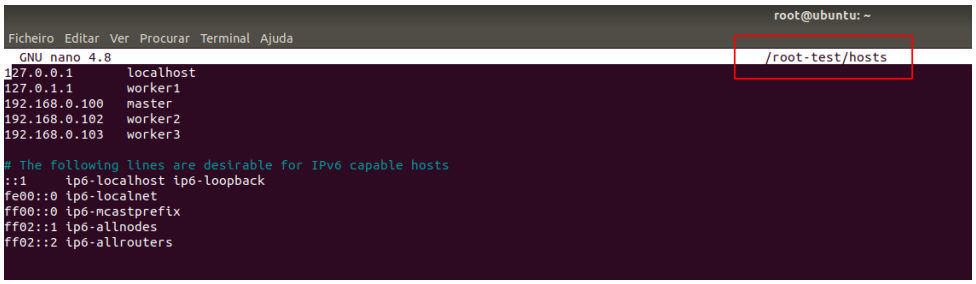


Figura 31: Acceso al fichero `/etc/hosts` de la máquina anfitriona desde dentro de un contenedor

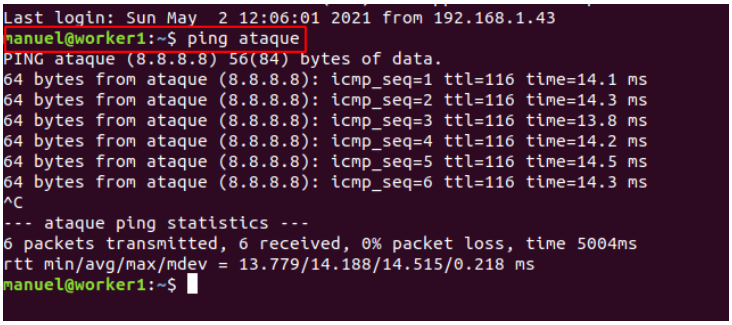


Figura 32: Comprobación, desde la máquina anfitriona, de que el fichero `/etc/hosts` fue modificado

es reconocido como tal fuera de él, por la máquina anfitriona, permitiendo la edición de ficheros críticos, en caso de tener acceso, de la forma que fuere, a ellos. Tal situación se ve representada en las figuras 31 y 32, donde el usuario `root` del contenedor Ubuntu consigue modificar el fichero `/etc/hosts` de la máquina anfitriona, en este caso, el *worker* 1. Para la prueba, se añadió la línea `ataque 8.8.8.8` a las líneas de configuración de `/etc/hosts`.

6.5. Comparativa de aislamiento entre virtualizaciones: máquinas virtuales VS contenedores

En esta sección trataremos de realizar una comparativa entre el nivel de aislamiento que ofrecen las máquinas virtuales en comparativa con los contenedores, en lo que se refiere a la protección del sistema anfitrión, según lo contemplado en el requisito no funcional RNF7, expuesto en la tabla 30. En realidad, debido a las limitaciones establecidas en la sección 4.1, el laboratorio de pruebas empleado a lo largo de este proyecto está pensado para funcionar empleando varias máquinas virtuales, sobre las que, a su vez, corren múltiples contenedores, tal y como fue narrado en la sección 5 y se puede observar en la figura 5.

Por tanto, para poder realizar pruebas en las que uno o varios contenedores puedan ser directamente ejecutados sobre la máquina anfitriona, sin existir la capa intermedia de virtualización hardware, se ha realizado una pequeña modificación en el laboratorio de trabajo para las pruebas de esta sección. Concretamente, la modificación consistió en la incorporación de la máquina anfitriona como nuevo nodo *worker* del clúster controlado por el nodo *master*, ubicado, a su vez, en una máquina virtual albergada por la máquina

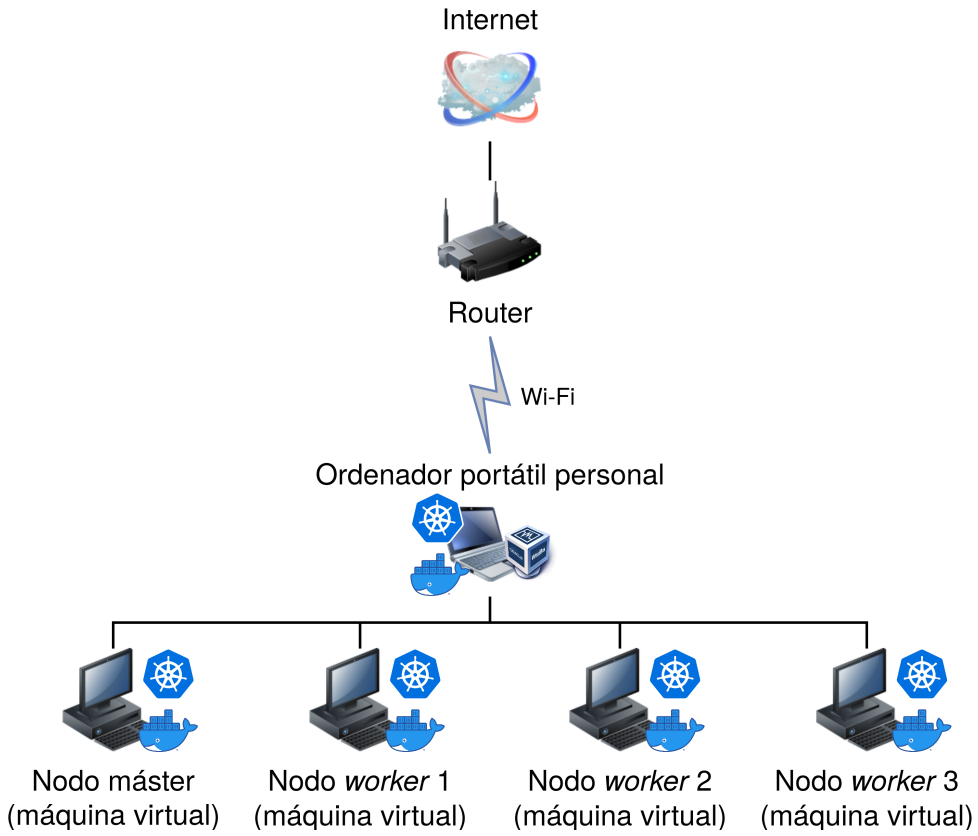


Figura 33: Laboratorio modificado

anfitriona. Por tanto, tras la repetición de los pasos especificados en la sección 5.3.1, pero sobre la propia máquina anfitriona, el laboratorio de trabajo modificado[17] sería el indicado en la figura 33.

Una vez tenemos un nodo *worker* corriendo directamente sobre la máquina anfitriona, sin la capa de virtualización de hardware corriendo por medio, comprobaremos si el aislamiento ofrecido por esta tecnología de virtualización resulta igual de eficiente o, por el contrario, no lo es. Para ello, replicaremos las pruebas ejercidas en la sección 6.3.1, ya que se trata de una prueba muy visual que nos permitirá observar cuáles serían las consecuencias reales sobre un servidor si no tuviera la protección ofrecida por virtualización hardware. De esta forma, se replica exactamente la misma prueba que en la sección 6.3.1, pero esta vez ejecutando el programa altamente consumidor de memoria en el nodo *worker* que se corresponde con la máquina anfitriona. En la figura 34 se puede observar el estado de la memoria de máquina anfitriona antes de ejecutar el programa, donde se puede ver que, si bien existe un consumo elevado (debido a la ejecución simultánea de múltiples máquinas virtuales), es un consumo que se mantiene totalmente estable.

No obstante, una vez ejecutamos el programa altamente consumidor de memoria dentro del contenedor ejecutado sobre la máquina anfitriona, comprobamos que el comporta-

[17] Aclarar que este laboratorio modificado solo será empleado para la prueba actual, pero se seguirá usando el diseño original para el resto de pruebas, a menos que se indique lo contrario.

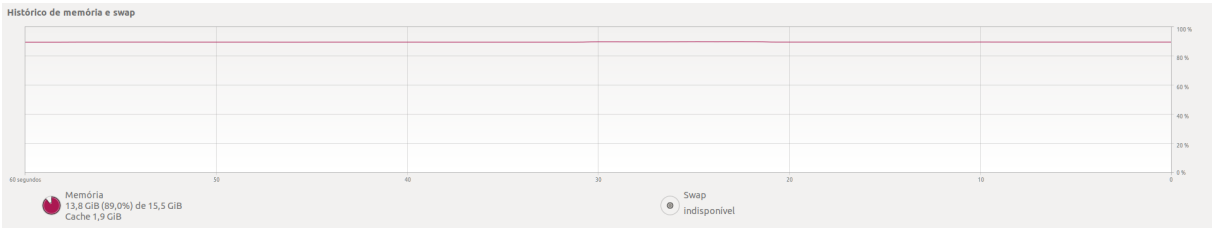


Figura 34: Mediciones del uso de la memoria RAM de la máquina anfitriona antes de la prueba

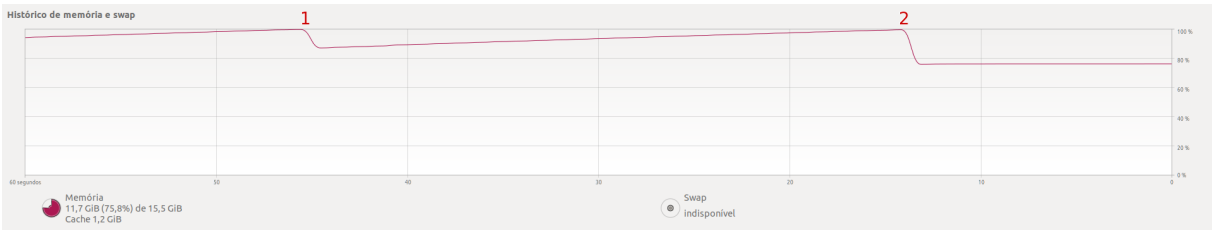


Figura 35: Mediciones del uso de la memoria RAM de la máquina anfitriona después de la prueba

miento de la memoria muda y además, ocurre un mal funcionamiento en la máquina como consecuencia de este uso excesivo de memoria. Las mediciones de la memoria después de la memoria pueden consultarse en la figura 35.

Podemos observar en la figura 35 como, al igual que en las pruebas realizadas en la sección 4.1, el uso de la memoria aumenta paulatinamente hasta llegar al 100 %, momento en el que la protección de *out-of-memory* se encarga de matar el proceso problemático, liberando así la memoria ocupada hasta el momento y volviendo al uso de memoria existente antes de la ejecución del programa. Esto se corresponde con la curva 1 indicada en la figura 35. No obstante, llama la atención que, una vez el consumo de memoria desciende abruptamente tras la muerte del proceso, éste vuelve a aumentar paulatinamente una vez más. Realmente, se desconocen las causas que provocaron esta nueva subida en el uso de la memoria: tal vez el programa continuó ejecutándose o quizás la máquina anfitriona no supo gestionar bien el uso de memoria tras esta situación de estrés continuado. Sea como fuere, el uso de memoria aumenta nuevamente hasta el 100 %, como respuesta, recordemos, de un uso excesivo de memoria por parte de un contenedor, al no existir ningún mecanismo de limitación de memoria establecido por Kubernetes. En esta segunda curva, una vez la máquina anfitriona llega nuevamente a su máximo, debe liberar nuevamente memoria. En este caso, esta liberación de memoria pasa por interrumpir la ejecución de una de las máquinas virtuales, concretamente la máquina "Worker1". Todo este comportamiento puede ser apreciado en la segunda de las curvas de la figura 35, indicada con el número 2. Podemos comprobar en la ventana de gestión de VirtualBox, reflejada en la figura 36, como la máquina virtual "Worker1" tuvo que ser abortada abruptamente.

Por tanto, tras este experimento podemos comprobar como, en caso de ocurrir algún fallo importante sobre los procesos ejecutados dentro de un contenedor, de no existir unas buenas limitaciones entre la máquina anfitriona y los contenedores, el mal funcionamiento

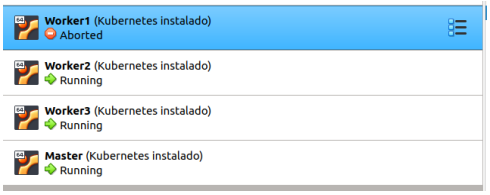


Figura 36: Máquina virtual "Worker1" abortada en VirtualBox

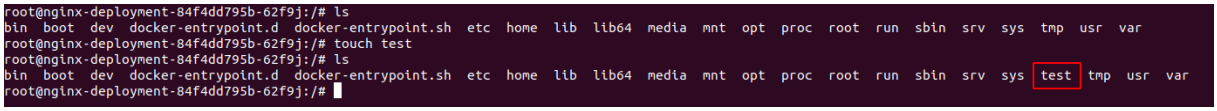


Figura 37: Escritura en disco de forma indiscriminada

podría extenderse hasta los propios procesos de la máquina anfitriona, alcanzando casos tan graves como éste, donde el propio sistema anfitrión se queda sin memoria disponible y se ve obligado a cerrar procesos que, en principio, no están relacionados con el proceso de contedorización. Recordemos que este sería el diseño y comportamiento propios de un clúster de Kubernetes bajo una instalación por defecto, en caso de que no exista ningún tipo de limitación o securización.

6.6. Escritura en disco

Al establecer el requisito no funcional RNF8, expuesto en la tabla 31, reflexionamos sobre la necesidad de realizar escrituras en disco, o no, dependiendo del tipo de aplicación que tengamos que desplegar un clúster. Por ejemplo, podemos tener aplicaciones que realicen cálculos y muestren los resultados como respuesta, pero no precisen almacenar en ningún momento estos resultados. En este tipo de situaciones, conceder permisos de escritura en disco, abre la puerta a nuevas vulnerabilidades que podrían ser evitadas en caso de contar con sistemas de solo lectura.

Puesto que nuestro único objetivo en esta sección es comprobar la posibilidad de escribir en disco en un sistema configurado sin ningún tipo de protección, podemos hacer uso de cualquiera de las aplicaciones ya empleadas hasta el momento. Por ejemplo, podemos entrar en un una consola de comandos interactiva dentro de un contenedor que se ejecute sobre cualquiera que los nodos *worker*, como puede ser uno de los *pods* configurados con el fichero YALM del anexo A.2, referente a un servidor Nginx. De este modo, una vez en la consola de comandos interactiva, intentamos realizar una escritura en disco y observamos que no existe ningún tipo de limitación, existiendo pues, la posibilidad de realizar escrituras indiscriminadamente, tal y como se puede observar en la figura 37.

6.7. Comprobación de la existencia de un mecanismo de validación mediante firmas digitales en Kubernetes

Tal y como fue contemplado, brevemente, en el requisito no funcional RNF9, reflejado en la tabla 32, un aspecto a tener en cuenta para aumentar la seguridad de un clúster Kubernetes se trata de la posibilidad de validar las imágenes con las que trabaja. Debemos

```
manuel@master:~$ kubectl create deployment kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1
deployment.apps/kubernetes-bootcamp created
manuel@master:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                  READY   STATUS    RESTARTS   AGE
default       kubernetes-bootcamp-57978f5f5d-5d8z8  1/1     Running   0          32s
kube-system   coredns-558bd4d5db-lfmtc              1/1     Running   0          14d
kube-system   coredns-558bd4d5db-n742w              1/1     Running   0          14d
kube-system   etcd-master                           1/1     Running   2          14d
```

Figura 38: Despliegue de una imagen con origen Google Cloud sin validar su firma

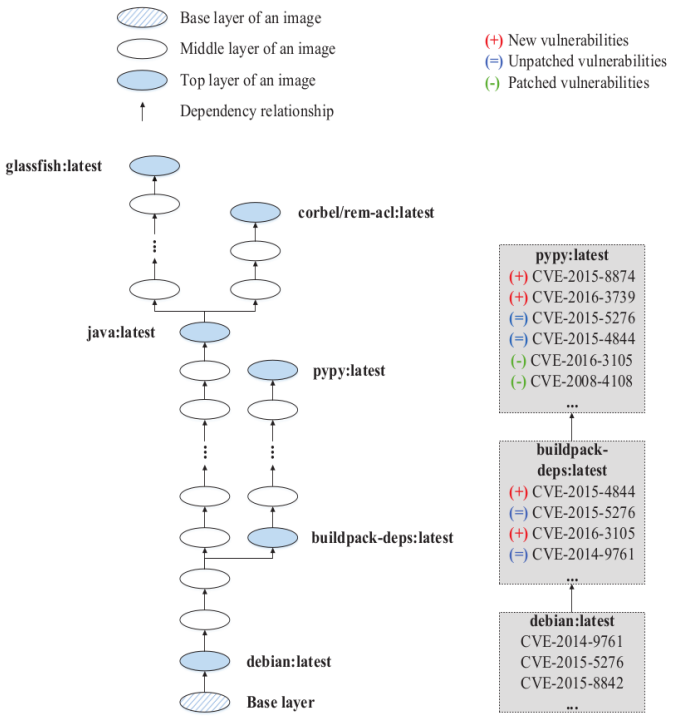
tener en cuenta que en este tipo de entornos de trabajo, muchas de las imágenes con las que se trabaja viajarán a través de la red, moviéndose entre diferentes sistemas. Este hecho puede acarrear implicaciones de seguridad, puesto que es necesario preservar y comprobar que los datos existentes en el interior de las imágenes se correspondan con los datos originales y no fueron modificados por una tercera persona. De esta forma, ya que la fuente de datos puede provenir desde un medio no confiable, como es Internet, es esencial garantizar aspectos como:

- **Autenticación:** el creador del contenido es quién dice ser.
- **Integridad:** la imagen no fue modificada durante la transmisión.
- **No repudio:** el creador de la imagen no puede negar que fue él quién realmente creó la imagen.

Este conjunto de características compondrán, conjuntamente, la validación de imágenes mediante firma digital.

No obstante, como hemos visto gracias a las pruebas realizadas hasta el momento, Kubernetes no realiza por defecto ninguna comprobación en la firma de las imágenes. Este comportamiento resulta lógico, pues debemos añadir un conjunto de firmas públicas para que el sistema las pueda validar. No obstante, investigando sobre el uso de firmas digitales sobre un clúster Kubernetes, podemos ver como Kubernetes no soporta este servicio por defecto. Es decir, a pesar de que la tecnología de contedorización Docker sí presenta una herramienta para la validación de imágenes, como es el "Docker Content Trust", Kubernetes no ofrece ninguna integración con esta herramienta por defecto [20][21]. De esta forma, en caso de que no se exploren alternativas de terceros para integrar la validación mediante firmas, un sistema basado en Kubernetes sería siempre vulnerable a posibles ataques de modificación indeseada o suplantación de identidad en la autoría de las imágenes.

Así, a lo largo de las otras secciones vimos como fue posible desplegar contenedores en el laboratorio de pruebas sin ningún tipo de validación de firmas. Todas las imágenes usadas hasta el momento provenían de la misma fuente, el Docker Hub, a pesar de que no es necesario que vengan de esta fuente concreta para comprobar, o no, la validez de su firma. Para comprobar este hecho, se realizó un despliegue tomando como origen una imagen albergada en Google Cloud, tal y como se puede observar en la figura 38, en donde podemos observar como desde diferentes orígenes la firma tampoco es comprobada.



(a) Image dependency graph example (b) Vulnerability propagation results

Figura 39: Ejemplificación de herencia de vulnerabilidades entre imágenes dependientes

Fuente: [8]

6.8. Herencia de vulnerabilidades entre imágenes

En esta sección se estudiará acerca de la posibilidad de que las imágenes empleadas en un clúster Kubernetes puedan heredar vulnerabilidades desde otras imágenes, tal y como fue especificado en el requisito no funcional RNF10, reflejado en la tabla 33. Para comprender este posible suceso, debemos tener en cuenta que, tal y como fue indicado en la sección 3.3, las imágenes que darán lugar a futuros contenedores no tienen por qué ser hechas desde cero, sino que es posible que las imágenes tomen como base otras imágenes ya existentes, creando así una relación padre-hijo entre ellas. Continuando con esta idea en mente, debemos remarcar que, si bien ésta es una característica muy útil para reducir costes y añadir una gran flexibilidad, la dependencia entre imágenes también puede provocar que cualquier vulnerabilidad software que la imagen padre contuviera en su interior sea transferida instantáneamente a la imagen hijo, si no se aplican medidas de protección.

Para evidenciar esta situación, se mostrará uno de los resultados del estudio "*A Study of Security Vulnerabilities on Docker Hub*", realizado por Rui Shu, Xiaohui Gu y William Enck, en el que se realizan numerosos análisis y entre los cuales se encuentra el estudio de herencia de vulnerabilidades entre imágenes dependientes. Concretamente, en la figura 39 podemos observar como una imagen padre, `debian:latest`, sirve como base para la creación de otras imágenes dependientes, como `buildpack-deps:latest` o `java:latest`, que, a su vez, pueden asumir el rol de padre y crear nuevas imágenes dependientes, como `pypy:latest`, en el caso de `buildpack-deps:latest`; o `glassfish:latest`

```
manuel@master:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                    READY   STATUS    RESTARTS   AGE
default       kubernetes-bootcamp-57978f5f5d-5d8z8    1/1     Running   0          21h
default       nginx-deployment-84f4dd795b-78l9p       1/1     Running   0          22m
default       nginx-deployment-84f4dd795b-7th5l       1/1     Running   0          22m
default       ubuntu-deployment-7ffd97df99-2z9ww      1/1     Running   0          22m
default       ubuntu-deployment-7ffd97df99-j88wp      1/1     Running   0          22m
default       ubuntu-deployment-7ffd97df99-rtpct      1/1     Running   0          22m
default       ubuntu-deployment-7ffd97df99-vtbxn      1/1     Running   0          22m
kube-system   coredns-558bd4d5db-lfmtc                1/1     Running   0          15d
kube-system   coredns-558bd4d5db-n742w                1/1     Running   0          15d
kube-system   etcd-master                             1/1     Running   2          15d
kube-system   kube-apiserver-master                   1/1     Running   2          15d
kube-system   kube-controller-manager-master          1/1     Running   2          15d
kube-system   kube-flannel-ds-6gd6k                   1/1     Running   0          3d1h
kube-system   kube-flannel-ds-p9p6x                   1/1     Running   1          12d
kube-system   kube-flannel-ds-vc7dl                   1/1     Running   0          12d
```

Figura 40: Comprobación del espacio de nombres de las ejecuciones corriendo sobre el clúster

y `corbel/rem-acl:latest`, en el caso de `java:latest`. Todas estas herencias provocan que vulnerabilidades conocidas en la imagen base, `debian:latest` fueran heredadas hasta las ramas más alejadas, junto con la adición de nuevas vulnerabilidades, correspondientes al software último o incluso correspondiente a imágenes intermedias. [8]

6.9. Comprobación de los espacios de trabajo existentes por defecto

Como reflexionamos en la especificación del requisito no funcional RNF11, representado en la tabla 34, el entorno de trabajo más común en el que querremos trabajar bajo un clúster contedorizado, será un entorno en el que sea posible definir espacios de trabajo diferenciados para los diferentes equipos y/o aplicaciones. Dicha separación es posible mediante el uso de espacios de nombre (*namespaces*). En este caso concreto, la separación de diferentes ejecuciones de *pods* en diferentes espacios de nombres, podría entenderse como la creación de un clúster virtual separado mediante software del resto, aunque siempre corriendo todos ellos bajo el mismo clúster físico.

A pesar de este hecho, si revisamos las ejecuciones realizadas hasta el momento, siempre bajo una configuración por defecto, podremos observar como todas ellas están corriendo sobre un mismo espacio común, tal y como se muestra en la figura 40.

Además del espacio de nombres *default*, existen otros tres que son creados automáticamente en la inicialización de un clúster Kubernetes:

- `kube-system`**:** el espacio para los objetos creados por el sistema de Kubernetes.
- `kube-public`**:** un espacio accesible para todos los usuarios, incluso sin autenticación. Está reservado para aquellos recursos del clúster que queramos hacer visibles y legibles de forma pública. El aspecto público de este espacio de nombres es solo una convención, pero no un requisito.
- `kube-node-lease`**:** espacio dedicado a aquellos objetos compartidos asociados a cada nodo, que permiten una mejora del rendimiento a medida que el clúster escala. [31]

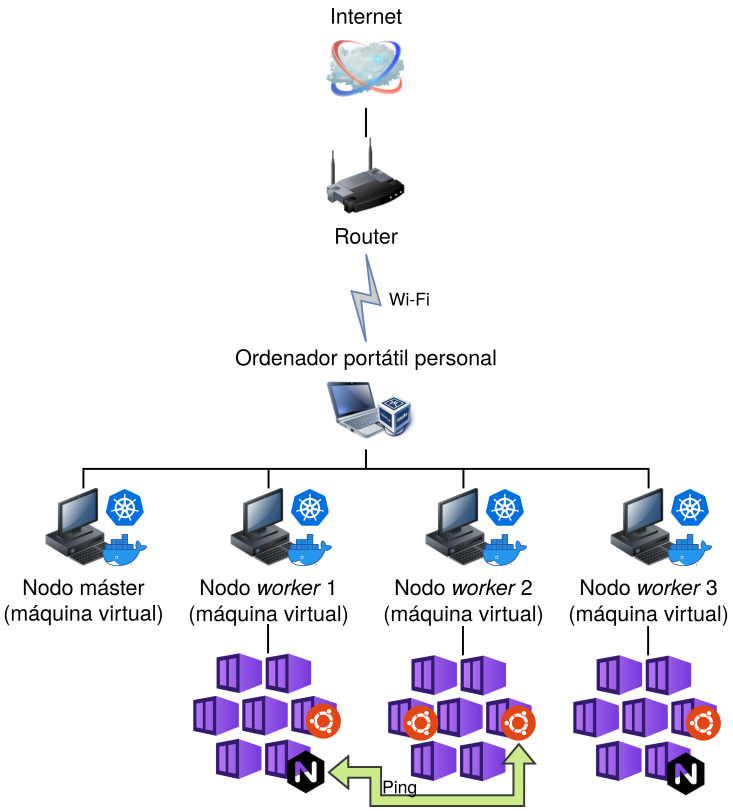


Figura 41: Representación gráfica de la prueba de interconexión de contenedores en red

6.10. Comprobación de la interconexión por red de contenedores en diferentes nodos

Tal y como fue especificado en el requisito no funcional RNF12, establecido en la tabla 35, en caso de que un ataque de red acontezca, el hecho de permitir que todos los contenedores ejecutándose en el clúster posean interconexión de red entre ellos, solo pondría en mayor riesgo la arquitectura desplegada. Normalmente, muchas de las aplicaciones no necesitarán interactuar entre sí, además de que existe la posibilidad de que se comuniquen de forma "externa", pasando antes por el nodo *master*, al que siempre tendrán que estar conectados.

Por tanto, para poder realizar la comprobación de la interconexión de contenedores corriendo sobre diferentes nodos, se desplegaron sobre el clúster diferentes aplicaciones sobre diferentes nodos. Concretamente, se emplearon los ficheros de configuración YAML para las imágenes de Nginx y Ubuntu, representados en los anexos A.2 y A.5, respectivamente. De esta forma, lo que se pretende es realizar el despliegue de dichas aplicaciones en el clúster y, posteriormente, comprobar si es posible, desde dentro de uno de los contenedores en ejecución, establecer comunicación con otro contenedor, referente a la otra aplicación y ubicado en un nodo *worker* diferente. Un resumen gráfico de la prueba a realizar puede observarse en la figura 41.

Para comprobar la posibilidad de intercomunicación de red, se realizaron los siguientes

pasos:

1. Realización de los despliegues según los ficheros de configuración YAML establecidos previamente.

2. Comprobación de que el despliegue fue correcto y obtención del nombre de los *pods* desplegados.

```
manuel@master:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                   READY   STATUS    RESTARTS   AGE
default       kubernetes-bootcamp-57978f5f5d-5d8z8   1/1     Running   0          22h
default       nginx-deployment-84f4dd795b-78l9p      1/1     Running   0          88m
default       nginx-deployment-84f4dd795b-7th5l      1/1     Running   0          88m
default       ubuntu-deployment-7ffd97df99-2z9ww     1/1     Running   0          88m
default       ubuntu-deployment-7ffd97df99-j88wp     1/1     Running   0          88m
default       ubuntu-deployment-7ffd97df99-rtpct     1/1     Running   0          88m
default       ubuntu-deployment-7ffd97df99-vtbxn     1/1     Running   0          88m
kube-system   coredns-558bd4d5db-lfmtc               1/1     Running   0          15d
kube-system   coredns-558bd4d5db-n742w               1/1     Running   0          15d
kube-system   etcd-master                            1/1     Running   2          15d
```

Figura 42: Comprobación del despliegue y obtención del nombre de los *pods*

3. Obtención de un *pod* de cada aplicación ubicados en diferentes nodos.

```
manuel@master:~$ kubectl describe pod nginx-deployment-84f4dd795b-78l9p
Name:         nginx-deployment-84f4dd795b-78l9p
Namespace:    default
Priority:     0
Node:         worker1/192.168.1.101
Start Time:   Sat, 08 May 2021 00:07:03 +0200
Labels:       app=nginx
              pod-template-hash=84f4dd795b
Annotations:  <none>
Status:       Running
IP:           10.244.1.30
IPs:
  IP:           10.244.1.30
Controlled By:  ReplicaSet/nginx-deployment-84f4dd795b
```

Figura 43: *Pod* Nginx en nodo *worker* 1

```
manuel@master:~$ kubectl describe pod ubuntu-deployment-7ffd97df99-2z9ww
Name:         ubuntu-deployment-7ffd97df99-2z9ww
Namespace:    default
Priority:     0
Node:         worker2/192.168.1.102
Start Time:   Wed, 05 May 2021 20:48:38 +0200
Labels:       app=ubuntu
              pod-template-hash=7ffd97df99
Annotations:  <none>
Status:       Running
IP:           10.244.2.8
IPs:
  IP:           10.244.2.8
Controlled By:  ReplicaSet/ubuntu-deployment-7ffd97df99
```

Figura 44: *Pod* Ubuntu en nodo *worker* 2

4. Obtención de la dirección IP interna en el contenedor ejecutado en el *pod* Nginx del nodo *worker* 1.

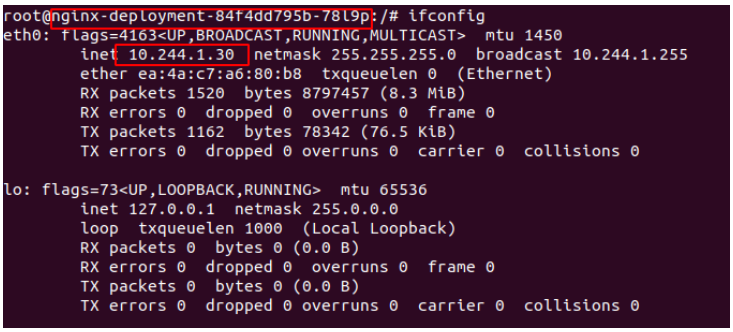


Figura 45: Obtención de la IP interna del contenedor Nginx en el nodo *worker* 1

5. Comprobación práctica de la interconexión entre contenedores de aplicaciones diferentes corriendo sobre diferentes nodos en un clúster Kubernetes. Para la prueba se empleó el comando `ping` indicando como parámetro la dirección IP interna del otro contenedor.

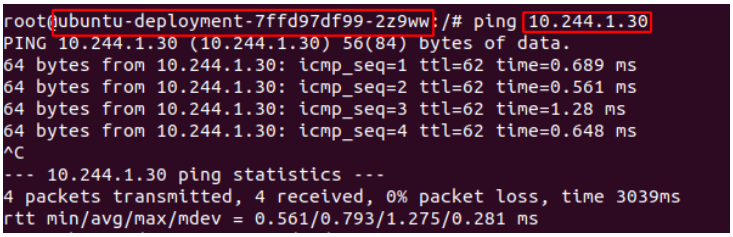


Figura 46: Prueba de establecimiento de conexión

Finalizada esta prueba, podemos confirmar que una configuración por defecto de un clúster Kubernetes, la comunicación de red entre *pods* es posible sin ningún tipo de restricción, lo que podría suponer un mayor riesgo en caso de producirse un ataque de red.

7. Securización

Finalizada la fase de análisis, a lo largo de esta sección nos centraremos en la aplicación de medidas de corrección o prevención, ante la carencia de protecciones existentes en un clúster de contenedores orquestado por Kubernetes en su configuración por defecto. Tomando como base los avances obtenidos gracias a las pruebas y los diferentes entornos empleados en la sección 6, se aplicarán medias de corrección o prevención que se consideren necesarias. Posteriormente, se comprobará la eficacia de la securización realizada, preferentemente mediante la repetición de las pruebas ya realizadas a lo largo de la sección de análisis.

7.1. Análisis estático sobre imágenes de contenedores

Vimos a lo largo del análisis de vulnerabilidades, concretamente en la sección 6.1, como existe la posibilidad de desplegar contenedores basados en imágenes con vulnerabilidades conocidas, incluso siendo provenientes de fuentes que podrían ser consideradas "de confianza". Para comprobar la existencia de vulnerabilidades en dichas imágenes, también en la sección 6.1, fue realizada una prueba de análisis estático de vulnerabilidades sobre una imagen bien conocida y ampliamente empleada.

En esta sección mostraremos posibles mecanismos de securización ante esta problemática, completando así la realización del RNF1, reflejado en la tabla 24. No obstante, puesto que en el momento de realizar esta securización, la imagen seleccionada para la realización de las pruebas en la sección 6.1, Nginx:1.20.0, sigue tratándose de una de las imágenes más recientes existentes en el Docker Hub, se repetirá la prueba seleccionando una imagen con mayor antigüedad, de tal forma que este proceso de securización resulte más notorio. La imagen seleccionada en esta ocasión, se trata de la versión 1.11.0[18], igualmente disponible públicamente en el Docker Hub. Como se puede observar en la figura 47, esta imagen tiene una antigüedad notoria, de aproximadamente 5 años, por lo que es de esperar que se encuentren más vulnerabilidades en dicha imagen, al trabajar con software ya considerado desactualizado.

De esta forma, serán repetidos los mismos pasos para la realización de un análisis estático de vulnerabilidades sobre esta nueva imagen objetivo. Concretamente, resulta especialmente interesante el número de vulnerabilidades reconocidas tras dicho análisis.

[18] `https://hub.docker.com/layers/nginx/library/nginx/1.11.0/images/sha256-b1fcb97bc5f6effb44ba0b5d60bf927e540dbdcfe091b1b6cd72f0081a12207c?context=explore`

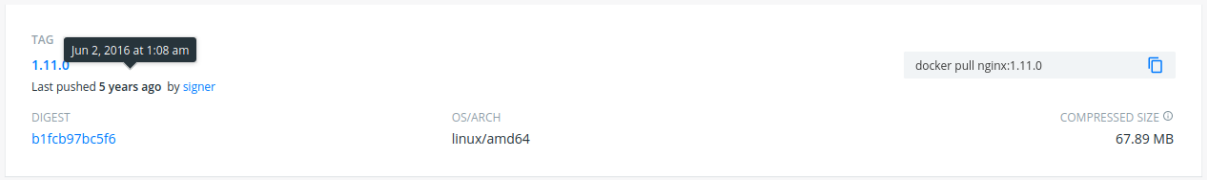


Figura 47: Imagen de Nginx en Docker Hub. *Tag* 1.11.0.

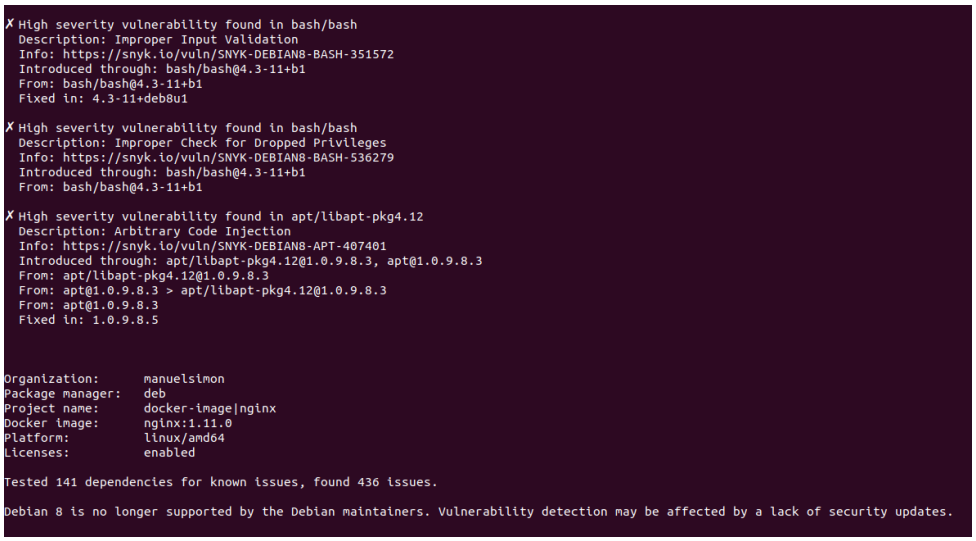


Figura 48: Fin del resultado del análisis estático de la imagen Nginx:1.11.0

En esta ocasión, el número de vulnerabilidades localizas ha alcanzado las 436, contando además con numerosas vulnerabilidades con una severidad alta. Además, de esta vez, el análisis estático también nos indica como el SO base de la imagen ya no mantiene un soporte oficial, lo que puede incrementar el número de vulnerabilidades localizadas, ante la falta de actualizaciones de seguridad. Todo esto puede ser observado en la figura 48.

Replicado el proceso de análisis estático, razonaremos ahora alguno de los mecanismos de securización posibles a aplicar. Puesto que la imagen seleccionada se trata de una imagen que contiene software ya desactualizado, al tener ya cierta antigüedad, un posible mecanismo de securización podría ser, simplemente, actualizar el software existente en la imagen a través de sus propios repositorios. Se mostrará una pequeña demostración a continuación para mostrar la utilidad de este método.

7.1.1. Posible ejemplo de securización

Una posible metodología de securización para disminuir el número de vulnerabilidades localizadas tras la realización de un análisis estático sobre una imagen, podría resumirse en los siguientes pasos:

1. Creación de un contenedor basado en la imagen con vulnerabilidades sobre cualquier máquina en la que tengamos acceso. Preferiblemente ésta debería ser una máquina aislada del resto, ya que vamos a desplegar, conscientemente, un sistema con vulnerabilidades conocidas.

2. Realización de un proceso de actualización de software. Este proceso puede resultar extenso, en caso de querer actualizar todo el software afectado por las vulnerabilidades detectadas, o limitarse a un proceso más rápido y automático haciendo uso de los repositorios asociados al sistema operativo del contenedor. En esta ocasión, simplemente haremos uso de los comandos `apt-get update` y `apt-get upgrade`.

   Además, puesto que se trata de un proceso que siempre realizaremos nada más instanciar el contenedor, es posible añadir los comandos de actualización en un *script*

o en un fichero de arranque Dockerfile[19], de tal forma que se consiga automatizar el procedimiento y, a su vez, minimizar el tiempo en el que el contenedor contendrá tantas vulnerabilidades conocidas.

Listing 4: Fichero Dockerfile de ejemplo

```
FROM nginx:1.11.0
CMD apt-get update && apt-get -y upgrade && bash
```

3. Una vez en posesión del contenedor con el software actualizado, en el que entendemos que se han solventado algunas de las vulnerabilidades localizadas previamente, debemos crear una imagen en función del mismo. Dicha imagen' poseerá el mismo software que la imagen original, pero en su versión más actualizada. Para crear la imagen basada en dicho contenedor, aplicamos un `commit` sobre el mismo, incluyendo el que será el nombre del repositorio y su respectivo *tag*.

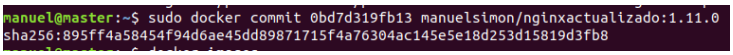


Figura 49: Creación de imagen local basada en el contenedor en ejecución actualizado

4. Disponiendo ya de la imagen local, la exportaremos hasta un repositorio en el Docker Hub. En este punto, debemos tener en cuenta que debe existir un repositorio coincidente con los datos dados en el punto anterior, que puede ser creado a través de la interfaz web del servicio de repositorio de imágenes.

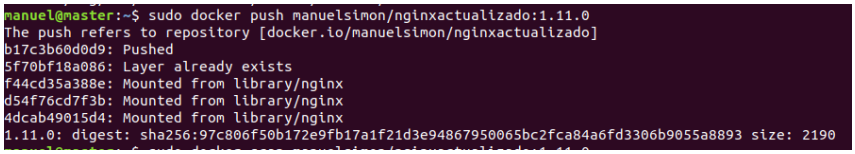


Figura 50: *Push* de la imagen hacia un repositorio en el Docker Hub

5. Por último, comprobaremos la eficacia del proceso de actualización de software, haciendo nuevamente uso de la herramienta para el análisis de vulnerabilidades estático, "Docker Scan". Tras los pasos realizados, podremos comprobar como, aún existiendo vulnerabilidades conocidas, éstas han descendido desde las 436 iniciales a unas 136, consiguiendo una reducción de casi el 70 %, después de este procedimiento de securización, lo que demuestra la importancia de una actualización constante del software. Por tanto, esta prueba ha servido igualmente para ejemplificar la gran importancia de mantener un entorno de trabajo actualizado. Esta recomendación no se aplica solamente a las imágenes empleadas en el entorno virtualizado, sino que también es vital mantener actualizado el software que se ejecuta sobre el propio entorno de trabajo, como por ejemplo, las tecnologías de virtualización o de orquestación.

[19] `https://docs.docker.com/engine/reference/builder/`

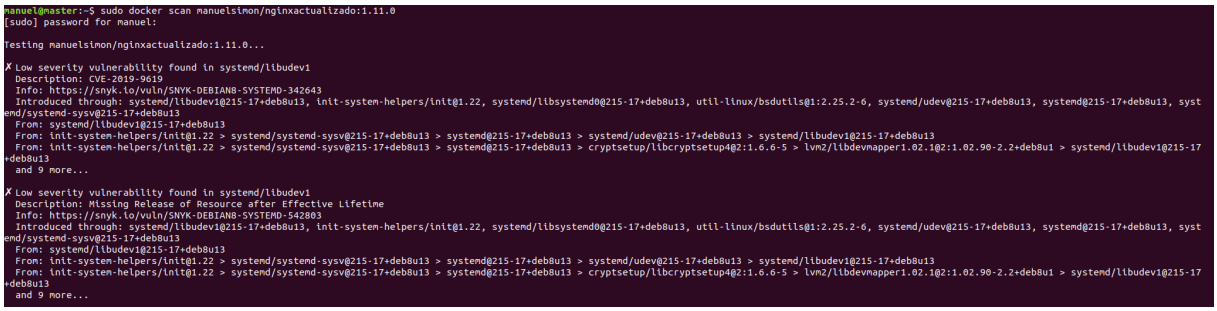


Figura 51: Ejecución de análisis estático sobre la imagen actualizada

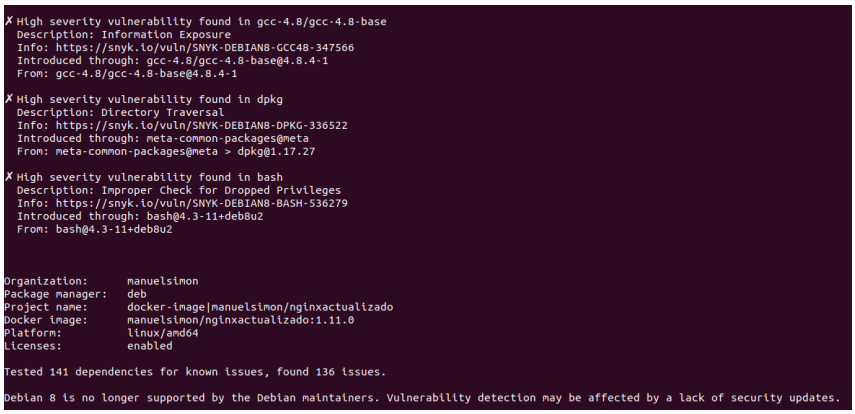


Figura 52: Fin del resultado del análisis estático de la imagen actualizada

7.1.2. Procedimiento estandarizado

Realizados todos los pasos en la demostración previa, resumiremos en este punto los pasos esenciales para realizar la securización después de la ejecución de un análisis de vulnerabilidades estático:

1. Seleccionamos la imagen base sobre la que queremos trabajar, entre las miles disponibles en el catálogo del Docker Hub.

2. Realizamos un análisis estático de vulnerabilidades sobre la imagen escogida, haciendo uso para ello de la herramienta "Docker Scan".

3. Obtenemos un informe detallado de las vulnerabilidades de seguridad conocidas encontradas en la imagen. En función de los resultados arrojados, el operador debe actuar según considere oportuno:

   - Continúa ejecutando la imagen original, siendo conocedor de los riesgos a los que se expone ejecutando dicho software. No recomendado.
   - Comienza un proceso de corrección de las vulnerabilidades encontradas, cuanto menos de las más críticas. Para ello, puede realizar procesos de actualización de software automatizados con ayuda de plantillas Dockerfile o similares, o medidas más exhaustivas, en función de las vulnerabilidades localizadas.

- Finalizado el proceso de corrección de vulnerabilidades, crea una nueva imagen que sustituirá a la original.

Para concluir esta sección, debemos tener en cuenta que este no es el único procedimiento posible para detectar y corregir vulnerabilidades en imágenes. Existen otras herramientas que podrían ser consideradas, como Clair[20], en función de las necesidades de nuestro proyecto. Por ejemplo, Clair tiene una configuración inicial más compleja, pero a cambio, es posible realizar análisis sobre imágenes locales, sin depender de la publicación en el Docker Hub, de forma que será más oportuno para proyectos con imágenes que contengan información considerada secreta en su interior. No obstante, "Docker Scan" se trata de la herramienta recogida en la documentación oficial de Docker para el análisis estático de imágenes [42], contando así con un mayor soporte por parte de los desarrolladores de esta tecnología de contenerización.

7.2. Limitación de recursos

Observamos a lo largo de la sección 6.3 como, por defecto, los contenedores instanciados en un clúster Kubernetes pueden llegar a consumir el total de la capacidad existente de los recursos físicos, como pueden ser la memoria, la CPU o el uso de red, desde un único contenedor. De esta forma, en caso de que un contenedor mal configurado o empleado por un usuario malintencionado, podría llegar a provocar un ataque de DoS, al dejar sin capacidad computacional al resto de contenedores corriendo sobre la misma máquina, o inclusive la máquina anfitriona que lo contiene. Para evitar este tipo de situaciones, Kubernetes provee un sistema de limitación de recursos que pondremos en uso a lo largo de esta sección, a la vez que repetimos las pruebas efectuadas a lo largo de la sección 6.3, para comprobar su efectividad.

7.2.1. Limitación de memoria

En esta primera subsección de la securización aplicada a la limitación de recursos, aplicaremos las correcciones pertinentes para limitar el uso de memoria, completando así la ejecución del RNF3, descrito en la tabla 26.

Para poder ejercer una limitación en el máximo de memoria que un contenedor puede consumir, basta con indicar en el archivo de configuración YAML, en el que se especifica el despliegue a realizar, una serie de parámetros. Concretamente, podemos especificar dos tipos de parámetros, aunque ambos deben estar bajo la categorización `resources`:

- `requests`: se trata de un límite flexible, que puede ser superado en caso de que el clúster tenga recursos disponibles suficientes en dicho nodo o pueda balancearlos sin problema.
- `limits`: se trata de un límite estricto, que no puede ser superado. [30]

Por tanto, para poder evitar la problemática del consumo excesivo de memoria, bastaría con añadir unas líneas como las siguientes, adaptables según cada caso de uso:

[20] `https://github.com/quay/clair`

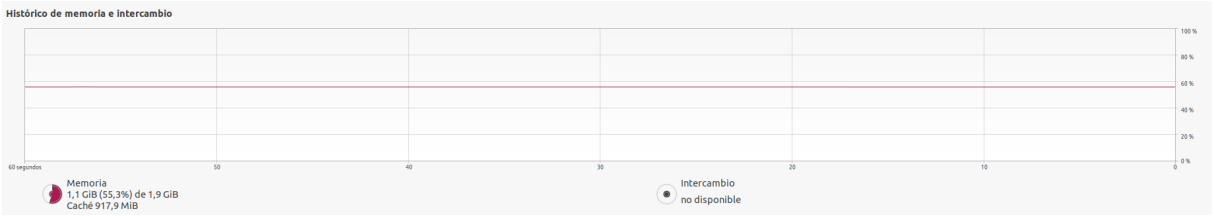


Figura 53: Mediciones del uso de la memoria RAM antes de la prueba (correcciones aplicadas)

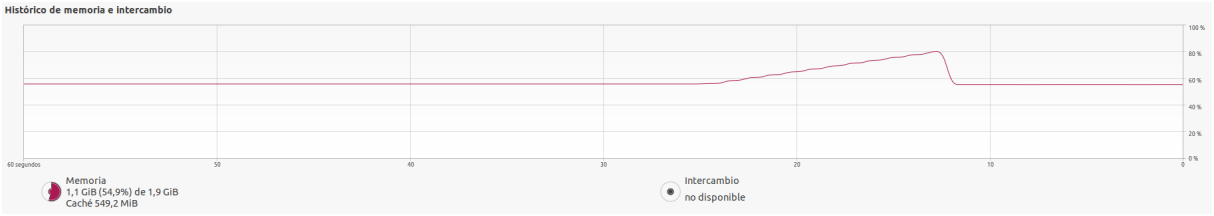


Figura 54: Mediciones del uso de la memoria RAM después de la prueba (correcciones aplicadas)

Listing 5: Ejemplo de limitación de memoria en un fichero YAML

```
resources:
  limits:
    memory: "500Mi"
```

Como se puede observar, en el ejemplo mostrado se ha establecido un límite estricto de 500MiB, por lo que cada contenedor que se inicialice desde un fichero YAML con estas líneas, bien sea para establecer la definición de un *pod* o de una instancia, nunca podrá superar los 500MiB de memoria. Para comprobar su funcionamiento, repetiremos la misma prueba que fue indicada en la sección 6.3.1, haciendo uso nuevamente del código disponible en el anexo A.3.

En esta ocasión, antes de ejecutar la prueba, podemos ver como existe un consumo de memoria totalmente estable, reflejado en la figura 53. Posteriormente, ejecutando el programa consumidor de memoria, observamos que su consumo va aumentando, pero una vez el contenedor alcanza los 500MiB máximos establecidos, este consumo no aumenta más. Además, ya que dentro del contenedor se ha alcanzado el máximo de memoria disponible, en esta ocasión también salta la protección "*out-of-memory*", interrumpiendo la ejecución del programa y haciendo que el consumo de memoria vuelva al punto inicial. Todo este comportamiento descrito puede observarse en la figura 54.

La gran diferencia entre esta prueba y la realizada sin ningún tipo de protección en la sección 6.3 es que, en esta ocasión, la máquina anfitriona no dejó de responder en ningún momento, pudiendo ofrecer así un servicio continuado y sin interrupciones, logrando evitar posibles ataques de DoS derivados de un consumo excesivo de memoria. Por tanto, tras aplicar las medidas de securización, podemos observar como la máquina anfitriona tiene siempre cierta capacidad de memoria disponible para atender otros posibles procesos, bien fueran otros contenedores albergados en la máquina "Worker1", u otras aplicaciones no contenerizadas corriendo igualmente sobre dicha máquina.

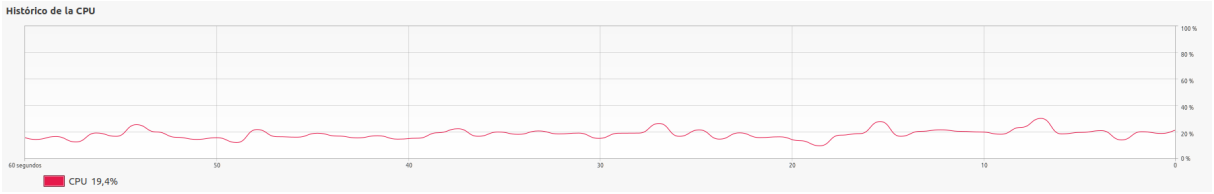


Figura 55: Mediciones del uso de CPU antes de la prueba (correcciones aplicadas)

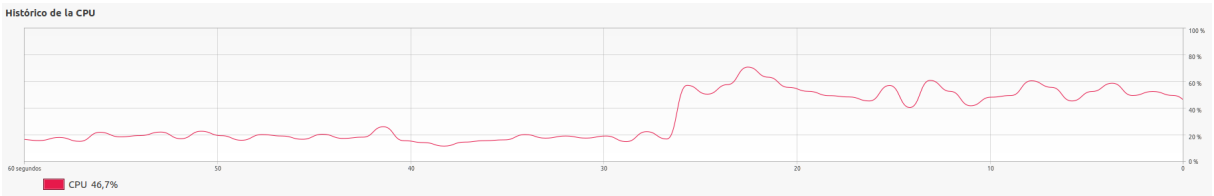


Figura 56: Mediciones del uso de CPU después de la prueba (correcciones aplicadas)

7.2.2. Limitación de CPU

Prosiguiendo con los mecanismos de securización para la aplicación de límites de recursos, en esta sección abordaremos las correcciones necesarias para aplicar una limitación de CPU, realizando después una prueba de concepto y concluyendo así el RNF4, descrito en la tabla 27.

La limitación de la CPU se realiza de una manera prácticamente idéntica a la limitación de memoria, descrita en la sección 7.2.1, por lo que no es preciso realizar nuevamente explicaciones teóricas sobre la securización a aplicar. En esta ocasión, un ejemplo de limitación de CPU en un fichero YAML podría ser el siguiente:

Listing 6: Ejemplo de limitación de CPU en un fichero YAML

```
resources:
  limits:
    cpu: "500m"
```

Por tanto, en esta ocasión, todo contenedor instanciado mediante un fichero de configuración YAML con dichas líneas, podría emplear media CPU. Una vez más, la limitación de CPU deberá ser ajustada según las circunstancias o restricciones de cada caso de uso.

Para poder comprobar el correcto funcionamiento de la limitación establecida, en esta ocasión no será posible repetir la utilización del software altamente consumidor de memoria, ya que, a pesar de crear un bucle, debido a las restricciones ya existentes en el uso de memoria, este bucle no bastará para alcanzar un uso excesivo de CPU de manera continuada. Por lo tanto, para la realización de esta prueba, se ha ejecutado el siguiente comando en el interior de un contenedor configurado con la limitación de CPU: `cat /dev/zero >/dev/null`. Con la ejecución de este comando, creamos un bucle infinito en el que escribimos caracteres nulos en un "dispositivo nulo". En las figuras 55 y 56 podemos ver el uso de CPU antes y después de la ejecución de este comando, respectivamente.

Antes de la ejecución, podemos observar que existe un uso de CPU de aproximadamente el 20 %, que se mantiene relativamente constante. Después de la ejecución del bucle

infinito, puede observarse que el consumo de CPU crece instantáneamente, pero este se mantiene en unos valores de aproximadamente el 50 % de CPU. En ciertos momentos, este valor se ve ligeramente superado, aunque siempre rondando la misma cifra, de forma el resultado obtenido no parece tan estricto como el dado en el caso de la limitación de memoria. Igualmente, debemos tener en cuenta que estamos realizando las mediciones desde una de las máquinas virtuales que conforman el laboratorio y no desde un nodo correspondiente con una máquina física, por lo que debemos suponer cierto margen de error.

Finalmente, podemos concluir que, una vez más, las limitaciones establecidas impiden un uso de recursos de CPU exhaustivos y exclusivos por parte de un único contenedor, impidiendo correctamente la ejecución de potenciales ataques de DoS.

7.3. Limitación de los permisos de ejecución dentro de los contenedores

En la especificación del RNF6, representado en la tabla 29, y a lo largo de la sección 6.4, reflexionamos acerca de la necesidad de ejecutar los contenedores con permisos de superusuario. Concretamente, en la sección de análisis pudimos observar como existen ciertas implicaciones de seguridad importantes provocadas por la utilización de contenedores en modo superusuario, no solamente dentro del propio contenedor, sino pudiendo llegar a afectar también a la máquina anfitriona. Por lo tanto, el objetivo de esta sección es la aplicación de medidas correctivas y comprobar su correcto funcionamiento, concluyendo así el desempeño del RNF6.

Para la limitación de los permisos de ejecución dentro de los contenedores, haremos uso de las "*pod security policies*", unas políticas de seguridad a nivel de clúster para controlar aspectos sensibles de seguridad [32]. En esta ocasión, dentro del fichero YAML de configuración, añadiremos unas líneas como las siguientes, a adaptar según nuestro caso de uso:

Listing 7: Ejemplo de limitación de CPU en un fichero YAML

```
securityContext:
  allowPrivilegeEscalation: false
  runAsUser: 1000
  runAsGroup: 3000
```

Con dichas opciones de configuración, estamos indicando que los contenedores deben ejecutarse con un usuario con un UID = 1000 y un GID = 3000, además de impedir la escalada de privilegios. Es decir, con estas medidas de securización estaremos forzando a la utilización de un usuario sin privilegios e impidiendo que se puedan obtener permisos de superusuario dentro del contenedor. Una vez establecidas las medidas de securización, podemos comprobar su eficacia en práctica.

7.3.1. Comprobación de la securización: implicaciones dentro del contenedor

Tal y como reflexionamos en la sección de análisis, si un ataque tuviera éxito y se hiciera con el control del contenedor, dicho ataque puede verse amplificado o minimizado en función de si el atacante obtiene permisos de superusuario o no. Por tanto, resulta

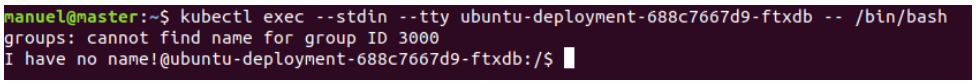


Figura 57: Comprobación de iniciación del contenedor sin permisos `root`

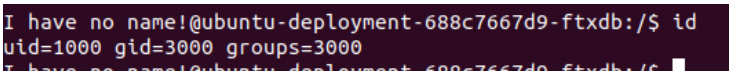


Figura 58: Comprobación de UID y GID dentro del contenedor

importante limitar, en la mayor parte de los casos, el tipo de usuario y sus permisos de ejecución. Por tanto, una vez limitados los permisos gracias a las medidas de securización aplicadas, comprobaremos sus implicaciones dentro de un contenedor.

El primer paso será entrar dentro del contenedor a través de una consola de comandos interactiva, para observar el usuario que es asignado. Puesto que no ha sido configurado previamente ningún usuario con UID = 1000, ni tampoco un GID = 3000 obtendremos un pequeño aviso al iniciar la consola interactiva, pero esto no impedirá que podamos trabajar dentro de ella como un usuario con permisos limitados, tal y como se muestra en la figura 57. También comprobaremos que los parámetros de UID y GID establecidos son coincidentes y no se corresponden con los del usuario `root` (0), tal y como sí correspondían en la prueba realizada en la sección 6.4.1. Este comportamiento puede observarse en la figura 58.

Para finalizar esta pequeña serie de pruebas dentro del contenedor, intentaremos ejecutar un comando para el que son precisos permisos de superusuario. Comprobamos en la figura 59 como, al tener un usuario sin este tipo de permisos, no podemos ejecutar este tipo de acciones privilegiadas.

7.3.2. Comprobación de la securización: implicaciones en la máquina anfitriona

En la sección 6.4.2, observamos como el UID y el GID de un usuario es mapeado con una relación directa uno a uno dentro y fuera del contenedor. Bajo estas circunstancias, en caso de que un atacante consiguiera "romper" el aislamiento del contenedor, obtendría lo mismos permisos que tuviera dentro del contenedor, pero en la propia máquina anfitriona. Mismamente, sin llegar a casos tan extremos, pudimos ver como, en caso de compartir un directorio entre la máquina anfitriona y el contenedor, un usuario `root` dentro del contenedor puede llegar a modificar archivos para los cuales son precisos permisos de superusuario en la máquina anfitriona, consiguiendo afectar significativamente al comportamiento de la máquina anfitriona.

Para poder ofrecer un aislamiento mayor entre máquina anfitriona y contenedores,

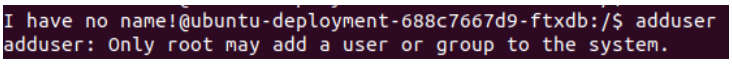


Figura 59: Intento fallido de ejecución de comando privilegiado

```
manuel@worker1:~$ sudo adduser dockermap
Añadiendo el usuario `dockermap' ...
Añadiendo el nuevo grupo `dockermap' (1001) ...
Añadiendo el nuevo usuario `dockermap' (1001) con grupo `dockermap' ...
Creando el directorio personal `/home/dockermap' ...
Copiando los ficheros desde `/etc/skel' ...
Nueva contraseña:
Vuelva a escribir la nueva contraseña:
passwd: contraseña actualizada correctamente
Cambiando la información de usuario para dockermap
Introduzca el nuevo valor, o presione INTRO para el predeterminado
        Nombre completo []:
        Número de habitación []:
        Teléfono del trabajo []:
        Teléfono de casa []:
        Otro []:
¿Es correcta la información? [S/n] S
manuel@worker1:~$ sudo sh -c 'echo dockermap:5000:65536 > /etc/subuid'
manuel@worker1:~$ sudo sh -c 'echo dockermap:5000:65536 > /etc/subgid'
```

Figura 60: Creación de nuevo usuario `dockermap` y sus respectivos `subuid` y `subgid`

Docker puede hacer uso de los denominados espacios de nombre (*namespaces*), siempre que se trabaje con una versión de Docker igual o superior a la 1.10. Sin embargo, esta protección, que puede ser de gran importancia, no está configurada por defecto, sino que dicha responsabilidad recae en el administrador del sistema.

Por tanto, en esta ocasión, lo que queremos conseguir es un contenedor corriendo con el usuario `root` dentro de él, pero que sea identificado como un usuario no privilegiado en la máquina anfitriona, de tal forma que no pueda acceder y modificar ficheros y/o procesos sensibles. Es decir, lo que se procura en esta securización es remapear el UID y el GID del usuario fuera del contenedor. Así pues, para realizar esta securización, activaremos esta configuración, realizando las pruebas desde una de las máquinas virtuales que alberga un nodo *worker*, puesto que se trata de una configuración propia de Docker y no de Kubernetes.

Primeramente crearemos un nuevo usuario, `dockermap`, que servirá para remapear en la máquina anfitriona. Además, para poder realizar el remapado, debemos añadir un `subuid` y un `subgid`, que estarán contenidos en los ficheros `/etc/subuid` y `/etc/subgid`, respectivamente, tal y como se muestra en la figura 60.

Para poder aplicar las configuraciones, debemos parar el servicio de Docker (por ejemplo, con `systemctl`) e iniciarlo nuevamente, indicando la opción de `-userns-remap`. Una posibilidad para iniciarlo nuevamente podría ser indicando que haga uso del remapeado con el siguiente comando: `sudo dockerd -userns-remap=default &`. [28]

Una vez aplicadas estas configuraciones, ya tendremos Docker corriendo con el remapeado activo, gracias al uso de los espacios de nombres. Para comprobar su efectividad, repetiremos la misma prueba de concepto que fue realizada en la sección 6.4.2, donde se montará un contenedor que tendrá acceso al directorio `/etc` de la máquina anfitriona. Desde una consola de comandos interactiva dentro del contenedor, se intentará modificar el fichero `/etc/hosts`, que solamente podría ser modificado por el superusuario de la máquina anfitriona, pero no por el usuario `root` de un contenedor. Una vez desplegado el ambiente de pruebas, recreando la misma configuración que en la sección de análisis, intentamos modificar el fichero `/etc/hosts` de la máquina anfitriona. No obstante, al

```
root@93f2f12210d1:~# echo "ataque    8.8.8.8" > /etchost/hosts
bash: /etchost/hosts: Permission denied
root@93f2f12210d1:~# id
uid=0(root) gid=0(root) groups=0(root)
root@93f2f12210d1:~#
```

Figura 61: Permiso denegado al intentar sobreescribir un fichero protegido

contrario de lo que ocurrió en la primera prueba, en esta ocasión el fichero no pudo ser modificado, a pesar de estar realizando la operación como un usuario `root` con UID=0 y GID=0 dentro del contenedor, lo que quiere decir que la remapeado del espacio de nombres se ha realizado correctamente y, por lo tanto, la securización ha sido un éxito, completando así la realización del RNF6. La realización de la prueba puede observarse en la figura 61.

Finalmente, indicar que esta configuración debe ser repetida en cada uno de los nodos que conformen el clúster, pues estamos trabajando con configuraciones propias de Docker y no directamente con Kubernetes.

7.4. Compromiso entre portabilidad y seguridad

Pudimos ver en la sección 3 como ambas tecnologías de virtualización, máquinas virtuales y contenedores, ofrecen diferentes ventajas y desventajas. Por un lado, las máquinas virtuales ofrecen un nivel de aislamiento más avanzado, por lo que se pueden llegar a considerar más seguras. Sin embargo, dicho aislamiento viene dado debido a la replicación de múltiples componentes que las virtualizaciones a nivel de sistema operativo no realizan, convirtiéndose así en una alternativa más ligera, flexible y portátil. En resumen, ambas tecnologías son capaces de proveer entornos aislados para la ejecución de aplicaciones bajo una máquina anfitriona compartida, pero desde perspectivas bien diferenciadas.

No obstante, no es necesario tratar ambas tecnologías como una confrontación directa, sino que, además de poder funcionar de manera independiente, también es posible trabajar con ellas en conjunto, en función de las necesidades propias del entorno. Por ejemplo, una buena solución de compromiso entre seguridad y portabilidad podría ser la ejecución de contenedores dentro de máquinas virtuales que, a su vez, corren sobre una máquina anfitriona física. Este enfoque aumenta la seguridad, al introducir dos capas de virtualización: las máquinas virtuales y los contenedores, además de poder ofrecer los beneficios de celeridad y flexibilidad otorgados por los contenedores. Una aproximación de esta propuesta se puede observar en la figura 62. A pesar de que este enfoque puede resultar interesante, debemos tener en cuenta que no es aplicable a todos los entornos de trabajo. Por ejemplo, aplicaciones de rendimiento crítico o que hagan uso de hardware especializado, en cuyo caso el aislamiento ofrecido por las máquinas virtuales supondrían un problema difícilmente resoluble. En este tipo de entornos, esta solución mixta no sería viable, por lo que la solución tendría que pasar por el uso de únicamente la virtualización a nivel de sistema operativo.

Puesto que en el estudio que nos ocupa no existe, a priori, ninguna limitación hardware ni de rendimiento crítico, esta fusión de tecnologías de virtualización formará parte de las

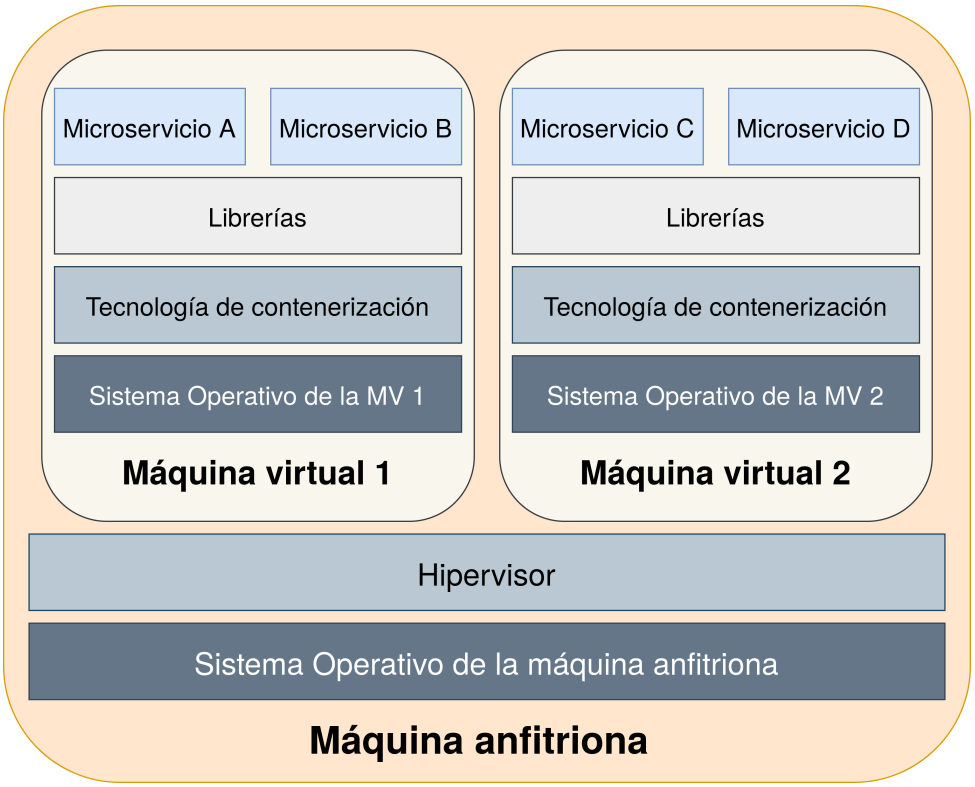


Figura 62: Propuesta de combinación de ambas tecnologías de virtualización

pruebas a realizar para la securización del sistema. El hecho de que Kubernetes sea capaz de gestionar contenedores corriendo en diferentes tipos de entorno (sobre una máquina física, sobre una máquina virtual, entornos basados en la nube, etc.) lo convierte en una herramienta de orquestación ideal para este tipo de aproximación.

De esta forma, la aproximación y problemas expuestos en la sección 6.5, en la que se comprobaban los problemas que puede afrontar un clúster Kubernetes con contenedores ejecutándose directamente sobre una máquina, sin la existencia de una capa de virtualización hardware corriendo por medio, puede ser securizada con la solución planteada en la figura 62. Realmente, este entorno de trabajo es el que ha sido empleado a lo largo de todo el proyecto, debido a las limitaciones hardware ya indicadas en la sección 4.1. Por tanto, a lo largo de toda la memoria existen numerosos ejemplos de esta securización, combinando las dos tecnologías de virtualización, para obtener un entorno más seguro, más ágil y más flexible, de forma simultánea.

Por ejemplo, la prueba realizada en la sección 6.3.1 ya es, por sí, una demostración de la securización que otorga este modelo combinado. A pesar de que en dicha prueba podemos observar como la inexistencia de limitaciones en el uso de RAM por parte de un contenedor puede llevar a comportamientos erráticos dentro de la máquina virtual que lo contiene; realmente, la máquina anfitriona física que contiene dicha máquina virtual y, a su vez, este contenedor problemático, no ha sufrido ningún tipo de perjuicio, puesto que la máquina virtual cuenta con un aislamiento mayor, dado por una virtualización hardware de la memoria RAM. De esta forma, tras la presentación de este modelo combinado,

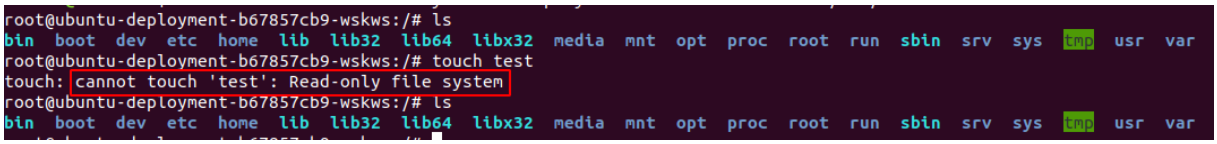


Figura 63: Comprobación del sistema de solo lectura

podemos dar completada la ejecución del RNF7, reflejado en la tabla 30.

7.5. Sistemas de solo lectura

Establecimos en el RNF8, reflejado en la tabla 31, la necesidad de ejecutar, en ciertas ocasiones, aplicaciones que no precisan de realizar escrituras en disco para su correcto funcionamiento. En este tipo de ocasiones, permitir que las aplicaciones contenerizadas realicen escrituras en disco puede suponer la aparición innecesaria de vulnerabilidades.

A lo largo de la sección 6.6, observamos la capacidad de realizar escrituras en disco de forma indiscriminada dentro de cualquier contenedor, en caso de que no se apliquen correcciones. Por lo tanto, en esta sección veremos como aplicar correcciones y obtener, de esta forma, sistemas de solo lectura, completando así la realización del RNF8.

Para poder crear contenedores con sistemas de solo lectura, haremos uso de las "*pod security policies*", al igual que en la sección 7.3. Concretamente, especificaremos la política `readOnlyRootFilesystem`, indicándola dentro del fichero de configuración YAML en el que se basará la creación de los contenedores. Por tanto, para la aplicación de esta política de seguridad, indicaremos las siguientes líneas dentro del fichero YAML:

Listing 8: Ejemplo de limitación de CPU en un fichero YAML

```
securityContext:
  readOnlyRootFilesystem: true
```

Así pues, una vez creado un contenedor con un sistema de solo lectura, podemos proceder a introducirnos en él e intentar crear un nuevo fichero, recreando la prueba realizada en la sección 6.6. En esta ocasión, podemos ver como no es posible crear el fichero, cumpliéndose así la securización perseguida, tal y como se puede observar en la figura 63.

7.6. Validación de imágenes mediante firma digital

A lo largo de la sección 6.7 reflexionamos acerca de la importancia de contar con un mecanismo de validación de imágenes mediante firmas digitales, para poder asegurar propiedades tan relevantes como la autenticación, la integridad y el no repudio. Igualmente, en dicha sección ya fue indicado el hecho de que, por defecto, Kubernetes no provee ningún tipo de mecanismo para la integración de un sistema de validación de firmas digitales. Así pues, en caso de querer sacar provecho de la herramienta "Docker Content Trust" y poder validar imágenes mediante firmas digitales, será necesario emplear software de terceros que consigan ampliar las funcionalidades de Kubernetes. En esta sección presentaremos

una solución alternativa, denominada Connaisseur[21], realizaremos una pequeña prueba de demostración y profundizaremos un poco más en el mecanismo existente en Docker para este tipo de securización, "Docker Content Trust".

7.6.1. Docker Content Trust

Para garantizar la validación de imágenes, Docker provee la herramienta "Docker Content Trust", que permite el uso de firmas digitales, consiguiendo verificar la integridad en el lado del cliente y el conocimiento del autor de *tags* específicos de una imagen. Por tanto, las firmas digitales son realizadas sobre los *tags*, siendo posible que existan *tags* firmadas de una misma imagen y otras que no lo estén. En caso de que el usuario tenga dicha funcionalidad activada, solamente podrá trabajar con imágenes firmadas, quedando inhabilitadas todas aquéllas que no lo estuvieran.

Para realizar este control, existen una serie de claves, que son creadas cuando se invoca por primera vez una operación de este sistema de integridad. Dicho conjunto de claves estará conformado por:

- Una clave *offline*, que es la raíz del contenido considerado fiable para un *tag* de una imagen.
- La clave de los *tags*.
- Las claves mantenidas por un servidor, como la clave del *timestamp*.

La clave *offline* es usada para crear las diferentes claves que se utilizarán con los *tags*. Esta clave *offline* pertenecerá a una persona u organización, y se encontrará en todo momento en el lado del cliente, por lo que debe ser almacenada en un lugar seguro y preferentemente con copias de seguridad, pues su recuperación resulta complicada y requiere de la intervención del soporte del equipo de Docker. Las claves de los *tags* estarán asociadas a una imagen de un repositorio, siendo los creadores de dicha clave los que podrán realizar las operaciones de *push* y *pull* sobre cualquier *tag* del repositorio. La clave asociada al *timestamp* también estará asociada con una imagen de un repositorio, pero en esta ocasión la clave será almacenada en el lado del servidor [21]. Este comportamiento está resumido en la imagen 64.

7.6.2. Connaisseur

Como ya fue explicado anteriormente, Kubernetes no posee un soporte nativo para "Docker Content Trust", lo que nos lleva a la utilización de este software de terceros para controlar e interceptar, en caso de ser preciso, la entrada de imágenes en nuestro clúster. De manera resumida, Connaisseur verifica las firmas existentes en todas las imágenes que intercepta. En caso de que la imagen interceptada no contenga una firma digital, dicha imagen será rechazada y no podrá ser desplegada en el clúster gestionado por Kubernetes. Para realizar esta verificación, Connaisseur buscará todas las referencias de imágenes que existan en el clúster y las recopilará en una lista. Posteriormente, buscará los archivos

[21] `https://github.com/sse-secure-systems/connaisseur`

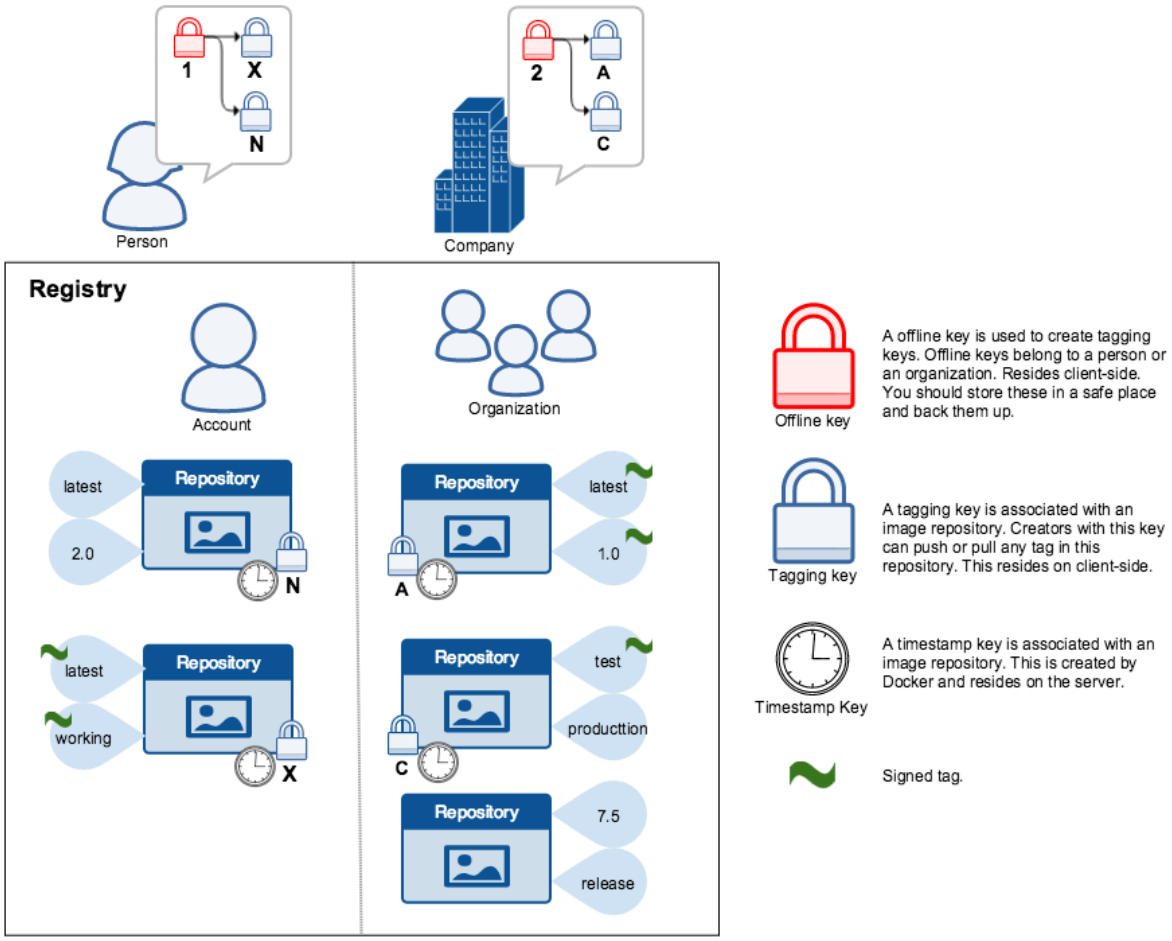


Figura 64: Relación entre las claves usadas por "Docker Content Trust"

Fuente: [21]

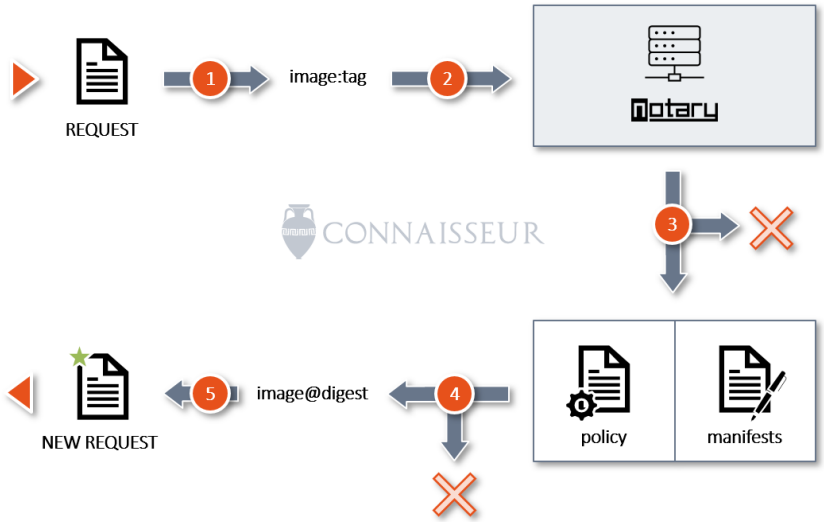


Figura 65: Resumen gráfico del funcionamiento de Connaisseur

Fuente: `https://github.com/sse-secure-systems/connaisseur`

manifest y validará tales firmas. En caso de que la firma se considere válida, la imagen (concretamente, el tag que fuese requerido) será aceptada y podrá pasar a formar parte del clúster. En caso contrario, si la firma no es válida o si el *tag* no está presente en los archivos consultados por Connaisseur, se denegará el recurso, impidiendo su despliegue en el clúster. Un resumen de este comportamiento puede observarse en la figura 65.

Ejemplo de uso :

Una vez realizada una explicación teórica del método de securización a aplicar, procederemos a ejecutar una pequeña prueba de uso, configurando "Docker Content Trust" e instalando Connaisseur en el clúster.

El primer paso consistirá en crear un *tag* de una imagen no firmado, de tal forma que una vez sea configurada la validación de imágenes mediante firma digital, debería ser rechazada. Para ello, crearemos una imagen personalizada que tomará como base una imagen Debian[22]. En primer lugar, bajamos desde el Docker Hub la última imagen disponible para Debian (`debian:latest`), realizando para ello un *pull*, e iniciamos un contenedor con una consola de comandos interactiva, en el que basaremos la imagen personalizada (figura 66). Posteriormente, realizaremos un `commit` sobre el contenedor en funcionamiento, para crear la imagen personalizada, indicando para ello el nombre del repositorio personalizado, que deberemos crear en el Docker Hub, así como el *tag* "*unsigned*", para indicar que este será el *tag* que no irá firmado (figura 67). Una vez en posesión de la imagen personalizada con la información necesaria, realizamos un `push` hacia el Docker Hub, sin ningún tipo de indicación para realizar firmas digitales, como hasta el momento (figura 68).

[22] `https://hub.docker.com/_/debian`

```
root@master:/home/manuel# docker run --name DebianFirma --rm -i -t debian:latest bash
Unable to find image 'debian:latest' locally
latest: Pulling from library/debian
d960726af2be: Pull complete
Digest: sha256:acf7795dc91df17e10effee064bd229580a9c34213b4dba578d64768af5d8c51
Status: Downloaded newer image for debian:latest
```

Figura 66: Descarga de imagen Debian

```
root@master:~# docker commit ea40ea9ca6b3 manuelsimon/pruebafirmas:unsigned
sha256:392522f9eff69a777c4cb5ab96002a1dd8ce9129d8de1c0cececf0e30b9c7fc8
```

Figura 67: Operación `commit` sobre el contenedor en funcionamiento

Subida la imagen sin firma, procederemos a configurar el servicio "Docker Content Trust" en el nodo máster del clúster. Para su configuración, bastará con exportar las siguientes variables:

Listing 9: Activación de "Docker Content Trust"

```
export DOCKER_CONTENT_TRUST=1
export DOCKER_CONTENT_TRUST_SERVER=https://notary.docker.io
```

Hecho esto, repetimos las operaciones de `commit` y `push`, pero esta vez creando un nuevo *tag* que denominaremos *signed* y sobre el cual aplicaremos una firma digital. Puesto que es la primera vez que realizamos una operación `push` con "Docker Content Trust" configurado, se nos solicitará una serie de contraseñas para establecer las claves detalladas previamente. Este comportamiento puede observarse en la figura 69.

Una vez establecida nuestra firma *offline*, recuperaremos la parte pública de la firma, puesto que será usada por Connaisseur para realizar su posterior verificación. Este proceso está recogido en la figura 70.

El siguiente paso es realizar la descarga de Connaisseur y todas sus dependencias así como proceder a su instalación. Antes de realizar la instalación, debemos indicar en el fichero de configuración `helm/values.yaml` las variables `notary auth user`, `notary auth password` y `notary rootPubKey`, correspondiéndose con las credenciales del Docker Hub y la clave pública recuperada en el paso previo. Modificados estos parámetros, lanzamos el proceso de instalación con el comando `make install` [20], tal y como se muestra en la figura 71.

Listing 10: Instalación de dependencias

```
git clone https://github.com/sse-secure-systems/connaisseur.git
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/master/
    ↪ scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```

```
root@master:~# docker push manuelsimon/pruebafirmas:unsigned
The push refers to repository [docker.io/manuelsimon/pruebafirmas]
688e187d6c79: Mounted from library/debian
unsigned: digest: sha256:56261b7b355e65df3e105bd10eded390d8a7870e07475946c6d5e5ed83e02f3f size: 529
```

Figura 68: Subida al repositorio personal del *tag* no firmado

```
root@master:~# export DOCKER_CONTENT_TRUST=1
root@master:~# export DOCKER_CONTENT_TRUST_SERVER=https://notary.docker.io
root@master:~# docker commit ea40ea9ca6b3 manuelsimon/pruebafirmas:signed
sha256:4e1416c47322f9fac0aa6a0783947b583e8daeb5ba03bc8a1baba7230a8efda1
root@master:~# docker push manuelsimon/pruebafirmas:signed
The push refers to repository [docker.io/manuelsimon/pruebafirmas]
688e187d6c79: Layer already exists
signed: digest: sha256:73b25a81f59939c68c246b264e0636dff42d2c905e0622830f59c53d4cac05e8 size: 529
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with ID 643623f:
Repeat passphrase for new root key with ID 643623f:
Enter passphrase for new repository key with ID 21dea31:
Repeat passphrase for new repository key with ID 21dea31:
Finished initializing "docker.io/manuelsimon/pruebafirmas"
Successfully signed docker.io/manuelsimon/pruebafirmas:signed
```

Figura 69: Configuración y primer uso de "Docker Content Trust"

```
root@master:~# cd ~/.docker/trust/private
root@master:~/.docker/trust/private# sed '/^role:\sroot$/d' $(grep -iRl "role: root" .) > root-priv.key
root@master:~/.docker/trust/private# openssl ec -in root-priv.key -pubout -out root-pub.pem
read EC key
Enter PEM pass phrase:
writing EC key
```

Figura 70: Recuperación de la parte pública de la firma creada

```
snap install yq
apt-get install jq
```

Finalmente, una vez Connaisseur complete el proceso de instalación, comprobamos su eficacia intentando trabajar con una imagen no firmada, correspondiente al primer *tag* realizado, que es rechazada (figura 72) y finalmente con una imagen firmada, correspondiente con el segundo *tag*, que sí es aceptada en el clúster (figura 73).

De esta forma, gracias al uso de un software externo, Connaisseur, conseguimos realizar una validación de las imágenes que se ejecutarán en el clúster mediante un mecanismo de firma digital, quedando así abordados todos los objetivos definidos en el RNF9.

7.7. Herencia de vulnerabilidades entre imágenes

En este caso, siendo conscientes de la posible herencia de vulnerabilidades desde una imagen padre a una imagen hijo, tal y como estudiamos a lo largo de la sección 6.8, el

```
root@master:~/connaisseur# make install
bash helm/certs/gen_certs.sh
Generating RSA private key, 4096 bit long modulus (2 primes)
.....................................................................++++
.....................................++++
e is 65537 (0x010001)
Signature ok
subject=CN = connaisseur-svc.connaisseur.svc
Getting Private key
kubectl create ns connaisseur || true
Error from server (AlreadyExists): namespaces "connaisseur" already exists
kubectl config set-context --current --namespace connaisseur
Context "kubernetes-admin@kubernetes" modified.
#
#############################################
#
# The installation may last up to 5 minutes.
#
#############################################
#
```

Figura 71: Proceso de instalación de Connaisseur

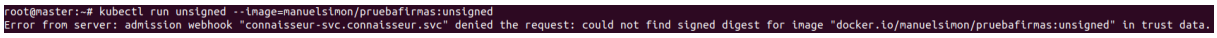


Figura 72: Connaisseur rechaza imagen sin firma digital

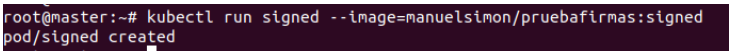


Figura 73: Connaisseur acepta imagen con firma digital

proceso de securización no difiere de la localización de vulnerabilidades en una imagen cualquiera. Para ello, ya establecimos un procedimiento basado en el análisis estático sobre imágenes de contenedores en la sección 7.1.

Por tanto, en este caso, el proceso de securización se basaría en la realización de un análisis estático para detectar vulnerabilidades, preferiblemente sobre la imagen padre. De esta forma, una vez creada la nueva imagen dependiente, podremos proceder a corregir dichas vulnerabilidades, por ejemplo, mediante la actualización del software implicado. Normalmente, dicha corrección será aplicada sobre la imagen hijo, ya que es posible que la imagen de la que toma su origen pueda ser una imagen externa que no podamos modificar directamente. Aplicado este procedimiento, damos por concluida la realización del RNF10, reflejado en la tabla 33.

7.8. Diferenciación de espacios

En el RNF11, representado en la tabla 34, establecimos la posibilidad de crear diferentes espacios de trabajo para las distintas aplicaciones y/o equipos de trabajo, de tal forma que se creara una nueva capa de separación para evitar posibles conflictos. Por ejemplo, con la existencia de diferentes capas según los entornos, se podrán evitar potenciales confusiones como que un miembro del equipo de desarrollo realice un despliegue inestable directamente en producción. Sin embargo, vimos en la sección 6.9 como, por defecto, todos los despliegues realizados se ejecutan automáticamente en el espacio de nombres por defecto (*default*). A lo largo de esta sección veremos como crear nuevos espacios de nombres en un clúster Kubernetes.

Podemos comprobar como, en un principio, solamente existen los espacios creados por defecto, ya explicados anteriormente en la sección 6.9, tal y como se observa en la figura 74.

Posteriormente, para realizar la creación de nuevos espacios de nombre, podemos hacer uso de ficheros JSON, como los que podemos encontrar en el anexo A.7. Haciendo uso de estos ficheros, podemos crear dos espacios de nombre de ejemplo, como se puede observar en la figura 75.

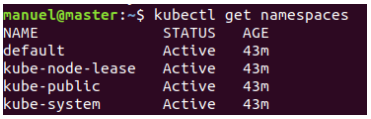


Figura 74: Espacios de nombre creados por defecto en un clúster Kubernetes

```
manuel@master:~$ kubectl create -f crear-namespace2.json
error: the path "crear-namespace2.json" does not exist
manuel@master:~$ nano crear-namespace2.json
manuel@master:~$ kubectl create -f crear-namespace2.json
namespace/espacio2 created
manuel@master:~$ kubectl get namespaces --show-labels
NAME              STATUS   AGE   LABELS
default           Active   57m   kubernetes.io/metadata.name=default
espacio1          Active   85s   kubernetes.io/metadata.name=espacio1,name=espacio1
espacio2          Active   12s   kubernetes.io/metadata.name=espacio2,name=espacio2
kube-node-lease   Active   57m   kubernetes.io/metadata.name=kube-node-lease
kube-public       Active   57m   kubernetes.io/metadata.name=kube-public
kube-system       Active   57m   kubernetes.io/metadata.name=kube-system
```

Figura 75: Creación y comprobación de la existencia de nuevos espacios

```
manuel@master:~$ kubectl config set-context dev --namespace=espacio1 --cluster=kubernetes --user=kubernetes-admin
Context "dev" created.
manuel@master:~$ kubectl config set-context prod --namespace=espacio2 --cluster=kubernetes --user=kubernetes-admin
Context "prod" created.
```

Figura 76: Asociación de los nuevos espacios de nombre a un contexto

Una vez creados los nuevos espacios de nombres, existe la posibilidad de asociarlos a un determinado contexto. Por ejemplo, imaginemos que queremos usar uno de los nuevos espacios para aquellos despliegues que vaya a usar un hipotético equipo de desarrollo, mientras que queremos usar el segundo espacio, separadamente, para el equipo que se encarga de hacer los despliegues finales en producción. En este caso, podemos asociar los espacios de nombres a contextos que hagan referencia a su utilidad, como se muestra en la figura 76.

Finalmente, ya creados y asociados los nuevos espacios de nombres, ejemplificaremos su uso. Podemos cambiar de espacio de nombre, por ejemplo, al primero de los espacios creados, que asociamos a un ambiente de desarrollo, tal y como vemos en la figura 77. Allí, podemos lanzar un *deployment* y comprobar como, efectivamente, solamente existen los *pods* asociados a dicho *deployment*. Posteriormente, cambiando al espacio de nombres asociado con el ambiente de producción, podemos observar como allí los *pods* recién creados no son accesibles, ni tan siquiera visibles, tal y como podemos observar en la figura 78.

Hecho esto, podemos observar como es posible crear, gracias al uso de *namespaces*, diferentes espacios para las distintas aplicaciones y/o equipos que pueden estar haciendo uso de un clúster, evitando posibles conflictos entre ellos. De esta forma, se da por concluida la realización del RNF11.

7.9. Configuración y uso del componente *Secret*

A lo largo de esta sección, realizaremos una prueba de configuración y uso del componente *secret*, siguiendo las indicaciones del RNF13, reflejado en la tabla 36. Esta securización no posee una fase de análisis asociada, como el resto de securizaciones realizadas hasta el momento. Esto es así, puesto que ya mencionamos en la sección 3.4.1, donde se realiza una explicación teórica de los principales componentes de Kubernetes, que se trata

```
manuel@master:~$ kubectl config use-context dev
Switched to context "dev".
manuel@master:~$ kubectl config current-context
dev
```

Figura 77: Cambio al contexto de desarrollo, asociado al "espacio1"

```
manuel@master:~$ kubectl get deployment
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
ubuntu-deployment   3/4     4            3           50s
manuel@master:~$ kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
ubuntu-deployment-688c7667d9-cflf8    1/1     Running   0          64s
ubuntu-deployment-688c7667d9-j4bs6    1/1     Running   0          64s
ubuntu-deployment-688c7667d9-rqtxh    0/1     Pending   0          64s
ubuntu-deployment-688c7667d9-wrfxz    1/1     Running   0          64s
manuel@master:~$ kubectl config use-context prod
Switched to context "prod".
manuel@master:~$ kubectl get pods
No resources found in espacio2 namespace.
```

Figura 78: Demostración del aislamiento entre *namespaces* dentro del mismo clúster

```
manuel@master:~$ echo -n 'user' | base64
dXNlcg==
manuel@master:~$ echo -n 'pass' | base64
cGFzcw==
```

Figura 79: Obtención en base-64 de la información privada

de un componente que no es posible emplear por defecto, sino que es preciso realizar una configuración previa. [22]

Para poder almacenar las informaciones consideradas secretas de manera cifrada, el primer paso será obtener la representación en base-64 de dicha información. Por ejemplo, imaginemos que queremos almacenar un nombre de usuario y contraseña para una aplicación contenerizada, información que considerada secreta. En primer lugar debemos obtener la representación en base-64 de dichos datos, tal y como puede observarse en la figura 79.

Siendo conocedores de la representación en base-64 de la información a privatizar, podemos comenzar ya con la creación de un *secret*, que puede ser realizado bien desde un fichero YAML, bien desde un fichero de texto. En dicho fichero, estableceremos los pares de nombre y contenido del *secret*, siendo este contenido la representación en base-64 obtenida previamente. Por mantener la coherencia con el trabajo realizado hasta el momento, haremos uso de un fichero YAML como el que se puede observar a continuación:

Listing 11: Ejemplo de creación de *secret*

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-manuel
type: Opaque
data:
  USER_NAME: dXNlcg==
  PASSWORD: cGFzcw==
```

Creado el fichero de configuración YAML, podemos aplicarlo sobre el clúster para su almacenamiento, tal y como es posible observar en la figura 80.

Una vez dicho *secret* fue almacenado en el clúster, es posible hacer referencia a él desde un nuevo fichero de configuración YAML que de lugar a un nuevo *pod* o *deployment*, en cuyos contenedores existirá la referencia a este valor cifrado. Debemos tener en cuenta que, aunque en el clúster la información se almacena de manera cifrada, dentro de los con-

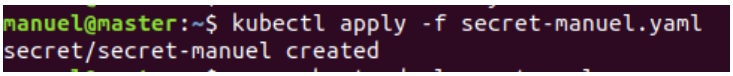


Figura 80: Almacenamiento del *secret* en el clúster Kubernetes

tenedores existirá un acceso directo a la información, sin necesidad de ejecutar cualquier proceso de descifrado previo o similar. Por ejemplo, en caso de querer usar los dos valores indicados en el *secret* creado previamente, podemos hacerlo indicando las siguientes líneas dentro del fichero YAML de configuración del despliegue en el que deseemos poseer tal acceso[23]:

Listing 12: Asociación de *secret* en un fichero YAML para realizar un despliegue

```
env:
- name: envuser
  valueFrom:
    secretKeyRef:
      name: secret-manuel
      key: USER_NAME
- name: envpass
  valueFrom:
    secretKeyRef:
      name: secret-manuel
      key: PASSWORD
```

Como se puede observar, en el fichero YAML de configuración, que es un fichero que tal vez queramos publicar o compartir públicamente, en ningún momento se escribe la información privada (en este caso, el usuario y la contraseña de la aplicación contenerizada), sino que se hace una referencia indirecta gracias al uso del componente *secret*. Para indicar cuál es el valor que se desea obtener, indicaremos:

- `env.name:` el nombre de la variable que existirá dentro del contenedor.
- `env.valueFrom.secretKeyRef.name:` el nombre del *secret* almacenado en Kubernetes. Dentro de un *secret* pueden estar almacenados varios valores, como en el caso que se nos presenta.
- `env.valueFrom.secretKeyRef.key:` la clave dada en el fichero de configuración del *secret* para la variable que deseemos recuperar.

Una vez se haya realizado el despliegue basado en el fichero YAML con el uso del componente *secret* configurado en su interior, comprobar su uso es tan sencillo como el acceso a una variable de entorno cualquiera. Este comportamiento puede observarse en la figura 81. Una vez comprobado su correcto funcionamiento, podemos dar por finalizada la ejecución del RNF13.

[23]El acceso a la información almacenada en los *secrets* puede ser consultada desde el interior de un contenedor bien desde una variable de entorno, bien desde un volumen que se añade al despliegue. En el ejemplo presentado, se ha optado por la posibilidad de las variables de entorno.

```
manuel@master:~$ kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
ubuntu-deployment-799449f7c5-92swg    1/1     Running   0          74s
ubuntu-deployment-799449f7c5-b7rm7    1/1     Running   0          74s
manuel@master:~$ kubectl exec --stdin --tty ubuntu-deployment-799449f7c5-92swg -- /bin/bash
root@ubuntu-deployment-799449f7c5-92swg:/# echo $envuser
user
root@ubuntu-deployment-799449f7c5-92swg:/# echo $envpass
pass
```

Figura 81: Comprobación del correcto uso de un *secret* desde dentro de un contenedor

```
manuel@master:~$ sudo docker run -i kubesec/kubesec:v2 scan /dev/stdin < ubuntu-deployment.yaml
[
  {
    "object": "Deployment/ubuntu-deployment.default",
    "valid": true,
    "message": "Passed with a score of 7 points",
    "score": 7,
    "scoring": {
      "advise": [
        {
          "selector": ".metadata .annotations .\"container.seccomp.security.alpha.kubernetes.io/pod\"",
          "reason": "Seccomp profiles set minimum privilege and secure against unknown threats",
          "points": 1
        },
        {
          "selector": "containers[] .securityContext .capabilities .drop",
          "reason": "Reducing kernel capabilities available to a container limits its attack surface",
          "points": 1
        },
        {
          "selector": "containers[] .securityContext .capabilities .drop | index(\"ALL\")",
          "reason": "Drop all capabilities and add only those required to reduce syscall attack surface",
          "points": 1
        },
        {
          "selector": ".metadata .annotations .\"container.apparmor.security.beta.kubernetes.io/nginx\"",
          "reason": "Well defined AppArmor policies may provide greater protection from unknown threats. WARNING: NOT PRODUCTION READY",
          "points": 3
        },
        {
          "selector": ".spec .serviceAccountName",
          "reason": "Service accounts restrict Kubernetes API access and should be configured with least privilege",
          "points": 3
        }
      ]
    }
  }
]
```

Figura 82: Análisis estático sobre fichero YAML mejorado

7.10. Análisis estático sobre ficheros de configuración YAML

Después de todas las modificaciones realizadas en los ficheros YAML de configuración para mejorar diversos aspectos de la seguridad, ejecutadas a lo largo de las diferentes pruebas de securización realizadas en esta sección, tendremos, llegados a este punto, ficheros de configuración más seguros que el empleado en la prueba realizada en la sección 6.2. Por tanto, después de las correcciones realizadas en los ficheros YAML de configuración, podemos aplicar nuevamente un análisis estático sobre un fichero que incorpore todas las securizaciones realizadas a lo largo de esta sección.

Nuevamente, haremos uso de la herramienta Kubesec, pero esta vez sobre el fichero de configuración mejorado, localizado en el anexo A.8. En esta ocasión, a pesar de que la herramienta de análisis aún muestra ciertos aspectos a mejorar, el nivel de seguridad ofrecido por este fichero YAML es mucho mayor que los ficheros empleados en la sección de análisis de este proyecto. Tal y como refleja la imagen 82, conseguimos pasar de una puntuación de cero puntos hasta un total de 7 puntos.

Hecho esto, damos por concluida la realización del RNF2, recogido en la tabla 25. Igualmente, los aspectos todavía recogidos por la herramienta de análisis estático pueden suponer un índice interesante para un posible trabajo futuro con el que conseguir todavía una mayor securización del clúster.

7.11. *Service Mesh*

7.11.1. Introducción teórica a *Service Mesh*

Service Mesh se trata de una solución especialmente diseñada para la gestión de las comunicaciones entre los diferentes microservicios desplegados en un clúster.

Debemos tener en cuenta que cuando pasamos de una infraestructura monolítica cara una infraestructura basada en microservicios, aunque estemos incorporando una gran cantidad de ventajas, también estaremos introduciendo una nueva serie de problemas o retos técnicos nuevos, que precisarán de una solución eficiente. Por ejemplo, algunos de los problemas a los que se debe enfrentar cualquier infraestructura Kubernetes basada en microservicios, y que no hemos solventado hasta ahora, son los siguientes:

- Es preciso que cada microservicio cuente con su propia **lógica de negocio** (*business logic*), toda aquella algoritmia precisa para poder intercambiar información. Por ejemplo, para poder intercambiar la información entre la interfaz de usuario y una base de datos. Al subdividir una aplicación en numerosos microservicios, todos estos microservicios deben contar con una lógica de negocio para poder intercambiar la información entre ellos y que la aplicación funcione correctamente. Esto supone una carga de trabajo mayor para los desarrolladores e incluso un atraso con respecto a las aplicaciones monolíticas, donde había un menor número de puntos donde era preciso configurar esta lógica.

- Íntimamente relacionada con la lógica de negocio, también existe la necesidad de configurar para cada microservicio un ***endpoint***, una interfaz de comunicación estandarizada para que los microservicios puedan intercambiar la información necesaria entre sí y, de esta forma, hacer funcionar la aplicación que conforman. Esto quiere decir que cada vez que incorporemos un nuevo microservicio al clúster, será preciso que tenga configurada en su interior una lógica de negocio y una configuración de comunicación con el resto de microservicios.

- En lo referente a la seguridad, aspecto en el que se centra este estudio, a pesar de que hemos incorporado numerosos mecanismos para incrementar la seguridad dentro del clúster, todavía falta por **securizar** un punto esencial, **la comunicación existente entre todos los *pods*** que forman parte del clúster. Por defecto, las comunicaciones existentes dentro de un clúster Kubernetes se realizan de manera insegura, esto es, usando protocolos no cifrados como por ejemplo HTTP. Además, como comprobamos en la sección 6.10, en un clúster Kubernetes existe la posibilidad de que cada servicio dentro del mismo clúster pueda comunicarse libremente con cualquier otro servicio. Estas dos características existentes en el funcionamiento por defecto de Kubernetes hacen que, desde el punto de vista de la seguridad, si un atacante consigue acceso a un servicio dentro del clúster, éste tendría la posibilidad de moverse libremente o escalar entre los diferentes servicios corriendo dentro del clúster, al no existir una conexión segura. Por ejemplo, podría leer los paquetes de red que se están transmitiendo entre los diferentes *pods* y conseguir acceder a información confidencial que es completamente legible, al no viajar cifrada entre los diferentes microservicios.

- En muchas ocasiones, los desarrolladores querrán comprobar si la comunicación entre los diferentes microservicios está siendo fluida, implementando para ello una **lógica de *retry***, consiguiendo así una aplicación más robusta. Por ejemplo, en esta lógica se comprobará si un microservicio no puede ser alcanzado, o tarda mucho tiempo en dar respuesta, en cuyo caso la aplicación intentará generar nuevas peticiones. Una vez más, la necesidad de implementar esta lógica en cada microservicio, supone un desarrollo más lento y menos centrado en la propia lógica que debe incorporar el microservicio.
- **Obtención de métricas y trazabilidad**: cuando poseemos un entorno basado en numerosos servidores y múltiples aplicaciones, normalmente querremos monitorizar el funcionamiento de dicho entorno, para localizar posibles errores, que pueden tener orígenes muy dispares. Para ello, contamos con la ayuda de herramientas como Zabbix[24] o Grafana[25].

En resumen, todos estos requisitos, que son reclamados habitualmente en cualquier entorno conformado por múltiples aplicaciones y máquinas, deberán ser atendidos y solucionados por el equipo de desarrollo de cada microservicio, suponiendo una mayor carga de trabajo que impide avanzar en el verdadero desarrollo del producto final. Además, al ser necesario introducir toda esta lógica en cada microservicio, esto incrementa su complejidad, existiendo una clara contradicción con los principios seguidos por los entornos basados en microservicios, donde lo que se pretende es que cada uno de ellos sea lo más sencillo y simple posible, para poder separar adecuadamente la lógica de la aplicación. Por tanto, para poder poner solución a todos estas necesidades comunes, de las que Kubernetes por defecto no se encarga, nacen los *Service Mesh*.

El enfoque realizado por los *Service Mesh* consiste en separar toda esta lógica de los microservicios e incorporarla en las denominadas aplicaciones sidecar, comportándose como un *proxy*, como una aplicación intermedia para gestionar las comunicaciones y su lógica asociada, consiguiendo así que los desarrolladores se tengan que despreocupar de ella. Además, el despliegue de dichas aplicaciones sidecar está pensado para que se pueda realizar de forma automática, de forma que cuando un nuevo microservicio es desplegado, éste incorporará, dentro del mismo *pod*, un nuevo contenedor que funcionará como *proxy*, y que será incorporado automáticamente, sin que los desarrolladores se tengan que preocupar por ello. Una vez el *Service Mesh* esté correctamente configurado, las comunicaciones entre los microservicios se realizarán a través de estas aplicaciones sidecar [18]. Un resumen visual sobre los conceptos explicados acerca *Service Mesh* hasta el momento, puede observarse en la figura 83.

7.11.2. Introducción teórica a Istio

Service Mesh se trata de un patrón o paradigma, pero para poder implementarlo debemos escoger alguna de las implementaciones existentes, como pueden ser Istio[26], Linkerd[27]

[24] `https://www.zabbix.com/`
[25] `https://grafana.com/`
[26] `https://istio.io/`
[27] `https://linkerd.io/`

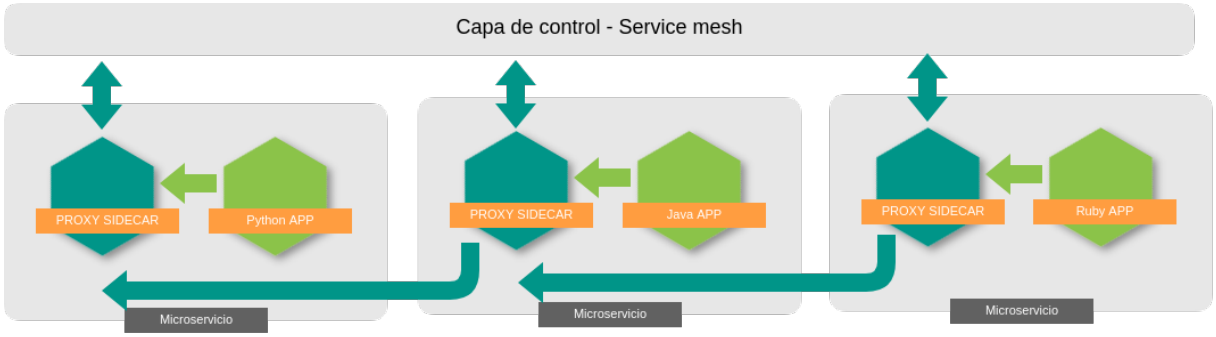


Figura 83: Resumen visual de *Service Mesh*

Fuente: [18]

o Consul[28]. Para el despliegue de un servicio *Service Mesh*, hemos optado por Istio, al tratarse de una de las soluciones más completas y populares en el momento de la realización de este estudio. Algunas de las características más destacables de Istio son:

- **Servicio de descubrimiento dinámico:** además de desplegar los *proxies* en cada microservicio, Istio cuenta con un servicio centralizado de descubrimiento dinámico, de tal forma que no es preciso configurar manualmente los *endpoints* de cada microservicio, sino que estos son descubiertos y registrado de forma totalmente automática por Istio.
- **Gestión de certificados:** para poder establecer las comunicaciones seguras entre los diferentes microservicios, Istio también se comporta como una autoridad de certificación (AC o CA por sus siglas en inglés), encargándose de generar los certificados para cada uno de los microservicios corriendo en el clúster, asegurando una comunicación segura con TLS.
- **Obtención de métricas:** Istio se encarga de obtener diferentes datos y métricas desde los *endpoints* para poder monitorizarlos mediante cualquier servicio externo que lo permita. De esta forma, no será preciso desplegar y configurar un servicio de monitorización para cada recurso del clúster, sino que dicha configuración será realizada automáticamente por Istio.
- **Istio *Ingress Gateway*:** se trata de un componente que funciona como un punto de entrada para el clúster, siendo el encargado de recibir cualquier tráfico de entrada a dicho clúster y redirigirlo al servicio virtual adecuado, consiguiendo así llegar al microservicio correcto.

Podemos observar un resumen visual del funcionamiento de la implementación de *Service Mesh* realizada por Istio en la figura 84; así como una representación más detallada de su funcionamiento en la figura 85.

[28] `https://www.consul.io/`

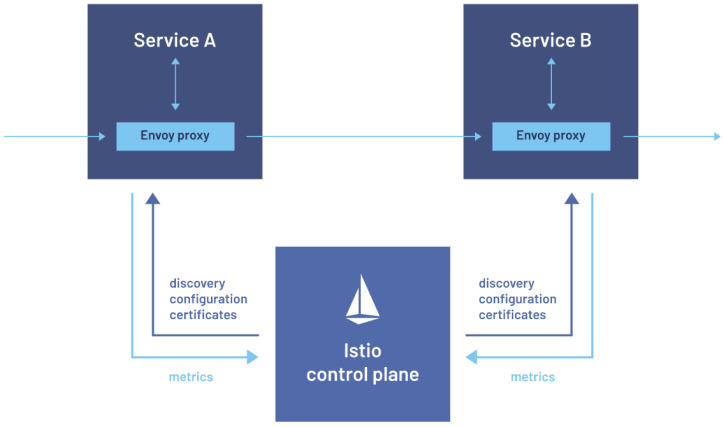


Figura 84: Implementación de *Service Mesh* de Istio

Fuente: [39]

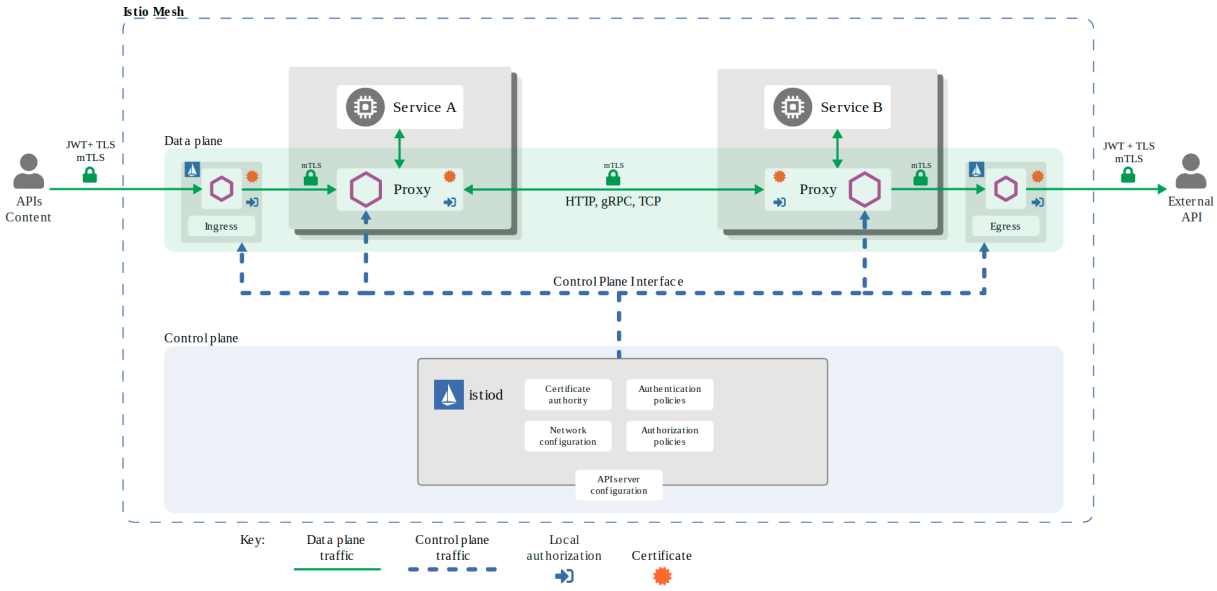


Figura 85: Representación gráfica del funcionamiento de Istio

Fuente: [37]

7.11.3. Instalación y configuración del *Service Mesh* Istio en el clúster

Realizada una explicación teórica y comprendidos los problemas que pueden ser abordados desplegando una implementación de *Service Mesh* como puede ser Istio, procederemos a realizar su instalación y puesta en funcionamiento en el clúster de pruebas. Puesto que el clúster existente, cuyas especificaciones fueron detalladas en la sección 5.1, no cumplía con los requisitos mínimos de memoria para la instalación de Istio[29], el clúster de pruebas sufrió las siguientes modificaciones:

- Eliminación de la máquina "Worker3", eliminando así uno de los nodos *worker* que componían el clúster.
- Modificación del hardware virtualizado con el que contaba la máquina virtual "Worker1", que pasó a tener las siguientes especificaciones, con el fin de alcanzar los requisitos mínimos dictados por Istio:
  - **CPU:** Intel Core i7-9750H CPU @ 2.60GHz×6
  - **RAM:** 9GB

De esta forma, el clúster pasó de tener 3 nodos *worker* con una capacidad conjunta de CPU de 3 procesadores y memoria de 6GB, a un clúster de 2 nodos *worker* con una capacidad conjunta de 7 procesadores y 10GB de memoria RAM.

Una vez realizada la modificación de las especificaciones del clúster, los pasos seguidos para realizar la instalación de Istio fueron los siguientes:

1. **Descarga** en el nodo máster de la última versión disponible de Istio (Istio 1.10.0, en el momento de la realización de este estudio) desde el GitHub oficial[30] y descompresión del fichero descargado (`istio-1.10.0-linux-amd64.tar.gz`).

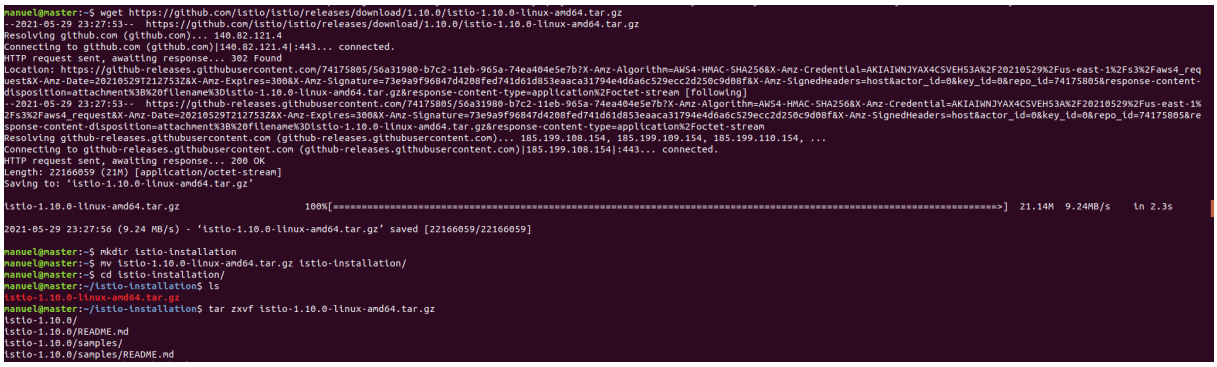


Figura 86: Descarga y descompresión de la última versión de Istio

2. **Incorporación al** `PATH` del directorio `bin` de Istio, para poder contar con el comando `istioctl` a través del cual realizaremos la instalación.

[29] `https://istio.io/latest/docs/ops/deployment/performance-and-scalability/`
[30] `https://github.com/istio/istio/releases/`

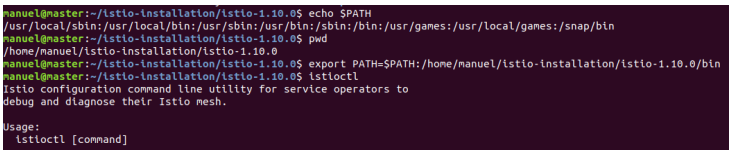


Figura 87: Incorporación al `PATH` de `istioctl`

3. **Instalación** de Istio gracias al comando `istioctl install`. Después de algunos minutos de espera, obtenemos por pantalla un aviso de la correcta finalización de la instalación de Istio. Igualmente, podemos comprobar los *pods* existentes corriendo sobre el clúster para confirmar que, efectivamente, los servicios de Istio, el *Control Plane* `istiod` y el *Ingress Gateway* ya están corriendo sobre el clúster.

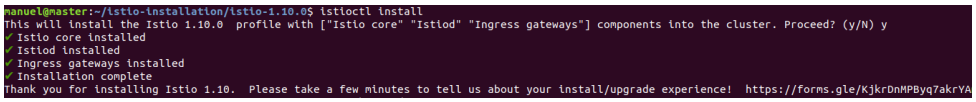


Figura 88: Proceso de instalación de Istio

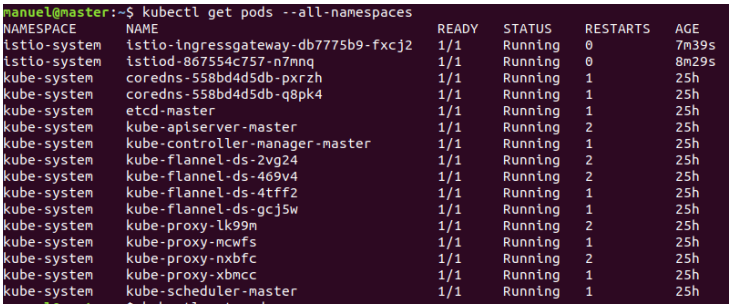


Figura 89: Comprobación de que Istio ya está corriendo en el clúster

4. Una vez Istio esté instalado y corriendo sobre el clúster, podemos realizar un despliegue para comprobar si los nuevos microservicios ya funcionan con su correspondiente sidecar:

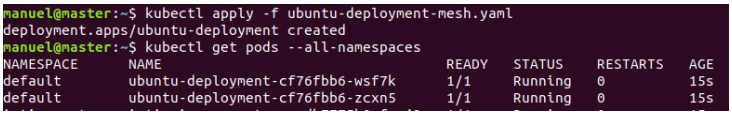


Figura 90: Comprobación de despligue sin sidecar

Al contrario de lo que se podría pensar, la funcionalidad de autoinyectado del *proxy* no se realiza de forma totalmente automática por defecto, sino que es preciso realizar una pequeña configuración. Para poder **incluir la funcionalidad del autoinyectado de *proxies* de forma automática**, debemos modificar la etiqueta (*label*) del espacio de nombres en el que queramos autoinyectar los *proxies*, incluyendo el valor `istio-injection=enabled`. Hecho esto, podemos comprobar como los siguientes

despliegues ya incluyen un sidecar acoplado para la gestión de las comunicaciones, como ya fue explicado previamente.

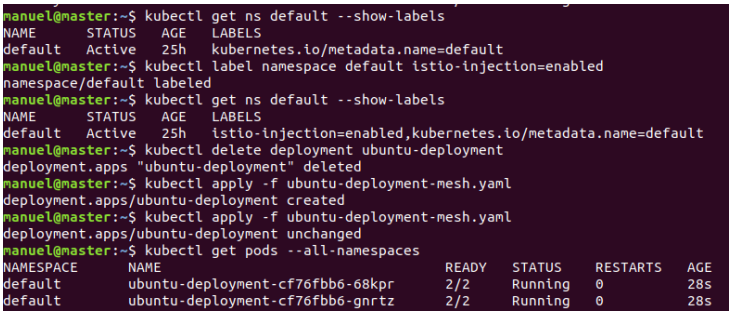


Figura 91: Configuración del autoinyectado automático de *proxies*

Finalizados estos pasos, ya tendremos Istio correctamente instalado y configurado para los posteriores despliegues a ejecutar en el clúster. Hecho esto, nuestro clúster ha mejorado significativamente su seguridad en lo referente a la interconexión de los diferentes microservicios. Por ejemplo, a partir de este momento, todas las comunicaciones entre los microservicios se realizarán de manera cifrada gracias al uso de TLS y certificados, gestionados por Istio. Además, puesto que las comunicaciones entre los diferentes microservicios se realizarán, a partir de ahora, mediante el uso de los *proxies*, ya no existirá una comunicación directa entre los diferentes contenedores. De esta forma, gracias a la instalación y configuración en el clúster del *Service Mesh* Istio, podemos dar por solucionados los problemas que se planteaban en los requisitos RNF12 y RNF14, reflejados en las tablas 35 y 37, respectivamente.

7.11.4. Funcionalidades "extendidas" de Istio

Vimos en la sección 7.11.1 como una de las problemáticas que se pretenden solventar con el uso de *Service Mesh* es la obtención de métricas y trazabilidad. Igualmente, en la sección 7.11.2, vimos como Istio implementa esta obtención de métricas y datos desde los *endpoints*, para facilitar una posible monitorización del clúster en el que reside. Por lo tanto, en esta sección, veremos, de forma breve, como es posible observar estos datos que están siendo recogidos por Istio. Istio incluye una serie de integraciones[31] que podemos emplear para este fin. Por ejemplo, incluye integraciones con software de monitorización como Prometheus[32], Grafana[33] o Kiali[34]. Su instalación y configuración está pensada para ser ejecutada desde ficheros YAML incluidos en la descarga que realizamos previamente de Istio.

Para realizar la instalación de estas integraciones, basta con realizar un comando `kubectl apply -f` sobre la carpeta raíz que incluye todos estos softwares, tal y como se puede observar en la figura 92. Una vez desplegados, podemos comprobar su dirección de acceso con el comando `kubectl get svc -n istio-system`, tal y como se observa en la

[31] `https://istio.io/latest/docs/ops/integrations/`
[32] `https://prometheus.io/`
[33] `https://grafana.com/`
[34] `https://kiali.io/`

figura 93.

```
manuel@master:~/istio-installation/istio-1.10.0$ kubectl apply -f samples/addons/
serviceaccount/grafana created
configmap/grafana created
service/grafana created
deployment.apps/grafana created
configmap/istio-grafana-dashboards created
configmap/istio-services-grafana-dashboards created
deployment.apps/jaeger created
service/tracing created
service/zipkin created
service/jaeger-collector created
customresourcedefinition.apiextensions.k8s.io/monitoringdashboards.monitoring.kiali.io created
serviceaccount/kiali created
configmap/kiali created
clusterrole.rbac.authorization.k8s.io/kiali-viewer created
clusterrole.rbac.authorization.k8s.io/kiali created
clusterrolebinding.rbac.authorization.k8s.io/kiali created
role.rbac.authorization.k8s.io/kiali-controlplane created
rolebinding.rbac.authorization.k8s.io/kiali-controlplane created
service/kiali created
deployment.apps/kiali created
serviceaccount/prometheus created
configmap/prometheus created
clusterrole.rbac.authorization.k8s.io/prometheus created
clusterrolebinding.rbac.authorization.k8s.io/prometheus created
service/prometheus created
deployment.apps/prometheus created
```

Figura 92: Despliegue de las integraciones de Istio

```
manuel@master:~/istio-installation/istio-1.10.0$ kubectl get svc -n istio-system
NAME                   TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)                                      AGE
grafana                ClusterIP      10.106.97.34     <none>        3000/TCP                                     111s
istio-ingressgateway   LoadBalancer   10.111.248.230   <pending>     15021:30334/TCP,80:32634/TCP,443:32102/TCP   26m
istiod                 ClusterIP      10.101.245.201   <none>        15010/TCP,15012/TCP,443/TCP,15014/TCP        26m
jaeger-collector       ClusterIP      10.98.156.237    <none>        14268/TCP,14250/TCP                          111s
kiali                  ClusterIP      10.108.70.35     <none>        20001/TCP,9090/TCP                           111s
prometheus             ClusterIP      10.106.168.141   <none>        9090/TCP                                     111s
tracing                ClusterIP      10.107.14.131    <none>        80/TCP                                       111s
zipkin                 ClusterIP      10.97.170.113    <none>        9411/TCP                                     111s
```

Figura 93: Obtención de las direcciones de acceso de las nuevas integraciones

Por tanto, una vez tengamos estas integraciones desplegadas, podemos emplearlas para la realización de la monitorización del clúster:

- **Prometeus:** es un software que podemos emplear para la recopilación y monitorización de datos.
- **Grafana:** permitirá la visualización gráfica de las monitorizaciones realizadas por Prometeus.
- **Tracing:** servicio que nos permitirá el rastreo de las diferentes peticiones que lleguen a los microservicios corriendo sobre el clúster.
- **Jaeger:** permitirá la visualización gráfica del rastreo realizado por Tracing.
- **Zipkin:** software alternativo a Jaeger.
- **Kiali:** herramienta de visualización y gestión gráfica de los microservicios corriendo sobre el clúster.

Puesto que conocemos las direcciones asociadas a estas integraciones, podemos acceder a su interfaz gráfica a través de un navegador. Por ejemplo, contando con dos despliegues sobre el clúster, Ubuntu y Nginx, comprobamos el uso de la herramienta Kiali, a través de la cual pudimos comprobar como es posible observar y gestionar dichos *deployments* gráficamente. Por ejemplo, en las imágenes 94, 95 y 96, en comienzo los despliegues no tienen ningún tipo de conexión establecida, pero al crear una consola de comandos interactiva sobre alguno de sus contenedores desde la máquina máster, podemos ver como

dicha conexión es detectada y reflejada por Kiali.

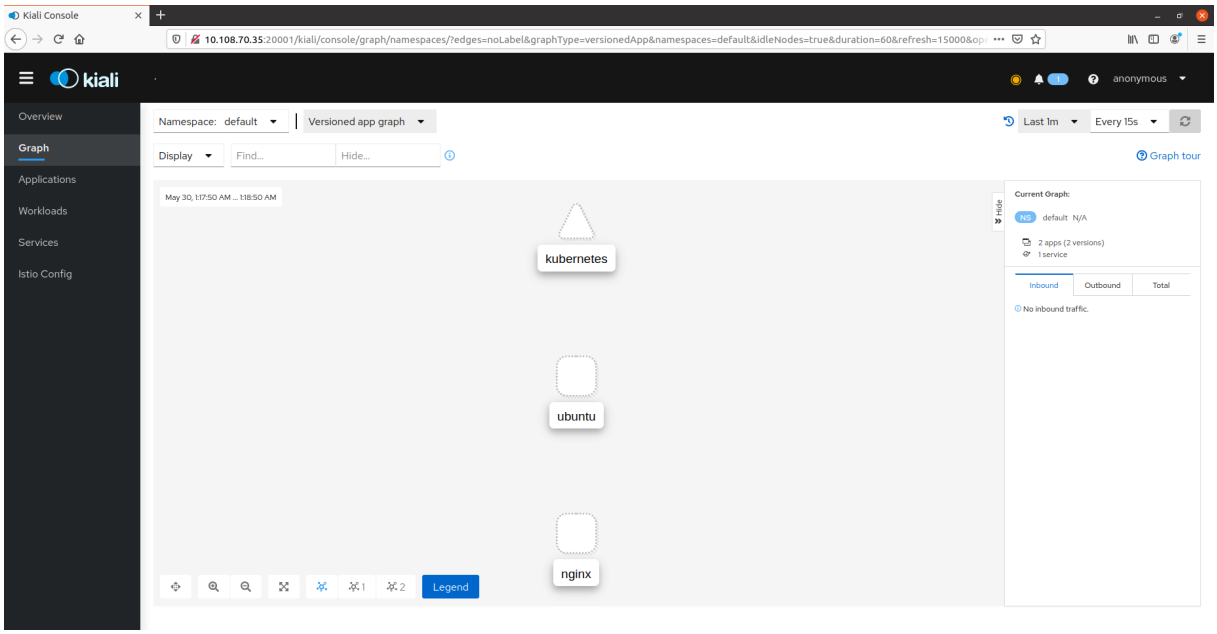


Figura 94: Ejemplo de uso de Kiali, parte 1

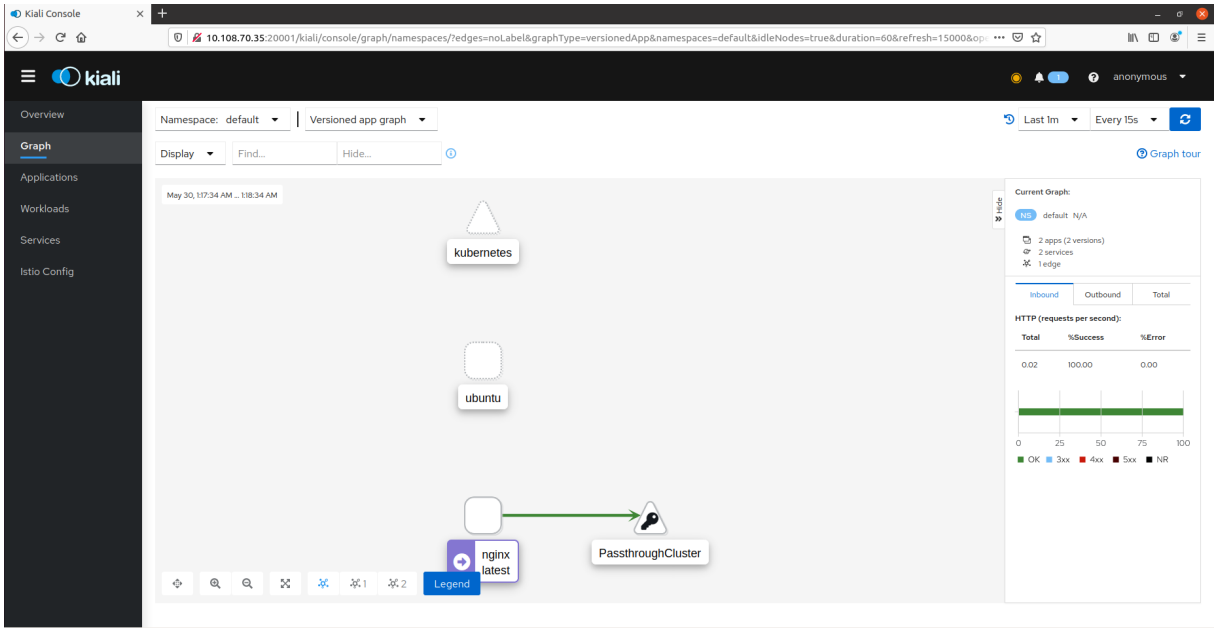


Figura 95: Ejemplo de uso de Kiali, parte 2

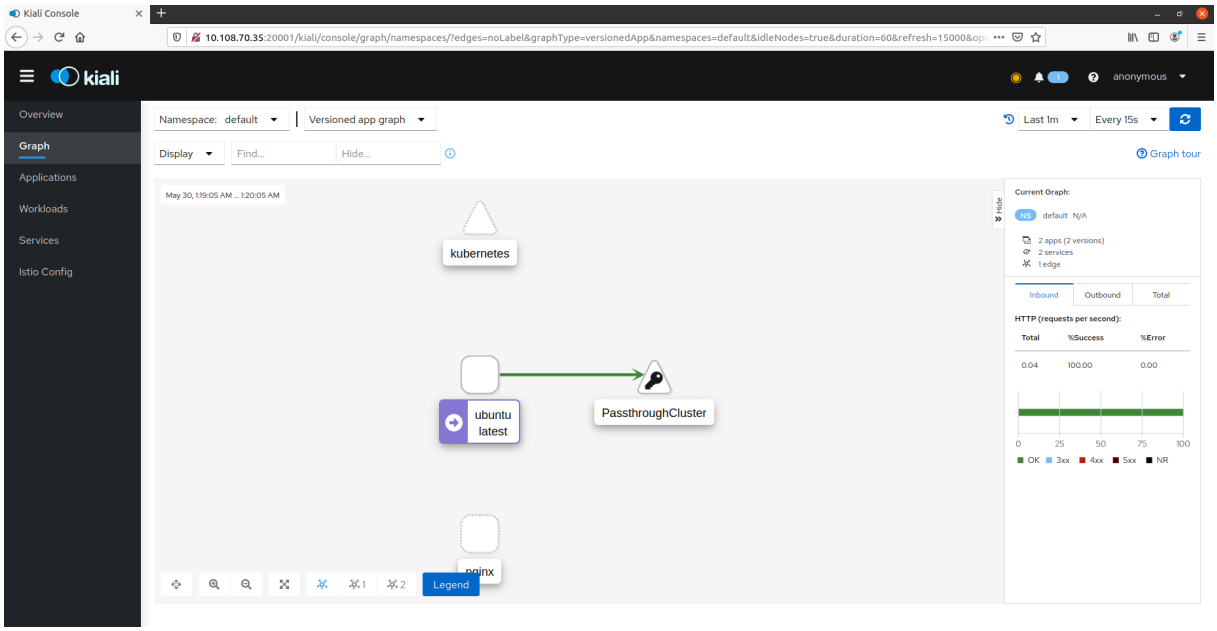


Figura 96: Ejemplo de uso de Kiali, parte 3

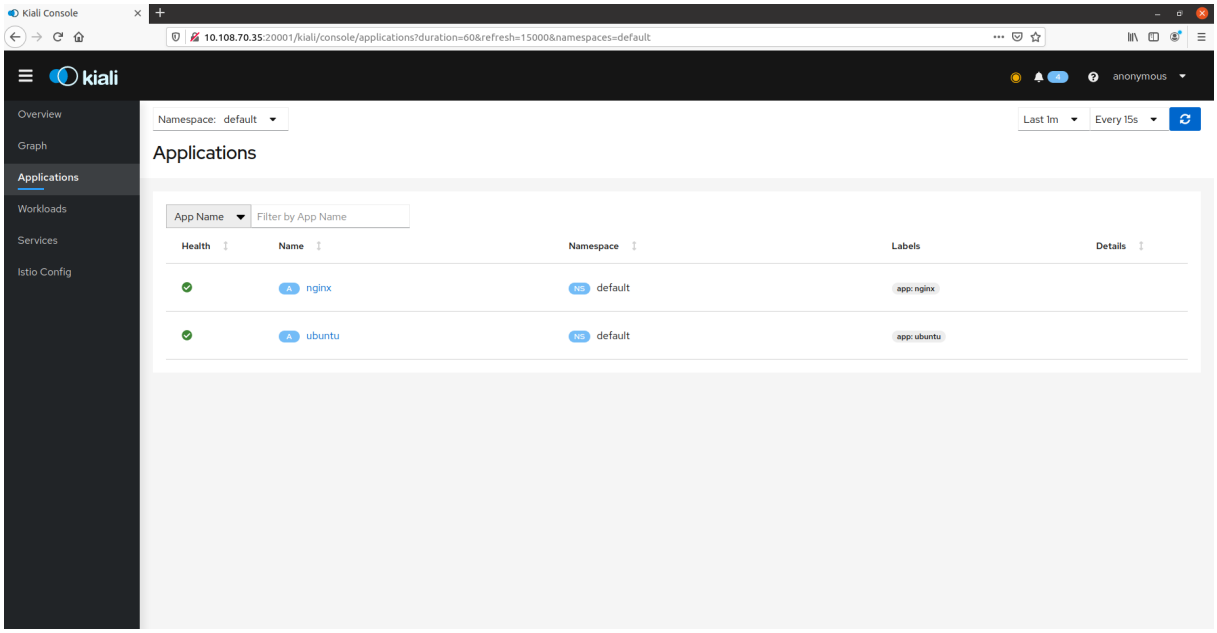


Figura 97: Ejemplo de uso de Kiali, parte 4

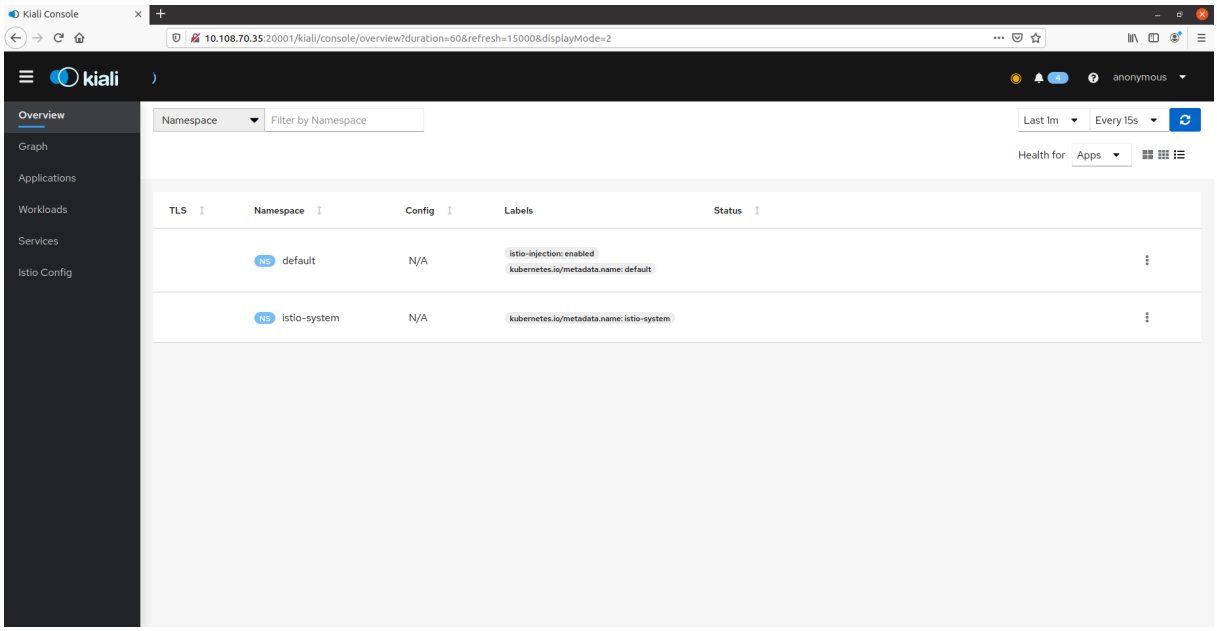


Figura 98: Ejemplo de uso de Kiali, parte 5

De esta forma, podemos observar como gracias al uso de *Service Mesh* es posible realizar una monitorización sencilla del clúster, completando así la realización del RNF15, reflejado en la tabla 38. Debemos tener en cuenta que esta ha sido una introducción breve a la monitorización y que es preciso realizar algunas configuraciones más para tener estos servicios funcionando plenamente. Una tarea de implementación futura podría consistir en la correcta configuración de Prometeus y Grafana, para completar la total monitorización de los microservicios del clúster.

8. Conclusiones

Finalizada la sección de securización, donde fueron aplicadas aquellas medidas precisas para la detección y corrección o prevención de las vulnerabilidades identificadas previamente en la sección de análisis, podemos dar por concluido el proceso de estudio para la realización de un análisis de seguridad de arquitecturas basadas en Kubernetes.

A lo largo del desarrollo de este proyecto hemos podido ver, de una forma integral, la securización de un clúster mínimo, pero funcional, basado en esta tecnología de orquestación. Decimos que dicho desarrollo ha sido integral, puesto que ha incluido un proceso de planificación, en el que se han incluido todos aquellos aspectos a considerar para la viabilidad del proyecto; un proceso de estudio teórico, exponiendo los principales conceptos sobre los que el trabajo formaría su base; una especificación de limitaciones y requisitos, que guiarían el recorrido a seguir por unos estudios teóricos posteriores más profundos, así como la realización de futuras pruebas prácticas; un procedimiento detallado de la creación del laboratorio de pruebas, con el fin de poder recrear este escenario para una futura continuidad del estudio; una fase de análisis de vulnerabilidades, donde se estudiaron y explotaron, de forma teórico-práctica, diversas vulnerabilidades presentes en clústeres Kubernetes, demostrando las debilidades presentes en un sistema desprotegido; y, finalmente, la securización de estas vulnerabilidades, una vez más, desde una perspectiva teórico-práctica, demostrando la efectividad de las medidas adoptadas y completando así el ciclo de desarrollo de este estudio. Para finalizar, también serán indicadas una serie de posibles mejoras futuras al proyecto, en la sección 9, de tal forma que funcionen como base para una potencial continuación del mismo.

8.1. Clasificación final de los análisis de seguridad realizados

Concluidos así todos los aspectos contemplados en la sección de planificación (sección 2), podemos realizar, como punto final, una clasificación recopilatoria de los análisis y securizaciones realizados, de forma que se alcance una división de más de alto nivel sobre las capas analizadas a lo largo de las diferentes pruebas, siempre desde una perspectiva de la seguridad. Esta clasificación se realizará atendiendo a las 4C de la seguridad en *Cloud Native*. Concretamente, esta clasificación sigue un enfoque en capas, con el fin de aumentar la defensa en profundidad de la seguridad, y está conformada por protecciones en: Código, Contenedor, Clúster y *Cloud*, de ahí el nombre de las 4C. Una representación gráfica de esta clasificación puede observarse en la figura 99.

Aplicando esta clasificación a las pruebas realizadas a lo largo del proyecto, podemos resumir todas las securizaciones estudiadas y realizadas en la siguiente lista:

- **Securización *Cloud* o securización de la infraestructura:**
  - **Acceso a la red:** ver propuesta de trabajo futuro de securización de la red externa, reflejada en la sección 9.3.
  - **Acceso y encriptación del componente `etcd`:** ver propuesta de trabajo futuro de securización del componente `etcd`, reflejada en la sección 9.2.

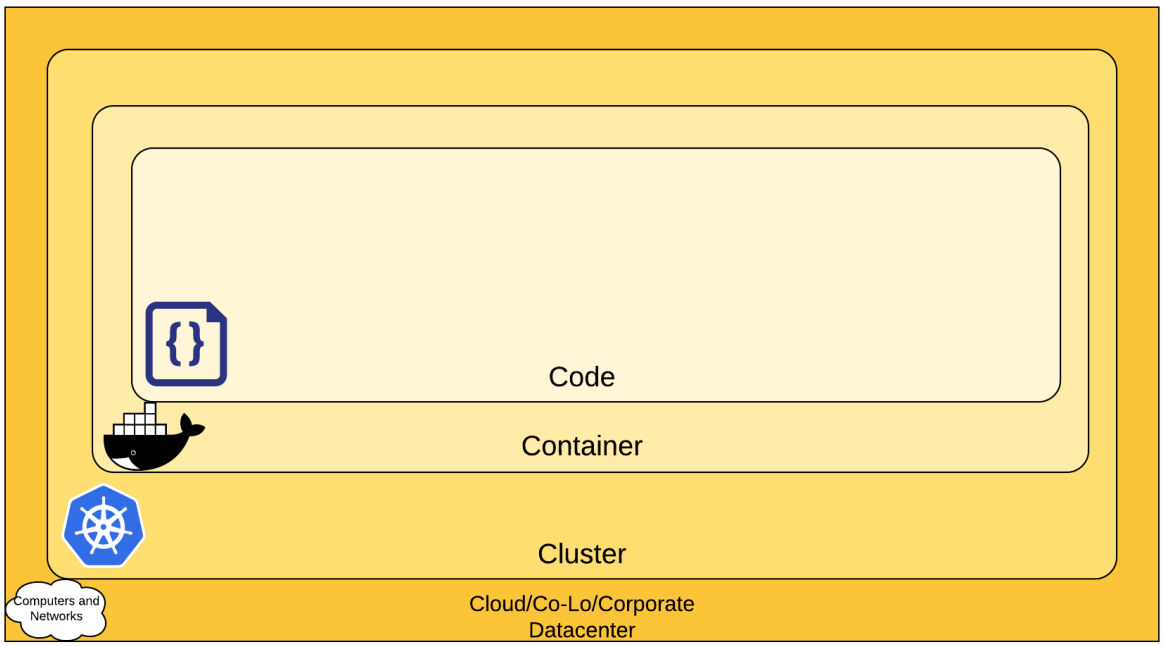


Figura 99: Representación gráfica de las 4C de seguridad en *Cloud Native*

Fuente: [29]

- **Securización del Clúster:**
  - **Autenticación y autorización RBAC:** ver propuesta de trabajo futuro de control de acceso basado en roles, reflejada en la sección 9.4.
  - **Administración de *secrets*:** ver el requisito RNF13, reflejado en la tabla 36.
  - ***Pod Security Policies*:** ver los requisitos RNF6, reflejado en la tabla 29; y RNF8, reflejado en la tabla 31.
  - **Aplicación de QoS sobre *pods*:** ver los requisitos RNF3, reflejado en la tabla 26; RNF4, reflejado en la tabla 27; y RNF5, reflejado en la tabla 28.
  - **Políticas de red:** ver el uso de *Service Mesh* a lo largo de la sección 7.11.
  - **Uso de TLS:** ver el requisito RNF14, reflejado en la tabla 37, así como el uso de *Service Mesh* a lo largo de la sección 7.11.
  - **Diferenciación de espacios** (no incluido en la clasificación original base): ver el requisito RNF11, reflejado en la tabla 34.
  - **Monitorización del clúster** (no incluido en la clasificación original base): ver el requisito RNF15, reflejado en la tabla 38.
- **Securización de los Contenedores:**
  - **Escáneres de vulnerabilidades de contenedores y seguridad de dependencias del sistema operativo:** ver los requisitos RNF1, reflejado en la tabla 24; RNF2, reflejado en la tabla 25; y RNF10, reflejado en la tabla 33.

- **Firmas digitales de imágenes:** ver el requisito RNF9, reflejado en la tabla 32.
- **Prohibición de usuarios privilegiados:** ver el requisito RNF6, reflejado en la tabla 29.
- **Uso de contenedores de solo lectura** (no incluido en la clasificación original base): ver el requisito RNF8, reflejado 31.
- **Combinación de diferentes tecnologías de virtualización** (no incluido en la clasificación original base): ver el requisito RNF7, reflejado en la tabla 30.

- **Securización del Código:**
  - **Acceso a través de TLS:** ver el requisito RNF14, reflejado en la tabla 37, así como el uso de *Service Mesh* a lo largo de la sección 7.11.

9. Trabajo futuro

Dado por finalizado el estudio que se pretendía alcanzar con la realización de este proyecto, debemos tener en cuenta que el proceso de securización de un clúster se trata de un procedimiento muy amplio y en constante evolución, debido a la gran explosión en el uso de microservicios que está teniendo lugar desde hace algunos años y que, previsiblemente, continuará avanzando y creciendo en un futuro. Es por ello que un estudio total, donde queden recogidas todas las securizaciones posibles o necesarias para el despliegue de un clúster Kubernetes, resulta extremadamente complicado de ser completado, considerando las limitaciones temporales establecidas en la sección 2.3, así como el hecho de que el desarrollo de este proyecto fue llevado a cabo por un único autor.

Teniendo en cuenta estos hechos, a lo largo de esta última sección se recopilarán todas aquellas configuraciones necesarias para la obtención de un clúster Kubernetes seguro, o más seguro, pero que no han podido ser estudiadas detenidamente a lo largo del desarrollo del proyecto, fundamentalmente debido a las limitaciones citadas en la sección 4.1. Por ejemplo, todas aquellas soluciones para las que fuera preciso la utilización de un hardware más potente o requirieran de procesos de instalación y/o configuración muy extensos y, por tanto, se vieran restringidos por la limitación de este proyecto, al poder poner en riesgo los plazos de entrega existentes.

Como ya fue mencionado, debemos comprender que la securización de un clúster Kubernetes se trata de un proceso muy amplio y que a lo largo de este proyecto se ha tratado de realizar una visión lo más completa posible del mismo, partiendo desde cero. Es por ello que algunos aspectos de la securización han quedado fuera del estudio. No obstante, en esta sección se indicarán algunas de las securizaciones que fueron tenidas en cuenta para su estudio, pero que desafortunadamente, finalmente fueron descartadas en las fases de análisis y/o securización.

Así pues, aunque posiblemente la realización de las siguientes pruebas y securizaciones no podrán garantizar un 100 % de seguridad en el uso de un clúster Kubernetes, en caso de que este proyecto fuera continuado en un futuro, los siguientes estudios conformarían el conjunto de pruebas que el autor ha considerado más interesantes para completar esta mejora de la seguridad, y que suponen un punto de partida o de continuación a considerar.

9.1. Limitación de red

Revisando el conjunto de pruebas realizadas en la sección de análisis (sección 6) y confrontándolas con aquellas medidas de corrección o prevención establecidas a lo largo de la sección de securización (sección 7), podremos observar como uno de los requisitos no funcionales, el RNF5, reflejado en la tabla 28, ha quedado sin securización propia.

Esta exclusión viene dada, esencialmente, por la necesidad de aplicar la medida de miniaturización ante la materialización del riesgo R2, recogido en la tabla 15. Esto es, ante la aparición de un suceso imprevisto que produjo una demora en la realización de alguna de las entregas parciales del proyecto, se optó por descartar alguna de las tareas

planificadas. Concretamente, la securización descartada se trata de la limitación del uso de red, puesto que no presenta una solución homogénea clara que pudiese ser aplicada y, por tanto, requería de un gran tiempo de estudio. No obstante, algunas de las propuestas valoradas para la realización de esta securización podrían ser:

- **Aplicación de políticas de QoS** mediante los servicios ofrecidos por sistemas GNU/Linux: utilización de mecanismos que ofrezcan un trato preferenciado a diferentes tráficos y aplicaciones, usando como base la red virtual creada por cada contenedor Docker, lo que podría suponer un gran trabajo de configuración.
- **Delegación de esta limitación en el Service Mesh de Istio:** una vez el clúster de pruebas está configurado con un Service Mesh como Istio, éste ofrece mecanismos para aplicar limitaciones de red, como, por ejemplo, la velocidad nativa del sidecar, para conseguir así limitar dinámicamente el tráfico realizado por un microservicio [17]. Persiguiendo la coherencia del desplazamiento de la gestión de las comunicaciones entre los diferentes microservicios cara el Service Mesh, esta es quizás la opción más interesante para ser abordada.

9.2. Securización del componente `etcd`

Según lo expuesto en la sección 3.4.2, el componente `etcd` se trata de uno de los elementos más importantes dentro de un clúster Kubernetes, puesto que almacenará en su interior todos los cambios que sucedan en el clúster. Por este motivo, la securización del componente `etcd` requiere un estudio más específico que el resto de componentes, al comportarse como la base de datos interna usada por Kubernetes para la correcta gestión del clúster. Puesto que `etcd` almacena el estado del clúster e información secreta, se trata de un recurso que debemos tratar con cautela, al resultar sensible y un objetivo común para los atacantes. Por tanto, en caso de que un usuario malintencionado consiguiera obtener acceso a `etcd`, entonces dicho usuario podría obtener toda la información sensible allí almacenada, tal como nombres de usuario, contraseñas o consultas, pudiendo llegar a hacerse con el control total del clúster. Incluso un acceso de solo lectura resulta extremadamente peligroso, ya que un atacante podría emplear los datos allí almacenados para conseguir una posterior escalada de privilegios. [27]

En su configuración por defecto, `etcd` almacena toda la información, que debería ser en todo momento secreta, en modo texto. A pesar de que la información no resulta legible directamente, sino que se encuentra encriptada, la clave para dicha encriptación se encuentra almacenada en modo texto en el fichero de configuración del nodo máster. [3]

Por este motivo, el componente `etcd` debería ser protegido con más cautela que el resto del clúster, protegiéndolo debidamente mediante las herramientas que se consideren oportunas, como podrían ser:

- **Separación física** del componente `etcd` del resto del clúster y **protección** específica de dicho componente mediante el uso de **cortafuegos**.
- **Habilitación de TLS** para la comunicación cliente-servidor y servidor-servidor, haciendo uso de claves público-privadas que garanticen la seguridad en el intercambio de datos.

- **Activación del cifrado** para la información secreta almacenada en `etcd`. El cifrado de la información contenido en el interior de este componente es crucial y no está activado de forma predeterminada. Es posible habilitarlo a través del proceso `kube-apiserver`, mediante el uso del argumento `-encryption-provider-config`. Para realizar el proceso de cifrado será necesario contar con un proveedor para realizar el cifrado y definir las claves secretas, de modo que no resulta de una configuración directa, al depender de terceros y ser necesarios numerosos pasos. [27]

9.3. Securización de la red externa

A lo largo de este estudio, hemos visto como realizar numerosas securizaciones que afectan directamente al comportamiento interno de un clúster Kubernetes. Sin embargo, al igual que cualquier otro tipo de servidor o clúster, también resulta importante realizar una securización de la red externa, es decir, de los posibles accesos que se pueden producir desde Internet o incluso desde otras subredes de la misma organización.

Con este objetivo en mente, muchas veces esta securización pasa por la aplicación de sistemas cortafuegos, sistemas encargados de separar redes informáticas, efectuando un control del tráfico que transcurre entre ellas. Dicho control consiste, de forma general, en permitir o denegar el paso de la comunicación de una red a otra mediante el control de protocolos de red. La implementación o arquitectura que puede poseer esta protección cortafuegos puede ser muy variada: existen arquitecturas de un solo punto, más sencillas, que consiste en separar la red que se quiere proteger (una red interna) del exterior por un solo dispositivo cortafuegos. Una alternativa más segura, aunque también más compleja, es la creación de arquitecturas con redes perimetrales, permitiendo añadir un nivel adicional de seguridad. En este último caso, se incorporaría una subred intermedia entre la red a proteger y el exterior, permitiendo que actúe de barrera ante posibles ataques o incursiones. Esta red intermedia suele recibir el nombre de zona desmilitarizada. [14]

Por tanto, una securización a nivel externo que puede resultar interesante es la mejora de la red presentada en la arquitectura del laboratorio de pruebas, representada inicialmente en la figura 5, por una arquitectura de red mejorada, con la existencia de cortafuegos y una red desmilitarizada. Un ejemplo de esta propuesta de arquitectura puede observarse en la figura 100.

9.4. Control de Acceso Basado en Roles

El Control de Acceso Basado en Roles (RBAC) se trata de un método de regular el acceso a los recursos de una máquina o red, basándose en los roles de cada usuario de la organización. Al poseer un acceso basado en roles y no en perfiles, se permite que un mismo rol sea compartido por múltiples usuarios, consiguiendo distribuir los permisos a lo largo del clúster para que todos los usuarios puedan tener el acceso correcto que les corresponda. Es posible configurar un control RBAC en Kubernetes, permitiendo configurar dinámicamente las políticas a través de la API de este sistema orquestador.

Como una aproximación inicial, resulta interesante conocer que, para su funciona-

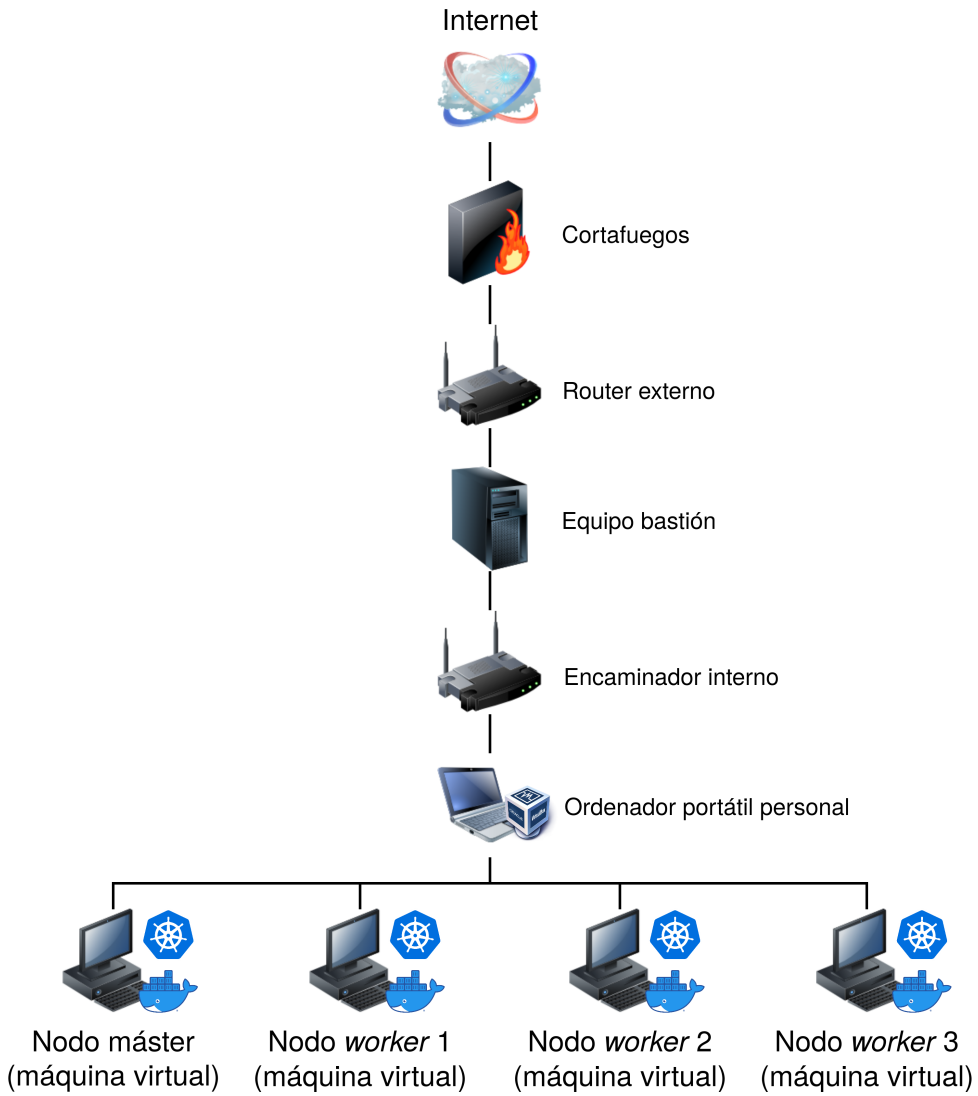


Figura 100: Propuesta de securización de la red externa

miento, Kubernetes hace uso de, esencialmente, cuatro recursos: *Roles*, *RoleBindings*, *ClusterRoles* y *ClusterRoleBindings*. Los *Roles* y los *ClusterRoles* contienen las reglas que representan la serie de permisos que queremos otorgar. Su principal diferencia radica en que los *Roles* indicarán una serie de permisos dentro de un espacio de nombres concreto, mientras que, por el contrario, los *ClusterRoles* no están limitados a un espacio en concreto, siendo expandibles a la totalidad del clúster. Un *RoleBinding* se encarga de otorgar los permisos definidos en un rol a uno o varios usuarios sobre un espacio de nombres. Los *ClusterRoleBidings* tendrán la misma funcionalidad, pero siendo estos últimos aplicables a permisos que afecten a todo el clúster.

Para la realización de la implementación de RBAC en Kubernetes, la documentación oficial presenta una serie de explicaciones teóricas y prácticas que pueden resultar adecuadas como punto inicial. [40]

9.5. Mejora continua de los ficheros de configuración YAML

En la securización llevada a cabo en la sección 7.10 observamos como, a pesar de conseguir una mejora substancial en la seguridad de los ficheros YAML de configuración para la realización de despliegues, la herramienta Kubesec aún proponía ciertas configuraciones para la mejora de estos ficheros. Por ejemplo, algunos de los puntos a aplicar para conseguir una mejora en la seguridad existente en el clúster de pruebas, podrían ser:

- **Utilización de `seccomp`**[35], una funcionalidad del *kernel* de GNU/Linux que puede ser empleado para restringir las acciones permitidas dentro de un contenedor y cuya compactibilidad con Docker está bien documentada. [35]
- **Restringir y/o reducir las capacidades del *kernel* disponibles para ser ejecutadas desde dentro de un contenedor**, consiguiendo así la minimización de la superficie de ataque. Para realizar esta securización, es posible emplear las "*pod security policies*", de forma similar a las limitaciones de los permisos de ejecución de la sección 7.3, o a la creación de sistemas de solo lectura de la sección 7.5.
- **Empleo de modelos de Control de Acceso Obligatorio:** puesto que este punto resulta bastante extenso, será explicado en la sección 9.6.
- **Utilización de *Service Accounts*** para configurar el principio del mínimo privilegio a cada *pod*. Los *Service Accounts* pueden ser usados para autenticar los procesos a nivel de máquina para obtener acceso a un clúster Kubernetes, siendo la API del servidor la responsable de tal autenticación. [19]

9.6. Modelo centralizado: MAC

El Control de Acceso Obligatorio se trata un modelo en el que las políticas del sistema no pueden ser alteradas por usuarios de forma individual, sino que, en este modelo, los objetos incluyen un nivel de seguridad y un conjunto de categorías. Para establecer el acceso que los usuarios tengan sobre los objetos, éste no se establece directamente sobre

[35] `https://man7.org/linux/man-pages/man2/seccomp.2.html`

el objeto, sino que los usuarios poseerán unos permisos para cada categoría.

Los MAC pueden ser utilizados para ofrecer un aislamiento entre los diferentes contenedores y la máquina anfitriona. Esta securización puede llevarse a cabo gracias a la integración existente en el *kernel* de GNU/Linux; no obstante, debido a las complejas reglas que lo conforman, puede resultar difícil de configurar. Algunas de las soluciones más comunes para su utilización son AppArmor[36] y SELinux[37].

Gracias a la aplicación de este modelo, conseguiríamos añadir una nueva capa de seguridad sobre el clúster, consiguiendo que, en caso de que un ataque tuviera éxito dentro de un contenedor y un usuario malintencionado consiguiera cualquier tipo de acceso sobre la máquina anfitriona, el MAC conseguiría frustrar dicho ataque. Por ejemplo, en caso de que un proceso ejecutado dentro de un contenedor consiguiera alcanzar un archivo de la máquina anfitriona, una implementación de este modelo se encargaría de verificar los permisos sobre tal operación de lectura o escritura, impidiendo que el ataque se materializara, al detectar la falta de privilegios sobre la categoría asociada a este objeto para este usuario. [38]

9.7. Otras referencias de consulta

Además de las propuestas de trabajo futuras presentadas a lo largo de esta sección, en las que se realizó una pequeña introducción al procedimiento de implementación, en mayor o menor medida, también resulta interesante indicar una serie de posibles estándares de securización, iniciativas o similares. Estas fuentes son las que se han considerado más interesantes después de la realización de la fase de estudio y que, junto con las ideas propias del autor, dieron lugar a la realización de la lista de requisitos del proyecto. De esta forma, en caso de querer continuar con la evolución del proyecto más allá de las propuestas de trabajo futuro presentadas hasta el momento, una posible guía de consulta podría ser el seguimiento de las siguientes fuentes:

- ***11 Ways (Not) to Get Hacked*** [15]:

  Una recopilación de posibles securizaciones realizada por Andrew Martin y recogidas en el blog oficial de Kubernetes. En su propuesta, Andrew Martin nos presenta numerosos aspectos a tener en cuenta para securizar un clúster Kubernetes, dividiendo su clasificación en: 1) *Control Plane*, 2) carga de trabajo (*Workloads*) y 3) una introducción de *Service Mesh*, presentándolo como la siguiente etapa de la seguridad en computación basada en la nube.

- ***XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices*** [3]:

  Estudio presentado por M. S. Islam Shamim, F. Ahamed Bhuiyan y A. Rahman en la conderencia "IEEE Secure Development (SecDev)" del año 2020. En este estudio se contempla una presentación teórica de Kubernetes como sistema de orquestación, sus diferentes componentes y, finalmente, una recopilación sistematizada de las mejores prácticas de seguridad para aplicar contra dicho sistema.

[36] `https://apparmor.net/`
[37] `https://github.com/SELinuxProject/selinux`

- **Las 4C de Seguridad en *Cloud Native*** [29]:

  Este es el sistema de clasificación que sirvió como base para la organización recopilatoria final de los análisis y securizaciones realizados, el cual ya fue presentado en la sección 8.1. A pesar de ya haber realizado una clasificación tomando este sistema como base, la composición original del mismo incluye más aspectos que no han sido contemplados a lo largo de este proyecto.

- **Las secciones de seguridad de la documentación oficial de Kubernetes y Docker:** [36][25]

  Aún siendo, probablemente, el recurso más obvio, las páginas oficiales contendrán siempre la información más actualizada con respecto a sus productos, además de servir como un excelente índice a través del cual alcanzar procesos y ejemplos de securización que sirvieron como base para la realización de alguna de las pruebas de este proyecto.

- ***Computing Native Computing Foundation*[38]:**

  La *Cloud Native Computing Foundation* se trata de un proyecto de la *Linux Foundation*[39], creado con la intención de ayudar en el avance de las tecnologías de contenerización. Entre sus múltiples iniciativas, resulta especialmente relevante, desde la perspectiva de este proyecto, las múltiples certificaciones en Kubernetes y los cursos presentados por esta organización, destacando la certificación *Certified Kubernetes Security Specialist*[40].

[38] `https://www.cncf.io/`
[39] `https://linuxfoundation.org/`
[40] `https://www.cncf.io/certification/cks/`

A. Anexos

A.1. Comandos para la instalación y configuración de Kubernetes

Fuente: [16].

A.1.1. Nodos *worker*

Listing 13: Comandos para la instalación de Kubernetes en un nodo *worker*

```
sudo apt install docker.io
sudo docker --version
sudo systemctl enable docker
sudo systemctl status docker
sudo apt-get install curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add
sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
sudo apt install kubeadm kubelet kubectl
sudo hostnamectl set-hostname worker1
sudo swapoff -a
sudo kubeadm join 192.168.1.100:6443 --token ndy92w.wj8rdp7uaq7nbhp8 --discovery
    ↪ -token-ca-cert-hash sha256:3
    ↪ cc4146c956567f67d3526556bc9b91169c51674b1586157fbe6a69d756a9254
```

A.1.2. Nodo *master*

Listing 14: Comandos para la instalación de Kubernetes en un nodo *master*

```
sudo apt install docker.io
sudo docker --version
sudo systemctl enable docker
sudo systemctl status docker
sudo apt-get install curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add
sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
sudo apt install kubeadm kubelet kubectl
sudo hostnamectl set-hostname master
sudo kubeadm init --pod-network-cidr = 10.244.0.0/16
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
kubectl get nodes
sudo kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/
    ↪ Documentation/kube-flannel.yml
kubectl get pods --all-namespaces
sudo kubectl get nodes
```

A.2. Fichero YAML para el despliegue de la imagen Nginx 1.20.0

Fuente: modificaciones sobre el fichero YAML disponible en: [33]

Listing 15: Fichero YAML para el despliegue de la imagen Nginx 1.20.0

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # indica al controlador que ejecute 2 pods
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.20.0
        ports:
        - containerPort: 80
```

A.3. Código C de un programa altamente consumidor de memoria

Fuente: propia.

Listing 16: Código C de un programa altamente consumidor de memoria

```
#include <stdlib.h>
#include <unistd.h>

void f(long double *a){
        a = (long double *)malloc(sizeof(long double) * 1000);
}

int main(void) {
        long double *a;

        while(1) {
            f(a);
            sleep(0);
        }
        return 0;
}
```

A.4. Fichero YAML para el despliegue de un pod con un contenedor Ubuntu

Fuente: propia.

Listing 17: Fichero YAML para el despliegue de un pod con un contenedor Ubuntu

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu
  labels:
    app: ubuntu
spec:
  containers:
  - name: ubuntu
    image: ubuntu:latest
    command: ["/bin/sleep", "3650d"]
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - mountPath: /root-test
      name: root-test
  restartPolicy: Always
  volumes:
  - name: root-test
    hostPath:
      # ubicacion en la maquina anfitriona
      path: /etc
      type: Directory
```

A.5. Fichero YAML para el despliegue de un *deployment* de Ubuntu con 4 réplicas

Fuente: propia.

Listing 18: Fichero YAML para el despliegue de un *deployment* de Ubuntu con 4 réplicas

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ubuntu-deployment
spec:
  selector:
    matchLabels:
      app: ubuntu
  replicas: 4 # indica al controlador que ejecute 4 pods
  template:
    metadata:
      labels:
        app: ubuntu
    spec:
      containers:
      - name: ubuntu
        image: ubuntu:latest
        command: ["/bin/sleep", "3650d"]
        imagePullPolicy: IfNotPresent
```

A.6. Fichero JSON para la creación de nuevos espacios de nombres

Fuente: propia.

Listing 19: Fichero JSON para la creación del espacio "espacio1"

```
{
  "apiVersion": "v1",
  "kind": "Namespace",
  "metadata": {
    "name": "espacio1",
    "labels": {
      "name": "espacio1"
    }
  }
}
```

Listing 20: Fichero JSON para la creación del espacio "espacio2"

```
{
  "apiVersion": "v1",
  "kind": "Namespace",
  "metadata": {
    "name": "espacio2",
    "labels": {
      "name": "espacio2"
    }
  }
}
```

A.7. Fichero JSON para la creación de nuevos espacios de nombres

Fuente: propia.

Listing 21: Fichero JSON para la creación del espacio “espacio1”

```
{
  "apiVersion": "v1",
  "kind": "Namespace",
  "metadata": {
    "name": "espacio1",
    "labels": {
      "name": "espacio1"
    }
  }
}
```

A.8. Fichero YAML para el despliegue de un *deployment* de Ubuntu securizado

Fuente: propia.

Listing 22: Fichero YAML para el despliegue de un *deployment* de Ubuntu securizado

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ubuntu-deployment
spec:
  selector:
    matchLabels:
      app: ubuntu
  replicas: 2 # indica al controlador que ejecute 2 pods
  template:
    metadata:
      labels:
        app: ubuntu
    spec:
      containers:
      - name: ubuntu
        image: ubuntu:latest
        command: ["/bin/sleep", "3650d"]
        imagePullPolicy: IfNotPresent
        resources:
          requests:
            memory: "300Mi"
            cpu: "300m"
          limits:
            memory: "500Mi"
            cpu: "500m"
        env:
        - name: envuser
          valueFrom:
            secretKeyRef:
              name: secret-manuel
              key: USER_NAME
        - name: envpass
          valueFrom:
            secretKeyRef:
              name: secret-manuel
              key: PASSWORD
        securityContext:
          runAsNonRoot: true
          allowPrivilegeEscalation: false
          readOnlyRootFilesystem: true
          runAsUser: 10001
          runAsGroup: 30001
```

Referencias

[1] L. Bass, *"The Software Architect and DevOps", in IEEE Software, vol. 35, no. 1, pp. 8-10, January/February* 2018, doi: 10.1109/MS.2017.4541051.

[2] T. Combe, A. Martin y R. Di Pietro, *"To Docker or Not to Docker: A Security Perspective", IEEE Cloud Computing, vol. 3, no. 5, Sept.-Oct. 2016, pp. 54-62.*

[3] M. S. Islam Shamim, F. Ahamed Bhuiyan y A. Rahman, *"XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices", 2020 IEEE Secure Development (SecDev)*, 2020, pp. 58-64, doi: 10.1109/SecDev45635.2020.00025.

[4] H. Kang, M. Le y S. Tao, *"Container and Microservice Driven Design for Cloud Infrastructure DevOps", 2016 IEEE International Conference on Cloud Engineering (IC2E), Berlin, Germany*, 2016, pp. 202-211, doi: 10.1109/IC2E.2016.26.

[5] Kurtzer, M. Gregory, V. Sochat y M. W. Bauer, "*Singularity: Scientific containers for mobility of compute*", *PLoS ONE 12(5):e0177,459*, 2017. `https://doi.org/10.1371/journal.pone.0177459`

[6] E. Reshetova, J. Karhunen, T. J. Nyman y N. Asokan, "*Security of OS-level virtualization technologies*", *Secure IT Systems: 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*, pp. 77-93.

[7] G. Shobha Sahana Upadhya, J. Shetty y R. Rajeshwari, "*A State-of-Art Review of Docker Container Security Issues and Solutions*", *American International Journal of Research in Science, Technology, Engineering and Mathematics Volume 17*, 2017, pp. 33-36.

[8] R. Shu, X. Gu y W. Enck, "*A Study of Security Vulnerabilities on Docker Hub*", *CODASPY '17 Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 269-280.

[9] P. Deemer, G. Benefield, C. Larman e B. Vodde, "*The Scrum Primer*". 2012. `http://scrumprimer.org/`

[10] Dirección General de Educación, Juventud, Deporte y Cultura de la Comisión Europea, "Guía de uso del ECTS". 2015. `https://op.europa.eu/es/publication-detail/-/publication/da7467e6-8450-11e5-b8b7-01aa75ed71a1`

[11] *Docker Team, White Paper: "Introduction to Container Security"*, 2015.

[12] Project Management Institute, "*A guide to the project management body of knowledge (PMBOK guide)*", Newtown Square, 2004.

[13] I. Sommerville, "Ingeniería del software", 5ª edición, Pearson Addison Wesley, Madrid, 2002.

[14] Sistemas de cortafuegos - Apuntes de la materia "Seguridad en Redes", módulo 1 - Guillermo Navarro Arribas - *Universitat Oberta de Catalunya.*

[15] *11 Ways (Not) to Get Hacked*. Kubernetes Blog (https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/#7-statically-analyse-yaml). Consultado el 28 de abril de 2021.

[16] ¿Cómo instalar Kubernetes en Ubuntu y derivados y crear dos nodos?. Ubunlog (https://ubunlog.com/como-instalar-kubernetes-en-ubuntu-y-derivados-y-crear-dos-nodos/). Consultado el 24 de abril de 2021.

[17] *Enabling Rate Limits using Envoy*. Documentación oficial de Istio (https://istio.io/latest/docs/tasks/policy-enforcement/rate-limit/). Consultado el 31 de mayo de 2021.

[18] Hablando de microservicios... ¿Qué es *Service Mesh*?. Guillermo Alvarado (https://galvarado.com.mx/post/hablando-de-microservicios-que-es-service-mesh/). Consultado el 29 de mayo de 2021.

[19] *Configure Service Accounts for Pods*. Documentación oficial de Kubernetes (https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/). Consultado el 31 de mayo de 2021.

[20] *Container Image Signatures in Kubernetes*. Medium (https://medium.com/sse-blog/container-image-signatures-in-kubernetes-19264ac5d8ce). Consultado el 07 de mayo de 2021.

[21] *Content trust in Docker*. Documentación oficial de Docker (https://docs.docker.com/engine/security/trust/). Consultado el 07 de mayo de 2021.

[22] *Distribute Credentials Securely Using Secrets*. Documentación oficial de Kubernetes (https://kubernetes.io/docs/tasks/inject-data-application/distribute-credentials-secure/). Consultado el 29 de mayo de 2021.

[23] *Docker overview*. Documentación oficial de Docker (https://docs.docker.com/get-started/overview/). Consultado el 27 de abril de 2021.

[24] *Docker Hub Quickstart*. Documentación oficial de Docker (https://docs.docker.com/docker-hub/). Consultado el 27 de abril de 2021.

[25] *Docker security*. Documentación oficial de Docker (https://docs.docker.com/engine/security/). Consultado el 05 de mayo de 2021.

[26] *Getting Started With Kubernetes*. C Sharp Corner (https://www.c-sharpcorner.com/article/getting-started-with-kubernetes-part2/). Consultado el 27 de marzo de 2021.

[27] *Kubernetes Security Best Practices: 10 Steps to Securing K8s*. Aquasec (https://www.aquasec.com/cloud-native-academy/kubernetes-in-production/kubernetes-security-best-practices-10-steps-to-securing-k8s/). Consultado el 08 de mayo de 2021.

[28] *Lab: User Namespaces*. Collabnix (https://dockerlabs.collabnix.com/advanced/security/userns/). Consultado el 23 de mayo de 2021.

[29] Las 4C de Seguridad en Cloud Native. Documentación oficial de Kubernetes (https://kubernetes.io/es/docs/concepts/security/_print/#pg-04eeb110d75afc8acb2cf7a3db743985). Consultado el 01 de junio de 2021.

[30] *Managing Resources for Containers*. Documentación oficial de Kubernetes (https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/). Consultado el 20 de mayo de 2021.

[31] *Namespaces*. Documentación oficial de Kubernetes (https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/). Consultado el 28 de mayo de 2021.

[32] *Pod Security Policies*. Documentación oficial de Kubernetes (https://kubernetes.io/docs/concepts/policy/pod-security-policy/). Consultado el 22 de mayo de 2021.

[33] *Run a Stateless Application Using a Deployment*. Documentación oficial de Kubernetes (https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/). Consultado el 03 de mayo de 2021.

[34] *Runtime options with Memory, CPUs, and GPUs*. Documentación oficial de Docker (https://docs.docker.com/config/containers/resource_constraints/). Consultado el 03 de mayo de 2021.

[35] *Seccomp security profiles for Docker*. Documentación oficial de Docker (https://docs.docker.com/engine/security/seccomp/). Consultado el 30 de mayo de 2021.

[36] *Securing a Cluster*. Documentación oficial de Kubernetes (https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/). Consultado el 31 de mayo de 2021.

[37] *Security*. Documentación oficial de Istio (https://istio.io/latest/docs/concepts/security/). Consultado el 30 de mayo de 2021.

[38] *SELinux Mitigates container Vulnerability* . Red Hat blog (https://www.redhat.com/en/blog/selinux-mitigates-container-vulnerability). Consultado el 30 de mayo de 2021.

[39] *The Istio Service Mesh*. Documentación oficial de Istio (https://istio.io/latest/about/service-mesh/). Consultado el 30 de mayo de 2021.

[40] *Using RBAC Authorization*. Documentación oficial de Kubernetes (https://kubernetes.io/docs/reference/access-authn-authz/rbac/). Consultado el 31 de mayo de 2021.

[41] *Virtual Networking*. Documentación oficial de VirtualBox (https://www.virtualbox.org/manual/ch06.html). Consultado el 24 de abril de 2021.

[42] *Vulnerability scanning for Docker local images*. Documentación oficial de Docker (`https://docs.docker.com/engine/scan/`). Consultado el 25 de abril de 2021.

[43] *What is Kubernetes?*. Documentación oficial de Kubernetes (`https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`). Consultado el 01 de marzo de 2021.