

Seguridad en Docker

Juan Gamboa Fernández

Máster en Seguridad de las Tecnologías de la Información y de las Comunicaciones
Trabajo Final de Máster

Pau del Canto Rodrigo

Víctor García Font

30 de diciembre de 2018

© Juan Gamboa Fernández

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

FICHA DEL TRABAJO FINAL

Título del trabajo:	<i>Seguridad en Docker</i>
Nombre del autor:	<i>Juan Gamboa Fernández</i>
Nombre del consultor:	<i>Pau del Canto Rodrigo</i>
Nombre del PRA:	<i>Víctor García Font</i>
Fecha de entrega :	12/2018
Titulación:	<i>Máster en Seguridad de las Tecnologías de la Información y de las Comunicaciones</i>
Idioma del trabajo:	<i>Castellano</i>
Palabras clave	<i>Contenedores, Docker, Seguridad</i>

Resumen

Los contenedores se han implantado como una alternativa a las máquinas virtuales por su mayor rendimiento y menor demanda de recursos, entre otros motivos. Sin embargo las consideraciones de seguridad, especialmente la cuestión de garantizar un grado aceptable de aislamiento, son los mayores retos que plantean. Los mecanismos que proporciona el núcleo de Linux, *namespaces* y *cgroups*, presentan limitaciones en su implementación. La solución para mitigarlas, por otro lado, implica el uso un conjunto heterogéneo y complejo de medidas adicionales: control de acceso MAC, segregación de privilegios usando *capabilities* y filtrado de llamadas al núcleo con *secomp*. Estos mecanismos, no obstante, son complejos y requieren un conocimiento detallado de la tecnología en que se basa. Los proveedores de plataforma como servicio (*PaaS*) comerciales tienen la ventaja la economía de escala para enfrentarse a estos retos. Pero en una instalación privada de tamaño pequeño, la situación es muy diferente.

Este trabajo aborda la seguridad en el segundo contexto, tratando de determinar si la ejecución de contenedores en explotación en una infraestructura de tamaño reducido puede mantener un nivel de seguridad aceptable. Para ello, hemos realizado un breve estudio la tecnología de contenedores y de las guías de buenas prácticas de seguridad. Más adelante evaluamos la aplicación de un HIDS en un entorno de contenedores. Concluimos que usando guías como el *CIS Docker Benchmark*, que facilitan la aplicación de medias de seguridad aunque no se domine en detalle la tecnología de base, y empleando como medias adicional tales como un HIDS configurado de forma adecuada es posible alcanzar un nivel de seguridad aceptable.

Abstract

Containers have become a feasible alternative to virtual machines because of their higher performance and lower resource requirements, among other benefits. However, they pose security concerns, most notably how to achieve an adequate isolation level. The mechanisms supplied by the Linux kernel, namespaces and cgroups have several implementation shortcomings. In order to mitigate them, a complex and heterogeneous mixture of further measures must be applied: MAC access control, capabilities for privilege segregation, and syscalls filtering using seccomp. These mechanisms are nevertheless complex and they demand a thorough understanding of the technology. Commercial Platform as a Service (PaaS) providers can take advantage of economies of scale to address these challenges. But the situation is quite different when deployed in a smaller site.

This thesis addresses the security challenges in the latter context and tries to ascertain whether a small organization can achieve an adequate level of security for their containers. To that end, we have conducted a short study of container technology and guides of security best practices. At a later stage we assess a HIDS in a container environment. We conclude that, when applying security guidelines such as those provided by the CIS Docker Benchmark, which eases the application of security measures, and applying further measures, such as an adequately configured HIDS deployment, it is possible to reach an acceptable level of security.

Índice

1	Introducción.....	1
1.1	Contexto y justificación del trabajo.....	1
1.2	Objetivos del Trabajo.....	3
1.3	Enfoque y método seguido.....	3
1.4	Planificación del Trabajo.....	3
1.4.1	Identificación de tareas.....	3
1.4.2	Planificación temporal.....	4
1.5	Resultados previstos al terminar el trabajo.....	4
1.6	Estructura de esta memoria.....	4
2	Contexto de la tecnología.....	6
2.1	Repaso de la nomenclatura de seguridad.....	6
2.2	Virtualización.....	8
2.2.1	Hipervisores.....	9
2.2.2	Contenedores de sistemas.....	10
2.2.3	Contenedores de aplicaciones.....	11
2.3	Mecanismos de soporte para la creación de contenedores en Linux.....	13
2.3.1	Espacios de nombres (<i>namespaces</i>).....	13
2.3.2	Control de recursos (<i>cgroups</i>).....	14
2.3.3	Union filesystems.....	15
2.3.4	Control de acceso obligatorio.....	16
2.3.5	Capabilities.....	17
2.3.6	Otros mecanismos de confinamiento: <i>seccomp</i>	17
2.4	Clusters de contenedores y orquestación.....	18
2.5	Plataformas específicas para el despliegue de contenedores.....	19
3	Aspectos específicos de la Seguridad en contenedores Docker.....	21
3.1	Amenazas y vulnerabilidades.....	21
3.2	Guías de buenas prácticas.....	24
3.2.1	CIS Docker Benchmark.....	24
3.2.2	Publicaciones NIST.....	25
4	HIDS y sistemas de consolidación de logs.....	27
4.1	Introducción.....	27
4.2	Sistemas IDS.....	27
4.2.1	Herramientas complementarias a los IDS.....	28
4.3	Productos evaluados.....	28
4.3.1	OSSEC.....	29
4.3.2	Sysdig Falco.....	29
4.3.3	Wazuh.....	30
4.3.4	Graylog.....	30
4.4	Laboratorio de pruebas.....	31
4.4.1	Despliegue de máquinas.....	31
4.4.2	Configuración de envío y recepción de alertas.....	31
4.4.3	grayloga (recepción de logs).....	35
4.5	Pruebas de detección de intrusiones.....	37
4.5.1	Pruebas ejecutadas dentro los hosts.....	37
4.5.2	Pruebas de detección de intrusiones realizadas desde un nodo externo.....	38
4.6	Conclusiones de la evaluación.....	41
5	Conclusiones.....	42
	Referencias.....	43

Lista de figuras

Hipervisores de tipo 1 (bare metal) y tipo 2 (hosted).....	9
Virtualización a nivel de sistema operativo.....	11

1 Introducción

1.1 Contexto y justificación del trabajo

Aunque el empleo de la virtualización a nivel de sistema operativo se ha venido empleando durante años para crear lo que podría entenderse como "máquinas virtuales ligeras" (por ejemplo, Jails en FreeBSD, Zones en Solaris, y OpenVZ o LXC) las plataformas para la ejecución de contenedores de aplicaciones (tales como Docker o Rkt de CoreOS) conforman un paradigma que introduce novedades sustanciales, como el empleo de imágenes estructuradas en capas, y las facilidades para la distribución de imágenes. De forma paralela, este nuevo modelo introduce nuevas vulnerabilidades, que deben contrarrestarse con un conjunto de mecanismos poco homogéneo, y con frecuencia, mal documentados.

El uso de los contenedores de aplicaciones ha experimentado un crecimiento notable en los últimos años, especialmente cuando se compara con el crecimiento de las plataformas de virtualización basada en hipervisores en el mismo período. Este fenómeno se debe, entre otras causas, a los menores tiempos de arranque, menor uso de memoria, y mayor densidad de instancias por máquina de los contenedores con respecto a las máquinas virtuales. Otro factor decisivo en su difusión es que, a diferencia de otras tecnologías de virtualización a nivel de sistema operativo como OpenVZ o Jails, las plataformas para el despliegue y ejecución de contenedores de aplicaciones facilitan un modelo de uso muy orientado a las metodologías de desarrollo ágiles y a las propuestas del "movimiento" DevOps, que han asumido, al mismo tiempo, un papel cada vez más relevante en el sector del desarrollo de software. El sistema de distribución de imágenes permite el empaquetado de aplicaciones junto a toda su pila de dependencias, con bastantes garantías poder desplegarse y ejecutarse en cualquier entorno donde se encuentre instalada la plataforma de contenedores correspondiente. Por último, la aparición de servicios comerciales en la nube de tipo PaaS (*Platform as a Service*) basados en contenedores (por ejemplo, Google Cloud Platform¹ o RedHat OpenShift Container Platform²) ha consolidado una alternativa al uso de contenedores en instalaciones privadas que facilitado y extendido su uso.

Esta tendencia irá, previsiblemente, en aumento: una encuesta de Forrester, encargada por Dell, Intel y RedHat en 2017 [1] prevé un crecimiento del 80% en el número de aplicaciones desplegadas en contenedores para los próximos dos años. El *Global Perception Study* realizado por Cloudfoundry el mismo año [2], indica, por otro lado, que entre 2016 y 2017 ha habido un incremento significativo del número de empresas evaluando el uso de contenedores, pero no se ha producido un aumento sustancial de su uso efectivo y atribuye esta disparidad a las dificultades que presenta la administración de contenedores en producción. El análisis de DataDog [3] realizado en 2018 confirma la tendencia en el crecimiento de despliegues prevista por Forrester el año anterior. Es interesante destacar que este último informe muestra un porcentaje no despreciable de empresas que han abandonado la tecnología de contenedores.

1 <https://cloud.google.com/containers/>

2 <https://www.openshift.com/products/container-platform/>

Los estudios anteriores emplean metodologías distintas, por lo cual sus resultados no se pueden comparar directamente. Pero podemos concluir que, pese al mayor o menor crecimiento del empleo de contenedores, un número considerable de empresas está evaluando su empleo, pero no los emplea todavía, y cierto número de empresas abandona su uso.

No tenemos datos concluyentes que avalen esta percepción, pero creemos que probablemente el hecho de que las empresas evalúen detenidamente el uso de contenedores antes de emplearlos en producción o que algunas de ellas abandone su uso, se pueda atribuir a dos motivos: las dificultades de introducir un nuevo modelo de administración de sistemas y la complejidad que supone mitigar los riesgos que conlleva. El estudio de Portworx[4] realizado en 2017 apunta en este sentido el almacenamiento persistente como reto principal en la adopción de contenedores; en años anteriores se trataba de la seguridad.

Resulta evidente que los métodos que se emplean en Linux para crear, desplegar, interconectar y administrar los contenedores de aplicaciones no se ajustan particularmente bien a las prácticas tradicionales en la administración de sistemas.

En cuanto a la cuestión de la seguridad, el aislamiento de las cargas de trabajo ejecutadas en un contenedor con respecto a otros contenedores o con respecto al sistema anfitrión (host) está considerado, de forma bastante generalizada [5] [6], inferior al proporcionado por los hipervisores.

Un elemento que añade complejidad en la aplicación de controles de seguridad, es el hecho de que los contenedores en Linux no están soportados directamente como entidades definidas en el núcleo, a diferencia de otras implementaciones de contenedores, como OpenVZ o Solaris Zones. El núcleo de Linux ofrece solo un conjunto de mecanismos para la gestión de su seguridad: aislamiento mediante los espacios de nombres (namespaces), control de recursos (cgroups), control de privilegios (capabilities), y un marco genérico, LSM (Linux Security Modules) que soporta distintas implementaciones de control de acceso (MAC, Mandatory Access Control, RBAC Role Base Access Control) y un mecanismo de filtrado de llamadas al sistema (seccomp). La creación y gestión de los contenedores se realiza mediante utilidades en el espacio de usuario³, especialmente el llamado container runtime. Como consecuencia, el aislamiento, y en general, la seguridad de los contenedores depende de que estas utilidades del espacio de usuario gestionen adecuadamente las primitivas que ofrece el núcleo, por un lado, y las complementen con otros mecanismos implementados en el runtime. En la práctica, esto se traduce en que las exigencias de aislamiento y seguridad solo se pueden cubrir aplicando un conjunto diverso de mecanismos no siempre bien integrados entre sí.

Por último, un aspecto que añade confusión es el panorama complejo y en continuo cambio que se presenta en este campo, motivado, probablemente, por los intereses del mercado en este sector, y en menor medida por cuestiones puramente técnicas.

La literatura técnica proporciona un buen número de estudios detallados de distintos aspectos de la seguridad de los contenedores en general [7] [8]. En el caso particular de Docker, Bui [5] concluye que la seguridad que ofrece Docker de forma predeterminada es aceptable salvo por el nivel de aislamiento que proporciona, pero argumenta que esta debilidad se puede aminorar desplegando los contenedores sobre máquinas virtuales. Esta práctica, de hecho se ha generalizado: una buena parte, si no la mayoría de los despliegues de contenedores se ejecuta sobre

³ La parte de la memoria virtual que incluye las aplicaciones y procesos del sistema, por contraposición al espacio del núcleo, en el que se ejecuta el núcleo (*kernel*), módulos, y drivers.

máquinas virtuales para combinar la eficiencia de los contenedores con el mejor aislamiento que proporcionan los hipervisores [31]. El despliegue de contenedores sobre VMs, como es previsible, no garantiza la ausencia de riesgos.

1.2 Objetivos del Trabajo

Los objetivos de este trabajo son dos: por un lado pretendemos obtener una primera aproximación a los fundamentos sobre los que se basa la tecnología de contenedores que emplea Docker, así como una visión general de la tecnología misma, teniendo en cuenta los desarrollos realizados en este campo durante los dos últimos años. Con toda seguridad este objetivo requerirá más dedicación de la que en principio podría parecer, debido a la complejidad de algunos de los mecanismos en los fundamentos y al cambio acelerado y a la variedad de alternativas en la oferta de soluciones que se produce a más alto nivel, como ya se mencionó anteriormente.

En segundo lugar, intentaremos determinar, evaluando informalmente posibles alternativas, qué productos y configuraciones se ajustan mejor a los requisitos de nuestra infraestructura de pruebas para desplegar:

- Un sistema de detección de intrusiones a nivel de *host* (HIDS, *host-based intrusion detection system*). El empleo de HIDS en los hosts que alojan los contenedores, ha sido citado con frecuencia [10] como una de las medidas más efectivas para paliar un rango de amenazas (*kernel exploits* [11], uso de canales encubiertos para filtrar información [12]) en las plataformas de contenedores.
- Un sistema centralizado de recogida y gestión de alertas generadas en los contenedores y en los *hosts*.

1.3 Enfoque y método seguido

La parte práctica de este trabajo implica dos actividades diferentes. Por un lado, la oferta de productos aplicables, aunque solo consideremos los productos de código abierto, es considerable, y como en otros aspectos de esta tecnología, cambiante. La evaluación de herramientas consume tiempo, y en la mayor parte de los casos no produce resultados relevantes. Para este tipo de pruebas, con máquinas de "usar y tirar", usaremos máquinas virtuales en VirtualBox. Este producto ofrece una interfaz gráfica pulida, y al mismo tiempo permite distintas configuraciones de conectividad en red que permiten simular una instalación de producción. Pero, de modo más importante, esta herramienta facilita la evaluación de los productos, ya que en la mayor parte de los casos, éstos se encuentran disponibles en imágenes de discos que se pueden arrancar en VirtualBox, directamente, o mediante Vagrant.

1.4 Planificación del Trabajo

1.4.1 Identificación de tareas

Podemos establecer las siguientes tareas:

1. Seleccionar y estudiar la bibliografía necesaria para determinar el estado presente de la tecnología.
2. Redacción del borrador de memoria del TFM.

- 3 Determinar y configurar la infraestructura para efectuar las evaluaciones
- 4 Selección, instalación y configuración de herramientas específicas para ejecutar las pruebas.
- 5 Diseño y ejecución de las pruebas.
- 6 Consolidar resultados y redactar conclusiones.
- 7 Corrección y composición de la memoria del TFM.
- 8 Preparar la presentación del TFM.

1.4.2 Planificación temporal

Sería deseable hacer un cálculo de horas de dedicación a cada tarea, y cubrir el total de horas del TFM partir de una valoración de 25 a 30 horas por crédito ECTS, pero careciendo de referentes en los que basar las estimaciones, las asignaciones serían, en gran medida, arbitrarias. Por otro lado, resulta necesario fijar límites temporales, puesto que la entrega de las PEC son hitos fijados de antemano. Una solución de compromiso entre estos dos condicionantes podría ser emplear semanas como unidades de tiempo, según se indica en la tabla 1.

PEC	1			2			3			4								
Semana	19/09	24/09	01/10	08/10	15/10	22/10	29/10	05/11	12/11	19/11	26/11	03/12	10/12	17/12	24/12	31/12	07/01	14/01
Tarea 1	■	■	■															
Tarea 2				■	■	■	■	■	■	■	■	■						
Tarea 3				■	■	■	■	■										
Tarea 4							■	■	■									
Tarea 5								■	■	■								
Tarea 6										■	■	■						
Tarea 7													■	■	■			
Tarea 8															■	■		
Entrega PEC																		
Presentación																		
Defensa																		

Tabla 1: Planificación de tareas

1.5 Resultados previstos al terminar el trabajo

No pretendemos, ni estamos en condiciones de obtener un producto tangible como resultado de este trabajo, tal como una podría ser una comparativa rigurosa de productos: ello requeriría al menos cierta familiaridad con la materia de estudio. El único resultado previsto es obtener un primer acercamiento a la tecnología de contenedores de aplicaciones y a los de sistemas de detección de intrusiones, además de esta memoria.

1.6 Estructura de esta memoria

Este trabajo contiene dos partes claramente diferenciadas, correspondiente a los dos objetivos fijados anteriormente. En la primera, correspondiente a los capítulos segundo y tercero, realizaremos una revisión de la tecnología. El capítulo segundo tratará de los fundamentos de seguridad sobre los que se basan los mecanismos implementados en Docker. Consideramos que este capítulo es necesario para adquirir los conocimientos elementales que nos permitan comprender las recomen-

daciones de seguridad recogidas en las guías de buenas prácticas que revisaremos en el capítulo tercero, especialmente el *CIS Docker Benchmark for Docker CE* y algunas publicaciones del NIST.

En la segunda parte de este trabajo, que se recoge en el capítulo cuarto, trataremos de documentar una evaluación en nuestro entorno de pruebas uno o varios sistemas de detección de intrusiones a nivel de host y un sistema centralizado de recogida de logs.

2 Contexto de la tecnología

2.1 Repaso de la nomenclatura de seguridad

En este trabajo usaremos la nomenclatura siguiente:

Los *sujetos* son entidades que realizan acciones. De modo informal, podemos identificar un sujeto con un usuario (una persona); pero los usuarios solo pueden ejecutar acciones a través de un agente, esto es, un proceso. Podemos considerar que un sujeto es una abstracción de un proceso.

Los *objetos* son los recursos a los cuales se accede. En el caso más habitual en Linux, un objeto es un elemento de un sistema de ficheros (un fichero, directorio, etc.), pero un objeto puede ser cualquier otro tipo de recurso: una zona de la memoria virtual, una interfaz de red, etc. Un proceso puede ser un objeto al que se accede desde otro proceso que actúa como sujeto, como ocurre en el caso, por ejemplo en la ejecución de un programa que genera la traza de llamadas al sistema de otro programa en ejecución.

La *matriz de acceso* es una tabla con una fila por cada sujeto y una columna por cada objeto. La celda para unas determinadas fila y columna indica el modo de acceso del sujeto y objeto correspondiente. La matriz no se implementa como un array de dos dimensiones, porque la mayoría de las celdas están vacías. Se pueden emplear dos alternativas: almacenar los modos de acceso por filas, registrando para cada sujeto una lista de los objetos y los modos de acceso permitidos, o bien almacenar los modos de acceso por columnas, registrando para cada objeto una lista de los sujetos que tienen acceso al objeto, y los modos de acceso permitidos para cada sujeto. En el primer caso, hablamos de un sistema basado en *listas de capabilities*; en el segundo, de un sistema basado en *listas de control de acceso (ACLs Access Control Lists)*.

Entre los sistemas operativos más usados actualmente, solo FreeBSD proporciona una implementación de listas de *capabilities*. En Linux, el mecanismo de control de acceso descrito en *capabilities(7)*, y al que nos referiremos más adelante, no guarda relación con las *capabilities* que acabamos de describir.

El modelo de control de acceso más habitual es el llamado *discrecional (DAC, Discretionary Access Control)*. Este modelo permite que el propietario de un objeto modifique de forma arbitraria el modo de acceso del objeto. Normalmente, el propietario de un objeto es el sujeto que lo ha creado. Los sistemas basados en ACLs como Windows, Unix, Linux o VMS emplean este modelo. Un problema específico de los sistemas Unix, y que heredó Linux es que, independientemente de las limitaciones del control de acceso DAC, este tipo de control de acceso solo se aplica de forma sistemática a los objetos de un sistema de ficheros. No es posible, por ejemplo, controlar el acceso a una interfaz de red aplicándole una ACL.

El control de acceso obligatorio (*MAC, Mandatory Access Control*) designa un conjunto de modelos de control de acceso en los cuales los permisos de un sujeto sobre un objeto están determinados mediante una *política de control de acceso* que el usuario no puede modificar (aunque más adelante matizaremos este aspecto). Así, por ejemplo una determinada política MAC puede impedir que el propietario de un objeto conceda sobre éste permisos de escritura a otros sujetos.

Existen varios modelos MAC. En el contexto de este trabajo son relevantes los que aplica SELinux, principalmente *TE (Type Enforcement)*, y especialmente cuando se aplica en un entorno de contenedores, *MCS (Multi-Category Security)*. TE es un modelo clásico documentado en la literatura académica. MCS, sin embargo es un modelo reciente, pero basado en el modelo más antiguo: *MLS (Multi-Level Security)*. En los dos apartados siguientes describiremos los dos modelos históricos MAC: (MLS y TE), y el nuevo modelo MCS.

MLS (Multilevel Security)

Las implementaciones basadas en el modelo MLS asignan a los sujetos y objetos una etiqueta con dos atributos. El primero correspondiente a su sensibilidad, que es una clasificación jerárquica. El segundo atributo es un conjunto, esto es, un atributo multivalorado, de categorías. Las categorías son clasificaciones no jerárquicas, que se usan para agrupar los objetos según algún criterio arbitrario, como por ejemplo, proyectos o departamentos, de un modo similar a como se emplean los grupos en las implementaciones habituales de modelos DAC. Los modos de acceso se establecen aplicando alguna política predefinida que compara las etiquetas del sujeto y el objeto. Todo el modelo MLS se basa en esta etiqueta, que se denomina de cualquier forma según la fuente. En adelante emplearemos la expresión "nivel de seguridad" para referirnos a este tipo de etiqueta compuesta de una sensibilidad y un conjunto de categorías.

En Linux el modelo MLS se implementa como una política de SELinux. El "nivel de seguridad" que acabamos de describir corresponde al cuarto campo o atributo de un *contexto* SELinux (definiremos *contexto* más adelante). En este trabajo no haremos más referencias a los modelos MLS. La mencionamos únicamente porque es la base sobre la que se construye una política diseñada específicamente para la gestión de contenedores, *MCS (Multi-category Security)*, que describimos más adelante, y que reutiliza el campo "nivel de seguridad" de un *contexto* SELinux de un modo distinto.

2.1.1.1 TE (Type Enforcement)

En estos modelos se etiquetan los objetos y sujetos; los modos de acceso de sujeto a objeto, y de sujeto a sujeto se especifican en dos matrices. Las etiquetas asignadas a los sujetos se denominan *dominios*, y las de los objetos, *tipos*. La implementación de TE usada en Linux (la política de SELinux más empleada) simplifica el modelo TE: asigna un tipo tanto a los sujetos como los objetos, y maneja una sola tabla de modos de acceso permitidos. Es decir, se añade una segunda matriz de acceso (la matriz MAC) además de la matriz que controla el acceso DAC: cada proceso tiene los identificadores habituales de usuario y grupo *uid, gid* y además, un identificador de seguridad (*sid*).

2.1.1.2 MCS (Multi-Category Security)

A diferencia de los modelos históricos, MCS es un modelo basado en una implementación específica para SELinux, y solo se describe de modo informal en algunos artículos en línea de autores involucrados en su desarrollo. Según Morris [13]:

"MCS is in fact an adaptation of MLS. It re-uses much of the MLS framework in SELinux, including the MLS label field, MLS kernel code, MLS policy constructs, labeled printing and label encoding/translation. From a technical point of view, MCS is a policy change, along with a few userland hacks to hide some of the

unwanted MLS stuff. There are a couple of kernel changes, but only relating to making it easy to upgrade to MCS (or MLS) without invoking a full filesystem relabel."

Las diferencias que señala Morris entre MCS y MLS son:

- MCS ignora el atributo de sensibilidad: todos los sujetos y objetos se etiquetan con una sola sensibilidad (s0), con lo cual desaparece la clasificación jerárquica.
- MCS emplea las categorías MLS asignándolas a los sujetos de modo similar a como se emplean los grupos suplementarios de un proceso en Linux, descritos en `credentials(7)`; un proceso puede tener asignado varios grupos suplementarios que no determinan el modo de acceso de un proceso en un momento dado: es el grupo efectivo del proceso (*egid*) el que se emplea para determinar la decisión. De modo similar, en MCS un proceso puede tener asignado un conjunto categorías, pero no ejecutarse en un nivel de seguridad que incluya todas las categorías.
- MCS no proporciona protección alguna frente a software dañino, ni errores o abusos por parte de los usuarios. Esto es tarea de los controles de acceso DAC y de TE. MCS solo proporciona una decisión de acceso después de que se hayan pasado los controles de acceso DAC y TE.

MCS es útil en las siguientes situaciones [14] :

- Confinar VMs que se ejecutan sobre el mismo hipervisor: las categorías permiten que cada VM se ejecute en su propio dominio (*contexto de seguridad*).
- Confinamiento de contenedores en entornos multi-inquilino (*multitenant*), por ejemplo, en servicios *PaaS* como OpenShift.

2.2 Virtualización

En términos generales, la virtualización consiste en la implementación de recursos mediante software, aunque puede emplear recursos adicionales a nivel de firmware o hardware. Es una técnica que se puede aplicar en diversos contextos. Así, por ejemplo, en los sistemas operativos se han empleado desde hace décadas mecanismos de gestión de memoria virtual, que permiten que un proceso pueda emplear un espacio de direcciones mayor que la que ofrece el hardware donde se ejecuta. En el contexto de este trabajo, nos referiremos a virtualización de recursos a un nivel de abstracción mayor: máquinas virtuales (VMs) y contenedores.

En las secciones siguientes se presenta una revisión de hipervisores y contenedores. Estas dos tecnologías se han presentado a menudo como competidoras, si atendemos al número de publicaciones dedicadas a comparar su rendimiento [15] [16][17][18] [19]; sin embargo, en la práctica deberían considerarse complementarias, atendiendo al uso extendido de desplegar plataformas de contenedores sobre máquinas virtuales.

2.2.1 Hipervisores

Un hipervisor o VMM (*Virtual Machine Monitor*) es un componente implementado normalmente en software que permite crear y ejecutar instancias de máquinas virtuales, mediante la virtualización del hardware, y posiblemente, otras técnicas adicionales, como puede ser la traducción dinámica de código. Una máquina virtual creada sobre un hipervisor puede ejecutar un sistema operativo completo, incluyendo el núcleo, por lo que es posible ejecutar máquinas virtuales con distintos sistemas operativos (por ejemplo, Windows y Linux) sobre el mismo hardware.

Desde Goldberg [24], los hipervisores se han clasificado en dos tipos en función del entorno donde se ejecutan:

Tipo 1: Se ejecuta directamente en la CPU del ordenador anfitrión. También se denominan hipervisores nativos o *bare-metal hypervisors*. El hipervisor es un micronúcleo que se encarga de las operaciones esenciales del núcleo, como la planificación de tareas o procesos en la CPU (*scheduling*), manejo de interrupciones y gestión de memoria, y delega el resto de operaciones a los núcleos de las máquinas virtuales.

Tipo 2: Se ejecutan sobre el sistema operativo del ordenador anfitrión. Suelen componerse de dos partes: una que se ejecuta como un módulo del núcleo del sistema operativo anfitrión, y otra que se ejecuta como procesos a nivel de usuario en el sistema operativo anfitrión. La planificación de tareas se ejecuta en el núcleo del sistema anfitrión.

En la literatura aparecen con frecuencia los términos *bare metal hypervisor* y *hosted hypervisor* como sinónimos de los hipervisores de tipo 1 y 2 respectivamente. En la figura 1 [21] se muestra un esquema de ambas arquitecturas.

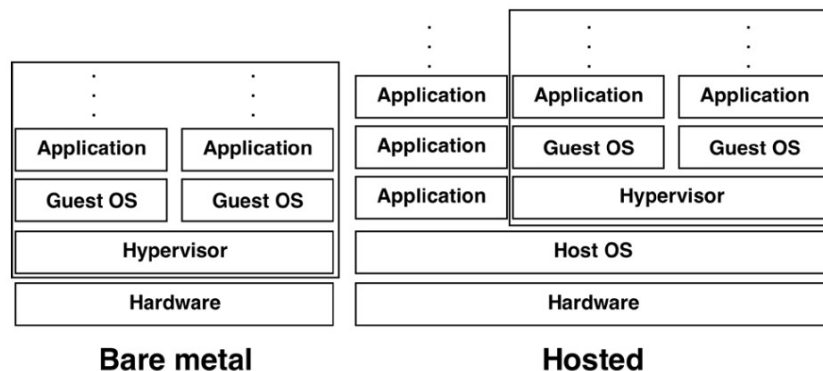


Figura 1: Hipervisores de tipo 1 (*bare metal*) y tipo 2 (*hosted*).

Algunos ejemplos de hipervisores empleados actualmente en arquitecturas x86 son:

- VMWare ESX, un hipervisor comercial de tipo 1 lanzado en 2001.
- Xen: un hipervisor de tipo 1 desarrollado en la Universidad de Cambridge en 2003.
- KVM: un hipervisor de tipo 2 para Linux desarrollado por Qumranet, una empresa adquirida por RedHat en 2008. Para su ejecución, el hipervisor

emplea un módulo del núcleo y un proceso en modo usuario (QEMU) para realizar traducción dinámica de código.

- Hyper-V: un hipervisor de tipo 2 incluido en todas las versiones de Windows Server desde 2008.

2.2.2 Contenedores de sistemas

Mientras que la función de un hipervisor es, de forma algún modo, hacer particiones de una máquina, la tecnología basada en contenedores de sistemas crea particiones de una instancia del sistema operativo. Cada partición mantiene la apariencia de ser una máquina independiente, aunque todas comparten el mismo núcleo del sistema operativo. Esto permite ejecutar con una sola instancia del núcleo del sistema operativo múltiples aplicaciones aisladas entre sí.

La técnica empleada en los contenedores tiene su origen en la llamada al sistema chroot(2), presente en Unix desde los años setenta, y que permite que un proceso cambiar el directorio raíz que se ha usado para arrancar el sistema por otro directorio. Jails [22], implementado en FreeBSD al final de los años noventa, emplea una técnica similar a chroot para virtualizar el sistema de ficheros de un conjunto de aplicaciones (esto es, modificar el espacio de nombres del sistema de ficheros visible desde esas aplicaciones), pero añade medidas similares para los espacios de nombres de los procesos y los recursos de red.

En 2004, una funcionalidad similar, denominada "zonas" [23] apareció en Solaris 10. Las zonas de Solaris 10 mejoraron el aislamiento de las particiones usando un esquema de agregaciones de recursos (*resource pools*), mediante el cual se permite asignar un conjunto de recursos (por ejemplo, memoria o CPUs) para que una partición lo use de forma exclusiva. De este modo se impide que la carga de trabajo de una partición acapare los recursos de la máquina.

Aunque en Linux, la funcionalidad para soportar contenedores ha tenido varias implementaciones, el primer proyecto que obtuvo una difusión apreciable fue OpenVZ⁴. Esta implementación usa un núcleo modificado de Linux cuyo código no se ha integrado en el núcleo oficial, aunque posteriormente se ha ido alineando con las funcionalidades ofrecidas en el núcleo estándar de Linux. La implementación de contenedores soportada directamente en el núcleo oficial se produjo en 2014 con LXC⁵, basado en el trabajo desarrollado en Google desde 2006 [24].

La figura 2 [25] muestra un esquema de la arquitectura de las zonas de Solaris, pero puede ilustrar las implementaciones de otros los otros sistemas sustituyendo las etiquetas de "zonas" por "jaulas" o contenedores. La zona global de Solaris se corresponde aproximadamente con lo que en el entorno de Linux se denomina de forma algo imprecisa el "host"; esto es, el entorno de ejecución desde el que se crean y administran los contenedores.

4 <http://www.openvz.org>

5 <http://www.linuxcontainers.org>

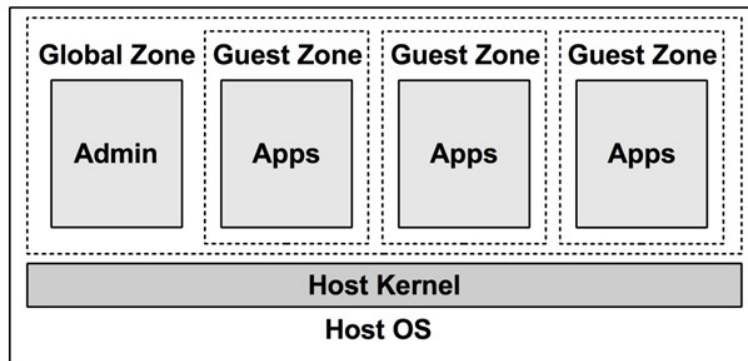


Figura 2: Virtualización a nivel de sistema operativo.

2.2.3 Contenedores de aplicaciones.

El patrón de uso habitual de las tecnologías de contenedores a las que nos hemos referido más arriba en proporcionar un entorno de ejecución similar al que se obtiene con una máquina virtual. Es decir, emplear un contenedor como una alternativa al uso de una VM ejecutada sobre un hipervisor. En el caso de Linux, caso de uso normal con LXC es iniciar que el contenedor ejecute `init(1)` o `systemd(1)`, y los procesos habituales: `bash`, `sshd`, etc.

La tendencia más reciente es, sin embargo, emplear contenedores de aplicaciones. En este modelo, el patrón de uso recomendado se orienta a los microservicios: cada cada contenedor ejecuta un grupo de aplicaciones o servicios determinados en función de alguna política (por ejemplo, segregando los grupos en función del tipo de aplicación, o de los consumidores de éstas), en vez de un entorno de ejecución completo que simule una VM. Es importante señalar que en Linux, la principal diferencia entre un contenedor de aplicaciones y un contenedor de sistemas es su patrón de uso: el núcleo proporciona los mismos mecanismos para su creación en ambos casos.

2.2.3.1 Docker

El uso de contenedores de aplicaciones ha venido impulsado por Docker. Dejamos para más adelante aclarar qué se entiende por Docker actualmente, puesto que este nombre ha sufrido cierto abuso. Nos referimos como Docker en este momento como la plataforma de contenedores implementada en el lenguaje *Go* y liberado en 2013. Las principales novedad que aportó este proyecto el concepto de *imagen* de un contenedor, similar en muchos aspectos a una fichero de alguno del los sistemas de paquetería (`rpm`, `deb`) de Linux.

Una imagen de contenedor es en esencia una jerarquía de directorios correspondiente a un sistema de ficheros raíz, archivadas en formato `tar(1)`. El archivo `tar` incluye, además del sistema de ficheros raíz un fichero de configuración con algunos metadatos. A veces se usa el término *bundle* para llamar a este archivo.

Las imágenes se distribuyen como sistemas de fichero de solo lectura. En el caso más simple, se trata de un solo sistema de ficheros, y se conoce como una *imagen base*. Las modificaciones que se necesiten aplicar a la a la imagen base se añaden como capas usando alguna implementación de un sistema de ficheros en capas

(*stacked filesystem*), o alguna otra alternativa de las que revisaremos más adelante. Este esquema permite compartir capas de solo lectura entre varios contenedores, manteniendo la apariencia de que los sistemas de ficheros están montados en modo lectura/escritura, limitando, de este modo el espacio de almacenamiento usado por las imágenes⁶.

Al incluir todas las dependencias de una aplicación en una imagen de contenedor se evitan los problemas habituales que aparecen cuando los entornos de desarrollo y explotación no son idénticos. Al haber una separación completa entre la ejecución de un contenedor y la creación de su contenido, la imagen se puede crear en cualquier plataforma de desarrollo, por ejemplo, Windows o MacOS, y posteriormente desplegarse en un servidor Linux, que posiblemente situado en una plataforma *PaaS (Platform as a Service)*. Se eliminan así las dependencias de aplicaciones o bibliotecas instaladas en el *host*; la única dependencia radica en el núcleo.

2.2.3.2 El ecosistema de Docker

A raíz de la popularidad de Docker ha aparecido un considerable número de proyectos relacionados, que introducen novedades en la tecnología, o proporcionan implementaciones alternativas. De igual modo, muchos proyectos desaparecen con la misma rapidez que aparecen otros nuevos. Es posible que la velocidad a la que producen estos nuevos desarrollos y el panorama en continuo cambio de las arquitecturas de los productos ya existentes sea la responsable del confuso vocabulario usado en estas tecnologías, donde a falta de definiciones claras, se emplean metáforas como *ciclo de vida* y *ecosistema*. En palabras de Ernst *et ál.* [26]

"a major issue of this ecosystem, which is mainly driven by open source projects, is the lack of clear functional demarcation of different components. In fact, capabilities and scopes of existing projects overlap. Similarly, we found terminology in literature to be inconsistent."

En particular, el nombre Docker se ha venido empleando con distintos significados. Cuando apareció en 2013, la situación estaba bien definida: se trataba de un proyecto de software libre implementado en el lenguaje *Go*, y desarrollado por un proveedor de *PaaS* llamado dotCloud. En la actualidad, Docker es una empresa (Docker Inc., una marca registrada) que comercializa varios productos servicios, y promociona varios proyectos de software libre relacionado con la tecnología de contenedores.

En el resto del trabajo, emplearemos el término Docker de modo informal para referirnos a un plataforma de gestión de contenedores compuesta de tres componentes principales:

- *Engine, daemon*: El servidor que gestiona el "ciclo de vida" de los contenedores: creación de imágenes, ejecución de contenedores, administración de red, etc. Se comunica mediante una API REST con el *container registry* para la carga y descarga de imágenes.
- *Docker client*: `docker(1)`. Es la interfaz estándar, en de modo línea de órdenes, para acceder a los servicios del *daemon*. El cliente Docker envía las peticiones al Docker daemon mediante una API HTTP.

⁶ Algunos autores consideran que el beneficio obtenido no justifica la complejidad que introducen en la creación y gestión de imágenes. Véase, por ejemplo el *microcontainer manifesto*

- *Docker registry*: Es el *container registry* al que nos referíamos anteriormente, denominaremos a partir de ahora simplemente "registro". El registro es repositorio de imágenes de contenedores. Puede ejecutarse desde una plataforma SaaS, por ejemplo Docker Hub, Oracle Container Registry, RedHat Registry, o instalarse como un servicio privado sobre la infraestructura del usuario. El Docker Registry envía o recibe las imágenes del Docker Engine (aquí el Docker Engine funciona como cliente del Docker Registry, no es el Docker client el que se encarga de esto) mediante un protocolo definido en el *Docker Registry HTTP API V2* [27]

2.2.3.3 Alternativas a Docker:

Rkt [28], llamado anteriormente *Rocket* es una plataforma de contenedores alternativa a Docker, o más concretamente, un *container runtime*, desarrollado por CoreOS. A diferencia de Docker, Rkt parece seguir la tradición de Unix de usar herramientas especializadas en hacer una cosa bien y que se puedan combinar para realizar operaciones más complejas. Según sus creadores, las herramientas para descargar imágenes, instalar y ejecutar contenedores deben estar bien integradas, pero ser independientes; la distribución de imágenes debe soportar diversos protocolos, y los despliegues en entornos privados no deben requerir el acceso a un *registry*. CoreOS proporciona, de todos modos un *registry* para imágenes tanto de rkt como de Docker.

2.3 Mecanismos de soporte para la creación de contenedores en Linux.

En Linux, los contenedores puede considerarse, en cierto modo, una ficción presentada por herramientas que se ejecutan en modo usuario: como se señaló en la introducción, no existe en el núcleo ninguna estructura que corresponda a un contenedor. Un contenedor en ejecución es básicamente un conjunto de procesos aislados por espacios de nombres y con sus consumos de recursos limitados por cgroups. El aislamiento que se obtiene usando estas dos primitivas no tan bueno como sería deseable, debido a limitaciones de su implementación, por lo que se pueden emplear herramientas adicionales para mejorarla: filtrado de llamadas al núcleo mediante seccomp, privilegios de grano más fino que los que se proporcionan al usuario root, mediante capabilities, y control de acceso con políticas MAC.

En los apartados siguientes revisamos estos mecanismos.

2.3.1 Espacios de nombres (*namespaces*)

Un espacio de nombres es un mecanismo de confinamiento de objetos del sistema (una interfaz de red, un sistema de ficheros, etc.) entre grupos de procesos. Cada proceso se puede vincular con varios espacios de nombres de distinto tipo, de tal modo que el núcleo muestre una visión distinta de los recursos disponibles a cada grupo de procesos. Este mecanismo proporciona un modo sencillo de dividir un sistema entre dominios. Por ejemplo el directorio `/usr/lib` pueden indicar un objeto distinto en dos procesos que no compartan el espacio de nombres del sistema de ficheros.

El núcleo actual ofrece los siete tipos de espacios de nombres que se muestran en la tabla 2.

Cgroup	Directorio raíz de Cgroup
IPC	Recursos IPC de tipo System y colas de mensajes POSIX.
Network	Dispositivos de red, puertos, etc.
Mount	Puntos de montaje.
PID	Identificadores de procesos (pid).
User	Identificadores de usuarios y grupos (uid, gid).
UTS	Hombre de <i>host</i> (<i>Unix Timesharing System</i>)

Tabla 2: Espacios de nombres en Linux

El espacio *cgroup* permite que un proceso pueda tener una vista local al contenedor del punto de montaje del pseudo-sistema de ficheros de cgroups (*/sys/fs/cgroup*) y del fichero */proc/self/cgroup*. El espacio *IPC* aísla recursos de comunicación entre procesos: señales, *pipes*, memoria compartida, etc. Un espacio *network* contiene dispositivos virtuales de red separados, direcciones IP, puertos y tablas de encaminamiento IP. El espacio *mount* confina un conjunto de puntos de montaje de sistemas de ficheros: en distintos espacios los procesos tiene vista distintas de la jerarquía del sistema de ficheros. El espacio PID virtualiza los identificadores de procesos (PIDs). Cada proceso tiene dos identificadores: uno dentro de su propio espacio y otro global, que es único en el *host*. Los procesos de un contenedor solo pueden ver los procesos del mismo espacio PID. El espacio *user* aísla los identificadores (números) de usuario y grupo. De este modo, se permite asignar el usuario *root* interno de un contenedor a un usuario no privilegiado del *host*. Así, un proceso puede tener al mismo tiempo privilegios totales dentro de un contenedor, y carecer de ellos en el *host*. El espacio *UTS* permite que cada contenedor tenga su propio nombre de *host*; de este modo un contenedor puede como nodo independiente.

2.3.2 Control de recursos (cgroups)

Se suele llamar "grupos de control", o de forma abreviada, *cgroups*(7), al mecanismo que proporciona el núcleo de Linux para aplicarle a los procesos límites de uso de recursos como CPU, memoria, ancho de banda en las operaciones de E/S de dispositivos de bloques, etc.

Linux ya ofrecía el mecanismo tradicional de *rlimits* (por *resource limits*, normalmente llamados *ulimits*, por la orden *ulimit* que se emplea para establecer los límites de la shell). Los límites establecen para cada proceso el máximo número de ciertos recursos (por ejemplo, número máximo de subprocesos, tiempo de CPU) que pueden consumir, mediante la llamada *setrlimit*(2). Para cada recurso hay dos límites: uno "duro", y otro "blando". El límite duro actúa como máximo del límite blando. Solo un proceso privilegiado (esto es, que tenga la capability *CAP_SYS_RESOURCE*) puede cambiar el límite duro. Un proceso sin esa capability solo puede reducir, de forma irreversible, su límite blando.

El módulo *pam_limits*(8), permite establecer el control de recursos sobre para cada sesión que inician determinados usuarios o grupos, según se determine en el fichero */etc/security/limits.conf*. Sin embargo, *rlimits* no proporciona el control de grano fino que se necesita para asignar límites individuales a cada contenedor (entendiendo aquí como contenedor el conjunto de procesos). Su principal problema es que no

permite especificar límites para un conjunto de procesos o definir la acción que se debe realizar cuando se alcance un límite.

Aunque más arriba decíamos que *cgroups* es un mecanismo, en sentido estricto, un *cgroup* es un grupo de procesos a los que se les impone un conjunto de parámetros o límites. Estos grupos pueden formar jerarquías, de modo que un subgrupo herede los límites impuestos en el grupo superior. Los componentes del núcleo encargados de imponer los límites del *cgroup* se llaman "controladores de recursos", "subsistemas", o simplemente "controladores". Así por ejemplo, el controlador *blkio* impone los límites sobre un dispositivo de E/S de bloques a un *cgroup*, o el controlador *cpu*, el "ancho de banda" de CPU asignado a un *cgroup*.

Además de imponer límites en el uso de recursos, *cgroups* proporciona otras funcionalidades, entre ellas:

- prioridades en el uso de recursos, como por ejemplo la controladora *net_prio* para los interfaces de red.
- contabilidad del uso de los recursos, como en el caso de la controladora *cpuacct*.
- monitorización del *cgroup*, por ejemplo, la controladora *cpuacct*.
- suspender y reanudar todos los procesos del *cgroup* (controladora *freezer*).

Las controladoras presentes en un sistema se muestran como un pseudo-sistema de ficheros montado en `/sys/fs/cgroup`, que se puede consultar con las órdenes habituales (`ls`, `findmnt`, etc.) o bien usar la orden `ls -ls /sys/fs/cgroup` ofrece para obtener una salida más clara.

2.3.3 Union filesystems

La otra novedad que aportó Docker a la tecnología de contenedores es el uso de sistemas de ficheros contruidos a bases de capas, usando un *union filesystem* u otro mecanismo que proporcione una funcionalidad similar.

Un *union filesystem*, o *union mount filesystem* es un tipo sistema de ficheros lógico (en el sentido de que no se construye a partir un dispositivo físico de bloques) que agrupa varias jerarquías de directorios (ramas) y las muestra como una sola jerarquía. Las ramas tienen asignada una prioridad, que determina el orden que ocupan en la pila de capas. La única capa de lectura/escritura es la superior, pero el *union filesystem* permite dar la apariencia de que todos los ficheros que incluye son de lectura/escritura, gracias al empleo de técnicas de CoW (*Copy on Write*): cuando se necesita modificar un fichero situado en una capa de solo lectura, se hace una copia del fichero a la capa de lectura/escritura. A partir de ese momento la copia del fichero sustituye al fichero original en la jerarquía de ficheros que presenta el *union filesystem*. Las primeras implementaciones de *union filesystems* aparecieron en la década de 1990, entre ellas las de Plan9 y 4.4BSD-Lite [29]

En Linux se han sucedido varias implementaciones de *union filesystems*: *unionfs* [30] *aufs*⁷, *overlayfs* [31]. La implementación más reciente es *overlayfs2*. Una alternativa a los *union filesystems* son los sistemas de ficheros con soporte nativo de CoW como *btrfs* y *ZFS*. Otras alternativas son usar el *device-mapper*, o emplear directamente las funcionalidades de CoW que ofrecen los fabricantes de cabinas SAN (*Storage Area Networks*). La documentación oficial recomienda distintas alterna-

7 <http://aufs.sourceforge.net/>

tivas en función de la distribución de Linux y de la edición de Docker (Community, Enterprise).

2.3.4 Control de acceso obligatorio

El núcleo de Linux proporciona la infraestructura para varias implementaciones de MAC mediante LSM [32]. Las principales son SELinux y AppArmor. La elección entre una y otra vendrá determinada en gran medida por la distribución de Linux elegida: RedHat emplea SELinux; AppArmor se emplea en SuSE y Ubuntu. En las versiones recientes de Debian, AppArmor se emplea de forma predeterminada, pero se proporcionan paquetes para configurar SELinux si es necesario.

2.3.4.1 SELinux

SELinux se basa en la arquitectura de seguridad llamada Flask [33], implementada en el sistema operativo Fluke, un sistema operativo experimental desarrollado a finales de los años 90. Esta arquitectura separa la definición de política de seguridad del mecanismo que la aplica. En Flask, todos los atributos de seguridad asociados con un sujeto o un objeto se especifican en un *contexto de seguridad*, uno de cuyos atributos es el tipo TE (*Type Enforcement*). Un *servidor de seguridad* asocia un *identificador de seguridad* (*sid*, *security identifier*) a cada contexto de seguridad. En Linux, el servidor de seguridad es un subsistema del núcleo que indica si un acceso se permite o no, usando una matriz de acceso que indica los modos de acceso correspondiente a los *contextos* de los sujetos y los objetos y de acuerdo con las reglas que indica la *política* que se emplee. El servidor de seguridad no controla el acceso; únicamente emite una decisión. El encargado de hacer cumplir la decisión es el *policy enforcement server*, que además mantiene una matriz de acceso llamada AVC (*Access Vector Cache*) donde se encuentran almacenadas las decisiones previas del servidor de seguridad. Si la decisión no se encuentra en la AVC, el *policy enforcement server* le pide al servidor de seguridad que consulte su matriz de acceso.

Las políticas son conjuntos de reglas que determinan como se construye la matriz de acceso del servidor de seguridad. La política se compila a un formato binario que se carga en el *servidor de seguridad*. Los ejemplos más comunes de políticas empleadas son: *targeted*, *mls*, *standard*, *standard*, *minimum* [14].

La política de referencia⁸ es un conjunto estándar de políticas en formato fuente que se usa para construir políticas SELinux. Cada distribución de Linux con soporte de SELinux modifica la política de referencia para adaptarla a sus requisitos y distribuye su versión particular de la política de referencia.

⁸ <https://github.com/SELinuxProject/refpolicy/wiki>

2.3.4.2 Uso de SELinux en Docker

Docker, cuando emplea SELinux, usa TE para proteger el Docker Engine y el host de posibles ataques iniciados desde los contenedores. Todos los procesos que se ejecutan en los contenedores tienen asignado en su contexto de seguridad un tipo que proporciona un modo de acceso restringido (`svirt_lxc_net_t`, definido en el fichero de políticas `lxc_contexts`). La exigencia de ejecutar todos los contenedores con el mismo tipo es una limitación severa: hay que permitirle a `svirt_lxc_net_t` los máximos privilegios que pueda necesitar cualquier contenedor. Por ejemplo, aunque dos aplicaciones utilicen cada una un puerto distinto de red, `svirt_lxc_net_t` les permite a ambas comunicarse sobre todos los puertos posibles. MCS permite restringir el acceso entre distintas instancias del mismo tipo TE. El acceso se permite si TE lo autoriza y además, el sujeto y el objeto tiene la misma categoría en su contexto de seguridad. Se asignan distintas categorías a diferentes contenedores, permitiendo así el aislamiento entre contenedores con el mismo tipo TE.

2.3.4.3 AppArmor

Apparmor es un mecanismo de monitorización que mantiene para cada aplicación protegida una lista de todos los trayectos de ficheros y directorios a los que tiene acceso, y que impide el acceso de la aplicación a los trayectos no incluidos en la lista. Es más fácil de usar que SELinux, pero mucho más limitado: emplea trayectos de un sistema de ficheros en vez de un identificador de seguridad (`sid`) inmutable para tomar las decisiones de acceso. El empleo de un trayecto de un sistema de ficheros como identificador es poco fiable: mover una aplicación de `/bin` a `/usr/bin`, por ejemplo, puede modificar el resultado de la decisión. No existen sujetos u objetos como entidades a nivel de sistema; tan solo se consideran los objetos de un sistema de ficheros. Las políticas se especifican con referencias a objetos individuales, sin que sea posible agruparlas en clases.

2.3.5 Capabilities

Con independencia del modelo DAC o MAC empleado, los sistemas operativos habituales emplean algún tipo de mecanismo para permitir realizar operaciones privilegiadas sin tener en cuenta el control de acceso MAC o DAC. Linux sigue el modelo clásico de Unix: los procesos que se ejecutan en el contexto del UID 0, (asignado tradicionalmente al nombre de usuario `root`) no están restringidos por las reglas DAC/MAC. El resto de los procesos se ejecutan sin ningún tipo de privilegio.

El borrador POSIX 1003.1e [34] fue un intento de dividir el privilegio absoluto del UID 0 en un conjunto de privilegios separados. El borrador no llegó a convertirse en un estándar, pero sirvió de base para la implementación del sistema de privilegios de Linux descrito en `capabilities(7)`.

2.3.6 Otros mecanismos de confinamiento: *seccomp*

El núcleo de Linux proporciona una facilidad llamada *Secure computing mode* (*seccomp*), para limitar las posibles llamadas al sistema que se pueden invocar desde un proceso. Linux ofrece un considerable número de llamadas al sistema: en el equipo que estamos usando para escribir este documento, `syscalls(2)` lista 412 llamadas.

Esta funcionalidad de filtrado de llamadas se puede emplear como un mecanismo de confinamiento de los contenedores: Docker proporciona la posibilidad de estab-

lecer el listado de las llamadas que se permiten invocar desde un contenedor definiéndolas en un fichero *json* llamado *perfil seccomp*. El perfil que proporciona Docker con la configuración predeterminada⁹ intenta encontrar un equilibrio entre protección y facilidad de uso: la protección es moderada y una compatibilidad con las aplicaciones es alta, o al menos así lo indica la documentación oficial de Docker. Traducido a cifras, esto significa que inhabilita 44 llamadas, lo cual supone alrededor del 10% de las 412 llamadas que referíamos más arriba.

Si se quiere restringir la lista, se puede crear un perfil *seccomp* alternativo y seleccionarlo al invocar `docker-run(1)` mediante la opción `--security-opt=<perfil.json>`. Si por el contrario, se quiere ignorar la funcionalidad de filtrado de llamadas, se puede usar la opción `--security-opt=unconfined`.

2.4 Clusters de contenedores y orquestación

El modelo de uso de contenedores orientado a microservicios implica con frecuencia la necesidad de mantener un número considerable de clusters de contenedores. En muchos casos, estos clusters puede estar dispersados geográficamente, y deben cumplir unos requisitos mínimos de disponibilidad y tolerancia a fallos. La complejidad de mantener una infraestructura de este tipo ha provocado la aparición de unos componentes de software que faciliten la integración y la gestión de contenedores. En general, el componente que proporciona varias o todas las funcionalidades necesarias se conocen como *orquestadores* de contenedores.

La terminología, como es habitual en este campo, es lo suficientemente elástica como para que no se puedan encontrar dos definiciones equivalentes de *orquestador*; vamos a usar como base la lista de funcionalidades que le asigna Khan[35], porque excluye la posibilidad de dejar algo fuera.

Así, en un sentido amplio, el orquestador se encarga de:

1. Gestión del estado del cluster y planificación. Se requiere mantener una visión global del estado del cluster para poder planificar las cargas de trabajo en función de los recursos libres del cluster y del horario. Esto, la planificación de colas de trabajo de un entorno convencional, pero teniendo en cuenta la complejidad que el cluster de contenedores.
2. Proporcionar alta disponibilidad y tolerancia a fallos: mecanismos para la habilitar detección de fallos y evitar los puntos únicos de fallos, balanceo de carga.
3. Seguridad. Integración de utilidades para verificar la integridad de las imágenes, servicios de gestión de identidades y gestión de accesos, etc.
4. Simplificar la gestión de la red: gestión dinámica de direcciones y puertos, comunicación entre contenedores situados en distintas máquinas a través de túneles
5. Habilitar el descubrimiento de servicios (*service discovery*). En el modelo tradicional, los servicios se asocian de forma relativamente estática a direcciones y puertos que se resuelven mediante alguna combinación de DNS, LDAP y URIs HTTP. En el modelo de contenedores, la asociación entre nombres de servicios y direcciones no puede ser persistente, entre otros motivos por la posibilidad de *autoscaling* (creación de dinámica de nuevas ins-

⁹ <https://github.com/moby/moby/blob/master/profiles/seccomp/default.json>

tancias de contenedores en función de la demanda). Se requiere por lo tanto un *registro de servicios*, esto es, un componente que vincule los servicios proporcionados en los contenedores con su localización en la red. Algunos ejemplos de servicios son *etcd*, *Consul* y *Zookeeper*.

6. Hacer posible el despliegue continuo. Cuando se emplea el modelo CI/CD, la plataforma de orquestación puede proporcionar mecanismos para integrar la gestión de herramientas como Jenkins.
7. Monitorización, tanto de la infraestructura donde se ejecutan los contenedores como de la actividad de los mismos.

La plataforma de orquestación de contenedores nativa de Docker es *Docker Swarm*. La que está recibiendo más atención, sin embargo, es *Kubernetes*, que desarrolló inicialmente Google, y actualmente es un proyecto de software libre¹⁰. Como consecuencia del éxito de Kubernetes, Docker Inc. lo ofrece como alternativa a Docker Swarm en muchos de sus productos, aunque no en la versión CE para Linux.

En algunos casos, las plataformas de orquestación se ofrecen como *SaaS*. Así, los servicios de computación en la nube de Amazon, AWS (*Amazon Web Services*) ofrecen la plataforma de orquestación ECS (*Elastic Container Service*). Por su parte, el servicio de computación Azure de Microsoft proporciona soporte de Swarm y Kubernetes como alternativas a su servicio nativo de orquestación.

2.5 Plataformas específicas para el despliegue de contenedores

Hasta ahora, hemos asumido que Docker es un paquete instalado en alguna distribución de Linux como RedHat, Debian o Ubuntu. Existen, sin embargo distribuciones destinadas de modo específico a ejecutar contenedores, que incluyen solo los componentes indispensables para funcionar como un *host*. Este enfoque ofrece ventajas en al menos un par de escenarios. Para desarrollo, puede ser conveniente manejar imágenes de VM desechables con VirtualBox o VMWare, posiblemente empleando la herramienta *docker-machine*.¹¹ En este caso, una distribución específica para contenedores ofrece la ventaja de su menor tamaño. En el caso de un entorno de explotación, una distribución mínima ofrece una menor superficie de ataque y disminuye las necesidades de parches.

Un ejemplo del primer caso es, o ha sido *boot2docker*¹², una distribución para desarrollo que se ejecuta completamente desde RAM. Sin embargo su desarrollo se ha abandonado; Docker Inc. recomienda ahora la migración a sus productos *Docker for Windows* o *Docker for Mac*.

Para el caso de explotación existen varias posibilidades, entre ellas:

- CoreOS, una distribución a la que nos hemos referido anteriormente, puede ejecutar Docker, además de su runtime nativo, Rkt.
- RedHat (o Fedora o CentOS) Atomic Host, basados en el proyecto *Atomic*¹³. Este producto presenta algunas particularidades interesantes, como el uso de sistemas de ficheros inmutables (los únicos directorios que permiten modificaciones

10 <http://kubernetes.io>

11 <https://docs.docker.com/machine/overview/>

12 <https://github.com/boot2docker/boot2docker>

13 <http://www.projectatomic.io/>

son /etc y /var). Otra característica es que, en lugar de emplear un sistema de paquetería tradicional, las actualizaciones de software se realizan generando un nuevo sistema de ficheros raíz que reemplaza al anterior y que, en caso necesario permite deshacer la operación, y recuperar el sistema en el estado anterior a la actualización. Es decir, algo parecido a un sistema de control de versiones, pero aplicado a sistemas de ficheros en vez de ficheros individuales.

El entorno empresarial en este campo varía rápidamente. RedHat adquirió a principios de 2018 la empresa CoreOS Inc., lo cual supone el abandono de la plataforma Atomic. La próxima versión de CoreOS, según se ha anunciado¹⁴ será Fedora CoreOS; en este momento no se encuentra todavía disponible. Recientemente, IBM ha adquirido RedHat, aunque probablemente esta operación no introducirá más modificaciones en la oferta de productos¹⁵.

14 <https://coreos.fedoraproject.org/new/2018/06/20/welcome-fedora-coreos.html>

15 <https://www.forbes.com/sites/adrianbridgwater/2018/10/29/ibm-buys-red-hat-for-the-hybrid-cloud-factor/amp/>

3 Aspectos específicos de la Seguridad en contenedores Docker

3.1 Amenazas y vulnerabilidades

Teniendo en cuenta la complejidad del ecosistema de Docker, o dicho de otro modo, la complejidad que impone por un lado, la combinación de un elevado número de componentes (los contenedores en sí mismos, los hosts donde se ejecutan, el origen de las imágenes, los elementos virtualizados de red), el software auxiliar para su administración (registros, orquestadores, sistemas de autenticación, gestión de autorizaciones), y la posible inclusión de una tubería CI/CD, no es de extrañar que la literatura sobre este asunto sea abundante.

El objetivo de este trabajo restringe el espectro de amenazas que debemos considerar: no nos ocuparemos de las amenazas introducidas durante el desarrollo, aunque sean la primera preocupación que debería tenerse en cuenta usando un enfoque global. Nuestra percepción es que el éxito de Docker tiene mucho que ver con el enfoque orientado al desarrollador que desde el primer momento adoptó, a diferencia de otras implementaciones de contenedores anteriores. El movimiento DevOps incluye un buen número de elementos que trastocan los principios tradicionales de separación de responsabilidades entre desarrollo y operaciones, fijado durante décadas en el folclore de cualquier centro que mantenga equipos de desarrollo y explotación para dar servicio a una población de usuarios de cierto tamaño. Pero no son solo tradiciones que se mantienen por la inercia. Sin ir más lejos, según se recoge en el ENS (*Esquema Nacional de Seguridad*), que es una norma de obligado cumplimiento en las Administraciones Públicas y el Sector Público Institucional, la separación de funciones y responsabilidades en estas áreas en uno de los controles de seguridad exigidos. Por otro lado, la normativa legal de protección de datos, prohíbe, por ejemplo, el desarrollo de aplicaciones o la ejecución de pruebas de aplicaciones que accedan a datos personales que no hayan sido anonimizados (RLOPD [36]) o pseudonimizados (RGPD [37]).

El replanteamiento de responsabilidades laborales y el cumplimiento legal no impiden que el modelo CI/CD pueda aplicarse en los entornos que hemos mencionado. Pero obligan a aplicar los controles de seguridad en un mayor número de puntos, y a hacerlo de modo continuo, por la agilidad que impone el modelo entre los entornos de desarrollo, pruebas y explotación, y la frecuencia de los despliegues.

Otro aspecto citado con frecuencia es la fiabilidad de las imágenes obtenidas de los registros. La facilidad que proporciona Docker para descargar una imagen de un registro remoto y crear un contenedor a partir de ella facilita al mismo tiempo la propagación de las vulnerabilidades incluidas en la imagen. Docker empezó a ofrecer en 2016 un servicio de escaneo de imágenes para registros, tanto para Docker Hub (con algunas limitaciones) como para DTR (*Docker Trusted Registry*, el producto de Docker para desplegar registros privados). Existen también alternativas de otros vendedores. Aún así, Shu *et ál.* [38] realizaron en 2017 un estudio de más de trescientas mil versiones de imágenes, tanto oficiales como comunitarias, y entre otros resultados encontraron que el 80% de las imágenes, incluyendo las oficiales, contenían al menos una vulnerabilidad de severidad alta.

Volviendo al objeto de este trabajo, centraremos nuestra atención en cuestiones de seguridad muy determinados de nuestro entorno, lo cual significa que ignoraremos la mayor parte de las posibles vulnerabilidades del ecosistema y

En el ámbito de este trabajo, requieren especial atención los siguientes aspectos:

1. Seguridad del host, tanto si es físico o una VM donde se ejecutan los contenedores. Esto por sí mismo es una cuestión que afecta a cualquier máquina, tanto si ejecuta contenedores como si no, pero es necesario destacar su relevancia en el caso una máquina dedicada a ejecutar contenedores: no afectan las mismas vulnerabilidades a un servidor de DNS, que ofrece un único servicio durante años, que las que afectan a una máquina donde los servicios aparecen, se actualizan o desaparecen de forma dinámica.
2. Segregación del tráfico de red entre redes administrativas, de datos de aplicación, y de almacenamiento. Al igual que el punto anterior, éste principio no es exclusivo de los contenedores, pero en el caso de estos se necesitan tener en consideración aspectos nuevos, como el tráfico de red entre contenedores, y cómo maneja el demonio de Docker los dispositivos de red virtuales. Las facilidades que ofrece Docker para crear bridges, asignar direcciones IP de forma dinámica, traducción NAT, etc., son unas facilidades incuestionables en un entorno de desarrollo. En un entorno de producción, sin embargo, toda esta magia es más un estorbo que una ayuda, ya que se necesita tener en cuenta cómo afectan estas modificaciones a nuestra configuración establecida desde el sistema operativo. Muy probablemente se deberá descartar la configuración proporcionada por el demonio de Docker para imponerle la nuestra.
3. Aislamiento de contenedores con respecto al host. Ésta es la principal debilidad en cuanto a seguridad de Docker, corregida en gran medida desde que Docker empezó a ofrecer la posibilidad de usar espacios de nombres de usuario. El espacio de nombres de usuarios permite que el usuario root de un contenedor corresponda a un usuario no privilegiado del host, lo cual resultaba imposible en las primeras versiones de Docker. Sin embargo quedan todavía recursos globales del host para los cuales no hay un espacio de nombres disponible, como, veremos más adelante.
4. Aislamiento de contenedores entre sí. Este aspecto es especialmente importante en despliegues de contenedores sobre un mismo host en un entorno multi-inquilino (*multitenant*), en cuyo caso la solución que se suele aplicar es dedicarle una VM a cada inquilino. Pero incluso si los contenedores pertenecen al mismo usuario, es necesario intentar que cada contenedor ofrezca la menor superficie de ataque posible desde los otros contenedores del host, por razones evidentes.

Una buena parte de la atención dedicada en las publicaciones sobre seguridad de contenedores se ha dedicado a los dos últimos apartados de la lista anterior, que podríamos considerar uno solo: aislamiento entre contenedor y host, puesto que el aislamiento entre de dos contenedores entre sí consigue impidiendo el acceso a recursos que proporciona el núcleo del host. En última instancia, un contenedor en ejecución es un conjunto de procesos que se ejecutan sobre sobre el sistema operativo del *host*, y el objetivo es restringir el acceso de estos procesos a lo recursos del sistema operativo: sistemas de ficheros, dispositivos.

Como hemos descrito previamente, los *namespaces* y *cgroups* son los mecanismos en los que se basa el aislamiento de los contenedores. Los *namespaces* proporcionan la forma más efectiva de aislamiento. Así, por ejemplo, el namespace PID permite que los procesos ejecutados en un contenedor no sean visibles desde otro contenedor. Este mecanismo permite proporcionar además interfaces de red o sockets a cada contenedor. Pero el aislamiento proporcionado depende de que el recurso global pertenezca a uno de los namespaces que ofrezca el núcleo. Linux ofrece actualmente los siete namespaces descritos en el capítulo 2, pero hasta la versión 3.8 del núcleo de Linux, que introdujo el namespace "User", el usuario root de un contenedor era necesariamente el mismo usuario root del host. Las versiones recientes de Docker permiten emplear el namespace user para asignar un usuario root del contenedor a un usuario no privilegiado del host empleando la opción "--userns-remap" del docker daemon y alguna otra configuración adicional en el host.

Los *cgroups*, por otra parte, permiten mantener controlado el consumo de recursos del host que realiza cada contenedor, impidiendo de este modo que un contenedor acapare los recursos del host deteriore el rendimiento de otros contenedores o del host. Las opciones --cpu-quota y --cpu-period de docker create y docker run permiten limitar el consumo de CPU de un contenedor. El mecanismo tradicional de *rlimits* (o *ulimits*), está también disponible mediante la opción --ulimit.

Sin embargo, estos mecanismos por sí solos no garantizan un aislamiento infalible. Varias fuentes [7] [39] [6]) han destacado limitaciones de *namespaces* y *cgroups* como las siguientes:

- No existen espacios de nombres de seguridad: aunque es posible emplear los mecanismos MAC que proporciona el núcleo para reforzar el aislamiento entre contenedores, y entre contenedores y el host, los mecanismos MAC son recursos globales, por lo que no es posible imponer políticas MAC independientes para cada contenedor.
- No existe un espacio de nombres de dispositivos (los dispositivos de red están incluidos en el espacio "Network") que permita solo el acceso a los drivers necesarios, y que permitiría aislar los dispositivos visibles desde cada contenedor. No es posible, por ejemplo aislar un dispositivo de bloque. Esta limitación afecta también a los módulos del núcleo. Todos los módulos cargados en el núcleo del host son visibles desde todos los contenedores.
- Implementación incompleta de cgroups: No se han integrado todas las características que ofrece *rlimits* para imponer límites de consumo de recursos en la implementación de cgroups. No existe, además, un espacio de nombres para los cgroups.

Estas limitaciones de los namespaces son fuentes de vulnerabilidades documentadas y explotadas, como por ejemplo, éstas de 2017:

- Jian *et ál.* [40]proporcionan el código de un mecanismo de "escape" del contenedor (esto es, obtener una shell de root en el host) alterando los namespaces.
- Gao *et ál* [12]explotan el acceso a recursos globales en /proc y /sys para generar para ataques de denegación de servicio que se basan en producir cargas de trabajo diseñadas para generar para picos de consumo eléctrico lo suficientemente elevados como para activar los interruptores automáticos

de protección de las PDU (*Power Distribution Unit*) de los armarios de los servidores y provocar su parada.

Las limitaciones de los namespaces y cgroups como mecanismos de confinamiento requieren el empleo de mecanismos adicionales para paliar sus deficiencias, como se señaló anteriormente en el capítulo 2, y tal como se recoge las guías de buenas prácticas de seguridad para contenedores. Las contramedidas adicionales se basan en el empleo de MAC (SELinux o Apparmor), en filtrado de llamadas (seccomp) y en el uso de capabilities. Estos tres mecanismos se solapan en algunos aspectos y son difíciles de usar sin un buen conocimiento tanto de la tecnología empleada como de los requisitos de la aplicación que se ejecuta en el contenedor, por lo que en la mayoría de los casos no quedará más opción que usar las políticas predefinidas por la distribución del sistema operativo (en el caso de MAC) o de la versión de Docker (en el caso de seccomp y capabilities).

3.2 Guías de buenas prácticas.

Las guías de buenas prácticas de seguridad más relevantes son las del NIST y el CIS Docker Benchmark. Éste último es especialmente interesante por dos motivos. Primero, han colaborado en él ingenieros de Docker involucrados en su desarrollo desde el comienzo. En segundo lugar proporciona una lista de verificación que se puede comprobar de forma automatizada mediante el Docker Bench Security¹⁶. Además, a diferencia de las guías del NIST está hecha de forma específica para Docker, y propone medidas muy concretas.

La documentación oficial de Docker incluye también una sección llamada "Securing Docker EE and Security Best Practices"¹⁷ que incluye tanto recomendaciones de seguridad como propaganda comercial de su producto Docker EE. Las buenas prácticas que describe ya están incluidas en las dos guías que hemos comentado anteriormente.

El informe de Grattafiori para el grupo NCC "Understanding and Hardening Linux Containers" [Grattafiori2016] dedica un capítulo a las recomendaciones de seguridad habituales. Este informe se elaboró en 2016, lo cual es mucho tiempo para Docker; aunque algunos datos estén desfasados con respecto a la versión actual de Docker, la guía completa continúa siendo una referencia particularmente útil para entender los detalles de implementación de los mecanismos de seguridad aplicables a los contenedores.

3.2.1 CIS Docker Benchmark

Nos referimos aquí al CIS Docker CE Benchmark [41]; hay otra guía anunciada para la versión EE. Este documento tiene trescientas páginas, pero están mayormente vacías: el documento entero es una lista de verificación con sus contramedidas, que cubre los siguientes apartados:

1. Configuración del host:
2. Configuración del Docker daemon.
3. Ficheros de configuración del Docker daemon.
4. Imágenes de contenedores y *Build File*.

¹⁶ <https://github.com/docker/docker-bench-security>

¹⁷ <https://success.docker.com/article/security-best-practices>

5. Runtime de contenedores.
6. Operaciones de seguridad de Docker

3.2.2 Publicaciones NIST

NIST SP 800-190 (Application Container Security Guide) [42] y NIST IR 8176 (Security Assurance for Linux Application Container Deployments) [58] describen las principales amenazas de seguridad (que denominan riesgos) en el entorno de los contenedores de aplicaciones y proponen contramedidas para aminorarlas.

El propósito de NIST SP 800-190, según indica el documento,

"is to explain the security concerns associated with application container technologies and make practical recommendations for addressing those concerns when planning for, implementing, and maintaining containers. Some aspects of containers may vary among technologies, but the recommendations in this document are intended to apply to most or all application container technologies."

NIST SP 800-190 no menciona a Docker, pero solo considera la tecnología de contenedores de aplicaciones. NIST SP 800-190 identifica vulnerabilidades y contramedidas sin entrar en detalles específicos. Por otro lado, NIST IR 8176 es un complemento a NIST SP 800-190 que incluye contramedidas específicas para los contenedores de aplicaciones en Linux. Un tercer documento, la "NIST Guidance on Application Container Security" es una breve introducción a NIST SP 800-190 y NIST IR 8176.

NIST SP 800-190 identifica en su capítulo tercero las amenazas sobre los componentes enumeradas a continuación, y recomienda las contramedidas correspondientes en el capítulo cuarto. Los componentes estudiados son:

- Imágenes..
- Registro
- Orquestador
- Contenedor
- Sistema operativo

Las recomendaciones de NIST IR 8176 se puede resumir en los puntos siguientes:

- Para proporcionar autenticidad e integridad de la pila de componentes software de los contenedores (SO del host, runtime, contenedores) se debe emplear cadenas de confianza basadas en hardware, como por ejemplo TPM¹⁸ (*Trusted Platform Module*).
- Usar medidas de protección basadas en el hardware para aislar los contenedores entre sí y entre el núcleo, como por ejemplo la que proporciona el modelo de ejecución de la arquitectura Intel SGX¹⁹
- Emplear las facilidades del núcleo: namespaces, cgroups, capabilities, LKM para la protección del núcleo y de los contenedores entre sí.

18 https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Summary_04292008.pdf

19 <https://software.intel.com/en-us/sgx>

- Usar medidas de protección para el runtime, imágenes de contenedores, registro y las herramientas de orquestación.

En el entorno de ejecución habitual de contenedores sobre VMs, son aplicables las recomendaciones de las guías del NIST siguientes:

- NIST SP 800-125 - Guide to Security for Full Virtualization Technologies
- NIST SP 800-125A Rev. 1 - Security Recommendations for Server-based Hypervisor Platforms
- NIST SP 800-125B - Secure Virtual Network Configuration for Virtual Machine (VM) Protection

4 HIDS y sistemas de consolidación de logs

4.1 Introducción

En este capítulo vamos a evaluar de modo informal dos sistemas de detección de intrusiones (IDS) que hemos seleccionado porque, según documentan, permiten monitorizar la actividad de los contenedores: Wazuh y Falco. Usaremos un sistema de gestión de logs, Graylog para la recogida de las notificaciones.

Falco es un IDS propiamente dicho. Wazuh, por el contrario, es un producto que incluye varios componentes que describimos más adelante, entre ellos, un *fork* de OSSEC, que es el verdadero IDS.

4.2 Sistemas IDS

La guía *NIST Special Publication on Intrusion Detection Systems* [44] define los sistemas de detección de intrusiones como "software or hardware systems that automate the process of monitoring the events occurring in a computer system or network, analyzing them for signs of security problems"

NIST SP 800-94 [45] clasifica en tres categorías los métodos de detección de intrusiones:

- Basados en firmas: se comparan las capturas que realiza el IDS contra cadenas o patrones conocidos de ataques.
- Basados en la detección de anomalías: también llamados basados en la conducta (*behavior-based*). Se considera anomalía es una desviación de comportamiento normal. Para poder establecer qué es un comportamiento normal es necesario monitorizar las actividades habituales: fallos de autenticación, intentos de conexiones de red, etc. durante un periodo de tiempo y generar un perfil.
- Basados en la inspección de paquetes sensible al estado (*stateful packet inspection, SPI*). No hay una traducción aceptable de SPI; lo que se indica es que el IDS es capaz de analizar protocolos cuyas especificaciones utilizan el concepto de estado de los autómatas finitos, por ejemplo TCP. El protocolo monitorizado se compara con un perfil proporcionado por el distribuidor del IDS en cada estado del protocolo. Este método también se conoce como inspección profunda de paquetes (*deep packet inspection*) o "basado en la especificación" (*specification-based*).

Otra clasificación habitual de IDSs, atendiendo al lugar donde se colocan es:

- Basados en el host: HIDS (*Host-based IDS*). En general, monitoriza una sola máquina física o una VM, aunque puede operar como un agente coordinado con un servidor central, encargado de recibir datos de varios nodos y generar las alertas. Dos ejemplos de HIDS son OSSEC y Samhain.
- Basados en la red: NIDS . Se suelen colocar en un punto de red, físico o virtual, con frecuencia en un router, aunque pueden operar en cualquier nodo que tenga acceso al tráfico de red que interesa monitorizar, por ejemplo, una VM conectada a una puerta de un switch virtual que permita

hacer *port mirroring*. Open vSwitch incluye esta funcionalidad. El bridge nativo de Linux no soporta directamente la configuración de una puerta para que haga *port mirroring*, pero es posible conseguir el mismo resultado con la acción "mirred" de la utilidad para el control de los parámetros de tráfico tc(8)²⁰. Ejemplos NIDS de red son Snort y Bro (renombrado a Zeek recientemente) y Suricata. Tanto Snort como Suricata son NIDS basados en la detección de firmas. Bro, por el contrario emplea SPI.

Una tercera categoría podrían ser los IDS situados en el hipervisor, como es el caso de Collabra [46] para el hipervisor Xen.

4.2.1 Herramientas complementarias a los IDS

Las funcionalidad ofrecida por los IDS se complementa a menudo con otras facilidades disponibles

- Herramientas para el escaneo de vulnerabilidades, basadas en el host o en la red. Algunos ejemplos de escaneros de vulnerabilidades para hosts son vuls, pompem y lynis. OpenSCAP incluye esta funcionalidad, además de otras. Como ejemplos de escáneres de vulnerabilidades en red se pueden mencionar Nessus y OpenVAS, aunque estos productos proporcionan también funciones para el escaneo de hosts.
- Herramientas para la comprobación de la integridad de ficheros (FIM, *File Integrity Monitoring*). Se emplean checksums de los ficheros críticos para compararlos con valores de referencia y generar una alerta en el caso de que haya habido una modificación. Ejemplos de estas herramientas son Tripwire, AIDE, y mtree(8). OSSEC y Samhain incluyen comprobaciones de integridad.
- Herramientas para comprobar la conformidad con políticas de seguridad. Estas herramientas comprueban la conformidad del host con las regulaciones de seguridad legales o de la industria. OpenSCAP es una herramienta de este tipo.
- Gestión de logs: los IDS suelen generar un considerable número de entradas de logs en respuesta a los eventos que detectan. Los IDS pueden incluir una funcionalidad para el envío de logs a un servidor remoto mediante syslog u otros protocolos, o bien delegar esta función en herramientas específicas como rsyslogd(8).

4.3 Productos evaluados

Hemos seleccionado para la evaluación dos sistemas HIDS: OSSEC, en la versión incluida en el producto Wazuh, y Falco. El criterio de selección de HIDS se ha basado en su capacidad para monitorizar actividades en los contenedores que se ejecutan en el host, según la información que proporciona la documentación de los dos productos.

Emplearemos un colector de logs centralizado, Graylog para consolidar las alertas generadas por los dos HIDS evaluados.

²⁰ <https://backreference.org/2014/06/17/port-mirroring-with-linux-bridges/>

4.3.1 OSSEC

OSSEC es un producto de software libre que incorpora las siguientes funciones:

- Detección de *rootkits*.
- Comprobación de integridad de ficheros (FIM).
- Análisis de ficheros de logs.
- Respuestas activas o alertas que se disparan de acuerdo a las reglas que se le proporcione.

OSSEC tiene dos componentes principales un manager (servidor) y los agentes, que se despliegan en cada host. En el manager se almacenan la base de datos para la comprobaciones de integridad de ficheros, los logs (aunque pueden reenviarse a otros destinos), las reglas de activación de alertas, la lista de agentes registrados, la lista de ficheros monitorizados (principalmente ficheros de logs y de auditoría de los hosts monitorizados) y los descodificadores que se deben emplear para procesar las entradas de los ficheros monitorizados. Los agentes se despliegan en los sistemas monitorizados para recoger los datos y enviarlos al servidor. Un agente pueden monitorizar tanto ficheros de logs (file monitoring) como la salida de programas (process monitoring) como, por ejemplo *uptime* o *df -h*. Para que un agente pueda enviar datos al servidor debe estar registrado. El procedimiento de registro establece una clave compartida que se usa para cifrar la comunicación entre cada cada agente y el servidor.

El servidor OSSEC recibe los datos de los agentes por el puerto 1514/udp, lo cual implica que la entrega no está garantizada. También puede recibir datos de sistemas que no tengan instalados el agente, mediante el protocolo syslog. Para ello, se configura el servidor para que reciba las entradas por el puerto 514/udp.

4.3.2 Sysdig Falco

Falco es un HIDS basado en la detección de anomalías en las llamadas al sistema. Las llamadas se interceptan con dos módulos del núcleo (*sysdig_probe* y *falco_probe*) que deben instalarse en cada host. La ejecución en el espacio del núcleo permite monitorizar la actividad del interior de los contenedores.

Sysdig (además del nombre de la empresa) es una utilidad de captura de llamadas al sistema, del tipo de *systemtap* o *dtrace* que puede emplear por separado como una herramienta de análisis de la actividad de los procesos y empleado *sysdig_probe*. Falco (el producto) es el IDS, y se compone de tres elementos: dos módulos del núcleo mencionados más arriba, y un demonio (*falco*) que recibe la información que generan los módulos, la filtra según las reglas que se le proporcione, y escribe las alertas a disco o a syslog.

Falco disparar una alerta a partir de cualquier evento que genere una llamada al núcleo de Linux y para el cual se le haya definido una regla que la active. Las reglas para la generación de alertas emplean las expresiones de filtrado de Sysdig, que son muy potentes pero notablemente complicadas. La documentación de Falco menciona, entre otros, los siguientes ejemplos de actividades que se pueden monitorizar:

- Ejecución de una shell en un contenedor.

- Lectura imprevista de un fichero sensible, por ejemplo, `/etc/shadow`.
- Escritura de un fichero que no corresponda a un dispositivo en `/dev`.
- Conexiones irregulares de red por parte de aplicaciones.

El demonio de Falco se puede ejecutar tanto en el host como en un contenedor; en cualquiera de los dos casos es posible monitorizar tanto el host como los contenedores. En las pruebas siguientes, lo ejecutaremos dentro de un contenedor.

4.3.3 Wazuh

Wazuh es un producto de software libre que reúne varios componentes de otros proyectos, principalmente:

- Un fork de OSSEC, descrito más arriba que será el HIDS que evaluaremos.
- OpenScap para la comprobación de conformidad con políticas de seguridad.
- La pila ELK (o "Elastic Stack", el nuevo nombre), compuesta a su vez de tres elementos:
 - Elasticsearch: una facilidad para la indexación y búsqueda de texto.
 - Logstash: encaminamiento y gestión de entradas de logs.
 - Kibana: una consola web para la visualización de resultados.

La consola de Wazuh se ejecuta como un componente instalado en Kibana. Esta consola junto con OSSEC serán los únicos elementos que emplearemos. OpenScap, pese a ser una herramienta de seguridad muy recomendable, queda fuera del alcance de la evaluación.

Hemos seleccionado la versión de OSSEC incluida en Wazuh, en vez de usar la versión nativa, porque la documentación de Wazuh afirma que es capaz de monitorizar la actividad de los contenedores con un componente adicional, Docker woddle.²¹ La actividad que registra este componente es, sin embargo, bastante limitada: se limita a señalar alertas para el arranque, parada, suspensión y reanudación de contenedores.

Wazuh resulta también interesante porque anuncia la posibilidad de integrar NIDS (Suricata y Bro) mediante un componente adicional, OwlH. La evaluación de NIDS, sin embargo está fuera del alcance de este proyecto, aunque sería el siguiente paso lógico.

4.3.4 Graylog

El colector de logs utilizado es Graylog. Hemos seleccionado este producto porque es una alternativa de software libre a Splunk, y porque está diseñado específicamente con el propósito de recoger y consolidar logs, a diferencia de otras soluciones basadas en ELK, como la que incluye Wazuh. Esta especialización le permite recoger entradas de los en prácticamente cualquier formato sin tener que escribir un decodificador específico para el formato de la entrada de log, como ocurre en el caso de Wazuh. Graylog emplea el componente Elasticsearch de ELK, pero proporciona sus propios componentes para la recogida y la interfaz de consola en vez de usar Logstash y Kibana.

²¹ https://documentation.wazuh.com/current/docker-monitor/monitoring_containers_activity.html

Las fuentes de las que puede recoger logs, según indica su documentación son:

- Syslog (TCP, UDP, AMQP, Kafka)
- GELF(TCP, UDP, AMQP, Kafka, HTTP)
- AWS - AWS Logs, FlowLogs, CloudTrail
- Beats/Logstash
- CEF (TCP, UDP, AMQP, Kafka)
- Trayecto JSON a través de una API HTTP
- Netflow (UDP) (Cisco, OVS)
- Texto plano (TCP, UDP, AMQP, Kafka)

De las opciones anteriores, emplearemos únicamente syslog y GELF, ambos por UDP.

En la primera evaluación de este producto, lo descartamos porque los requería para su ejecución más recursos de los que se le podía proporcionar en el primer laboratorio de pruebas. Después de migrar el laboratorio desplegado con VirtualBox a una máquina con más memoria y CPU ha funcionado sin problemas.

4.4 Laboratorio de pruebas

4.4.1 Despliegue de máquinas

Usaremos varias VMs sobre VirtualBox, en un ordenador portátil con 32GB de memoria. Cada VM tiene definida dos interfaces de red, conectadas a dos "redes" VirtualBox: una externa, de tipo "bridged" como red externa y otra interna, de tipo "Host only networking". Los tipos de red VirtualBox están descritos en su manual de usuario (<https://www.virtualbox.org/manual/ch06.html>).

VM	ext (bridged)	int (host-only adapter)	Componente desplegado
rh7a	192.168.100.135	10.1.0.135	Falco, Docker CE
rh7b	192.168.100.136	10.1.0.136	Agente OSSEC (Wazuh), Docker CE
wazuha	192.168.100.137	10.1.0.137	Manager OSSEC (Wazuh)
grayloga	192.168.100.138	10.1.0.138	Graylog
msi (host)	192.168.100.199	10.1.0.1	VirtualBox
asus (nodo externo)	192.168.100.106		nmap, ncrack, Nessus, etc.

Tabla 3: Nodos usados en las pruebas

Todos los nodos tienen instalados alguna distribución de Linux (RedHat, Ubuntu, CentOS, Debian).

4.4.2 Configuración de envío y recepción de alertas

Falco y OSSEC emplean dos modos distintos para comunicar alertas. Las de Falco son de tipo *push* (la inicia la máquina monitorizada); en OSSEC son de tipo *pull* (el manager le pregunta al agente). OSSEC permite, sin embargo, notificaciones de tipo *push* mediante syslog, pero se usa principalmente para monitorizar equipos en los cuales no es posible instalar un agente, por ejemplo dispositivos de red.

Configuramos los equipos del modo siguiente:

rh7a (falco) envía notificaciones:

- a graylog (Graylog) por syslog de forma predeterminada, el host .
- a graylog por GELF si así se especifica en la línea de órdenes, los contenedores.

rh7b (OSSEC) envia notificaciones:

- a wazuha (Wazuh) usando el protocolo manager-agente por un canal cifrado.
- a graylog por syslog.
- a graylog por GELF si así se especifica en la línea de órdenes, los contenedores.

4.4.2.1 rh7a (falco)

Falco necesita tener cargados dos módulos del núcleo, lo cual, dependiendo de la plataforma, puede que implique la instalación de un núcleo de desarrollo. En este caso (un nodo RedHat 7), debemos emplear los repositorios yum de Fedora para instalarlo:

```
[root@rh7a yum.repos.d]# yum install \
-y https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
[...]
[root@rh7a yum.repos.d]# yum install -y kernel-devel-$(uname -r)
[...]
```

Después, al instalar Falco se podrán generar los módulos con DKMS:

```
[root@rh7a yum.repos.d]# yum install falco
[...]
DKMS: build completed.
```

Configuramos de Falco para que envíe su salida un fichero en texto plano y a syslog. Rsyslog se encargará después de reenviarlo a Graylog. En el fichero `/etc/falco/falco.yaml` establecemos los valores siguientes:

```
# usamos syslog para enviar eventos a graylog
syslog_output:
  enabled: true

# usamos también la salida a un fichero local, para depuración
file_output:
  enabled: true
  keep_alive: false
  filename: /var/opt/log/falco.log
```

Las reglas para definir la respuestas a eventos se definen en el fichero `/etc/falco_rules.yaml`. Por ejemplo, la siguiente regla se activa cuando se ejecuta una shell de forma interactiva en un contenedor:

```
- rule: Terminal shell in container
  desc: A shell was used as the entrypoint/exec point into a container with an attached terminal.
  condition: >
    spawned_process and container
    and shell_procs and proc.tty != 0
    and container_entrypoint
  output: >
```

```
A shell was spawned in a container with an attached terminal (user=%user.name
%container.info
  shell=%proc.name parent=%proc.pname cmdline=%proc.cmdline terminal=%proc.tty)
priority: NOTICE
tags: [container, shell]
```

Cada regla contiene los campos siguientes:

- desc: una descripción de la regla, con el texto que elijamos.
- condition : la condición por la que se activa la alerta.
- output: el texto de salida mostrado cuando se activa la regla.
- priority: nivel de la alerta (INFO, WARNING, ALERT, DEBUG, CRITICAL).

Las alertas se pueden enviar a la salida estándar, a un fichero, a syslog, o a un programa que se ejecute al disparar la alerta. Se podrían enviar la salida por syslog a Wazuh pero sería necesario escribir descodificadores para las alertas. Graylog no requiere configuración adicional para recibir estas alertas por syslog. Así que enviaremos las salidas por syslog a Graylog, y usaremos el agente de OSSEC para enviar las alertas a Wazuh.

4.4.2.2 rh7b (agente ossec) y wazuha (manager ossec)

Para poder monitorizar un nodo con Wazuh, es necesario registrar el agente después de su instalación:

```
[root@wazuha ~]# /var/ossec/bin/manage_agents
*****
* Wazuh v3.6.1 Agent manager.          *
* The following options are available: *
*****
(A)dd an agent (A).
(E)xtract key for an agent (E).
(L)ist already added agents (L).
(R)emove an agent (R).
(Q)uit.
Choose your action: A,E,L,R or Q: A
- Adding a new agent (use '\q' to return to the main menu).
Please provide the following:
* A name for the new agent: rh7b
* The IP Address of the new agent: 10.1.0.136
* An ID for the new agent[003]:
[...]
Agent added with ID 003.
```

Desde el servidor, extraemos la clave del agente. Esta clave se comparte entre el agente y el manager para cifrar el canal de datos:

```
[root@wazuha ~]# /var/ossec/bin/manage_agents -e 003
Agent key information for '003' is:
MDAxIHJoN2EgMTAuMS4wLjEzNSA0OTNhNzczM2JiNDViZjU5ZDljOTRjNmVmNGI5ODd1MjYxMjFkNWU4ZTNjZjk1ZDY1YzgzMDQ2ZWNhOWM0OWE4
```

La añadimos en la máquina del agente y rearrancamos el agente.

```
[root@rh7b yum.repos.d]# /var/ossec/bin/manage_agents -i
MDAxIHJoN2EgMTAuMS4wLjEzNSA0OTNhNzczM2JiNDViZjU5ZDljOTRjNmVmNGI5ODd1MjYxMjFkNWU4ZTNjZjk1ZDY1YzgzMDQ2ZWNhOWM0OWE4
Agent information:
```



```
ID:003
Name:rh7b
IP Address:10.1.0.136
[...]
[root@rh7a yum.repos.d]# systemctl restart wazuh-agent.service
[...]
```

Comprobamos agente de OSSEC.

```
[root@wazuha bin]# /var/ossec/bin/agent_control -lc

Wazuh agent_control. List of available agents:
ID: 000, Name: wazuha (server), IP: 127.0.0.1, Active/Local
ID: 003, Name: rh7b, IP: 10.1.0.136, Active
```

4.4.3 graylog (recepción de logs)

4.4.3.1 Recepción por syslog

Graylog (appliance-syslog-udp) escucha directamente por udp/514 los envíos remotos de rsyslog. No es rsyslogd el que recibe los logs remotos:

```
root@grayloga:/etc/graylog# lsof -iUDP:514
COMMAND PID  USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
java     785  graylog  96u  IPv6  12823      0t0  UDP *:syslog
```

Configuramos el rsyslogd de rh7a y rh7b para que envíen entradas de syslog a Graylog. Añadimos al fichero /etc/rsyslogd

```
# envío por UDP
*. * @grayloga-int.hornillo:514;RSYSLOG_SyslogProtocol23Format
```

Probamos la generación de un evento en rh7a para que se notifique por syslog a Graylog:

```
[root@rh7a falco]# sysdig evt.type=open and fd.name contains /etc
[...]
```

Comprobamos la recepción por syslog en la consola de Graylog:

The screenshot shows the Graylog Search interface. The search results for the ID `d1e71c10-0854-11e9-a2e0-0800272d28a8` are displayed. The message content is `***Action write_etc#015`. Below it, another message with ID `d14d9b30-0854-11e9-a2e0-0800272d28a8` is shown with the content `***Action write_binary_dir#015`. The interface includes a sidebar with configuration options like Syslog Severity Mapper and Format String, and a main content area with message details and a table of search results.

4.4.3.2 Recepción por GELF

Graylog (appliance-syslog-udp) escucha por udp/12201 los envíos por GELF:

```
root@grayloga:~# lsof -iUDP:12201
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
svloggelf 783 root 3u IPv4 8106 0t0 UDP localhost:43506->localhost:12201
java 785 graylog 97u IPv6 10043 0t0 UDP *:12201
svloggelf 791 root 3u IPv4 8107 0t0 UDP localhost:54845->localhost:12201
svloggelf 793 root 3u IPv4 8111 0t0 UDP localhost:57319->localhost:12201
svloggelf 795 root 3u IPv4 8648 0t0 UDP localhost:53657->localhost:12201
svloggelf 821 root 3u IPv4 8659 0t0 UDP localhost:48294->localhost:12201
```

Probamos la recepción por GELF de los logs de contenedores. Ejecutamos un contenedor en rh7a con salida de logs por GELF dirigida al puerto udp/12201 de Graylog:

```
[root@rh7a log]# docker run --log-driver gelf --log-opt \
gelf-address=udp://grayloga.hornillo:12201 alpine echo Entrada log GELF
```

Comprobamos la recepción en la consola de Graylog:

rh7a - Entrada log GELF

2cdce6c0-0893-11e9-a2e0-0800272d28a8

Received by
appliance-gelf-udp on [P e09d764e / grayloga.homillo](#)

command
echo Entrada log GELF

4.5 Pruebas de detección de intrusiones

4.5.1 Pruebas ejecutadas dentro los hosts

Para generar operaciones sospechosas, o directamente agresivas, utilizaremos un programa llamado `event_generator` (<https://github.com/falcosecurity/falco/wiki/Generating-Sample-Events>). Este programa ejecuta, o lo intenta, operaciones para provocar alertas.

La tabla 4 muestra los parámetros correspondientes a las operaciones disponibles.

<code>write_binary_dir</code>	Escritura en ficheros bajo <code>/bin</code>
<code>write_etc</code>	Escritura en ficheros bajo <code>/etc</code>
<code>read_sensitive_file</code>	Leer un fichero sensible
<code>write_rpm_database</code>	Escritura en ficheros bajo <code>/var/lib/rpm</code>
<code>spawn_shell</code>	Ejecutar una shell (bash)
<code>db_program_spawn_process</code>	Intenta lanzar otro programa desde un programa de bases de datos en ejecución
<code>modify_binary_dirs</code>	Modificar un fichero bajo <code>/bin</code>
<code>mkdir_binary_dirs</code>	Crear un directorio bajo <code>/bin</code>
<code>change_thread_namespace</code>	Cambio de espacio de nombres
<code>system_user_interactive</code>	Convertirse en otro usuario del sistema e intentar ejecutar una orden interactiva
<code>system_procs_network_activity</code>	Abrir conexiones de red desde un programa que no debe realizar operaciones de red
<code>non_sudo_setuid</code>	Setuid como un usuario distinto de root
<code>create_files_below_dev</code>	Crear ficheros bajo <code>/dev</code>
<code>user_mgmt_binaries</code>	Convertirse en el programa "vipw", que dispara reglas relativas a los programas de administración de usuarios
<code>exfiltration</code>	Leer <code>/etc/shadow</code> y enviarlo por udp a una dirección y puerto específicos.

Tabla 4: operaciones ejecutadas por `event_generator`

Cuando se ejecuta sin parámetros, el programa ejecuta un bucle continuo con todas las acciones. La opción `-a` permite especificar una sola opción.

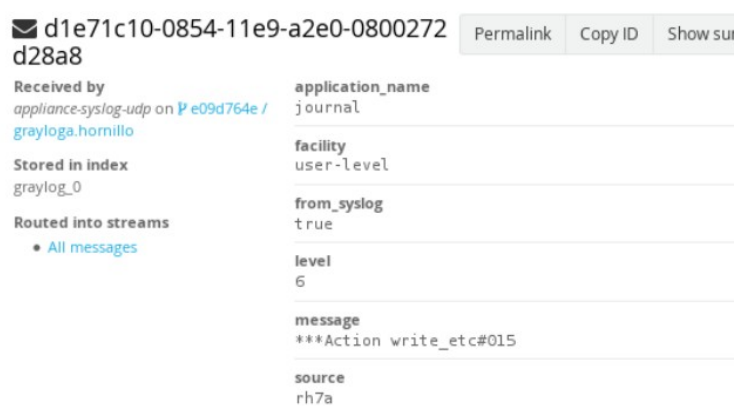
Ejecutaremos el `event_generator` en `rh7a` y `rh7b` para comparar cómo se activan las reglas de Falco y OSSEC, respectivamente. El programa se ejecutará dentro de un contenedor en cada máquina. Debemos tener en cuenta, sin embargo, que este programa está hecho por Sysdig (la empresa) para mostrar las cualidades de Falco, por lo que previsiblemente, Falco capturará todas las acciones. Tenemos que comprobar cuáles captura OSSEC.

4.5.1.1 Ejecución de event-generator en un contenedor:

rh7a (falco)

```
[root@rh7a falco]# docker pull sysdig/falco-event-generator
[...]
[root@rh7a falco]# docker run -it --name falco-event-generator sysdig/falco-event-generator
***Action change_thread_namespace
Calling setns() to change hhtopnamespace...
NOTE: does not result in a falco notification in containers, unless container run with --
privileged or --security-opt seccomp=unconfined
***Action create_files_below_dev
Creating /dev/created-by-event-generator-sh...
***Action db_program_spawn_process
Becoming the program "mysql" and then running ls
[...]
```

Comprobamos en la consola de Graylog de que se ha recibido por syslog eventos disparados:



The screenshot shows a message entry in Graylog with the ID `d1e71c10-0854-11e9-a2e0-0800272d28a8`. The message details are as follows:

Received by appliance-syslog-udp on <code>P e09d764e / grayloga.hornillo</code>	application_name journal
Stored in index graylog_0	facility user-level
Routed into streams • All messages	from_syslog true
	level 6
	message ***Action write_etc#015
	source rh7a

rh7b (OSSEC)

Ejecutamos el event_generator en un contenedor:

```
[root@rh7b etc]# date && docker unpause falco-event-generator
Wed 26 Dec 13:27:49 CET 2018
falco-event-generator
```

OSSEC no detecta ninguna de las actividades de falco-event-generator:

```
2018/12/26 13:04:49 ossec-syscheckd: INFO: Ending syscheck scan. Database completed.
2018/12/26 14:03:31 wazuh-modulesd:syscollector: INFO: Starting evaluation.
```

4.5.2 Pruebas de detección de intrusiones realizadas desde un nodo externo

4.5.2.1 Ataque por fuerza bruta al host por SSH.

Emplearemos la utilidad ncrack para intentar abrir una sesión de root por SSH desde otro nodo.

rh7a (falco)

Lanzamos el ataque rh7a para comprobar si Falco está generando alertas y enviándolas a syslog.

```
jgamboa@asus:~$ ncrack -v --user root rh7a:22
```

Las notificaciones llegan a la consola de Graylog:

```
2018-12-27 20:56:50.190      rh7a      security/authorization
Failed password for root from 192.168.100.106 port 9816 ssh2

2018-12-27 20:56:50.185      rh7a      security/authorization
PAM 1 more authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=asus.hornillo user=root
```

Las entradas que han llegado a Graylog desde rh7a provienen de /var/log/secure, están generadas por sshd, y reenviadas por rsyslog a Graylog. Falco no lo ha detectado. Estamos atacando a una aplicación (sshd) en el espacio del usuario solamente dando contraseñas inválidas hasta que sshd corta la conexión, así que es razonable que este caso no saque ventaja de sus módulos en el núcleo.

```
[root@rh7a ~]# tail -2 /var/log/secure
Dec 27 20:56:50 rh7a sshd[30871]: Failed password for root from 192.168.100.106 port 9816
ssh2
Dec 27 20:56:50 rh7a sshd[30871]: Connection closed by 192.168.100.106 port 9816 [preauth]
```

rh7b (OSSEC)

```
jgamboa@asus:~$ ncrack -v --user root rh7b:22
```

```
Starting Ncrack 0.6 ( http://ncrack.org ) at 2018-12-27 19:54 CET
```

Resultado:

Time	_source
▶ December 27th 2018, 19:54:52.310	id: 1545936892.507150 path: /var/ossec/logs/alerts/alerts.json rule.level: 5 rule.id: 5716 rule.description: sshd: authentication failed. rule.gdpr: IV_35.7.d, IV_32.2 rule.firedtimes: 125 rule.groups: syslog, sshd, authentication_failed rule.gpg13: 7.1 rule.pci_dss: 10.2.4, 10.2.5 rule.mail: false full_log: Dec 27 19:54:50 rh7b sshd[19562]: Failed password for root from 192
▶ December 27th 2018, 19:54:52.307	id: 1545936892.506700 path: /var/ossec/logs/alerts/alerts.json rule.level: 10 rule.id: 2502 rule.description: syslog: User missed the password more than one time rule.gdpr: IV_35.7.d, IV_32.2 rule.firedtimes: 34 rule.groups: syslog, access_control, authentication_failed rule.gpg13: 7.8 rule.pci_dss: 10.2.4, 10.2.5 rule.mail: false full_log: Dec 27 19:54:50 rh7b sshd[19534]: P

Estas alertas se han disparado también a partir de las entradas de /var/log/secure de rh7b

```
Dec 27 19:54:50 rh7b sshd[19534]: Failed password for root from 192.168.100.106 port 27320
ssh2
Dec 27 19:54:50 rh7b sshd[19534]: Connection closed by 192.168.100.106 port 27320 [preauth]
Dec 27 19:54:50 rh7b sshd[19534]: PAM 2 more authentication failures; logname= uid=0 euid=0
tty=ssh ruser= rhost=asus.hornillo user=root
```

Pero han llegado a wazuha necesariamente a través del agente OSSEC de rh7b, porque no tenemos configurado el rsyslog de rh7b para que envíe notificaciones a wazuha. Lo podemos confirmar al ver los detalles del evento en la consola de Wazuh:

@timestamp	December 27th 2018, 19:54:52.307
t_id	zskE8WcBbR_mW23i9Y6j
t_index	wazuh-alerts-3.x-2018.12.27
#_score	-
t_type	wazuh
t_agent.id	003
t_agent.ip	10.1.0.136
t_agent.name	rh7b
t_data.dstuser	root
t_data.srcip	asus.hornillo
t_decoder.name	sshd
t_decoder.parent	sshd
t_full_log	Dec 27 19:54:50 rh7b sshd[19534]: PAM 2 more authentication failures; logname=uid=0 euid=0 tty=ssh ruser= rhost=asus.hornillo user=root
t_id	1545936892.506700
t_location	/var/log/secure
t_manager.name	wazuha
t_path	/var/ossec/logs/alerts/alerts.json

Las entradas sí llegan a Graylog por syslog:

2018-12-27 19:54:50.643	rh7b	security/authorization	rh7b - PAM 1 more authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=asus.hornillo user=root
2018-12-27 19:54:50.641	rh7b	security/authorization	rh7b - Connection closed by 192.168.100.106 port 27330 [preauth]
2018-12-27 19:54:50.640	rh7b	security/authorization	rh7b - Failed password for root from 192.168.100.106 port 27330 ssh2

En este caso OSSEC detecta el ataque; Falco no.

4.5.2.2 Escaneo de puertos (rh7a, rh7b)

Ejecutamos desde otro nodo las órdenes

```
root@asus:/home/jgamboa# nmap -Pn --script vuln rh7a
[...]
root@asus:/home/jgamboa# nmap -Pn --script vuln rh7b
[...]
```

Ni Falco ni OSSEC lo detectan. Probablemente, un NIDS sería más adecuado en este caso.

4.6 Conclusiones de la evaluación

La evaluación realizada no tiene mucho rigor, entre otros motivos porque las pruebas no han cubierto muchos casos, como por ejemplo, ataques de denegación de servicio. Pero tampoco tendría demasiado sentido hacerlo, puesto que solo hemos empleado las reglas de OSSEC y Falco que vienen predefinidas. El verdadero trabajo estaría en crear las reglas de detección para Falco y OSSEC, y en este último caso, posiblemente también un descodificador de las notificaciones en el manager de OSSEC. En un despliegue en explotación, requeriría unos cuantos días o semanas configurar adecuadamente cualquiera de los dos HIDS para poder detectar los ataques y además, eliminar los falsos positivos. Una consola llena de alertas falsas no va a llamar la atención de nadie en el caso de un ataque real.

No es de extrañar, por lo tanto, que el modelo de negocio de las empresas que distribuyen Wazuh y Falco sea proporcionar como software libre el núcleo de un producto que requiere trabajo adicional, y ofrecer la funcionalidad ausente como SaaS o como versiones "Enterprise" que requieren una licencia. Esto, por supuesto, no es una crítica al modelo de negocio, sino una observación sobre las limitaciones en las versiones de tipo "community" de estos productos.

En cualquier caso, las pruebas que hemos realizado son suficientes como para pensar que en un host en el cual se usen contenedores, Falco es una solución posiblemente más apropiada por los siguientes motivos:

- Falco puede detectar directamente la actividad que se ejecuta en los contenedores y en los puertos de red, porque tiene acceso al núcleo a través de los módulos `sysdig_probe` y `falco_probe`. OSSEC, por el contrario solo tiene acceso a la actividad de los contenedores a través de los logs que generen el núcleo, las aplicaciones, o los contenedores.
- Para poder procesar un fichero de log que no venga ya incluido en la distribución del producto, OSSEC necesita que se le proporcione un descodificador específico, además de la regla que para indicarle que procese el fichero. Falco, por el contrario, le envía todo lo que detecta a un fichero o a `syslog`. En este caso, es Graylog, a través de Elasticsearch quien realiza el trabajo.
- Si configuramos OSSEC para que las notificaciones de la máquina monitorizada se envíen por `syslog`, el manager de OSSEC recibe la entrada por `syslog`, pero no aparecen automáticamente en la consola de Wazuh (Kibana) si no tenemos el descodificador adecuado.

Por otro lado OSSEC presenta las siguientes cualidades frente a Falco:

- Falco requiere la instalación de un módulo del núcleo, mientras que con OSSEC solo hay que instalar un servicio. En sistemas de explotación con contratos de soporte de software, la instalación de módulos de terceras partes puede ser motivo para rechazar la apertura de una incidencia en el servicio de soporte al ejecutarse el software en una "plataforma no soportada".
- OSSEC incluye características adicionales que no hemos tenido en cuenta, tales como la capacidad de realizar comprobaciones de la integridad de los ficheros.

5 Conclusiones

Contenedores de aplicaciones

Después de arañar un poco la superficie de las técnicas empleadas en los contenedores de aplicaciones para Linux hemos podido comprobar que algunas de las ideas que teníamos al principio del trabajo no eran del todo correctas, o estaban directamente equivocadas. La idea de comparar un contenedor Linux con una VM, en función de su rendimiento, densidad por host, y en particular en cuestiones de seguridad, tal como se reproducen en tantas publicaciones no nos parece ya tan acertada. Al terminar el trabajo, nuestra opinión es que VMs y contenedores de aplicaciones son especies distintas, y por lo tanto cualquier comparación tiene poco sentido: los contenedores son grupos de procesos más o menos confinados, y la comparación en cuanto a seguridad debería hacerse con respecto a los procesos que no se ejecutan en un contenedor. Desde este punto de vista, los contenedores en Linux aportan mejoras sustanciales en el plano de la seguridad.

SELinux

Una fracción apreciable del tiempo dedicado a este trabajo se ha empleado en un intento fallido de aplicar políticas SELinux a los contenedores de un modo efectivo antes de preparar el capítulo 4. La complejidad de la arquitectura Flask y la escasa o nula documentación sobre el modelo MCS no han ayudado a conseguir este objetivo. En nuestra opinión, la modificación de una política SELinux requiere unas habilidades muy especializadas de las que carecemos la mayoría de los administradores de sistemas. No nos ha sido posible adquirir estas habilidades durante el desarrollo del trabajo. Antes de empezar este trabajo, estábamos convencidos de que el despliegue de contenedores en producción exigiría el empleo de SELinux: el lado positivo de este fallo ha sido constatar que no se trata de una exigencia tan clara. Dos fuentes solventes comparten esta opinión:

“Due to this complexity, lack of up-to-date policies and general lack of understanding, SELinux suffers from what the author personally refers to the ‘setenforce 0 principal’. Disabling SELinux is such a common trend, it even has a website created to stopping the practice, stopdisablingselinux.com with an associated ‘setenforce 1’ t-shirt, put together by infamous SELinux advocate and RedHat employee Dan Walsh”. [8]

“Our general opinion regarding SELinux is that it’s capable of delivering more harm than benefit. Unfortunately, that harm can manifest not only as wasted time and aggravation for system administrators, but also, ironically, as security lapses. Complex models are hard to reason about, and SELinux isn’t really a level playing field; hackers that focus on it understand the system far more thoroughly than the average sysadmin.” [47]

Sistemas de control de intrusiones

Por el contrario, el escaso tiempo dedicado a los HIDS nos ha confirmado la idea de que deberían estar presentes en cada máquina de producción, con independencia de que ejecuten contenedores no. Llegar a dominar la configuración de reglas de disparo de alertas es, sin embargo, una tarea que requiere algún tiempo de aprendizaje. Falco nos ha parecido una herramienta eficaz, que además tiene la ventaja de usar el lenguaje de expresiones de sysdig, una utilidad de generación de trazas que seguramente, por sí sola merece que se le dedique el esfuerzo.

Referencias

- [1] Forrester Consulting, «Containers: Real Adoption And Use Cases In 2017», n.º March, 2017.
- [2] Cloudfoundry, «Cloud Foundry Foundation Global Perception Study Hope Versus Reality, One Year Later», pp. 1-6, 2017.
- [3] DataDog, «8 surprising facts about real Docker adoption», 2018. [En línea]. <https://www.datadoghq.com/docker-adoption/>. [Acceso: 06-oct-2018].
- [4] «Portworx Annual Container Adoption Survey 2017», 2017.
- [5] T. Bui, «Analysis of Docker Security», *arXiv preprint*. 2015.
- [6] A. Mouat, *Docker security*. O'Reilly Media, Inc., 2016.
- [7] E. Reshetova, J. Karhunen, T. Nyman, y N. Asokan, «Security of OS-level virtualization technologies», en *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014, vol. 8788, pp. 77-93.
- [8] A. Grattafiori, «Understanding and Hardening Linux Containers», *NCC Gr. Whitepapers*, n.º 1, pp. 1-122, 2016.
- [9] U. Gupta, «Comparison between security majors in virtual machine and linux containers», *Comput. Res. Repos.*, vol. abs/1507.0, p. 4, 2015.
- [10] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, y L. Foschini, «Securing the infrastructure and the workloads of linux containers», en *2015 IEEE Conference on Communications and Network Security, CNS 2015*, 2015, pp. 559-567.
- [11] Aqua Security Lab, «Dirty COW Vulnerability: Impact on Containers», *Aqua Blog*, 2016. [En línea]. <https://blog.aquasec.com/dirty-cow-vulnerability-impact-on-containers>. [Acceso: 21-oct-2018].
- [12] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, y H. Wang, «ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds», *Proc. - 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Networks, DSN 2017*, pp. 237-248, 2017.
- [13] J. Morris, «A Brief Introduction to Multi-Category Security (MCS)», *LIVEJOURNAL*, 2005. [En línea]. <https://james-morris.livejournal.com/5583.html>. [Acceso: 25-sep-2018].
- [14] «The SELinux Notebook (4th Edition)», 2014.
- [15] W. Felter, A. Ferreira, R. Rajamony, y J. Rubio, «An updated performance comparison of virtual machines and Linux containers», en *ISPASS 2015 - IEEE International Symposium on Performance Analysis of Systems and Software*, 2015, pp. 171-172.

- [16] R. Morabito, J. Kjällman, y M. Komu, «Hypervisors vs. lightweight virtualization: A performance comparison», en *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, 2015, pp. 386-393.
- [17] M. Chae, H. Lee, y K. Lee, «A performance comparison of linux containers and virtual machines using Docker and KVM», *Cluster Comput.*, dic. 2017.
- [18] A. M. Joy, «Performance comparison between Linux containers and virtual machines», *Conf. Proceeding - 2015 Int. Conf. Adv. Comput. Eng. Appl. ICACEA 2015*, n.º Lxc, pp. 342-346, 2015.
- [19] Z. Kozhirbayev y R. O. Sinnott, «A performance comparison of container-based technologies for the Cloud», *Futur. Gener. Comput. Syst.*, vol. 68, pp. 175-182, mar. 2017.
- [20] R. P. Goldberg, «Architectural Principles for Virtual Computer Systems», Harvard University, 1972.
- [21] R. Chandramouli, «Security recommendations for server-based hypervisor platforms», 2018.
- [22] P. Kamp y R. N. M. Watson, «Jails : Confining the omnipotent root», *Proceeding 2nd Int. Syst. Adm. Netw. Conf.*, pp. 3-14, 2000.
- [23] D. Price y A. Tucker, «Solaris Zones: Operating System Support for Consolidating Commercial Workloads», *LISA '04 Proc. 18th USENIX Conf. Syst. Adm.*, pp. 241-254, 2004.
- [24] P. B. Menage, «Adding Generic Process Containers to the Linux Kernel», *Proc. Ottawa Linux Symp.*, pp. 45-58, 2007.
- [25] B. Gregg, *Systems Performance: Enterprise and the Cloud*. Prentice Hall, 2013.
- [26] D. Ernst, D. Bermbach, y S. Tai, «Understanding the Container Ecosystem : A Taxonomy of Building Blocks for Container Lifecycle and Cluster Management», en *IEEE Second International Workshop on Container Technologies and Container Clouds*, 2016.
- [27] «Docker Registry HTTP API V2». [En línea]. <https://github.com/docker/distribution/blob/5cb406d511b7b9163bff9b6439072e4892e5ae3b/docs/spec/api.md>. [Acceso: 15-nov-2018].
- [28] A. Polvi, «CoreOS is building a container runtime, rkt», *CoreOS.com*, 2014. [En línea]. <https://coreos.com/blog/rocket.html>. [Acceso: 18-oct-2018].
- [29] J. Pendry y M. McKusick, «Union mounts in 4.4 BSD-lite», *AUUGN*, 1997.
- [30] D. Quigley, J. Sipek, C. P. Wright, y E. Zadok, «Unionfs: User and Community-Oriented Development of a Unification File System», en *Proceedings of the Linux Symposium*, 2006, vol. 2, pp. 349-362.
- [31] N. Brown, «Overlay Filesystem». 2014.
- [32] C. Wright, C. Cowan, J. Morris, S. Smalley, y G. Kroah-Hartman, «Linux security modules: General security support for the Linux Kernel», en

- Foundations of Intrusion Tolerant Systems, OASIS 2003*, 2003, vol. 8032, pp. 213-226.
- [33] J. Lepreau, R. Spencer, S. D. Smalley, P. Loscocco, M. Hibler, y D. Andersen, «The Flask Security Architecture: System Support for Diverse Security Policies», *Proc. 8th USENIX Secur. Symp.*, p. 11, 1999.
- [34] «IEEE1003.1e Draft Standard for Information Technology Portable Operating System Interface (POSIX)», *PSSG Draft 17*. p. 1, 1997.
- [35] A. Khan, «Key Characteristics of a Container Orchestration Platform to Enable a Modern Application», *IEEE Cloud Comput.*, vol. 4, n.º 5, pp. 42-48, sep. 2017.
- [36] Gobierno de España. Ministerio de Justicia, «Real Decreto 1720/2007, de 21 de diciembre, por el que se aprueba el Reglamento de desarrollo de la Ley Orgánica 15/1999, de 13 de diciembre, de protección de datos de carácter personal».
- [37] «Reglamento (UE) 2016/679 del Parlamento Europeo y del Consejo de 27 de abril de 2016 relativo a la protección de las personas físicas en lo que respecta al tratamiento de datos personales y a la libre circulación de estos datos».
- [38] R. Shu, X. Gu, y W. Enck, «A Study of Security Vulnerabilities on Docker Hub», en *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy - CODASPY '17*, 2017, pp. 269-280.
- [39] D. J. Walsh, «Are Docker containers really secure?», 2014. [En línea]. <https://opensource.com/business/14/7/docker-security-selinux>. [Acceso: 06-oct-2018].
- [40] Z. Jian y L. Chen, «A Defense Method against Docker Escape Attack», en *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy - ICCSP '17*, 2017, pp. 142-146.
- [41] Center for Internet Security, «CIS Docker Community Edition Benchmark 1.1.0», 2017.
- [42] M. Souppaya, J. Morello, y K. Scarfone, «Application Container Security Guide - NIST SP 800-190», 2017.
- [43] R. Chandramouli, «NISTIR 8176 Security Assurance Requirements for Linux Application Container Deployments», 2017.
- [44] R. Bace y P. Mell, «NIST Special Publication on Intrusion Detection Systems», *Natl. Inst. Stand. Technol.*, pp. 1-51, 2001.
- [45] K. A. Scarfone y P. M. Mell, «NIST SP 800-94 - Guide to Intrusion Detection and Prevention Systems (IDPS)». 2007.
- [46] S. Bharadwaja, W. Sun, M. Niamat, y F. Shen, «Collabra: A Xen hypervisor based collaborative intrusion detection system», *Proc. - 2011 8th Int. Conf. Inf. Technol. New Gener. ITNG 2011*, pp. 695-700, 2010.
- [47] E. Nemeth, G. Snyder, y T. Hein, *UNIX and Linux system administration handbook*, 5th ed. .

