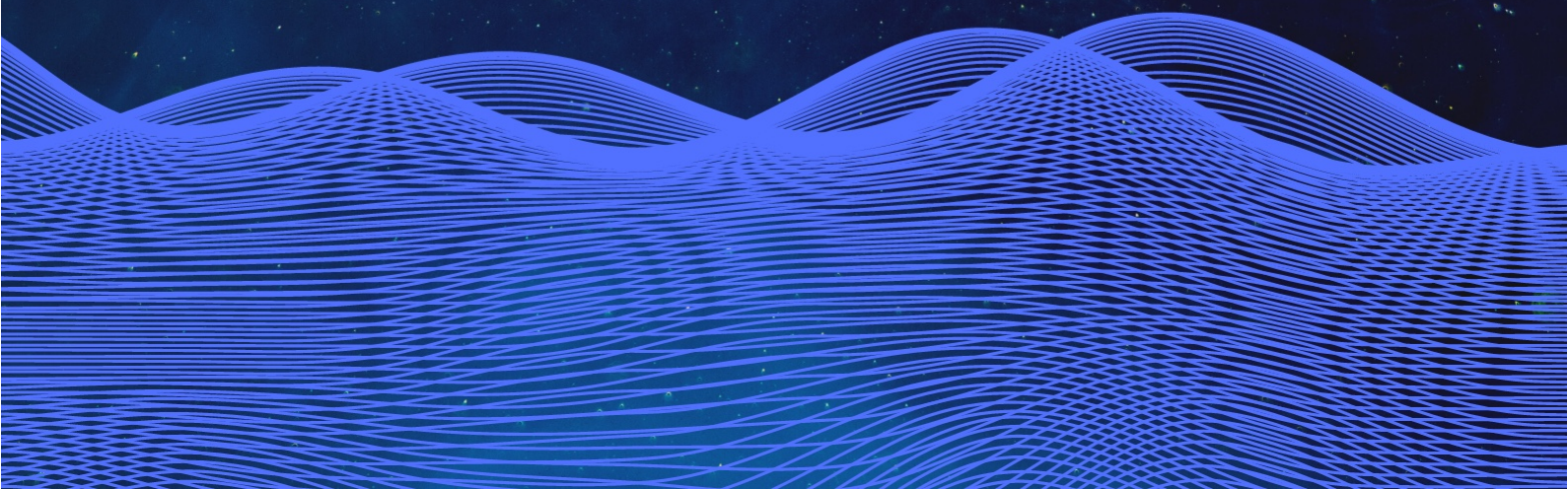
ULISES GASCÓN GONZÁLEZ

DOCKER SEGURO



Docker Seguro

Ulises Gascón González

Este libro está disponible en https://dockerseguro.ulisesgascon.com
Esta versión se publicó en 2022-07-23
ISBN *978-84-09-44492-2*
Versión: 1.0.0

Este libro está licenciado como Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License aunque no necesariamente las imágenes de su interior.

¡Un tweet sobre el libro!

¡Ayuda a Ulises Gascón hablando sobre el libro en Twitter!

El tweet sugerido para este libro es:
Acabo de conseguir el #ebook #Dockerseguro de @kom_256 en https://dockerseguro.ulisesgascon.com

Descubre lo que otra gente está diciendo sobre el libro con el hashtag *#DockerSeguro* en Twitter

“Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program”.
Linus Torvalds

Sobre el autor

Ulises Gascón (He/Him)

Open Source Developer & Senior Software Engineer

Más información:

- Biografía completa
- Open Source
- Blog
- Charlas
- Github
- Twitter
- Linkedin

Sumario

Introducción

Es importante remarcar, que este libro está orientado a un usuario medio/avanzado de Docker que ya tiene experiencia con el manejo de contenedores y con la creación de imágenes.

Si aún no tienes esta experiencia, te recomiendo el curso DOCKER DE NOVATO a PRO! de Pablo Fredrikson ("Pelado Nerd") y poner tus conocimientos en práctica antes de seguir la lectura.

Docker es una de las tecnologías más populares de los últimos años y es bastante probable que te tengas que enfrentar a crear imágenes y lanzar contenedores a lo largo de tu carrera.

Aunque es relativamente sencillo dar los primeros pasos en este universo y lanzar nuestro primer `hello-world` con un solo comando docker `run hello-world`, estamos aún lejos de sentirnos cómodos poniendo nuestros contenedores en producción.

En los entornos actuales donde trabajamos con metodologías ágiles y seguimos una cultura DevOps, nos damos cuenta rápidamente, que no es suficiente con tener nuestro código listo. También tenemos que dar un paso más e involucrarnos en la parte operacional de la tecnología, entendiendo qué ocurre, una vez que publicamos nuestros cambios en el repositorio del equipo.

Si ya te has tenido que enfrentar a poner contenedores en producción o estás abstraído de la tarea con orquestadores como Kubernetes, este libro te ayudará a entender todo el margen de maniobra que se te ofrece con Docker para crear imágenes mucho más seguras y sólidas (capítulo 3).

Así mismo, este libro te ayudará a comprender qué riesgos tenemos en cada fase del ciclo de vida (capítulo 1.2) y sobre todo qué herramientas (capítulo 4) y técnicas podemos usar en nuestro día a día para desplegar contenedores sin miedo.

Docker lleva dando vueltas desde hace casi 10 años. En este tiempo, muchas veces hemos encontrado vacíos sobre cómo securizar las imágenes, y aunque existen muchos recursos accesibles, suelen estar desperdigados y ser muy dogmáticos.

He querido condensar mi experiencia con Docker en un libro sencillo y ameno, de fácil lectura.

Veremos cómo resolver muchas situaciones cotidianas en entornos productivos, sin intentar dogmatizar sobre qué camino es mejor o peor, nos centraremos en entender los riesgos y mitigaciones para que tu saques tus propias conclusiones adaptadas a tus necesidades y experiencia.

Espero que disfrutes de la lectura de este libro, tanto como yo he disfrutado al escribirlo.

Capítulo 1: El reto de los contenedores

En este capítulo centraremos los esfuerzos en entender los retos y posibilidades que nos ofrece Docker.

Repasamos el ciclo de vida de los contenedores y los pilares sobre los que se asienta Docker. Finalmente veremos en profundidad todas las amenazas y vectores que tenemos cuando usamos y creamos contenedores con Docker, para poco a poco ir centrándonos en las mitigaciones disponibles en los siguiente capítulos.

1.1 Estado del Arte

Si vemos cualquier informe sobre el uso de Docker nos sorprenderá la gran aceptación que ha tenido. Este cambio de paradigma en el mundo del software ha facilitado mucho las cosas a base de añadir una abstracción sobre el despliegue de nuestras aplicaciones.

Entiendo que es responsabilidad de todos entender y manejar estas abstracciones lo suficiente, como para poder hacer despliegues de forma segura.

Cuando vemos informes como "The state of Open Source Security Report 2022" de Snyk, podemos ver claramente la tendencia actual de tener cada vez más dependencia de otras librerías y observamos que no estamos todavía entendiendo el problema que ello supone, sino tenemos unas políticas claras en nuestra relación con el open source y en cómo vamos a mantener esas dependencias actualizadas (capítulo 2.2.1).

Con Docker, también introducimos más dependencias ya que consumiremos imágenes creadas por terceros (capítulo 3.1.1).

Si revisamos el historial de vulnerabilidades reportadas y parcheadas de Docker, nos damos cuenta que esta tecnología es bastante compleja, y si no estamos familiarizados con su arquitectura o su ciclo de vida difícilmente podremos entender cuando esas vulnerabilidades reportadas son un riesgo real para nuestros proyectos o nuestra organización.

Existe una idea creciente de que los contenedores son tan aislados, robustos, inmutables y seguros como los contenedores marítimos. La realidad es un poco distinta a esa imagen idealizada que tenemos. Se puede llegar a tener contenedores robustos y seguros, pero no es tan sencillo como mover contenedores marítimos con nuestra grúa imaginaria.

Fotografía de Bernd Dittrich

Como veremos a lo largo del libro, Docker se apoya en muchas tecnologías existentes para lograr ese concepto idealista de contenedor marítimo, entre otras en Kernel namespaces y Control Groups (capítulo 2.3).

A finales de 2020 corrió como la pólvora el anuncio de que Kubernetes en su versión 1.20.0 "retiraba el soporte a Docker" y el pánico se hizo presa de la comunidad. En realidad lo que hizo Kubernetes fue retirar el soporte a `dockershim`.

Kubernetes llegó incluso a publicar una entrada en su blog explicando en detalle porque esto no sería un drama para nadie en realidad (salvo que usarás comandos de Docker directamente) y terminó añadiendo también un FAQ específico sobre este tema.

Personalmente, no me sorprende que se creara un gran alarmismo, ya que a día de hoy muchas personas siguen pensando que Docker es contenedores y que no existe nada más allá.

El concepto de los contenedores ya existía antes de Docker y aunque mucho de la adopción de este patrón se lo debemos a Docker Inc, lo cierto es que existe un mundo gigante donde ya tenemos la Cloud Native Computing Foundation y se han creado varios estándares e iniciativas para normalizar esta tecnología de una forma sólida y homogénea.

1.2 Los pilares de Docker

Aunque los contenedores han sido un gran avance en la adopción de la nube y las prácticas DevOps, no nos exime de conocer, prevenir y mitigar los posibles ataques que suframos. Este puede parecer una tarea abrumadora en un principio, sin embargo a lo largo del libro encontrarás estrategias y recursos que te ayudarán a superarlo.

El vector de ataque principal en el caso de los contenedores, se basa en la relación que existe entre el host y el propio contenedor. De esta forma, a medida que entendamos qué compartimos entre el host y el contenedor a nivel del sistema operativo y qué recursos podemos compartir o no con el contenedor, podremos prevenir o mitigar la mayoría de los vectores de ataques conocidos a día de hoy con Docker.

Arquitectura

Uno de los puntos más complicados de entender cuando trabajamos con Docker, es comprender que el Kernel es compartido entre el host (capítulo 2.1.3) y los contenedores, no como sucede con las máquinas virtuales que utilizan una solución Hypervisor

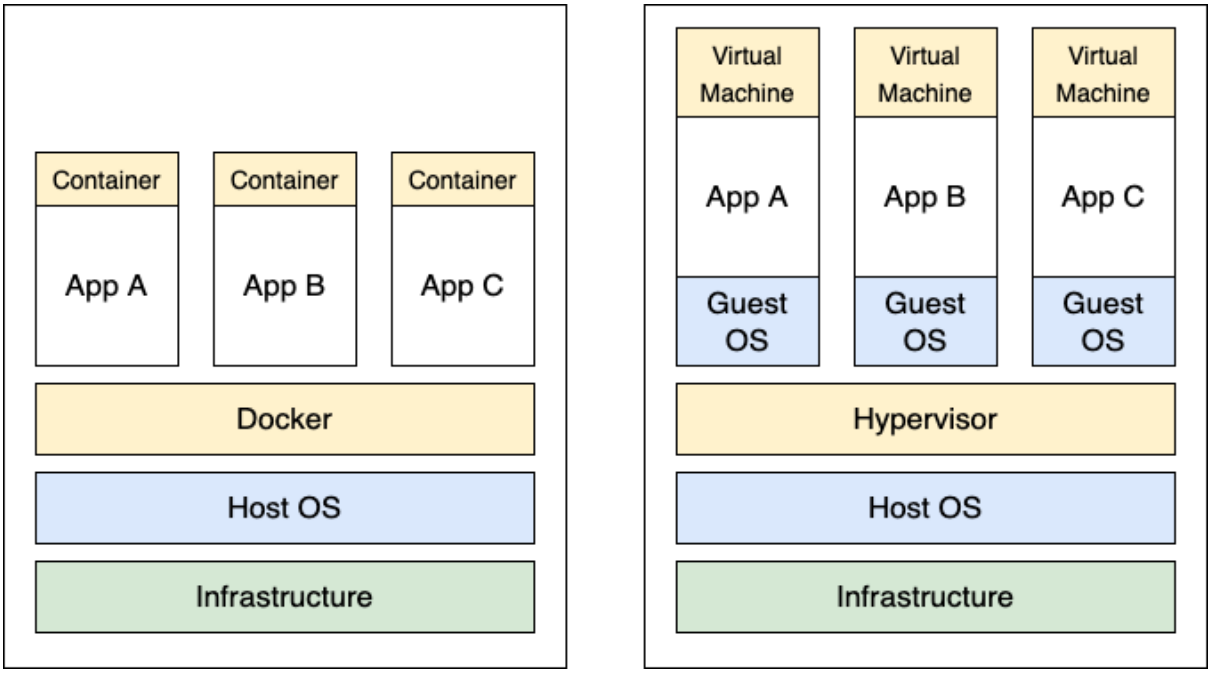


Imagen derivada de Geek Flare para adaptar el formato

Como resultado de compartir el Kernel podemos sufrir una serie de ataques:

- Cuando escapamos del contenedor y tenemos acceso a otros contenedores o al propio host. Ejemplo del uso de Dirty Cow (CVE-2016-5195) para salir del contenedor (capítulo 2.2.4).

- Cuando no gestionamos correctamente el uso de namespaces y permitimos que un contenedor pueda tener acceso root a la máquina host o muchos de los recursos y dispositivos presentes (capítulo 2.2.2).
- Al no imponer límites a los contenedores, podemos sufrir denegaciones de servicio dentro de la máquina host, impidiendo que recursos legítimos accedan a módulos y recursos del sistema host por falta de recursos (capítulo 2.2.5).

Ciclo de vida

Cada parte del ciclo de vida tiene sus propios retos, pero teniendo unas estrategias claras podremos hacer frente sin problemas.

Dockerfile

En muchas ocasiones usaremos imágenes de terceros directa o indirectamente como base para las que nosotros creemos (añadiendo nuestras propias capas).

Podemos hacer mucho a la hora de crear nuestras propias imágenes para mejorar la seguridad, como utilizar imágenes firmadas y para minimizar los riesgos por ejemplo tener una política bien definida sobre qué ficheros y carpetas formarán parte de la imagen (capítulo 3).

Imagen

Las imágenes deben poder subirse y descargarse de una forma segura de los registros públicos o privados que usemos (capítulo 3).

Contenedor

Crear imágenes es una cosa, pero mantener nuestros contendores corriendo en entornos productivos tiene muchos retos de seguridad que tienen que ver con cómo establecemos esa relación con la máquina host (redes, ficheros, dispositivos...) y cómo hacemos políticas efectivas (Seecomp, AppArmor, permisos, capacidades), sin olvidarnos de la configuración del propio demonio de Docker y las actualizaciones regulares de los contenedores y del propio host (capítulo 2)

Dependencias

Otro punto importante, es entender que Docker parte de la base de que el código que estamos ejecutando en esos contenedores es fiable y que nosotros como usuarios hemos

validado previamente que esa imagen se corresponda con lo que nosotros esperamos. Si nosotros decidimos ejecutar imágenes de Docker que no cuentan con garantías de seguridad (potencialmente maliciosas) o que entrañan algún riesgo, podríamos poner en peligro nuestra máquina host o nuestra propia infraestructura, por ejemplo compartiendo variables de entorno u otros datos sensibles con contenedores malintencionados (capítulo 2.2.3).

En ocasiones usamos imágenes que son confiables, pero sus dependencias no son inmutables, pudiendo generar vulnerabilidades a través de ellas, lo que se conoce como Supply Chain attacks

Muchas de las dependencias de nuestros proyectos dependen directa o indirectamente del Software Libre. Esto lleva años generando presión sobre una comunidad de maintainers que está muy lejos de ser sostenible con garantías.

Desde hace algún tiempo se enmascara este problema estableciendo una metáfora con el concepto de "Supply Chain".

> *"When you take on an additional dependency in a software project, often money does not change hands. npm install and cargo add do not bill your credit card. There is no formal agreement between a maintainer and its downstream users". **Iliana Etaoin***

1.3 Threat Modeling

La fundación (OWASP) incluye un modelo de amenazas específico para Docker que nos ayuda a visualizar en una sola imagen todo lo que tenemos que tener en cuenta

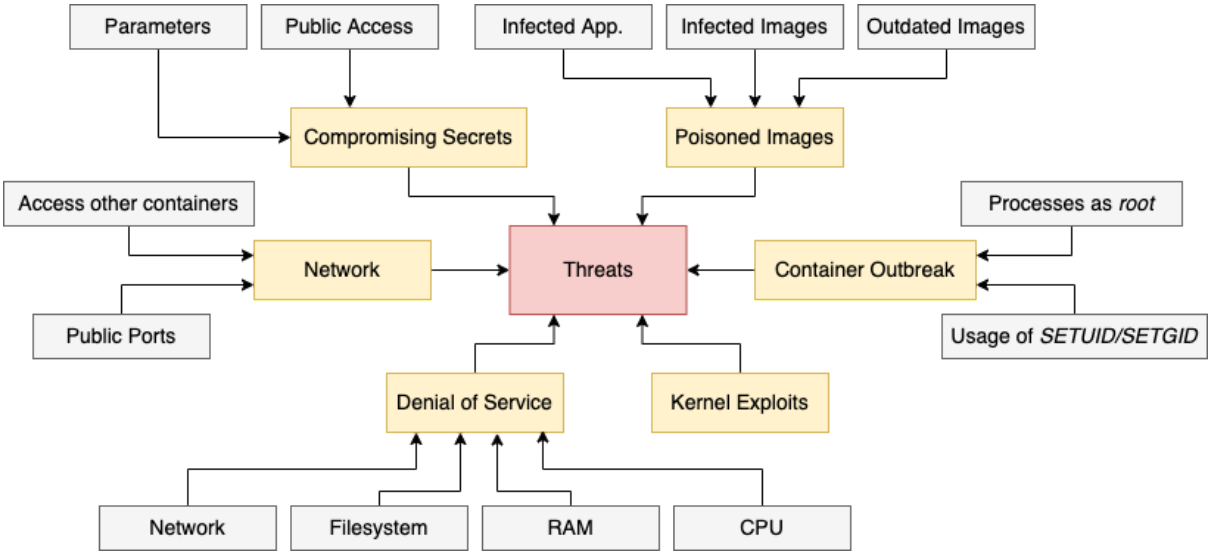
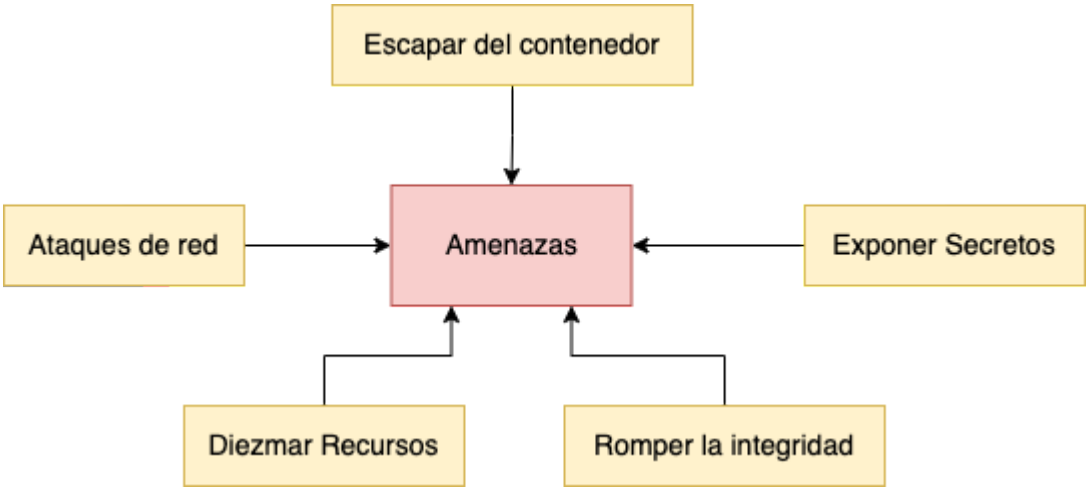


Imagen derivada de OWASP para adaptar el formato

Para hacerlo más sencillo y didáctico he decido hacer una clasificación alternativa al modelo de amenazas que plantea la fundación OWASP, tomándome la licencia de agrupar y reducir algunas casuísticas para facilitar la comprensión.



Escapar del contenedor

En un ataque contra contenedores, el primer paso es intentar hacerse con el control del contenedor y el siguiente paso será escalar permisos e intentar hacerse con el control de la máquina host.

Si realizamos una mala configuración del contenedor y permitimos el acceso `root` o permitimos la ejecución del contenedor con muchas capacidades/permisos, podemos acabar comprometiendo la máquina host (capítulo 2.1.3)

Ataques de red

Los contenedores muy frecuentemente están abiertos al tráfico a través de uno o múltiples puertos. La configuración de redes es un tema complejo que requiere su tiempo (capítulo 2.2.4).

Si hacemos una configuración incorrecta de red podemos acabar en escenarios de todo tipo cuando ese contenedor se ve comprometido con acceso a la red:

- Contenedor que comparte red con el Host. Esto permite hacer ataques a la interfaces vía red de los orquestadores presentes en el host, así como otras aplicaciones que tengan puertos abiertos, etc... En casos muy extremos podríamos incluso sufrir una expansión del ataque a la red donde la propia máquina host está conectada.
- Contenedor que comparte red con otros contenedores (sin necesidad de ello). Esto brinda la posibilidad de expandir el ataque a otros contenedores vía red.

Diezmar recursos

Ya sea porque estemos sufriendo un ataque intencionado o porque nuestro contenedor tenga un error de diseño como un memory leak, podemos fácilmente drenar los recursos de la máquina host si no establecemos unos límites a nuestros contenedores (capítulo 2.2.5).

Hacer que la máquina host se vuelva inestable por falta de recursos es otra forma de sabotear una infraestructura. Si además esa máquina host se encarga de orquestar otros servicios, podemos en ocasiones generar verdaderos problemas con nuestros clientes.

Este tipo de problemas de recursos pueden tornarse rápidamente muy costosos en términos económicos si estamos usando infraestructuras escalables en la nube y no hacemos una configuración correcta de las mismas.

Romper la integridad

Hablaremos de *poisoned images* (imágenes envenenadas) cuando estas estén desactualizadas y existan vulnerabilidades conocidas que puedan comprometer su contenido. También incluimos el caso de dockerizar un proyecto y que este incluya vulnerabilidades en sus propias dependencias o que las imágenes descargadas no sean las que esperamos, sufriendo así un ataque man in the middle (capítulo 2.3.3).

Un ejemplo sencillo: Creamos una imagen de Docker que por un error de diseño en nuestro propio código permite inyecciones SQL y que se exploten para la ejecución de comandos a través de una reverse shell.

Exposición de secretos

Este riesgo está siempre presente, podemos accidentalmente exponer información sensible por un error en nuestra lógica de negocio en la aplicación o por contar con alguna vulnerabilidad conocida, como un servidor web que permita un ataque tipo path traversial y acabe exponiendo nuestro fichero `.env` o similares (capítulo 2.2.3).

Pero también existen escenarios donde excedemos el nivel de información a la que accede un contenedor, por ejemplo dando acceso completo a un volumen del host cuando solo necesitaria una carpeta específica o garantizando acceso `root` al contendor con privilegios suficientes como para poder montar los volúmenes de la máquina anfitriona en el contenedor (capítulo 2.2.2).

1.4 OWASP Docker Top 10

La fundación OWASP ofrece desde hace años el OWASP TOP 10 y nos orienta sobre los 10 aspectos más críticos a tener en cuenta para securizar nuestras aplicaciones web según las tendencias que se han detectado en años anteriores.

Para docker existe un proyecto similar conocido como OWASP Docker Top 10. En este libro, expondremos de forma específica, como mitigar los retos que nos presenta la lista.

Haremos de esta sección, un índice rápido y alternativo para este libro que está muy relacionado con el modelo de amenazas (sección 1.2).

D01 - Secure User Mapping

Los privilegios en Docker pueden ser muy contraproducentes para nuestro host, para evitarlo debemos tener claro cómo manejar root (capítulo 2.1.3).

D02 - Patch Management Strategy

A lo largo del tiempo se van descubriendo vulnerabilidades en los sistemas que utilizamos y que los fabricantes o maintainers van parcheando. Es importante tener una política clara sobre cómo llevar a cabo el parcheado de nuestro sistema Host, así como de los contenedores asociados. Puedes encontrar toda la información en los capítulos 2.2.5 y 2.2.1.

Recordemos que a su vez este punto está conectado con el OWASP top 10 general. En esta edición se conoce como A06:2021 – Vulnerable and Outdated Components, ya que es común que no actualicemos, incluso cuando existen vulnerabilidades conocidas que están siendo explotadas de forma activa.

D03 - Network Segmentation and Firewalling

El diseño y la evolución de las redes que organices en tu sistema condiciona bastante los posibles vectores de ataque entre contenedores y sobre todo entre el host y los contenedores.

A lo largo de los últimos años se han realizado diversos cambios en el demonio de Docker, *depreciando* ciertos comandos que se han usado de forma asidua. Puedes encontrar todas las referencias en el capítulo 2.2.4.

D04 - Secure Defaults and Hardening

Independientemente de cómo controlemos el ciclo de vida de nuestros contenedores, necesitaremos tener claro que tanto los contenedores, como el host y el orquestador si lo hubiera, deben ser revisados y bastionados desde un principio, independientemente de que el entorno en el que nos encontremos sea local o productivo. Para ello dedicamos el capítulo 2 y especialmente la sección 2.2.1 a ello.

D05 - Maintain Security Contexts

Independientemente del sistema de orquestación que utilices, es importante evitar que los contenedores se mezclen o compartan recursos entre sí, cuando no sea necesario o por errores de diseño.

Por esta razón es importante introducir políticas de límites y artefactos que permitan mantener los contextos aislados, de lo que hablamos en el capítulo 2.3.

D06 - Protect Secrets

La gestión de secretos siempre es un reto, porque podemos acabar filtrando de forma accidental en los logs, o quedar expuestos a actores maliciosos. A esto dedicamos el capítulo 2.3.3.

D07 - Resource Protection

La limitación de recursos es necesaria cuando usamos contenedores para evitar escenarios de tipo Denegación de Servicios (DoS) en el host por falta de recursos. Ver capítulo 2.2.5.

D08 - Container Image Integrity and Origin

Docker parte de la premisa como vimos en la sección anterior (capítulo 1.3), que los contenedores que usamos en nuestro sistema consumen imágenes seguras y verificadas por nosotros.

Esto en ocasiones supone un reto porque debemos de elegir entre las muchas imágenes potenciales desarrolladas por la comunidad. Para ello hemos dedicado las secciones 2.2.1 y 3.1.1. También podemos hacer uso de mecanismos avanzados como la validación de la firma de imágenes (sección 2.3.3).

D09 - Follow Immutable Paradigm

La inmutabilidad es uno de los pilares que tenemos en el DevOps moderno, es fácil romper la inmutabilidad aunque Docker de base ya nos ofrezca un marco de trabajo con las herramientas para lograr la inmutabilidad de nuestras imágenes.

Para lograr la inmutabilidad debemos tener claro cómo elegir imágenes base y sus tags (secciones 2.2.1 y 3.1.1), así como la gestión de dependencias (sección 3.1.3 y 3.1.4).

D10 - Logging

A lo largo del libro recomendamos algunas herramientas que pueden hacernos la vida más fácil en cuanto a revisar nuestras imágenes y asegurarnos que seguimos las mejores prácticas de una forma implícita (ver capítulo 4). Pero independientemente de las

herramientas que usemos, es importante al menos tener logs de lo que pasa con nuestros contenedores, sobre todo en entornos productivos, de lo que hablamos en el capítulo 2.1.4.

Capítulo 2: Desplegando contenedores de forma segura

En este capítulo centraremos los esfuerzos en como mantener nuestros contenedores corriendo de forma segura en nuestra máquina(s) siguiendo recomendaciones de seguridad y de buenas prácticas.

Hablaremos de cómo fortificar nuestro host y cómo limitar recursos y permisos en nuestros contenedores. También veremos algunas estrategías sencillas para compartir recursos de forma eficiente con nuestros contenedores.

Así mismo hablaremos en profundidad de cómo podemos hacer uso de las capacidades de Red del demonio de Docker para aislar la comunicación de los contenedores entre sí.

2.1 Host

Cuando hablamos de Host nos referimos a cualquier máquina que esté corriendo el demonio de Docker y sea la capa que se interpone entre el demonio y la propia infraestructura.

En muchos entornos productivos, se suelen usar herramientas que permiten la orquestación de contenedores de una forma más sencilla, por ejemplo Kubernetes, Docker Swarm, Openshift... Cada una de esas herramientas tiene su propio enfoque de orquestación de contenedores y sus propias recomendaciones de seguridad y buenas prácticas. Nosotros en este capítulo queremos centrarnos en un entorno simple y puro, donde solo contamos con el sistema operativo del host y el demonio de Docker para orquestar nuestros contenedores.

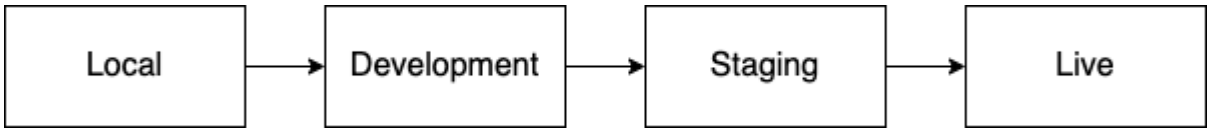
Muchas de las recomendaciones y estrategias que veremos a continuación son fácilmente portables a sistemas de orquestación de contenedores.

2.1.1 Bastionado

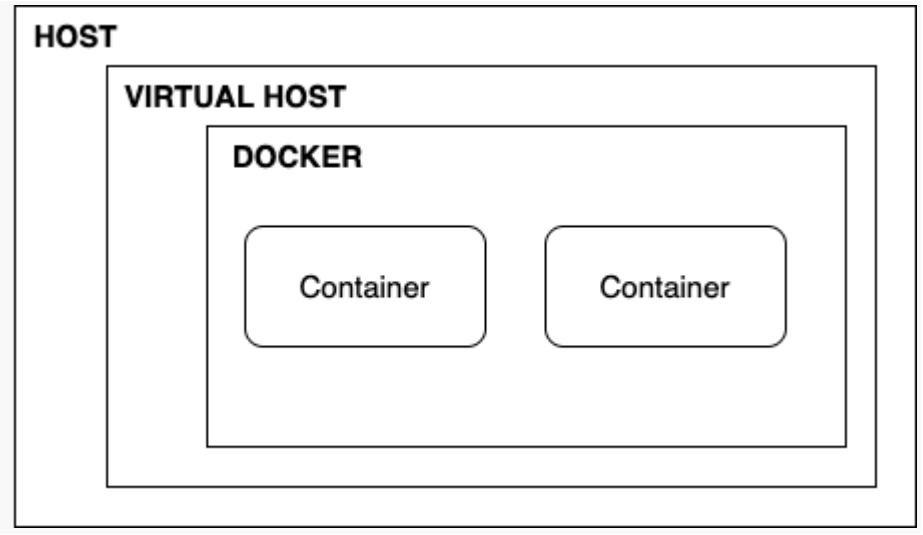
Aunque el bastionado es un proceso ciertamente complejo, podemos plantear al menos dos escenarios bien diferenciados a la hora de trabajar con Docker en entornos de desarrollo modernos (Agile, DevOps…).

Un primer entorno podría ser local. La máquina que se utiliza para las actividades de desarrollo de software, QA, etc.. donde esa máquina suele utilizarse como ordenador personal/trabajo con entorno gráfico, acceso a aplicaciones de mensajería como Slack, etc... y que no se dedica al despliegue de aplicaciones para recibir tráfico de fuera. Este entorno que se suele denominar local es donde más volatilidad y dinamismo tenemos con respecto a los contenedores, ya que los equipos suelen trabajar con diferentes contenedores y relaciones entre ellos, especialmente si se usan microservicios o se desarrollan librerías.

Otro entorno podría ser aquella(s) máquina(s) donde desplegamos los contenedores que esperamos sean estables a lo largo del tiempo y que soporten tráfico de internet o realicen un trabajo productivo definido. Este tipo de entornos suelen ser máquinas específicas que se usan solo para esto y que normalmente suelen ir evolucionando y modificando su estado a lo largo del tiempo, a través de trabajos y pipelines de integración continua o despliegue continuo. En entornos de trabajo moderno solemos encontrarnos al menos con 2 entornos `staging` y `live`. Siendo `live` el más crítico para nuestros objetivos empresariales y de seguridad.



En muchas ocasiones no prestamos la suficiente atención al entorno local que por su naturaleza distinta para cada individuo, hace que sea difícil de armonizar y bastionar desde un punto de vista IT / CISO. Un primer consejo sería el hacer uso de máquinas virtuales dentro de nuestro entorno local que nos permitan aislar mucho más el impacto de seguridad que pueda tener una mala configuración de Docker en nuestra máquina host.



Además sumariamos esta ventaja adicional, ya que podríamos desplegar esos contenedores en nuestra máquina virtual, compartiendo el mismo sistema operativo de nuestro entorno de `live` o `staging`, evitando así posibles limitaciones de Docker con MacOs o Windows. Obviamente este enfoque nos obliga a tener una máquina con recursos suficientes, y también cierto manejo de máquinas virtuales y su ciclo de vida, así como los volúmenes compartidos o los recursos entre `host <-> virtual host <-> docker <-> container`. Este tipo de enfoque es especialmente interesante cuando queremos tener un entorno más de laboratorio para explorar imágenes de terceros o cuando no confiemos en el contenido de las mismas.

Independientemente de usar máquinas virtuales o no, deberemos seguir ciertas recomendaciones.

Actualizaciones

Es fundamental mantener nuestro host actualizado, tanto el sistema operativo como el demonio de Docker. Históricamente la actualización del demonio de Docker podría arrastrar caídas de servicio ya que los contenedores se paran cuando paramos Docker (para su actualización). Un modo sencillo para evitarlo es hacer uso de la capacidad live-restore que no para los contenedores al cerrarse el demonio y que reconecta con ellos una vez se levanta el demonio de nuevo.

```
dockered --live-restore
```

Antivirus

Cada vez es más común en el mundo empresarial el uso de antivirus En la propia documentación de Docker se menciona que los escaneos sobre las carpetas que usan los contenedores pueden generar errores inesperados. La solución en este caso sería excluir los directorios de Docker pero hacer escáneres cuando los contenedores no están corriendo, usando cron jobs o una planificación adecuada.

2.1.2 Gestión de disco

Particiones específicas para los contenedores

Aunque Docker cuenta con un sistema bastante eficiente de gestión de las imágenes usando chunks y hashes, es muy habitual que nuestros contenedores ocupen cada vez más y más espacio en disco. Una buena recomendación sería mover la ubicación por defecto que usa Docker para la permanencia en disco `var/lib/docker` a un disco o partición independiente donde podamos hacer un buen mantenimiento y monitoreo del espacio.

Purgar Docker regularmente

Por la naturaleza de Docker, y de los contenedores efímeros, es común ocupar el espacio en disco con elementos que ya no son útiles. Una forma de purgar es usando el comando `prune` de docker y sus diversas variaciones.

Si todos los contenedores y sus elementos relacionados (volúmenes, redes, imágenes..) son prescindibles podemos hacer:

```
# Paramos todos los contenedores
docker kill $(docker ps -q)
# Borramos todos los contenedores
docker rm $(docker ps -a -q)
# Purgamos el sistema completamente (incluyendo volúmenes)
docker system prune --volumes
```

2.1.3 El poder de root

Si profundizamos en los pilares de Docker (capítulo 1.2) es fácil llegar a la conclusión de que en realidad al compartir el kernel si somos `root` en el contenedor también lo somos en el host.

Siendo la premisa siempre evitar la ejecución de un contenedor como `root`.

> *Repeat after me: "friends don't let friends run containers as root!"*
> ***Node.js Docker Cheat Sheet***

Root siempre es Root

Si no hacemos nada de forma proactiva o utilizamos un orquestador de contenedores bien configurado, es bastante probable que estemos lanzando los contenedores como root:

```
docker run -it busybox
/ # id
uid=0(root) gid=0(root) groups=10(wheel)
```

Contenedores Rootless

Si hacemos uso de Linux namespaces y la documentación oficial de Docker vemos que ya existen formas para poder ejecutar un contenedor sin ser root. En algunas ocasiones las imágenes que usamos ya definen este comportamiento usando la instrucción `USER` (capítulo 3.1.2), pero otras veces no y por ello necesitamos hacer uso del argumento `--user` del demonio de Docker, como veremos a continuación.

Modo non-root (con el usuario `nobody`):

```
docker run -it --user nobody busybox
/ # id
uid=65534(nobody) gid=65534(nobody)
```

De todas formas si utilizamos un orquestador como Kubernetes, encontramos que ya existen mecanismos que nos permiten gestionar este tipo de casos, simplemente ajustando las especificaciones de seguridad:

```
spec:
  securityContext:
    runAsNonRoot: true
```

Demonio de Docker Rootless

Desde la versión 19 de Docker, podemos hacer uso del modo rootless a nivel del demonio y desde la 20 se considera estable. Este modo tiene algunas limitaciones, pero tiene algunas ventajas que no podemos dejar escapar sin más.

Así pues la forma de instalarlo sería casi idéntica al demonio de Docker convencional (ejecutalo como usuario sin privilegios) aunque tendremos que instalar previamente otras dependencias como `newuidmap` y `newgidmap`:

```
curl -sSL https://get.docker.com/rootless | sh
```

Lo que nos ofrece esta versión del demonio de Docker es poder ejecutar el demonio bajo un usuario sin privilegios, lógicamente este comportamiento también se extiende a los contenedores que maneja el demonio.

Ahora bien, tiene algunas desventajas que aparecen al perder los privilegios. Como cabría esperarse no se podrá usar `--net=host`, algunos drivers de almacenamiento dejarán de funcionar y además perderemos capacidades de exponer puertos privilegiados (< 1024) entre otras..

Pero no deja de ser una buena opción si no te limita la pérdida de privilegios, especialmente en entornos productivos.

2.1.4 Monitorizar

Aún no siendo un entorno productivo, es siempre importante monitorizar nuestros contenedores para detectar anomalías que podrían alertarnos de potenciales vulnerabilidades.

Si contamos con pocos contenedores o no necesitamos una visión a largo plazo siempre podemos hacer uso del comando `stats` de Docker para sacar las métricas de nuestros contenedores

En el capítulo 4 hablamos sobre herramientas que pueden ayudarnos a gestionar múltiples aspectos:

- Con Dockprom (Capítulo 4.9) podemos monitorizar nuestros contenedores fácilmente y de una forma visual con Grafana pero además contamos con la posibilidad de incluir alertas
- Con K6 (Capítulo 4.10) podemos fácilmente detectar problemas a la hora de escalar nuestros contenedores, especialmente con servicios web. Para entender la fragilidad a la que nos encontramos, expuestos cuando no hacemos pruebas de carga las vulnerabilidades como CWE-1333: Inefficient Regular Expression Complexity basadas en una Expresión regular deficiente, pueden provocar una denegación de servicio
- Revisar los logs de nuestros contenedores es también importante ya sea usando los drivers de Docker o alguna solución más compleja.

2.1.5 Parches de seguridad

> *“Security is a process, not a product”.* ***Bruce Schneier***

Cuando damos el paso de usar contenedores, añadimos más capas a nuestra infraestructura. Ya que la virtualización es un ladrillo más en la pared que vamos construyendo.

Esto supone un reto a la hora de parchear y actualizar ya que cada parte (Sistema operativo, sistema de orquestación, imágenes, librerías...) deberá ser actualizado de forma frecuente.

Aunque es un proceso que parece sencillo, está intrínsecamente lleno de retos especialmente si estamos manteniendo sistemas arcaicos que hace uso de interfaces antiguas o obsoletas.

Esto se ha convertido en un problema cada vez más importante y ganado importancia en el OWASP top 10 (A09-2017 Using Components with Known Vulnerabilities y A06-2021 Vulnerable and Outdated Components) ya que el tiempo pasa y las vulnerabilidades descubiertas siguen creciendo en nuestra contra.

La mejor forma de abordar este tema es darle el peso que se merece y crear un plan sostenible en el tiempo que permita de una forma automatizada nos ayude con la tediosa tarea de actualizar y parchear de una manera frecuente a medida que las vulnerabilidades van apareciendo en nuestro extenso sistema de dependencias.

2.2 Contenedores

Los contenedores han supuesto toda una revolución en la experiencia del desarrollo para millones de personas en todo el mundo. A la hora del despliegue tenemos que tener algunos factores en cuenta para mantener un buen aislamiento y control de los recursos que compartimos entre el host y los contenedores, debemos cerciorarnos que las imágenes que utilizamos son de suficiente calidad para nuestros objetivos.

2.2.1 Imágenes base

Para crear un contenedor, primero necesitamos una imagen de la que partir. No es necesario crear nosotros mismos las imágenes, ya que en muchas ocasiones encontraremos imágenes ya disponibles en los registros públicos más populares.

Sin ir más lejos Docker Hub cuenta ya con más de 8.3 millones de repositorios, esto hace que elegir una imagen con la que trabajar sea una tarea complicada. Veamos algunos criterios que pueden ayudarnos.

Origen confiable

Docker hub divide las imágenes en varias categorías:

- **Imágenes Oficiales (Docker Official Image)**: Son un conjunto de imágenes que ayudan a la comunidad bien por ser la base de otras imágenes o por seguir las mejores prácticas, etc... .
- **Publicadores verificados (verified publisher)**: Suelen ser imágenes de mucha calidad y cuyos autores forman parte del programa de verificación de publicadores de Docker, normalmente entidades comerciales detras de productos relevantes
- **Programa Open Source (Open Source Program)**: Imágenes publicadas y mantenidas por miembros del Programa Open Source de Docker.
- **Las demás**: Aquellas cuyos autores no forman parte de ningún programa y están normalmente sujetas a límites de uso.

Muchas de las imágenes más descargas son imágenes oficiales, por ejemplo Mongo, Nodejs, Python, Mysql, entre otras. Esto no quiere decir que las imágenes no oficiales sean peores o inseguras, simplemente intentaremos priorizar aquellas que sean oficiales. Las imágenes no oficiales incluyen una referencia al usuario por ejemplo `ulisesgascon/check-my-headers` en comparación con las oficiales que no incluyen el usuario, como `busybox`.

Popularidad

Otro criterio que puede ayudarnos a decidir entre imágenes similares, es su popularidad. Por regla general cuantas más estrellas tenga un repositorio, más influencia tendrá y más extendido será su uso. Lo que facilitará enormemente la tarea de buscar ayuda o documentación cuando lo necesitemos.

Mantenimiento

Tanto si estamos usando imágenes de Docker finales o como base para crear otras, es muy recomendable entender cómo de activo y mantenido está un repositorio. En ocasiones podemos toparnos con imágenes que aunque son populares no están mantenidas de forma activa, introduciendo así un vector de ataque especialmente importante cuando se reportan vulnerabilidades que requieren de un parcheo rápido.

En ocasiones así, es interesante contar con la opción de ser nosotros mismos quienes demos un paso al frente y decidamos mantener esa imagenes de Docker.

Minimalismo

Cuanto más simple es una imagen de Docker menos superficie de ataque nos ofrece, siendo importante aprender a usar tags más allá de `latest`. Como podemos ver es muy notoria la diferencia:

- `node:18` usa Debian 11 y cuenta con al menos 35 vulnerabilidades críticas

- `node:18-slim` usa Debian 11 y cuenta con solamente 40 vulnerabilidades leves
- `node:18-alpine` usa Alpine 3.16.2 y no cuenta con vulnerabilidades conocidas

Es cierto que no todas las vulnerabilidades potenciales de la imágenes se convierten en vulnerabilidades explotables en nuestros proyectos, pero es importante entender que usar imágenes vulnerables, nos hace vulnerables

> *"Docker images almost always bring known vulnerabilities alongside their great value."* ***Liran Tal***

Otra forma de alcanzar el minimalismo, es hacer uso de imágenes que hagan uso del sistema multi-stage (ver capítulo 3.1.6)

Tags

La inmutabilidad es uno de los conceptos más importantes del mundo DevOps, y esto se refleja en el uso de tags dentro de Docker. Si estás familiarizado con el versionamiento semántico, esto es llevar el proceso un paso más allá. Tomemos como ejemplo Node `node:<version>-<os>` o Python `python:<version>-<os>`:

```
FROM node:latest
FROM node:18
FROM node:18-slim
FROM node:18-alpine
FROM node:18.1.0-alpine3.15
```

```
FROM python:latest
FROM python:3
FROM python:3.10-slim
FROM python:3.10-alpine
FROM python:3.10.5-alpine3.16
```

Como vemos, muchos repositorios en Docker Hub han adoptado este enfoque. En términos de inmutabilidad sobretodo si estamos usando entornos productivos, es necesario definir una política de equipo/empresa en lo referente a la especificidad de las versiones, ya que `node:18-alpine` ahora puede ser `node@18.1.0 & alpine@3.1` pero dentro de tres meses puede ser `node@18.5.0 & alpine@3.1` haciendo que nuestra inmutabilidad no lo sea tanto y acabemos teniendo versiones de nuestro software dispares entre los diversos entornos que manejemos (local, staging, live...) siendo los bugs ocasionados por esta discrepancia bastante difíciles de detectar en muchas ocasiones.

2.2.2 Archivos y carpetas

Una de las funcionalidades más populares en el mundo de los contenedores es poder compartir información a nivel de ficheros y carpetas entre el host y los contenedores, siendo esta una actividad de riesgo si no tenemos claro los retos y fortalezas que presentan los volúmenes en Docker.

Tomemos como ejemplo el siguiente código:

```
sudo docker run -it -v /:/var/host-hd ubuntu /bin/bash
```

Al ejecutar el ejemplo anterior, estamos usando el usuario root, en principio si somos root en el contenedor también lo somos en host como vimos en el capítulo 2.1.3. Además de ello, hemos montado el directorio `/` como `/var/host-hd` en nuestro contenedor de Docker haciendo que sea fácil para el contenedor acceder a información sensible, borrar y modificar ficheros o carpetas.

Evidentemente este es un caso extremo y ayuda a darnos cuenta lo fácil que resulta exponer información de más con un contenedor y además con demasiados permisos heredados del propio host.

Docker ya introduce bastantes mecanismos para gestionar esto, además cada orquestador suele tener funcionalidades adicionales que nos ayudarán a compartir información de una forma más eficiente y segura.

Tipología

Por definición cualquier fichero o carpeta que se genere dentro de un contenedor estará disponible de forma efímera, cuando el contenedor muera esa información desaparecerá.

En caso de necesitar persistencia tendremos que apoyarnos necesariamente en el host, y esto puede hacerse de varias formas:

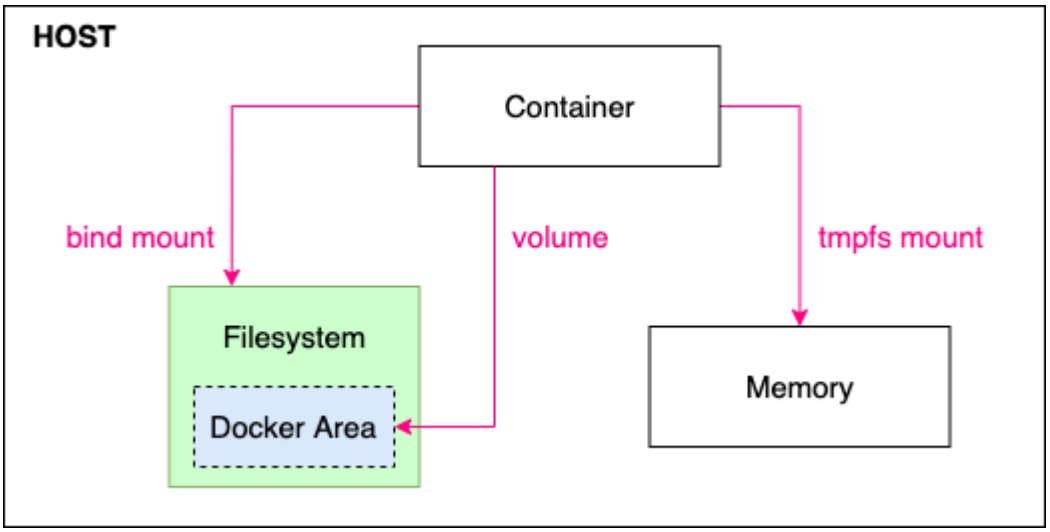


Imagen derivada de Docker para adaptar el formato

- **Volumes** se guardan en el host como designe Docker y no debería de usarse fuera del contexto de los contenedores
- **Bind mounts** son ficheros o carpetas (incluyendo información sensible) que se montan directamente en el contenedor. El contenedor puede modificar estos ficheros y carpetas en muchos casos
- **tmpfs** son ficheros y carpetas que usan solamente la memoria del host y no hacen uso del sistema de ficheros.

Lectura recomendada: Manage data in Docker

Modo lectura

Una forma efectiva de evitar la modificación de archivos en el host es hacer uso del modo solo-lectura cuando sea posible (ver `:ro`)

```
docker run -d -it -v /host_folder:/app:ro ubuntu
```

2.2.3 Gestión de secretos

Es muy frecuente que tengamos que compartir información sensible con nuestros contenedores, sobre todo en entornos productivos.

Esta información sensible puede ser desde pequeños artefactos como una llave privada o simples cadenas de texto que pueden contener contraseñas, tokens, etc.

Este tipo de información puede ser bastante sensible sobre todo si compartimos un token que permita el acceso a recursos clave fuera de nuestra red como un token de AWS con demasiados permisos o las credenciales para acceder nuestra base de datos de un entorno productivo.

Evidentemente, antes de compartir cualquier secreto deberemos de confiar que nuestra imagen es de confianza, siguiendo las recomendaciones del capítulo 2.1.1 y del resto del libro. Aunque no es muy frecuente, cada vez más podemos encontrar contenedores maliciosos en Docker Hub de los que podríamos ser víctimas potenciales.

Por supuesto, si almacenamos los secretos dentro de nuestras imágenes tendremos un problema ya que ese secreto será visible y extraíble con herramientas como Dive (capítulo 4.3), además dificultará enormemente su rotación, etc...

Una forma común de pasar secretos es mediante las variables de entorno haciendo posible que el secreto solo pase al contenedor que se ejecuta y no su imagen.

Podemos pasar las variables de entorno de forma individual con los argumentos `-e` o `--env`, pudiendo incluso hacer uso de variables ya presentes en la máquina host:

```
export MYVAR1=value1
docker run -e MYVAR1 --env MYVAR2=value2 ubuntu bash
```

O haciendo uso de un fichero específico tipo `.env` con el argumento `--env-file`.

```
cat .env
# This is a comment
MYVAR1=value1
MYVAR2=value2

$ docker run --env-file .env ubuntu bash
```

Este sistema tiene ciertas limitaciones que debemos conocer. Por un lado cualquier persona con acceso al contenedor podrá acceder a los secretos.

```
# El el host:
docker run -it --rm --name ubuntu -e MYSECRET=el_gran_secreto ubuntu
/bin/bash
# Dentro del contenedor:
env | grep MYSECRET
#MYSECRET=el_gran_secreto
```

Por otro lado si hacemos uso de comandos como `inspect` de docker podremos acceder a la misma información

```
docker inspect ubuntu -f "{{json: .Config.Env}}"
# ["MYSECRET=el_gran_secreto", ...]
```

Además, este sistema también es proclive a generar la exposición accidental de información sensible en nuestros logs, CWE-532: Insertion of Sensitive Information into Log File. Aunque esto dependerá enormemente de cómo hagamos esa configuración/centralización de logs.

Las variables de entorno son un mecanismo de mucha utilidad, pero para cierta información sensible en ocasiones necesitaremos otra estrategia. Una forma de mitigar este problema sería hacer uso de volúmenes (capítulo 2.2.2) o de Buildkit (capítulo 3.1.8). En entornos productivos debemos de confiar en los mecanismos que nos ofrecen los sistemas de orquestación de contenedores que usemos, por ejemplo con Docker Swarm o Kubernetes.

Como opción adicional, Docker Compose (capítulo 4.1) nos ofrece mayor comodidad al lidiar con volúmenes, ficheros tipo `.env` y variables de entorno, especialmente para uso local.

2.2.4 Redes

Sin duda uno de las funcionalidades más revolucionarias de Docker ha sido la facilidad con la que podemos gestionar la comunicación entre contenedores y el propio host mediante el uso de `network`.

La securización, aprovisionamiento y bastionado de redes es un tema en sí muy complejo que no está incluido en los objetivos de este libro, pero es importante remarcar algunos conceptos importantes a la hora de entender la seguridad de nuestros contenedores.

Cuando usamos orquestadores en entornos productivos, estos ya suelen incluir diversas formas de conectar y securizar los contenedores entre sí, y las formas de establecer subredes y mecanismos de comunicación seguros y eficaces.

Pero si usamos el demonio de docker en nuestro entorno local debemos tener varias cosas en consideración.

La configuración por defecto

Algo que resulta tan obvio en un primer vistazo en Docker, es que los contenedores pueden comunicarse entre sí aunque los puertos no hayan sido específicamente publicados o expuestos ya que usan `bridge` network por defecto.

Veamos un ejemplo ilustrativo con Nginx y busybox:

```
# Arrancamos Nginx
docker run --rm --name nginx-server --detach nginx

# Sacar la IP del Nginx (p.j: 172.17.0.2)
docker inspect nginx-server| grep IPAddress

# Arrancamos busybox y entramos en su terminal
docker run -it busybox sh

# hacemos un GET al Nginx
wget -q -O - 172.17.0.2:80
#<!DOCTYPE html>
#...
```

Una forma de evitar esto, es modificando el comportamiento del demonio de Docker usando el argumento `--icc=false`

```
dockerd --icc=false
```

Para asegurarnos que el cambio surtió efecto podemos buscar `com.docker.network.bridge.enable_icc:false` en las redes de docker disponibles

```
docker network ls --quiet | xargs docker network inspect --format '{{
.Name }}: {{ .Options }}'
```

Abusando de host

Una posibilidad que nos ofrece Docker es hacer uso de la red de la propia máquina host `--network=host` compartiendo así la red.

Al hacer esto, es fácil acabar exponiendo la máquina host a través de los contenedores si estos tienen alguna vulnerabilidad explotable.

Como ejemplo ilustrado de este tipo de ataques podemos leer este didáctico writeups como How to contact Google SRE: Dropping a shell in cloud SQL o Metadata service MITM allows root privilege escalation (EKS / GKE)

Uso de Link

Es común hoy en día encontrar referencias sobre el uso del argumento link cuando queramos que los contenedores puedan comunicar entre sí de forma explícita. A día de hoy, la idea sería portarnos al uso de `network`.

Establecimiento de redes

Docker hace uso del Container Network Model (CNM) que nos permite crear pequeñas redes segmentadas donde podemos hacer mejor control de los recursos, dejando así abierta la posibilidad de ir aislando y conectando elementos entre sí de una forma más segura.

Este trabajo es infinitamente más simple si hacemos uso de Docker Compose (capítulo 4.1) cuando tengamos que montar comunicaciones entre varios contenedores en el entorno local.

Lectura recomendada: Practical Design Patterns in Docker Networking de Dan Finneran

2.2.5 Limitación de recursos

Como en cualquier escenario de virtualización debemos preocuparnos de una forma proactiva y consciente del uso y sobre todo del mal uso de los recursos del sistema.

Los contenedores consumen los recursos que el host ofrece a través del demonio de docker, permitiendo así también limitar esos recursos para evitar ataques tipo DOS (ver CWE-400: Uncontrolled Resource Consumption) o que se comprometa la integridad del host por un memory leak en uno o varios contenedores.

Los orquestadores de contenedores en sistemas productivos suelen tener también mecanismos que podemos utilizar para poner límites a nuestros contenedores, algunos incluso proveen interfaces más avanzadas que las del propio demonio de Docker.

Limitar los reinicios

Docker ofrece la posibilidad de definir las políticas de reinicio con el argumento `--restart` evitando así la interrupción de servicio. Esto es especialmente útil si contamos con un sistema de escalado horizontal como en el caso de Kubernetes

La política más relevante a la hora de limitar recursos sería `on-failure` que nos permite limitar el número de reinicios máximos que permitimos a una imagen. En este ejemplo especificamos que 10 sería el máximo permitido.

```
docker run -d --restart on-failure:10 ubuntu
```

Evidentemente si una imagen ha terminado su tarea y decide cerrarse según se espera (codigo de salida 0) esta no será reiniciada.

Existen varias convenciones adicionales que deberemos tener presentes a la hora de entender como Docker aplica las políticas de reinicio. A su vez es interesante entender el papel que juegan los health checks en Docker, de lo que hablaremos en el capítulo 3.1.9.

También es importante remarcar que para poner límites a nuestros contenedores previamente tendremos que saber cuántos recursos necesita para funcionar de una forma correcta, siendo una primera opción utilizar herramientas como Dockprom (capítulo 4.9). Podemos hacer pruebas sin parar los contenedores para probar límites nuevos de CPU/Memoria ya que el comando `update` de Docker nos permite esa flexibilidad

```
# A un contenedor específico
docker update --cpuset-cpu ".5" --memory "500m" <ContainerNameOrID>

# A todos:
docker update --cpuset-cpus "1.5" --memory "350m" $(docker ps | awk
'NR>1 {print $1}')
```

Una forma interactiva de familiarizarnos con los límites en Docker es hacer uso de la imagen progrium/stress que ya nos provee una versión dockerizada de Stress.

Limitar CPU

La limitación de uso de CPU se puede hacer con diversos argumentos `--cpus`, `--cpu-period`, `--cpu-quota`, `--cpuset-cpus`, `--cpu-shares` lo que permite mucha flexibilidad.

En este ejemplo veremos cómo permitimos el uso de 1 CPU de dos formas equivalentes:

```
docker run -it --cpus="1" ubuntu /bin/bash
docker run -it --cpu-period="100000" --cpu-quota="100000" ubuntu
/bin/bash
```

Limitar memoria

En sistemas Linux es fácil que acabemos con una excepción `Out of Memory Exception (OOME)` si nuestros contenedores llegan a comprometer la memoría mínima necesaria para que el host sobreviva. Esto hace que Docker suba su prioridad (OOM Priority) haciendo que sea más fácil terminar con un contenedor que con el propio demonio de Docker. Lectura recomendada: Kernel: Out Of Memory Management y Kernel: Memory Resource Controller

Ram

Podemos limitar el uso de memoria RAM siendo lo minimo `6m` pudiendo llegar a varios Gigas sin problema

```
docker run -it --memory="1g" ubuntu /bin/bash
```

Swap

La memoria Swap nos permite usar espacio del disco duro como memoria en lugar de usar RAM directamente. Esto es muy útil si no tenemos mucha memoria RAM, pero el tener que acceder al disco hace que la aplicación sea mucho más lenta.

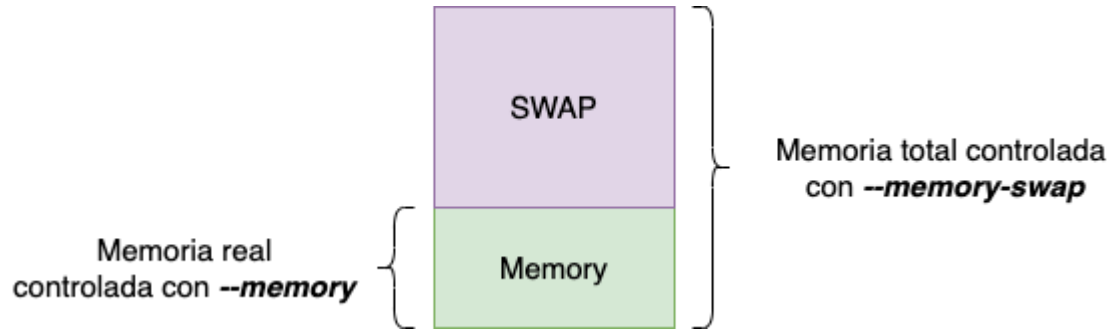


Imagen derivada de Thorsten Hans para adaptar el formato

Solo podemos configurar Swap si tenemos el argumento de `--memory` ya habilitado. Por defecto Docker igualará la memoria que especifiquemos en Swap. Haciendo así que la suma de `ram+swap` sea el doble que la ram que definimos.

Al combinar `--memory` y `--memory-swap` podemos llegara varios escenarios interesantes, veamos algunos ejemplos:

```
# Swap: 256m. Ram: 256mb. Total: 256mb + 256mb
docker run --memory="256m" ubuntu /bin/bash

# Swap: 1G. Ram: 256mb. Total: 1G + 256mb
docker run --memory="256m" --memory-swap="1g" ubuntu /bin/bash

# Swap: ilimitado. Ram: 256mb. Total: ? + 256mb
docker run --memory="256m" --memory-swap -1 ubuntu /bin/bash

# Swap: 0. Ram: 256mb. Total: 0 + 256mb
docker run --memory="256m" --memory-swap="256m" ubuntu /bin/bash
```

Otras limitaciones

En escenarios un poco más específicos contamos con la posibilidad de crear limitaciones para uso de GPUs y de la memoria del Kernel.

Cuando contamos con orquestadores ciertas tareas de limitación son más sencillas como la del propio demonio de Docker o del consumo de red especialmente útil en escenarios P2P o IPFS, pero en red local entrañan ciertas dificultades técnicas y deberemos de investigar formas de mitigarlo de una forma efectiva.

2.3 Conceptos Avanzados

2.3.1 USBs y otros dispositivos

Docker nos ofrece con el argumento `--device` la posibilidad de trabajar con comunicación serial de una forma efectiva. Para ello deberemos conectar el dispositivo (memorias, arduinos, sistemas embebidos, smartphones...) a la máquina anfitrión y después lanzar nuestro contenedor de Docker.

En este ejemplo conectamos un dispositivo en la máquina anfitrión al puerto `/dev/ttyUSB0` y permitimos el acceso desde el contenedor de Docker.

```
docker run -t -i --device=/dev/ttyUSB0 ubuntu bash
```

Evidentemente la máquina anfitrión estará expuesta a ese dispositivo y por ello debemos seguir las mismas políticas habituales al trabajar con dispositivos USB y entender que Docker no nos ofrece una protección adicional en este aspecto a nivel de seguridad y que seguimos exponiendo nuestra máquina anfitrión al dispositivo que hayamos conectado.

2.3.2 Permisos para los ficheros de configuración del demonio de Docker

Es una buena estrategia reducir los permisos de los ficheros y carpetas que hace uso el demonio de Docker. En la siguiente tabla podemos ver la configuración inicial recomendada por Docker, pero sería interesante intentar en la medida de lo posible reducir aún más esos permisos.

Recurso	Usuario:Grupo	Permisos
`/etc/default/docker`	root:root	644
`/etc/docker` (directory)	root:root	755
`/etc/sysconfig/docker`	root:root	644
`daemon.json`	root:root	644
Docker server certificate	root:root	444
Docker server certificate key	root:root	400
Docker socket	root:docker	660
docker.service	root:root	644
docker.socket	root:root	644
Registry certificate	root:root	444
TLS CA certificate	root:root	444

2.3.3 Verificar las imágenes

Por defecto el demonio de Docker nos permite descargarnos cualquier imagen que esté disponible dentro de un registro válido. Esta funcionalidad puede convertirse rápidamente en un problema si no podemos confiar en la autoría y la integridad del contenido por el canal de comunicación que usamos.

Para mitigar este tipo de escenarios podemos hacer uso de la validación de imágenes que incluye el propio demonio de Docker de la siguiente forma:

```
export DOCKER_CONTENT_TRUST=1
```

Desde este momento el demonio de Docker solo permitirá descargar aquellas imágenes que estén firmadas por sus autores a través del mecanismo de Notary que ofrece Docker.

En ocasiones tendremos otros entornos auto hosteados o similar y necesitamos definirlo de forma explícita de la siguiente forma:

```
export DOCKER_CONTENT_TRUST_SERVER=https://<URL>:<PORT>
```

2.3.4 Blindando el Daemon Socket

Es común que nos encontremos escenarios donde necesitemos exponer el Socket del demonio para que una aplicación pueda gestionar nuestros contenedores. Por ejemplo cuando hacemos uso de Portainer (capítulo 4.2), donde se nos pide que hagamos un bindeo entre el socket del host y el del contenedor de Portainer `-v /var/run/docker.sock:/var/run/docker.sock`.

Este paso hace que nuestro contenedor tome el control de los contenedores que está gestionando la máquina anfitrión, si confiamos en el software en el que estamos delegando esto no debería ser un problema, pero cuando esta relación ocurre fuera de la máquina anfitrión y por ejemplo queremos controlar el Socket de una máquina remota debemos ser proactivos y garantizar la comunicación a traves de SSH o TLS (HTTPS)

2.3.5 Privilegios y capacidades

Hace algunos años se decidió granularizar el poder de Root y dividirlo en una serie de capacidades que pueden ser activadas o desactivadas de forma independiente, esto se conoce como capabilities.

Docker nos ofrece una interfaz con tres argumentos para poder gestionarlo `--privileged`, `--cap-drop`, `--cap-add`.

En general el uso de `--privileged` es altamente desaconsejado porque concede todas las capacidades sin restricciones, son muy pocos los casos donde esto tenga una justificación real en un entorno productivo.

Carlos Polop de HackTricks presenta uno de los ejemplos más ilustrativos de lo fácil que resulta escalar privilegios rápidamente cuando no entendemos bien el alcance de `--privileged`

```
docker run --rm -it --privileged ubuntu bash
$ fdisk -l
#....
$ mkdir -p /mnt/hola
```

```
$ mount /dev/sda1 /mnt/hola
```

En realidad estamos permitiendo montar el volumen `/dev/sda1` del host como `/mnt/hola` en el contendor porque implícitamente hemos permitido esas capacidades al usar `--privileged`.

Una forma más moderna de enfocar las capacidades desde el punto de vista de la seguridad sería retirar todos los permisos de forma completa e ir gradualmente aperturando aquellos que sean estrictamente necesarios.

Un ejemplo sería este, donde quitamos todas las capacidades para finalmente permitir `kill` haciendo que un proceso que no es dueño de otro puedo terminar la ejecución:

```
docker run -d --cap-drop=all --cap-add=kill ubuntu /bin/bash
```

2.3.6 Políticas con AppArmor

AppArmor es un módulo de seguridad de Linux disponible desde la versión 2.6.36 del Kernel y que nos permite definir políticas que luego podemos agrupar en perfiles y vincularlos a la ejecución de programas.

Docker aplica por defecto una política moderada pensada de forma que no rompa la compatibilidad con la gran diversidad de imágenes disponibles. Esta política se conoce como `docker-default`.

Podemos hacer uso del argumento `--security-opt` para cargar otras políticas más restrictivas y acordes a los proyectos que tenemos entre manos.

En la propia documentación de Docker podemos encontrar un ejemplo bastante extenso sobre Nginx

2.3.7 Políticas Seccomp

Seccomp es una funcionalidad del Kernel de Linux que permite establecer políticas sobre las llamadas que un proceso puede realizar al sistema.

Docker aplica por defecto una política moderada que bloquea activamente 44 de las más de 300 directivas disponibles.

Podemos hacer uso del argumento `--security-opt` también en este caso para cargar políticas personalizadas.

En la propia documentación de Docker podemos encontrar un ejemplo bastante extenso sobre Nginx

```
docker run --rm -it --security-opt seccomp=/politica-personalizada.json
ubuntu
```

Capítulo 3: Creando y publicando imágenes seguras

Ahora que ya tenemos claro cómo consumir las imágenes de Docker, veamos ahora cómo podemos seguir buenas prácticas y recomendaciones de seguridad para que las imágenes que publiquemos sean mejores.

3.1 Creando Imágenes

Para crear una imagen siempre partiremos de un `Dockerfile` e iremos sumando instrucciones hasta lograr nuestra imagen ideal. Es importante remarcar la importancia de la inmutabilidad y la influencia que librerías de terceros pueden tener sobre la calidad de nuestra imagen final.

3.1.1 Elegir la imagen base

Como norma general siempre partiremos de una imagen base e iremos añadiendo nuestras capas hasta lograr el resultado final.

Elegir la imagen base es todo un reto, pudiendo ir desde lo más minimalista como Scratch hasta otras mucho más extendidas como Alpine. A la hora de construir una imagen siempre buscaremos que sea lo más compacta y simple posible. Todos los consejos del capítulo 2.2.1 se aplicarán en este caso.

Existe también la posibilidad de utilizar distroless como imagen base:

> *"Distroless" images contain only your application and its runtime dependencies. They do not contain package managers, shells or any other programs you would expect to find in a standard Linux distribution.*
> ***GoogleContainerTools/distroless***

Y aunque son prometedoras no han crecido en popularidad debido a que en realidad su uso tiene más sentido cuando intentamos estandarizar entornos de trabajo

Aquí un ejemplo con Node.js:

```
FROM node:18 AS build-env
COPY . /app
WORKDIR /app

RUN npm ci --omit=dev

FROM gcr.io/distroless/nodejs:18
```

```
COPY --from=build-env /app /app
WORKDIR /app
CMD ["hello.js"]
```

Preferentemente usaremos tags bien definidos para garantizar mayor inmutabilidad, aunque luego será nuestra responsabilidad ir actualizando nuestra versión a lo largo del tiempo. Esto es especialmente útil si la imagen que usamos de base no cuenta con soporte a [versionamiento semántico]

3.1.2 Privilegios

En el [capítulo 2.1.3] hablamos de las implicaciones que tiene ser root en los contenedores, por ello debemos dar un paso extra y asegurar que nuestro `Dockerfile` se preocupa de ello.

Podemos hacer uso del comando `groupadd` para añadir nuestro usuario y luego gestionar sus permisos `chmod` sobre la carpeta del proyecto dentro del contenedor, para finalmente definir a nuestro usuario como el `USER` a cargo del contenedor.

```
FROM ubuntu
RUN mkdir /app
RUN groupadd -r container && useradd -r -s /bin/false -g container
container
WORKDIR /app
COPY . /app
RUN chown -R container:container /app
USER container
CMD node index.js
```

Un ejemplo donde esto se hace es en Node.js donde las imágenes Alpine (`node:18.4.0-alpine3.16`) ya cuentan con un usuario `node` de tipo genérico. [Ver ejemplo]

3.1.3 Inmutabilidad en dependencias

Es común ver en imágenes de Docker que se use el comando `apt-get` para instalar algunas dependencias a nivel de sistema operativo. La recomendación en este caso es evitar hacer `apt-get upgrade` porque actualizará todas las dependencias a la versión disponible en ese momento, lo que claramente es mutable. Por lo que deberemos relegar esa tarea a la imagen de la que partimos `FROM`.

A la hora de instalar dependencias podemos hacer uso de binarios directamente (usando versionado) o de `apt-get install` teniendo en cuenta que si hacemos `apt-get update` este comando ya no es inmutable.

Además `apt-get update` y `apt-get install` debería de ejecutarse en el mismo comando de RUN para evitar que `apt-get update` quede cacheado y no se actualice.

```
# From https://github.com/docker-library/golang
RUN apt-get update && \
  apt-get install -y --no-install-recommends \
  git \
```

De la misma forma si tenemos que instalar dependencias para un lenguaje específico como Golang, Python, etc. deberíamos garantizar que definimos las versiones completas para intentar mantener la inmutabilidad lo más posible.

3.1.4 Integridad de dependencias

Siempre que instalemos dependencias o descarguemos recursos de la red deberemos de verificar la integridad de los datos.

En muchos casos basta con hacer un simple checksum o validación de firma para prevenir un Man In The Middle (MITM) y similares.

3.1.5 COPY o ADD

Existen dos formas de mover ficheros y carpetas a un contenedor cuando definimos la imagen de Docker: `COPY` y `ADD`

Existen varias diferencias entre ambos comandos que a nivel de seguridad nos hace inclinar la balanza hacia `COPY` ya que `ADD` tiene ciertos comportamientos implícitos que pueden suponer vulnerabilidades potenciales si no se comprenden:

- `COPY` solo permite copiar ficheros de la máquina host (llamados locales), mientras que `ADD` además permite descargar contenido remoto
- Cuando trabajamos con ficheros comprimidos `COPY` solo transporta el contenido mientras que `ADD` realiza el paso extra de descomprimir el contenido.
- Cuando usamos imagenes multi-stage (capítulo 3.1.6) no podremos pasarnos contenido entre las imágenes con `ADD`

En el caso de querer descargar recursos remotos es preferible hacer uso de `RUN` que de `ADD` ya que esto es una declaración explícita.

3.1.6 Multi-stage

Otra forma de alcanzar el minimalismo es hacer uso de imágenes multi-stage donde la imagen resultante es infinitamente más óptima:

```
FROM golang:1.16 AS builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    ./
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app ./
CMD ["./app"]
```

En este ejemplo de la documentación oficial de Docker vemos como primero usamos la imagen de `golang:1.16` para compilar la aplicación de Go y que posteriormente usamos `alpine:latest` para ejecutar el binario que se generó y recuperamos de la imagen anterior a la que llamamos `builder`.

Sería imposible compilar la aplicación de Go directamente con `alpine:latest` ya que no cuenta con el compilador.

3.1.7 Metadata

Los metadatos son importantes en nuestras imágenes porque nos aportan información adicional que en ocasiones es muy difícil de obtener si no se incluye de forma explícita. A su vez, estos metadatos pueden ser consumidos indistintamente por máquinas y personas, haciendo que esta información sea versátil y ayude en la toma de decisiones, especialmente en escenarios de integración continua como veremos en el capítulo 4.12.

Para incluir metadatos solo necesitamos hacer uso de la palabra reservada `LABEL` en nuestro Dockerfile, funciona igual que el resto de instrucciones y sigue una estructura de diccionario:

```
LABEL maintainer="ulises@linux.com"
```

Tenemos la posibilidad de incluir mucha información relevante relacionada con la propia build. Una forma de estructurar los metadatos es hacer uso de Label Schema Convention ya que nos ofrece un marco de trabajo muy determinado.

```
LABEL maintainer="ulises@linux.com"
LABEL org.label-schema.schema-version="0.1.0"
LABEL org.label-schema.description="Dockerizando un hello world!"
LABEL org.label-schema.url="https://ulisesgascon.com"
#...
```

3.1.8 Buildkit

Desde la versión 18.09 de Docker contamos con Buildkit que introduce una serie de mejoras que hacen la experiencia de construcción de las imágenes mucho más rápida, pero también incluye algunas mejoras en la seguridad que podemos usar en nuestros proyectos.

Habilitar

Para poder hacer uso del Buildkit tendremos que habilitarlo específicamente usando la variable de entorno `DOCKER_BUILDKIT` de la siguiente forma:

```
DOCKER_BUILDKIT=1 docker build .
```

Si queremos hacerlo de forma permanente podemos modificar `/etc/docker/daemon.json`, sin olvidarnos de reiniciar el demonio

```
{ "features": { "buildkit": true } }
```

Compartiendo secretos

Buildkit introduce la posibilidad de usar el argumento `--secret` a la hora de hacer la build junto con el uso de volúmenes de `type=secret` conseguimos así gestionar información sensible de una forma más correcta, evitando que los secretos sean guardados en la imagen final.

1. Guardamos el dato sensible en un fichero:

```
echo 'this-is-sensitive-data' > mysecret.txt
```

2. Referenciamos el volumen en el Dockerfile

```
# syntax=docker/dockerfile:1.2
FROM alpine
RUN --mount=type=secret,id=mysecret cat /run/secrets/mysecret
RUN --mount=type=secret,id=secret_token,dst=/foobar cat /foobar
#...
```

3. Pasamos el secreto en tiempo de build y referenciamos el id y nuestro fichero

```
DOCKER_BUILDKIT=1 docker build --secret id=mysecret,src=mysecret.txt .
```

Usando SSH de forma segura en la build

Es bastante común tener que acceder a recursos externos vía SSH para descargarnos archivos. En este ejemplo veremos como clonarnos el repositorio privado y ficticio `ulisesgascon/secret-project` en tiempo de build usando SSH para conectarnos, evitando exponer nuestra llave SSH.

1. Referenciamos el volumen en el Dockerfile usando el `type=ssh`

```
# syntax=docker/dockerfile:1
FROM alpine

# Instalamos el cliente ssh y git
RUN apk add --no-cache openssh-client git

# Descargamos y guardamos la llave publica de github.com
RUN mkdir -p -m 0700 ~/.ssh && ssh-keyscan github.com >>
~/.ssh/known_hosts

# Clonamos el repositorio
RUN --mount=type=ssh git clone
git@github.com:ulisesgascon/secret-project.git secret-project

#...
```

2. Pasamos la referencia de la llave SSH usando `--ssh` y usamos como argumento el perfil ssh que queremos usar (`default` en este caso) en tiempo de build.

```
DOCKER_BUILDKIT=1 docker build --ssh default .
```

3.1.9 Health checks

Como vimos en el capítulo 2.2.5 sobre la limitación de recursos Docker tienen la capacidad de detectar si está teniendo algún problema y el contenedor ha dejado de funcionar gracias a los código de salida que genera el proceso donde se ejecuta el contenedor.

Aunque esta referencia en ocasiones es más que suficiente no lo es en escenarios más complejos donde contamos con procesos que son más resilientes al fallo absoluto como un servidor web pero que están presentando algún tipo de anomalía que le impide su correcto funcionamiento.

En el caso de servidores web, podemos encontrar en muchas ocasiones que se incluye un endpoint que nos confirma si el servidor está funcionando correctamente o teniendo problemas, gracias a los códigos de error http.

```
docker run --health-interval=4s \
      --health-retries=8 \
      --health-timeout=13s \
      --health-cmd='curl -sS http://127.0.0.1:3001/health || exit 1' \
      myorg/mywebserver
```

Por supuesto este tipo de comprobaciones pueden añadirse al Dockerfile ofreciendo una forma adicional para saber cuando nuestra aplicación está dejando de funcionar de forma que el Demonio de Docker puede actuar en consecuencia

```
#...
HEALTHCHECK --interval=4s --timeout=13s --retries=8 CMD curl -sS
http://127.0.0.1:3001/health || exit 1
#...
```

3.2 Publicando

De nada sirve crear una imagen de Docker si no la distribuimos por el mundo. Así que sigamos el proverbio friki de Obijuan y demos el paso

> *"Más vale proyecto publicado con licencia libre, que ciento en el cajón"*
> ***Juan Gonzalez (Obijuan)***

3.2.1 Tags

Nuestras imágenes deben ir correctamente versionadas para permitir que otras personas o nosotros mismos podamos consumirlas con ciertas garantías de inmutabilidad.

En el capítulo 2.2.1 remarcaba lo importante que es hacer un uso correcto de los tags como uno de los factores decisivos a la hora de usar una imagen. Como individuos que publicamos imágenes tenemos la responsabilidad de usar tags de una forma coherente ya sea con el uso del versionamiento semántico u otra metodología.

Además tendremos que tomar una decisión sobre el polémico / mal entendido uso del tag `latest`, ya que en ocasiones podemos tener resultados inesperados que podrían generar frustración en los usuarios sobre todo a la hora de hacer un buen triage en el soporte. En general, cuanto más empujemos como colectivo hacía la inmutabilidad, más sencillo será la gestión de incidencias posteriormente.

3.2.2 Firmando imágenes

Como vimos en el capítulo 2.2.1 es importante que usemos imágenes que estén firmadas. El proceso de firmado de las imágenes es complejo y ha requerido mucho trabajo por parte del equipo de Docker y la comunidad.

A efectos prácticos tenemos dos vertientes que cubrir cuando queremos trabajar con imágenes firmadas. Por un lado queremos consumir imágenes que estén firmadas (dentro de lo posible) y por otro queremos firmar nuestras imágenes. Combinando ambos reducimos enormemente los riesgos de ataque man in the middle y similares.

Consumir imágenes firmadas

Como usuarios podemos hacer uso de la variable de entorno `DOCKER_CONTENT_TRUST=1` para forzar que todas las imágenes que usemos estén firmadas o no serán descargadas del registro.

Por defecto esta variable estaría desactivada `DOCKER_CONTENT_TRUST=0` por lo que sería nuestra responsabilidad asegurarnos que dejamos el valor habilitado, especialmente en las máquinas que destinamos a producción.

Publicar imágenes firmadas

Como cualquier sistema basado en encriptación asimétrica uno de los primeros pasos será crearnos unas llaves que nos permita firmar.

Lo primero será identificarnos contra el servidor de Docker Hub

```
docker login
```

En este caso, crearé una llave con nombre `personal-laptop-1` para ayudarme a identificarla cuanto tenga más llaves.

```
docker trust key generate personal-laptop-1
```

Ahora indicaremos a Docker que queremos que esa llave `personal-laptop-1` pueda firmar la imagen `ulisesgascon/signed-hello-world`

```
docker trust signer add --key dev1.pub personal-laptop-1
ulisesgascon/signed-hello-world
```

Desde este momento ya podremos firmar con esa llave ejecutando el siguiente comando (previamente debemos hacer la build con el tag correspondiente)

```
docker trust sign ulisesgascon/signed-hello-world:latest
```

Podemos inspeccionar la imagen en detalle de la siguiente forma:

```
docker trust inspect --pretty ulisesgascon/signed-hello-world:latest
```

Ya estaríamos listos para publicar la imagen como hacemos habitualmente. Podemos crear múltiples llaves, incluso podemos crear llaves para otras personas que colaboren con nosotros o máquinas.

Recuerda que siempre que uses llaves para firmar deberás guardar una copia de seguridad de las llaves.

3.2.3 Registros

Una vez tenemos nuestra imagen construida debemos hacerla accesible en alguna entidad que actúe como registro, donde podamos publicar, descargar y actualizar esas imágenes ya sea abierto al público o de carácter privado para una organización o grupo de recursos específico.

Con el auge de la nube pública, es común encontrar multitud de alternativas a Docker Hub. Incluso podemos auto-hostearnos nuestro propio registro con Registry

Independientemente de la solución que utilicemos, deberemos garantizar antes de publicar que el acceso a la imagen es el esperado y no debemos hacer públicas imágenes que deben ser privadas.

A su vez es importante seguir las recomendaciones generales en el uso de servicios de terceros como usar 2FA o rotar las contraseñas de forma frecuente, etc.

Esto es especialmente relevante si estás manteniendo imágenes populares que usan muchas personas, ya que en ocasiones las personas que mantienen esas librerías son objetivo de ataques como individuos por parte de agentes maliciosos que intentan publicar imágenes o librerías envenenadas en su nombre como ya pasó en 2018 cuando uno de los paquetes más populares de NPM (ecosistema de Node.js) fue corrompido

Una forma de mitigar este último escenario, es hacer uso de la firma de imágenes como vimos en el capítulo 3.2.2 y en el caso de utilizar tokens estos deberán ser correctamente custodiados y rotados de forma frecuente.

3.2.4 Dockerignore

Cuando usamos el comando `COPY` o `ADD` corremos el riesgo de copiar de manera recursiva ficheros sensibles. Por eso es importante mantener un fichero `.dockerignore` actualizado.

El comportamiento del fichero `.dockerignore` es muy similar al de `.gitignore`, por lo que resulta sencillo su uso.

Por ejemplo, para un proyecto de Node podríamos incluir logs, `node_modules`, referencias a ficheros `.env`, etc..

```
# Node
## Logs

logs
*.log
npm-debug.log*
yarn-debug.log*
yarn-error.log*

## Dependency directories
node_modules/

## Secrets
.env
.env.*
```

Capítulo 4: Herramientas

En este capítulo encontrarás un conjunto de herramientas recomendadas que hemos mencionado a lo largo del libro o que consideramos de especial relevancia.

Recordemos que en el ecosistema de Docker existen infinidad de herramientas, una forma de buscar nuevas herramientas o alternativas es en veggiemonk/awesome-docker

4.1 Docker compose

A medida que introducimos Docker en nuestro entorno de desarrollo, nos damos cuenta que rápidamente estamos en un escenario donde tenemos que orquestar múltiples contenedores a la vez por ejemplo: base de datos, backend, frontend, cache...

Para agilizarnos el proceso podemos hacer uso de Docker Compose una herramienta muy elegante que nos permite pasar de comandos como:

```
docker run -p 80:80 -v /var/run/docker.sock:/tmp/docker.sock:ro
--restart always --log-opt max-size=1g nginx
```

a ficheros yaml mucho más sencillos y legibles como:

```
version: '3.3'
services:
    nginx:
        ports:
            - '80:80'
        volumes:
            - '/var/run/docker.sock:/tmp/docker.sock:ro'
        restart: always
        logging:
            options:
                max-size: 1g
        image: nginx
```

Además podemos tener múltiples servicios corriendo y conectados entre sí con las políticas de red que nosotros queramos. Por ejemplo un Wordpress rápido y sencillo para hacer pruebas en local con MariaDB

```
version: '3.3'
services:
  db:
    # We use a mariadb image which supports both amd64 & arm64
architecture
```

```
    image: mariadb:10.6.4-focal
    # If you really want to use MySQL, uncomment the following line
    #image: mysql:8.0.27
    command: '--default-authentication-plugin=mysql_native_password'
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress
    expose:
      - 3306
      - 33060
  wordpress:
    image: wordpress:latest
    ports:
      - 80:80
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=wordpress
      - WORDPRESS_DB_NAME=wordpress
volumes:
  db_data:
```

También podemos incluir variables de entorno, que además tiene sus propios comandos y sintaxis que nos ayudan a gestionar el ciclo de vida de nuestros contenedores de una forma fácil e intuitiva.

Una forma fácil de empezar es simplemente migrando comandos de Docker que ya usamos en el día a día con la herramienta online Composerize

4.2 Portainer

Portainer es una de las herramientas esenciales para la orquestación de contenedores en entornos locales, principalmente aunque también productivos. Tiene una curva de aprendizaje muy suave en comparación con Kubernetes y cuenta con una interfaz web completa donde se pueden visualizar nuestros contenedores, volúmenes, redes, etc...

Es una forma sencilla y rápida sobre todo para acelerar la adopción de contenedores y nube pública en procesos de transformación digital o para empezar un homelab.

Evidentemente si lo usamos en entornos productivos deberemos de securizarlo a no ser que utilicemos la `Business Edition`

Portainer `Community Edition` ya viene dockerizado así que los haremos funcionar como un contenedor más que además de pasarle volúmenes y comunicar puertos le pasaremos el socket de Docker para que pueda orquestar los demás contenedores.

```
docker run -d -p 8000:8000 -p 9443:9443 --name portainer
--restart=always -v /var/run/docker.sock:/var/run/docker.sock -v
portainer_data:/data portainer/portainer-ce:latest
```

4.3 Dive

Dive nos permite adentrarnos de una manera cómoda en las diversas capas que componen una imagen de Docker siendo especialmente útil para buscar secretos o cambios entre capas.

Incluso podemos utilizarlo en sistemas de integración continua (sección 4.12) para validar la eficiencia de nuestras imágenes

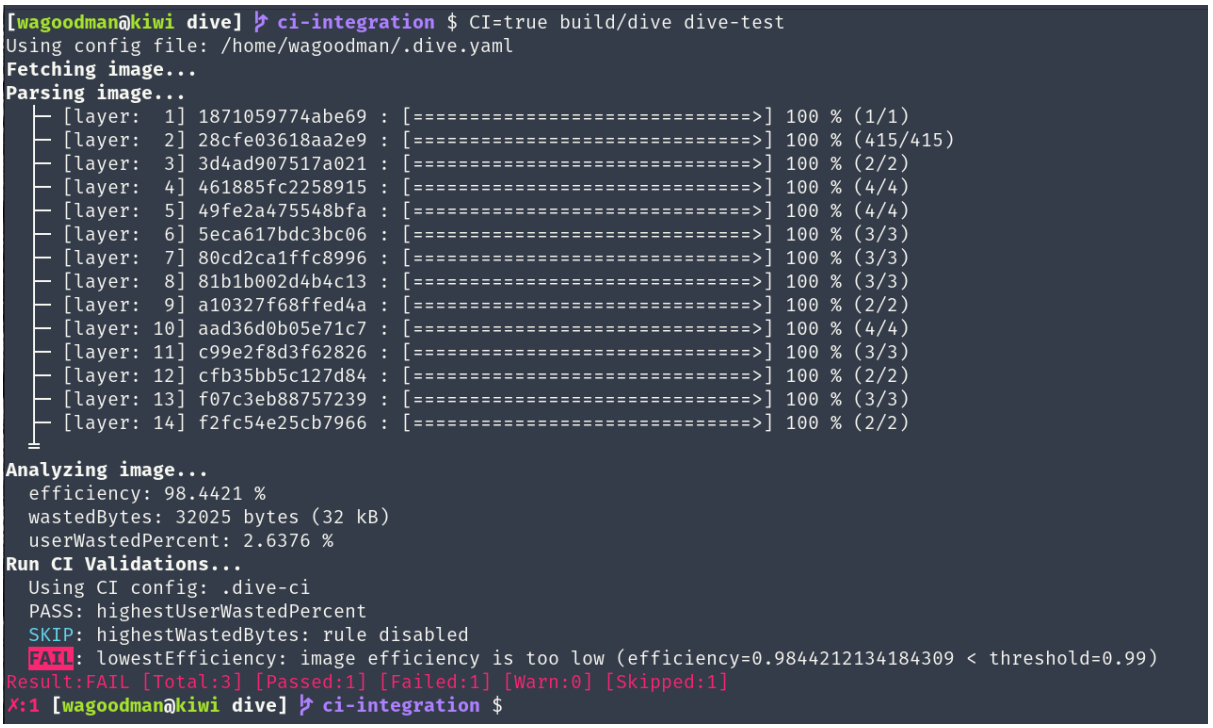


Imagen cortesía de Dive

4.4 Open Policy Agent (OPA)

Para escenarios donde buscamos el poder crear un entorno de desarrollo muy dogmático podemos hacer uso de políticas. Gracias a Open Policy Agent (OPA) podemos definir políticas uniformes en diferentes plataformas (aplicaciones, agentes, k8s, docker..) y asegurarnos que las políticas se cumplen haciendo uso de CI u otras herramientas adicionales.

4.5 Snyk

Snyk es una plataforma completa pensada para personas que se dedican al desarrollo de software y que ofrece varias soluciones que podemos utilizar para analizar, escanear y monitorizar nuestros proyectos a varios niveles (dependencias, análisis de código SAST, contenedores, infraestructura como código).

Nos permite escanear imágenes de Docker en busca de vulnerabilidades:

```
snyk container test hello-world
```

O incluso suscribirnos (monitorizando) una o varias imágenes que usemos, para así ser alertados cuando se publique una nueva vulnerabilidad en nuestras dependencias

```
snyk container monitor hello-word
```

Al igual que el propio Docker con su comando `scan` que hace uso de Snyk para escanear nuestras imágenes (ofreciendo 10 escaneos sin autenticación).

```
docker scan hello-world
```

También es útil para escanear las dependencias de nuestros propios proyectos a nivel de librerías, ya que Snyk cuenta con una base de datos abierta de vulnerabilidades.

Una de las formas más recomendadas de hacer uso de Snyk es a través de sus plugins para integración continua ya que de manera proactiva analizará el repositorio cuando detecte cambios y nos alertará cuando nuevas vulnerabilidades se publiquen incluso proponiendo cambios en el código.

4.6 Hadolint

Hadolint nos ofrece un linter completo con un conjunto de reglas sólidas y bien definidas. Lo que reduce la necesidad de recordar todas las buenas prácticas sobre todo cuando no usamos Docker a diario o añadimos nuevas personas al equipo.

Las reglas pueden modificarse fácilmente para adaptarse a las convenciones del equipo.

Y podemos consumirlo como un contenedor al que le pasamos el `Dockerfile` que queramos revisar:

```
docker run --rm -i hadolint/hadolint < Dockerfile
```

4.7 Dockle

Dockle es un linter similar en objetivos a Hadolint (sección 4.6) pero que incluye funcionalidades extra como la auditoría de seguridad, ya que se basa en imágenes de contenedores y no en `Dockerfile`. A la hora de instalarlo y usarlo tenemos más pasos.

Este sería la forma de usarlo como un contenedor:

```
$ export DOCKLE_LATEST=$(
 curl --silent
"https://api.github.com/repos/goodwithtech/dockle/releases/latest" | \
 grep '"tag_name":' | \
 sed -E 's/.*"v([^"]+)".*/\1/' \
)
$ docker run --rm goodwithtech/dockle:v${DOCKLE_LATEST}
[YOUR_IMAGE_NAME]
```

Si quisiéramos utilizar imágenes que solo existen en nuestra máquina deberíamos conectar el Socket de Docker.

```
$ docker run \
    -v /var/run/docker.sock:/var/run/docker.sock \
    --rm goodwithtech/dockle:v${DOCKLE_LATEST} \
    [YOUR_IMAGE_NAME]
```

4.8 Docker-slim

Docker-slim nos permite reducir el tamaño de nuestra imagen de Docker enormemente ya que utiliza un algoritmo muy efectivo para lograrlo. Además de reducir considerablemente el tamaño de la imagen final (hasta 30 veces) nos ofrece un conjunto de herramientas para entender en detalle este proceso y permitirnos influir en él.

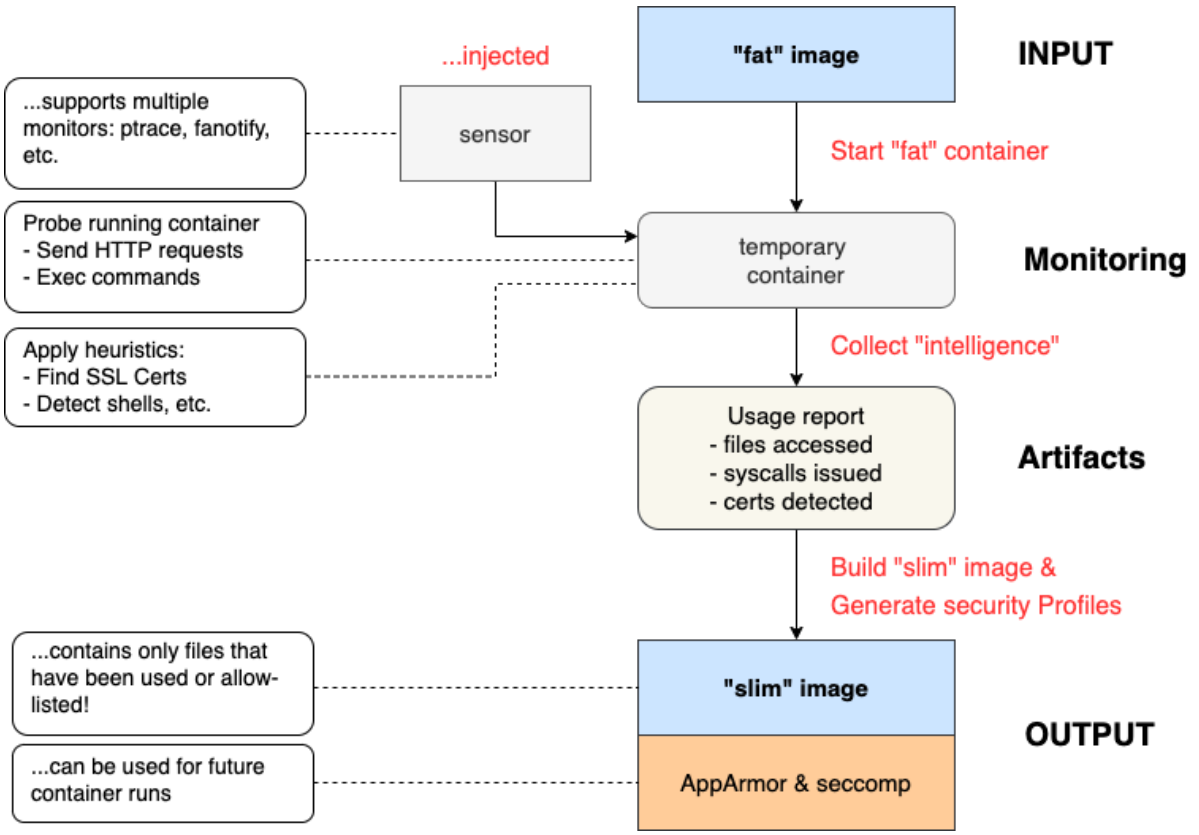


Imagen derivada de Docker-slim para adaptar el formato

Además de la reducción de tamaño nos ofrece una capa de seguridad adicional gestionando por nosotros AppArmor y Seccomp

Una vez instalado en nuestra máquina podemos hacer una build de la imagen:

```
docker-slim build my-company/my-app
```

4.9 Dockprom

Dockprom es una solución fantástica cuando necesitamos poder monitorizar nuestros contenedores, especialmente en el entorno local, ya que con poco esfuerzo podemos tener una solución de monitorización que incluye Prometheus, Grafana, cAdvisor, NodeExporter y alertas con AlertManager.

Se instala de forma simple (clonamos y desplegamos el Docker-compose) además podemos cambiar los usuarios y passwords por defecto fácilmente.

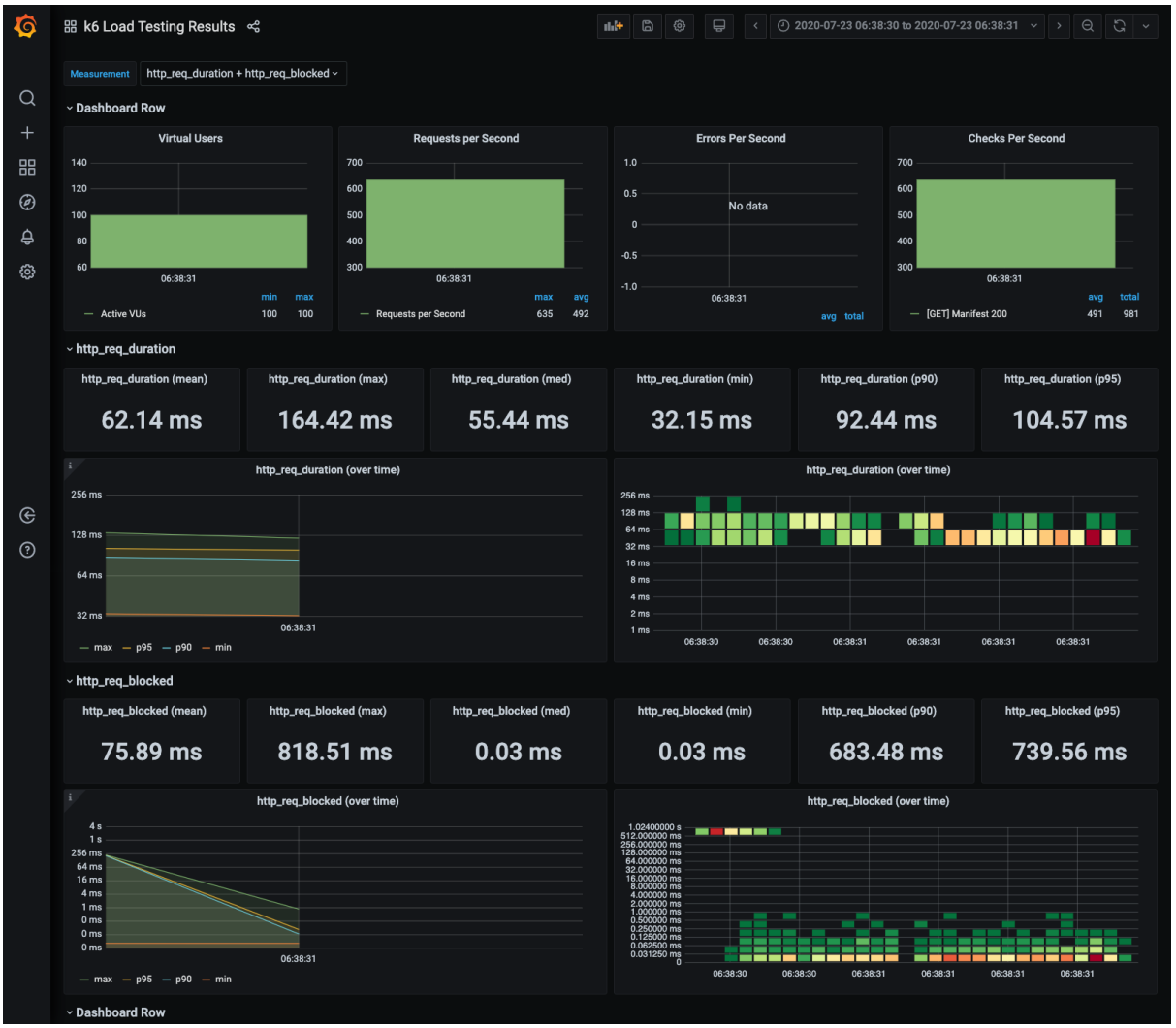
```
git clone https://github.com/stefanprodan/dockprom
cd dockprom
ADMIN_USER=admin ADMIN_PASSWORD=admin ADMIN_PASSWORD_HASH=JD...ZP
docker-compose up -d
```

Es bastante común que encontremos forks de este proyecto con customizaciones específicas como Promstack que resolvería el tema de logging con Loki.

4.10 K6

K6 es una de las mejores herramientas disponibles para hacer pruebas de carga usando Javascript como lenguaje para hacer nuestros scripts. Tiene mucho soporte de la comunidad haciendo que las pruebas de carga puedan hacerse más allá de servidores http.

Una forma de hacer uso de ello es desde el proyecto ulisesgascon/PoC-Load-test que ya incluye una versión dockerizada junto a Grafana para tener dashboard más detallados de las pruebas de carga.



4.11 Docker Bench for Security

docker-bench-security es un script que aglutina decenas de chequeos para verificar que seguimos las recomendaciones y buenas prácticas para el uso de docker en entornos productivos. El conjunto de chequeos se realizan en formas de tests siguiendo el CIS Docker Benchmark v1.4.0.

Puede ser instalado en la máquina host o ejecutado como imagen de docker

4.12 Integración continua

Es muy común que usemos un sistema de control de versiones para mantener los cambios de nuestros proyectos y sus releases de una forma estable y predecible.

Las imágenes de Docker no son una excepción a esta regla, lo que a su vez nos brinda la oportunidad de hacer uso de los diversos sistemas de integración continua para automatizar muchas de nuestras tareas como por ejemplo la publicación de imágenes en registros públicos o privados, revisión automática de las dependencias, etc...

Las posibilidades son infinitas, a su vez nos permite una gestión más óptima de los tokens, api keys y secretos que manejamos a nivel de la organización, evitando así la necesidad de compartir tokens con personas individuales, facilitando así la rotación de los mismos.

Veamos unos ejemplos

En esta ocasión usaremos Github Actions, pero fácilmente podría portarse a otros sistemas de CI sin mucho esfuerzo.

Imaginemos que tenemos un simple "Hola mundo" en un fichero `./src/server.js` y su correspondiente `package.json` con las dependencias y típicos scripts de Npm (start, test, lint...)

```
const express = require('express')

const app = express()

const PORT = 8080;
const HOST = '0.0.0.0';

app.get('/', (req, res) => {
    res.send('Hola mundo!')
})

app.listen(PORT, HOST () => {
    console.log('Listening on port ${PORT}}...')
```

```
})
```

Luego hacemos un Dockerfile siguiendo las buenas prácticas (multi-stage, gestión de privilegios, crear una imagen final liviana, metadatos...)

```
# Fase 1: dependencies

FROM node:14.15.5 as dependencies
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm i --only=production

# Fase 2: Imagen final
FROM node:14.15.5-alpine

# Distribution details.
ARG BUILD_DATE
ARG VCS_REF
ARG BUILD_VERSION

# Labels.
LABEL org.label-schema.schema-version="1.0"
LABEL org.label-schema.build-date=$BUILD_DATE
LABEL org.label-schema.name="ulisesgascon/hello-world-sample"
LABEL org.label-schema.description="Simple Hello World Sample in Nodejs
with Express"
LABEL
org.label-schema.url="https://github.com/ulisesgascon/hello-world-sample
"
LABEL
org.label-schema.vcs-url="https://github.com/ulisesgascon/hello-world-sa
mple"
LABEL org.label-schema.vcs-ref=$VCS_REF
LABEL org.label-schema.vendor="Ulises Gascon"
LABEL org.label-schema.version=$BUILD_VERSION
LABEL org.label-schema.docker.cmd="docker run -p 8080:8080 -d
ulisesgascon/hello-world-sample"

WORKDIR /app
RUN chown -R node:node /app
USER node
COPY --from=dependencies /usr/src/app/node_modules ./node_modules
COPY package*.json ./
COPY /src ./src
EXPOSE 8080
```

```
CMD [ "npm", "start" ]
```

Revisar los cambios

Un flujo bastante común se produciría cuando se creará una nueva pull request. Disparando un workflow que valide específicamente que nuestro `Dockerfile` está pasando un linter como Hadolint y que es capaz de hacer una build sin romperse (incluyendo metadatos).

```
name: Check Dockerfile
on: [pull_request]

jobs:
  check-dockerfile:
    runs-on: ubuntu-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v2

      - name: Get current time
        uses: josStorer/get-current-time@v2.0.1
        id: current-time

      - name: Linting with hadolint
        uses: reviewdog/action-hadolint@v1
        with:
          reporter: github-pr-review

      - name: Build the Docker image
        env:
          RELEASE_VERSION: ${{ steps.vars.outputs.tag }}
          BUILD_DATE: ${{ steps.current-time.outputs.time }}
          RELEASE_VERSION: "PR${{github.event.number}}"
        run: |
          docker build .
          --build-arg BUILD_DATE=${{ env.BUILD_DATE  }}
          --build-arg VCS_REF=${{ github.sha }}
          --build-arg BUILD_VERSION=${{ env.RELEASE_VERSION }}
          --tag ulisesgascon/hello-world-sample:$RELEASE_VERSION
          --tag ulisesgascon/hello-world-sample:latest
```

Publicar en un registro público

Con unas pocas modificaciones sobre el código anterior podemos hacer la publicación de la imagen de manera pública en Docker Hub cuando creamos una release en el proyecto.

```yaml
name: Publish Docker Image
on: [pull_request]

jobs:
  check-dockerfile:
    runs-on: ubuntu-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v2

      - name: Set RELEASE_VERSION
        run: echo "RELEASE_VERSION=${GITHUB_REF#refs/*/}" >> $GITHUB_ENV

      - name: Get current time
        uses: josStorer/get-current-time@v2.0.1
        id: current-time

      - name: Linting with hadolint
        uses: reviewdog/action-hadolint@v1
        with:
          reporter: github-pr-review

      - name: Build the Docker image
        env:
          RELEASE_VERSION: ${{ steps.vars.outputs.tag }}
          BUILD_DATE: "${{ steps.current-time.outputs.time }}"
        run: |
          docker build .
          --build-arg BUILD_DATE=${{ env.BUILD_DATE  }}
          --build-arg VCS_REF=${{ github.sha }}
          --build-arg BUILD_VERSION=${{ env.RELEASE_VERSION }}
          --tag ulisesgascon/hello-world-sample:$RELEASE_VERSION
          --tag ulisesgascon/hello-world-sample:latest

      - name: Docker Hub Login
        env:
          DOCKER_USER: ${{secrets.DOCKER_USER}}
          DOCKER_PASSWORD: ${{secrets.DOCKER_PASSWORD}}
        run: |
          docker login -u $DOCKER_USER -p $DOCKER_PASSWORD

      - name: Docker Hub Publish
        run: docker push --all-tags ulisesgascon/hello-world-sample
```

Nota: debemos incluir los secretos que solicitamos en la configuración del repositorio de Github (`DOCKER_USER` y `DOCKER_PASSWORD` para Docker Hub)

Conclusiones

En primer lugar, felicidades por haber logrado llegar hasta el final del libro. Espero que hayas disfrutado del aprendizaje y te sirva en tu día a día. Recuerda que puedes seguir formándote en temas relacionados con el mundo de los contendores. Mi recomendación para el siguiente paso sería intentar entender en profundidad y manejar un orquestador de contenedores como Kubernetes ya que te ayudará a poner en práctica muchos de estos conocimientos.

Como hemos visto a lo largo del libro, los contendores como tecnología vienen para quedarse sea cual sea su forma. Aún quedan muchos años por delante para seguir disfrutando de Docker hasta que en el sector se produzca el siguiente cambio de paradigma.

Si aún no has podido introducir los contenedores en tu entorno, igual puedes empezar por ir paso a paso metiendo Docker en el entorno local y luego en los sistemas de integración continua.

Tomar decisiones

Ahora que sabes más sobre la seguridad en Docker, puedes tomar decisiones más conscientes y entender mucho mejor cómo te impactan los CVEs que se publican sobre tu stack y que seguramente te generan incertidumbre.

Entender cómo funciona Docker, sus mecanismos internos y el ciclo de vida de los contenedores te permitirá asumir más riesgos calculados en el futuro.

Hacer buenas imágenes que sean inmutables (en la medida de lo posible) te ayudará a evitar muchos conflictos y errores entre entornos.

Implementar políticas de Zero Trust desconfiando de tu red, te ayudará a mitigar que un ataque se propague del contenedor al host o del contenedor a otros contenedores ajenos. Del mismo modo, partir de lanzar contenedores con los mínimos permisos y accesos posibles e ir abriendo puertas poco a poco, te ayudará a prevenir ataques que terminan con una escalada de permisos exitosa.

Aprovecha el potencial de las imágenes multi-stage para hacer imágenes más compactas y sencillas que vayan reduciendo tu superficie de ataque.

Vivimos en la era del SASS y de la automatización, usalo para que te empuje en la buena dirección.

Ve despacio

Después de leer el libro, seguro que estás deseando cambiar muchas cosas en tus contenedores e imágenes. Pero antes de empezar con ello te recomendaría encarecidamente que te tomes tu tiempo para determinar un plan sólido que puedas implementar y mantener en el tiempo.

Un plan en el que priorices las cosas que son más críticas en tu escenario. Por ejemplo, es más importante actualizar los contenedores que están usando `--privileged` o un modelo de permisos sobredimensionado que añadir los metadatos en el Dockerfile.

Por otro lado hay que tener en cuenta que no todos los cambios son fáciles de implementar. No es lo mismo hacer ajustes para que nuestros contenedores tengan solamente permisos de lectura sobre ciertos volúmenes, que asegurarnos que cada imagen que utilizamos está firmada.

Entiende tus vulnerabilidades adquiridas

Actualmente podemos afirmar que prácticamente toda la industria depende de otros para sacar adelantes sus aplicaciones, librerías y productos. Es normal que todos hagamos un uso extensivo −y en ocasiones abusivo− del software libre, pero debemos asumir que tenemos la responsabilidad de preocuparnos de nuestras dependencias.

Preocuparnos de nuestras dependencias tiene muchísimas vertientes; desde actualizar periódicamente e ir parcheando nuestro software, hasta contribuir de vuelta con nuestros propios parches y mejoras o preocupándonos de una forma proactiva de la sostenibilidad económica las personas que están haciendo ese trabajo por nosotros. Pensemos que al final del día, todos dependemos tecnológicamente de otros, hasta tal punto que el chiste se hace solo:

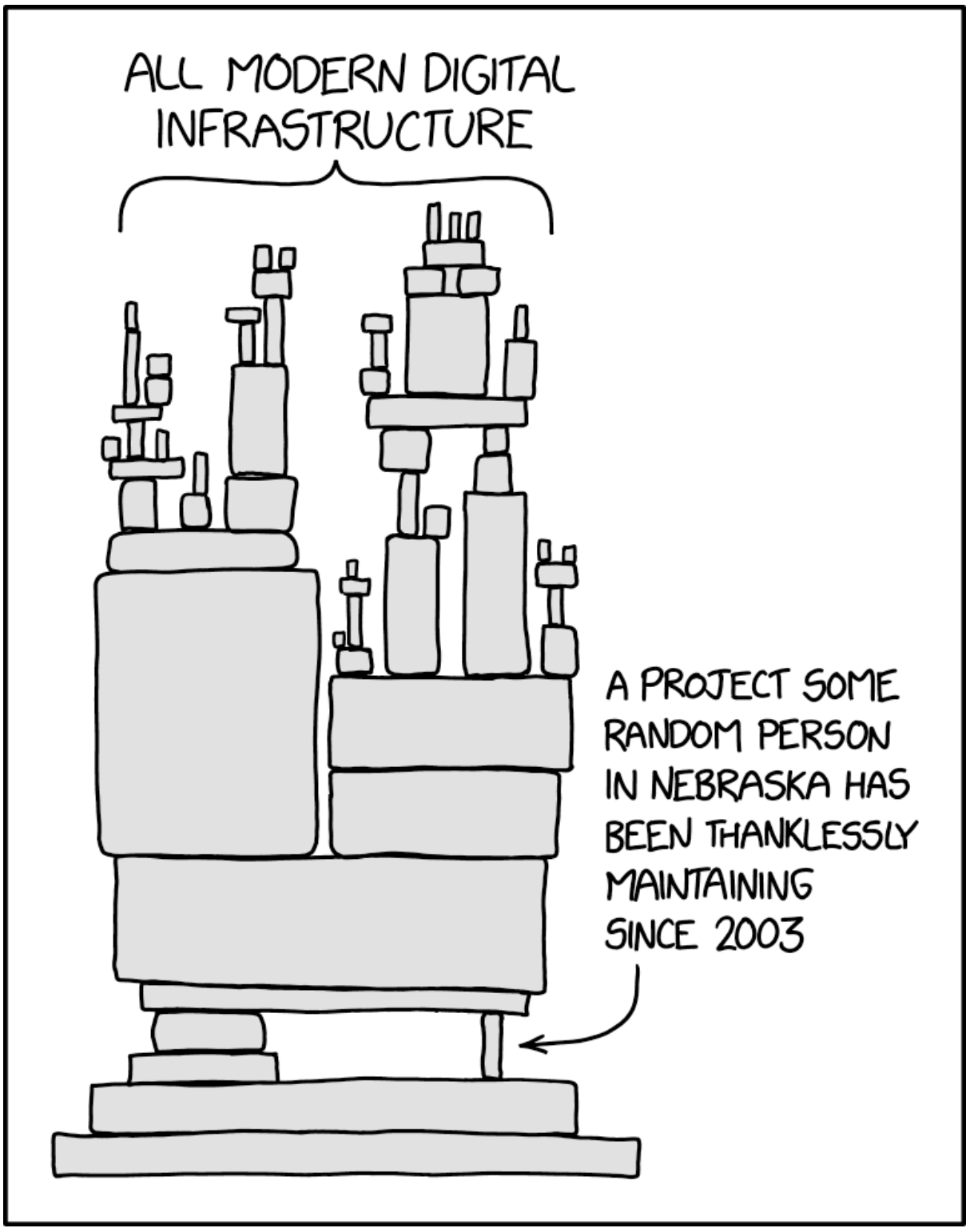


Imagen cortesía de XKCD

La seguridad es cultura

Y por encima de todo, creo que debemos empezar a asumir que la seguridad debe formar parte activa de la cultura de la empresa, y que todos, en nuestra medida somos

responsables de contribuir a que la seguridad mejore y no se degrade, exactamente igual que sucede con la práctica de hacer "buen código".

Recursos

Libros recomendados

- Docker: SecDevOps
- Mastering Docker
- Docker Cookbook
- Docker Security

Documentación Oficial

- Docker | Docker security non-events
- Docker | Docker security
- Docker | The Impacts of an Insecure Software Supply Chain
- Docker | Keep containers alive during daemon downtime
- Docker | Antivirus software and Docker
- Docker | Prune unused Docker objects
- Docker | Run the Docker daemon as a non-root user (Rootless mode)
- Docker | Isolate containers with a user namespace
- Docker | Runtime metrics
- Docker | Configure logging drivers
- Docker | Understanding Docker Hub Rate Limiting
- Docker | Use volumes
- Docker | Manage data in Docker
- Docker | docker inspect
- Docker | Legacy container links
- Docker | Start containers automatically
- Docker | docker update
- Docker | AppArmor security profiles for Docker
- Docker | Environment variables in Compose
- Docker | Back up your keys

Documentación

- OWASP Docker Threat Modeling
- OWASP Docker Top 10
- OWASP Top 10 - 2021
- Node.js Docker Cheat Sheet
- HackTricks
- Docker Swarm | Manage sensitive data with Docker secrets
- Kubernetes | Secrets
- Kubernetes | Set the security context for a Pod
- Kubernetes | Horizontal Pod Autoscaling

Información útil

- DOCKER DE NOVATO a PRO! (CURSO COMPLETO EN ESPAÑOL) de Pelado Nerd
- 8 Surprising Facts About Real Docker Adoption
- State of Open Source Security 2022
- Kubernetes is Moving on From Dockershim: Commitments and Next Steps
- A Brief History of Containers: From the 1970s Till Now
- Dirty COW - (CVE-2016-5195) - Docker Container Escape
- How to add Reverseshell to host from the privileged container
- Use SQL Injection to Run OS Commands & Get a Shell
- Understanding and Securing Linux Namespaces
- Top ten most popular docker images each contain at least 30 vulnerabilities
- Threat Alert: Supply Chain Attacks Using Container Images
- Practical Design Patterns in Docker Networking
- How to contact Google SRE: Dropping a shell in cloud SQL
- Metadata service MITM allows root privilege escalation (EKS / GKE)
- Bash command line exit codes demystified
- Linux Performance: Why You Should Almost Always Add Swap Space
- Docker Container Memory Limits Explained
- The misunderstood Docker tag: latest
- What's Wrong With The Docker :latest Tag?
- A Docker image walks into a Notary - Diogo Mónica
- Man-in-the-Middle (MITM) Attack
- Postmortem for Malicious Packages Published on July 12th, 2018
- A positive security culture
- Path Traversal
- Your Secret's Safe with Me. Securing Container Secrets with Vault
- Secure Your Containers with this One Weird Trick

Herramientas

- PodMan
- Versionamiento Semántico
- Composerize
- Portainer
- Dive
- Open Policy Agent (OPA)
- Snyk
- Hadolint
- Dockle
- Docker-slim
- Dockprom
- Prometheus
- Grafana
- cAdvisor
- Node_exporter
- Alertmanager

- Promstack
- Loki
- K6
- UlisesGascon/PoC-Load-test
- Docker Bench Security
- UlisesGascon/useful-npm-scripts
- appArmor
- seccomp
- veggiemonk/awesome-docker
- Docker Hub