Universitat Oberta de Catalunya

uoc.edu

Seguridad en Docker

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

Pau del Canto Rodrigo
Víctor García Font

4 de Junio de 2018

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Seguridad en Docker*
Nombre del autor:	*Xavier Fernández Ginés*
Nombre del consultor/a:	*Pau del Canto Rodrigo*
Nombre del PRA:	*Víctor García Font*
Fecha de entrega:	06/2018
Titulación:	*Máster en Seguridad de las Tecnologías de la Información y de las Comunicaciones*
Idioma del trabajo:	*Español*
Palabras clave	*Docker, Seguridad, "Best Practices"*

Resumen del Trabajo

Cuando empiezas a investigar sobre la tecnología de virtualización de Docker, lo primero que te llama la atención son las expectativas que ha levantado y está levantando por todo el mundo.

Docker es un proyecto que permite crear aplicaciones en contenedores de software que son ligeros, portátiles y autosuficientes.

La principal diferencia entre Docker con los modelos tradicionales de virtualización, es que utilizan la infraestructura nombrada contenedor en vez de las máquinas virtuales.

Los contenedores son un paquete de soluciones de software que te permite crear un entorno donde correr aplicaciones independientemente del sistema operativo que haya de base.

El propósito de este proyecto es investigar el uso de la virtualización de servidores basado en contenedores Docker, entender la tecnología y como funciona internamente, y por último, realizar un estudio de las mejores prácticas de seguridad que se pueden aplicar a los diferentes elementos que conforman el universo de Docker, así como soluciones que nos permitan mejorar la seguridad en los contenedores.

Abstract

When you start a research on Docker's virtualization technology, the first thing that catches your eye is the future expectations it has raised all over the world.

Docker is a project that allows you to create applications in software containers that are light, portable and self-sufficient

The main difference between Docker and traditional virtualization models is that it uses containers instead of virtual machines.

Containers are a package of items that allow you to create an environment in which applications run independently of the operating system.

The purpose of this project is to investigate the use of server virtualization based on Docker containers, understand the technology running behind it, and finally, conduct a study of the best security practices that can be applied to the different elements that make up Docker's universe, as well as solutions that allow us to improve security in containers.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y Justificación del Trabajo

Actualmente, y cada día va en aumento, las empresas se encuentran en el punto de mira de ataques de seguridad externos en los que el objetivo principal es acceder a información dentro de los sistemas, como podría ser información confidencial, y de esa forma poder chantajear a la empresa, pedir un rescate o divulgar esa información.

En el primer caso, en caso que el usuario consiga obtener información confidencial, este acontecimiento afectará tanto a nivel de reputación como el vínculo de confianza que había entre empresa-cliente.

En el segundo caso, normalmente suelen ser las páginas web públicas de las empresas, y en caso que el usuario consiga realizar cualquier ataque satisfactorio en éstas, la empresa se puede ver perjudicada dado que el usuario colgará la foto en las redes sociales, además de informar cómo se puede vulnerar la seguridad de esta empresa.

A día de hoy, muchas de las empresas están adoptando una arquitectura basada en micro servicios, dejando de lado la arquitectura monolítica. Esto es debido a que nos aporta mejor rendimiento, además de escalado de servidores, costes económicos, etc.

Actualmente una de las soluciones en las que muchas empresas están confiando es Docker, basado en contenedores.

El problema surge cuando, únicamente nos tenemos que fijar en las mejoras que nos aporta, alto rendimiento, escalado de servidores, coste económico, etc… dejando de lado la propia seguridad.

El propósito de este proyecto es investigar la seguridad existente en la virtualización de servidores basada en contenedores (Docker), entender la tecnología que se está ejecutando detrás de ella, revisar como poder realizar un “hardening” o una “*securización*” de los servidores delante de nuevas amenazas en forma de vulnerabilidades dentro del software instalado, así como poder revisar las diferentes maneras de poder auditar los contenedores una vez ha habido algún evento o ataque dejando un rastro mediante “*logs*”, y de esa forma detectar y analizar el vector de ataque y como poder mitigarlo en futuras ocasiones.

1.2 Objetivos del Trabajo

Para un resultado satisfactorio del proyecto, se plantean los siguientes objetivos:

- Analizar la arquitectura y el funcionamiento interno de Docker
- Revisar las imágenes oficiales del Docker Hub, para asegurar que los diferentes servicios instalados no tienen vulnerabilidades explotables.
- Evaluar posibles formas de realizar un "*hardening*" o mitigar posibles riesgos a raíz de vulnerabilidades de los contenedores de producción "en caliente".
- Generación y revisión de *logs* por parte de los diferentes servicios instalados en cada uno de los contenedores para poder realizar un análisis "post mortem" en caso de ataque.

1.4 Planificación del Trabajo

A continuación se han definido las diferentes etapas del proyecto, con una breve descripción de cada una de ellas:

Proyecto: Seguridad en Docker	PEC 1
Título: Documentación inicial	Cuadro 1 de 4
Descripción: Generación documentación inicial. Estudio de la tecnología y la arquitectura de virtualización de contenedores Docker.	Fecha inicial: 21/02/2018
	Fecha final: 12/03/2018

Proyecto: Seguridad en Docker	PEC 2
Título: Instalación de Docker y análisis del funcionamiento interno.	Cuadro 2 de 4
Descripción: Instalación de Docker, adquirir conocimientos sobre la gestión de contenedores y familiarizarse con las imágenes.	Fecha inicial: 13/03/2018
	Fecha final: 09/04/2018

Proyecto: Seguridad en Docker	PEC 3
Título: Estudio, creación y configuración de todo el entorno de Docker	Cuadro 3 de 4
Descripción: Creación de los ficheros para la generación de los contenedores, creación de contenedores y revisión del proceso de “hardening” de éstos. Creación de rastro de las acciones de los contenedores mediante un fichero de “log”	Fecha inicial: 10/04/2018
	Fecha final: 07/05/2018

Proyecto: Seguridad en Docker	PEC 4
Título: Memoria Final	Cuadro 4 de 4
Descripción: Recopilación de toda la información y generación del entregable final	Fecha inicial: 08/05/2018
	Fecha final: 04/06/2018

A continuación podemos observar el Diagrama de Gantt con las diferentes fases del proyecto:

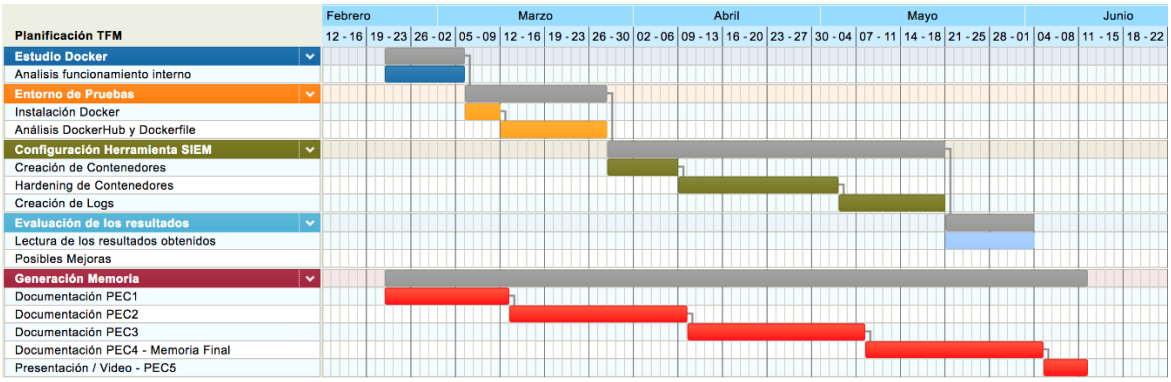

ILUSTRACIÓN 1: DIAGRAMA DE GANTT CON FASES DEL PROYECTO

1.5 Breve Sumario de Productos Obtenidos

Una vez finalizado el proyecto se esperan tener los siguientes resultados:

- Conocimiento funcionamiento interno Docker.

- Conocimiento de mejores prácticas a nivel de seguridad en Docker aplicables en:
    - Sistema Host
    - Demonio
    - Contenedores
    - Dockerfile

- Conocimiento de soluciones para realizar análisis de vulnerabilidades a contenedores.

- Conocimiento sobre la persistencia de trazas de los contenedores una vez finalizada su ejecución.

2. Estado del Arte

Cuando uno empieza a investigar sobre la tecnología de virtualización de contenedores Docker, lo primero que le llama la atención son las expectativas de futuro que ha levantado por todo el mundo.

Muchos analistas prevén que Docker es el siguiente paso en las tecnologías de virtualización y que en un futuro los contenedores reemplazarán a las máquinas virtuales.

Docker fue lanzado al mercado como un proyecto de código abierto por “dotCloud” en 2013, pero la idea de los contenedores no es nueva, ya que utiliza conceptos que han estado presentes desde los primeros días de Unix, como pueden ser los *namespace* y *cgroup*, para asegurar el aislamiento de los recursos y el empaquetado de una aplicación, así como de sus dependencias. Este empaquetado de dependencias permite al desarrollador escribir una aplicación en cualquier lenguaje y poder ejecutarla en diferentes entornos muy diferentes, independientemente de su sistema operativo.

Los contenedores de Linux (LXC) son la tecnología en la que se basa el software de Docker, que salió al mercado el 6 de agosto de 2008.
Docker, Inc es la compañía que está detrás del desarrollo de este software. Tiene su base en San Francisco (California) y en 2015 tenía más de 120 empleados trabajando.

Después de que Docker saliera al mercado, los desarrolladores empezaron a darse cuenta de cómo este nuevo enfoque podía solucionar muchos de sus problemas.
En agosto de 2013 Docker lanzó su tutorial interactivo, que fue probado por más de 10000 desarrolladores. En un año compañías como Red Hat o Amazon incorporaron soporte comercial para Docker y cuando Docker anunció su versión 1.0 en junio de 2014, el Docker Engine ya tenía 2,75 millones de descargas. Hoy en día ya tiene más de 100 millones.

Este gran éxito de Docker ha llamado la atención de otras compañías, que intentan darle otro enfoque a la virtualización de contenedores o incluso crear los suyos propios. Como respuesta a Docker, el CEO de la compañía CoreOS, Alex Polvi, lanzó al mercado Rocket, ya que según él, Docker no es seguro, ya que utiliza un daemon de Docker centralizado, en cambio Rocket utiliza un daemon de systemd para crear los contenedores.

Hay un gran debate sobre si los contenedores sustituirán a las máquinas virtuales, ya que los contenedores son más eficientes, por lo tanto, es una forma de hacer lo mismo que se hacía con los hipervisores, pero a un menor coste.

Como bien sabemos la industria de los hipervisores ha estado liderada por VMware y su más acérrimo competidor, Microsoft, para los cuales los contendores suponen tanto una amenaza como una oportunidad de negocio.

3. Docker

3.1 Introducción

Docker es una plataforma OpenSource creada para que los usuarios puedan desarrollar, enviar y ejecutar aplicaciones.

Los contenedores son un paquete de elementos que te permite crear un entorno donde correr aplicaciones independientemente del sistema operativo.

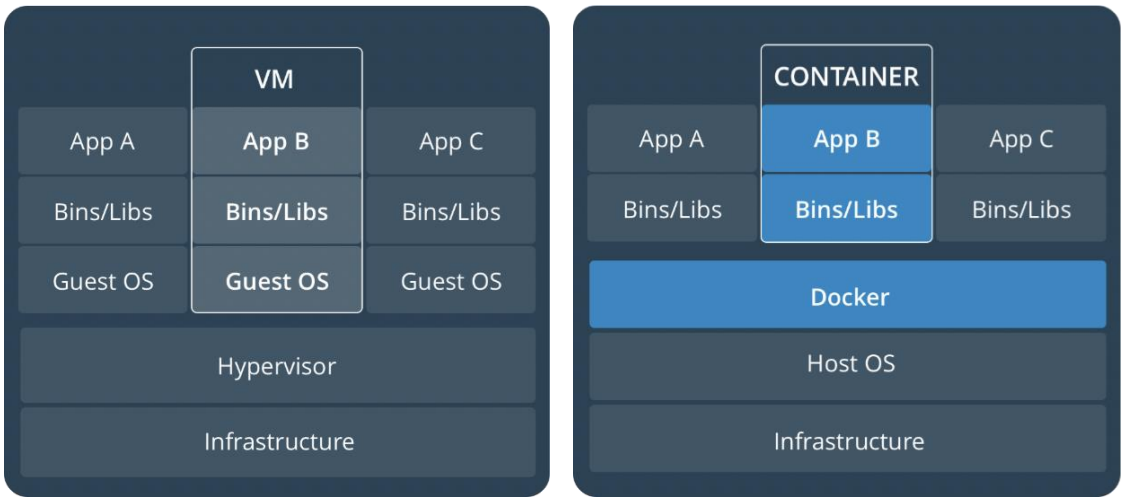

ILUSTRACIÓN 2: ARQUITECTURA CONTENEDORES

El código corre en estos contenedores totalmente aislado de otros contenedores y todos ellos comparten los recursos de la máquina sin la sobrecarga que aporta la capa del hipervisor.

Los contenedores son mucho más ligeros que las máquinas virtuales ya que, mientras que las máquinas virtuales requieren de la instalación de un sistema operativo, asignación de disco, CPU y memoria RAM para funcionar, un contenedor de Docker sólo necesita el sistema operativo que corre en la máquina en la que está el contenedor. Docker se compone de las siguientes partes:

- Docker Daemon
- Docker Client CLI
- Docker Hub
- Imágenes
- Contenedores

Los pasos que sigue Docker para correr un contenedor son los siguientes:
El Docker Client CLI (Command Line Interface) contacta con el Docker daemon, este demonio se encarga de descargar la imagen deseada del repositorio de Docker, llamado Docker Hub, y despliega un contenedor a partir de esa imagen. Si se ejecuta un programa en ese contenedor, la salida será enviada al Docker Client y el cliente lo enviará al terminal.

La capa en la que se ejecuta Docker se llama Docker Engine que fue escrito en el lenguaje Golang y corre en sistemas nativos Linux.
Docker actualmente no soporta checkpointing, restoring o migraciones en caliente entre hosts, pero es una funcionalidad que podría llegar en un futuro.

3.2. Docker Network

Hay varias formas en las que los contenedores se pueden conectar entre ellos y con el host. Las principales redes son las siguientes:

- None
- Bridge
- Host
- Overlay

3.2.1. None

None es sencillamente que el contenedor no tiene interfaz de red, aunque sí que tiene interfaz de loopback. Este modo se utiliza para contenedores que no utilizan la red, como pueden ser contenedores para pruebas que no necesiten comunicación externa o contenedores que pueden dejarse preparados para ser conectados a una red posteriormente.

3.2.2. Bridge

Un bridge de linux proporciona una red interna en el host a través de la cual los contenedores se pueden comunicar. Las direcciones IP asignadas en esta red no son accesibles desde fuera del host. La red “bridge” aprovecha las iptables para hacer NAT y mapeado de puertos. La red bridge es la red por defecto de Docker.

La creación del contenedor incluye los siguientes pasos respecto a la red:

- Se proporciona al host una red bridge.
- Un namespace para cada contenedor es proporcionado en cada bridge.
- Las interfaces de red de los contenedores se mapean a las interfaces privadas del bridge.
- Las iptables con NAT se usan para mapear entre cada contenedor y la interfaz pública del host.

3.2.3. Host

En este caso el contenedor comparte su namespace de red con el host, lo que se traduce en un mejor rendimiento ya que elimina la necesidad de NAT, pero esto también provoca conflictos de puertos. Por lo tanto, el contenedor tiene acceso a todas las interfaces de red del host. La red “host” es la que se utiliza por defecto en “Mesos”.

3.2.4. Overlay

Overlay utiliza los túneles de red para la comunicación de contenedores en diferentes hosts. Esto permite que dos contenedores que estén en hosts diferentes se comporten como si se encontraran en el mismo host, ya que hacen túneles de subredes de un host a otro.

La tecnología de tunneling que utiliza Docker es VXLAN (Virtual Extensible Local Area Network), que se incluye de forma nativa desde el lanzamiento de la versión 1.9.

Las redes multi-host requieren de parámetros adicionales cuando lanzamos el demonio de Docker, así como un registro key-value.

Este tipo de red es la utilizada en la orquestación de contenedores distribuidos en diferentes hosts, ya que no se podrían comunicar entre sí por la red “bridge” comentada anteriormente.

3.3. Docker Swarm

Docker Swarm nos permite gestionar un grupo de hosts de Docker como un solo host de Docker virtual. Swarm utiliza la API estándar de Docker, por lo que cualquier herramienta que se comunique con el Docker Daemon puede utilizar Docker Swarm, que permite la escalabilidad a varios hosts.

Para desplegar una imagen de una aplicación cuando Docker Engine está en modo Swarm hay que crear un servicio. Normalmente el servicio será la imagen de un micro servicio dentro de una aplicación más grande. Por ejemplo, un servicio puede ser un servidor HTTP, una base de datos, o cualquier otro tipo de programa ejecutable que se desee correr en un entorno distribuido.

Cuando se crea un servicio hay que especificar qué imagen de contendor usar, así como que comandos se ejecutarán en esos contenedores. También se definen las diferentes opciones de configuración del servicio como pueden ser:

- El puerto por el que Docker Swarm hará el servicio accesible desde el exterior.
- La red Overlay que permitirá conectar al servicio con otros servicios del clúster.
- Límites y reservas de memoria y CPU.
- Políticas de actualizaciones.
- Número de réplicas de la imagen que corren en el clúster.

En el momento de despliegue del servicio en Swarm, el gestor de Swarm acepta la definición como el estado deseado del servicio y distribuye el servicio en los nodos del clúster (dependiendo del número de réplicas). Estos procesos se ejecutan independientemente en cada uno de los nodos del clúster y se llaman task.

Por ejemplo, en la figura que podemos ver a continuación tenemos un servidor web Nginx que reparte la tarea de escucha HTTP en tres réplicas. Cada una de estas instancias es un proceso diferente en el clúster.

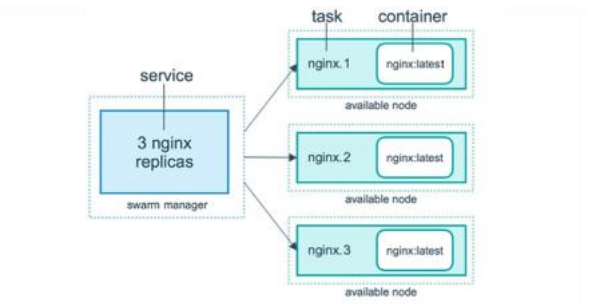

ILUSTRACIÓN 3: DIAGRAMA DE UN SERVICIO REPLICADO EN SWARM

En el modelo de Swarm cada task se corresponde con un contenedor.

Una task es la unidad básica de scheduling en un clúster de Swarm. Cuando declaras el estado deseado de un servicio mediante la creación de un servicio, el orquestador obtiene ese estado mediante el scheduling de tasks en los diferentes nodos.

Una task sigue un mecanismo unidireccional, ya que progresa siempre por una serie de estados (assigned, prepared, running). Si la task falla, el orquestador elimina la tarea y su contenedor y crea una nueva que la reemplaza para asumir el estado especificado en la creación del servicio.

El principal propósito de Docker Swarm es el de ser programador y orquestador de contenedores, por lo que los servicios y tasks tienen otro nivel de abstracción que les permite no ser conscientes de las aplicaciones que corren en los contenedores que implementan.

La siguiente figura muestra la manera en la que el gestor de Swarm acepta la creación de un servicio y envía la orden de ejecutar la task a los nodos.

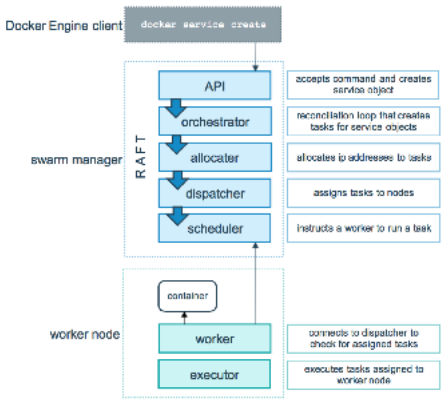

ILUSTRACIÓN 4: FLUJO DE TAREAS DEL GESTOR DE SWARM

Un servicio puede ser configurado de manera que ningún nodo del clúster pueda ejecutar sus tasks, en este caso el servicio se quedará en "*pending*".

Hay dos tipos de implementaciones de servicios: replicados y globales.
En el caso de los servicios replicados se especifica el número de tasks idénticas que se quieran ejecutar.

Un servicio global es un servicio que ejecuta una task en cada nodo. No hay un número especificado de tareas. Cada vez que añades un nodo al clúster, el orquestador de Swarm crea una task y el *scheduler* asigna la tarea a un nuevo nodo.

En el siguiente diagrama podemos observar en amarillo un servicio replicado de tres replicas en amarillo y uno global en gris.

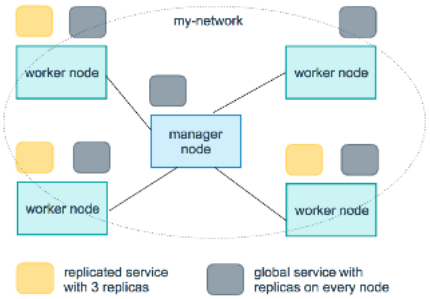

ILUSTRACIÓN 5: DIAGRAMA DE UN SERVICIO REPLICADO I UNO GLOBAL

Existen dos modos de disponibilidad en los nodos de Docker Swarm: "*active*" y "*drain*".

El modo de disponibilidad "*drain*" permite que el nodo no reciba nuevas tasks de Swarm, también implica que las tareas que se están ejecutando en ese nodo se paren y se relancen en otros nodos que estén en modo de disponibilidad activa.

Actualmente Docker Swarm tiene tres estrategias para decidir en qué nodos del clúster se deben ejecutar los contenedores:

- Spread: es el utilizado por defecto. Intenta balancear la carga de contenedores por igual en todos los nodos del clúster. Para ello tiene en cuenta la CPU y memoria RAM disponible, y el número de contenedores corriendo en cada nodo.
- BinPack: es todo lo contrario a Spread, este ejecuta todos los contenedores en el siguiente nodo disponible. El objetivo de ello es utilizar el menor número de nodos posible.
- Random: por último, como sugiere su nombre, esta estrategia utiliza un patrón aleatorio para la colocación de contendores en los nodos.

Swarm también permite el uso de cinco filtros que nos ayudan a gestionar los contenedores:

- Restriction: también llamados etiquetas de nodos, las restricciones son pares key/value asociados a un nodo en particular.
- Affinity: Este filtro se utiliza para hacer que diferentes contenedores se ejecuten en una misma red.
- Dependency: Cuando dos contenedores dependen uno del otro, este filtro permite situarlos en el mismo nodo.

- Health: Si un nodo no funciona correctamente, este filtro previene el despliegue de contendores en ese nodo.

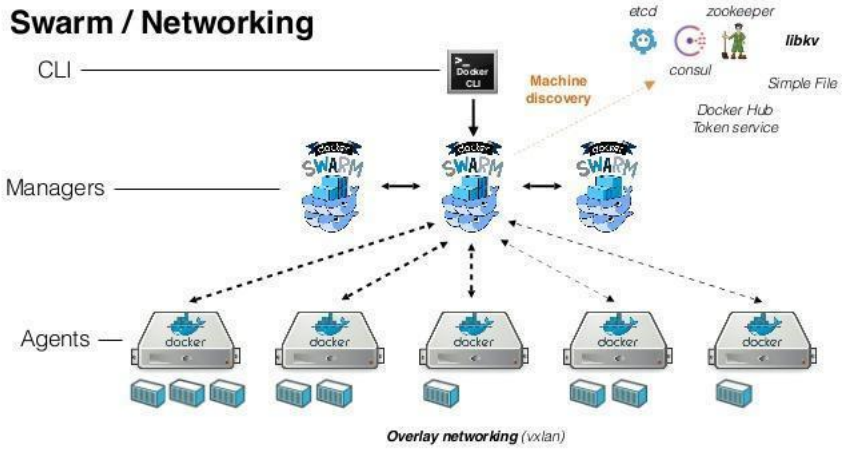

ILUSTRACIÓN 6: ARQUITECTURA CLÚSTER DE SWARM

3.4. Repositorios de Imágenes

Docker provee a su comunidad una fuente de recursos centralizada para el descubrimiento y distribución de imágenes de contenedores.

Actualmente Docker mantiene dos tipos de repositorios con imágenes ya creadas.

3.4.1. Docker Hub

Docker Hub es un repositorio de imágenes creados por la comunidad, en el que cualquier usuario puede crear su imagen y subirla en el repositorio para poder compartirlo con la comunidad.

Dentro de este repositorio existen dos tipos de imágenes, en función del origen de éstas. En primer lugar encontramos las oficiales, que son las que mantienen los principales distribuidores, como podría ser Apache, Nginx, MongoDB, Ubuntu, Alpine, etc.

Por otro lado, podemos encontrar las imágenes que han sido creadas por usuarios, por tanto, éstas, han sido customizadas y adaptadas en función a sus necesidades para el proyecto. Lo más probable es que haya customizado a partir de una imagen oficial.

3.4.2. Docker Store

Docker Store es un repositorio de imágenes mantenido por la propia empresa Docker, en el que este estaría más orientado al entorno empresarial, donde podemos encontrar plugins para contenedores, diferentes paquetes para ampliar las funcionalidades de nuestro entorno Docker e imágenes comerciales, en el que la empresa, en este caso Docker, nos garantiza que podemos confiar en el contenido de éstas.

3.5. Dockerfile

Para la creación de un contenedor, no es necesario obtener la imagen en uno de los dos repositorios indicados en el apartado anterior. En Docker, existe de posibilidad de poder crear tu propia imagen, en función de tus necesidades, mediante el fichero Dockerfile. Esta es una funcionalidad muy importante de cara a poder aplicar capas de seguridad en las diferentes imágenes que pueda crear un usuario, ya que se deberá de establecer pautas para poder cumplir con los básicos de seguridad.

El funcionamiento es el siguiente: el usuario en primer lugar crearía el fichero Dockerfile con el *formato* y las *directivas* marcadas por Docker. Una vez creado el fichero, se procede a la creación de la imagen mediante el comando *build*. Por último, sería necesario construir el contenedor a partir de la imagen creada anteriormente.

El fichero Dockerfile, tiene un formato muy en concreto, pero a la vez muy sencillo:

- Una línea que empieza por la almohadilla significa comentario:

  ```
  # comentario
  ```

- Para construir el fichero es necesario la introducción de Instrucciones seguido de los argumentos que queremos ejecutar, con el siguiente formato:

  ```
  INSTRUCCIÓN argumento
  ```

Dentro el apartado de Instrucciones, podemos encontrar muchas opciones.
A continuación se muestra un resumen con una breve explicación de las instrucciones que podemos introducir en la construcción de nuestra imagen:

`FROM:` define la imagen base sobre la que se construirá la nueva imagen.

`RUN`: ejecuta comandos dentro de la imagen creada. El comando tiene que tener lógica dentro de la imagen base.

`CMD`: provee valores por defecto al contenedor.

`ENTRYPOINT`: cuando es necesario correr un servicio en el contenedor.

`LABEL`: añade metadatos a una imagen.

`EXPOSE`: define los puertos por los cuales escuchara el contenedor, así como define el protocolo de red.

`ENV`: crea variables de entorno dentro del contenedor.

`ADD`: copia ficheros, directorios o ficheros remotos al directorio de destino dentro del contenedor.

`COPY`: copia ficheros y directorios locales dentro del contenedor.

`VOLUME`: crea un punto de montaje con el nombre indicado.

`USER`: establece el nombre de usuario (UID) y el nombre del grupo (GID) dentro del contenedor.

`WORKDIR`: establece el directorio de trabajo actual dentro del contenedor.

`ARG`: define variables de usuario en las que se pasan como parámetro durante la creación del contenedor.

`ONBUILD`: añade una instrucción disparador o *trigger* durante la creación de una imagen, en caso que está instrucción falle durante su ejecución, se cancelara la creación del contenedor.

`STOPSIGNAL`: define la señal que se le enviará al contenedor cuando este se pare.

`HEALTHCHECK`: comunica el estado de un contenedor.

`SHELL`: indica al contenedor que interprete de comandos se quiere usar por defecto.

A continuación, se muestra un ejemplo de un fichero Dockerfile de una imagen oficial de Nginx:

Dockerfile

```
FROM debian:stretch-slim
# all images must have a FROM
# usually from a minimal Linux distribution like debain or (even better) alpine
# if you truly want to start with an empty container, use FROM scratch

ENV NGINX_VERSION 1.13.6-1~stretch
ENV NJS_VERSION   1.13.6.0.1.14-1~stretch
# optional environment variable that's used in later lines and set as envvar when container is running

RUN apt-get update \
  && apt-get install --no-install-recommends --no-install-suggests -y gnupg1 \
  && \
  NGINX_GPGKEY=573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62; \
  found=''; \
  for server in \
    ha.pool.sks-keyservers.net \
    hkp://keyserver.ubuntu.com:80 \
    hkp://p80.pool.sks-keyservers.net:80 \
    pgp.mit.edu \
  ; do \
    echo "Fetching GPG key $NGINX_GPGKEY from $server"; \
    apt-key adv --keyserver "$server" --keyserver-options timeout=10 --recv-keys "$NGINX_GPGKEY" && found=yes && break; \
  done; \
  test -z "$found" && echo >&2 "error: failed to fetch GPG key $NGINX_GPGKEY" && exit 1; \
  apt-get remove --purge -y gnupg1 && apt-get -y --purge autoremove && rm -rf /var/lib/apt/lists/* \
  && echo "deb http://nginx.org/packages/mainline/debian/ stretch nginx" >> /etc/apt/sources.list \
  && apt-get update \
  && apt-get install --no-install-recommends --no-install-suggests -y \
            nginx=${NGINX_VERSION} \
            nginx-module-xslt=${NGINX_VERSION} \
            nginx-module-geoip=${NGINX_VERSION} \
            nginx-module-image-filter=${NGINX_VERSION} \
            nginx-module-njs=${NJS_VERSION} \
            gettext-base \
  && rm -rf /var/lib/apt/lists/*
# optional commands to run at shell inside container at build time
# this one adds package repo for nginx from nginx.org and installs it

RUN ln -sf /dev/stdout /var/log/nginx/access.log \
  && ln -sf /dev/stderr /var/log/nginx/error.log
# forward request and error logs to docker log collector

EXPOSE 80 443 8080
# expose these ports on the docker virtual network
# you still need to use -p or -P to open/forward these ports on host

CMD ["nginx", "-g", "daemon off;"]
# required: run this command when container is launched
# only one CMD allowed, so if there are mulitple, last one wins
```

ILUSTRACIÓN 7: DOCKER FILE DE IMAGEN OFICIAL NGINX

4. Seguridad en Docker

La seguridad es un factor cada vez más importante en el día de día de las empresas.

A continuación se propondrán diferentes guías de buenas prácticas para aumentar la seguridad en los contenedores una vez desplegados.

4.1. Configuración Entorno Docker

Dentro del mundo Docker, existen diferentes buenas prácticas que son recomendables llevar a cabo cuando se quiere desplegar un servicio con Docker.

4.1.1. Host

La máquina *Host* podemos definirla como la parte más importante del entorno Docker, dado que es donde se apoya toda la infraestructura y donde se ejecutaran los contenedores. Es por ello muy recomendable y a la vez muy importante seguir unas pautas de *Buenas Prácticas* a la hora de configurar la máquina donde se alojaran los diferentes contenedores.

4.1.1.1. Partición Específica para Docker

El primer punto a tener en cuenta cuando se quiera instalar Docker es el particionado del disco duro, debido a que Docker necesita almacenar las diferentes imágenes que el usuario ha ido utilizando (previamente descargada).

Para eso, es imprescindible que Docker sea instalado en la partición */var/lib/docker*, por defecto en Linux es instalado en esta partición, donde además, esta partición debe estar montada sobre el directorio raíz "*/*" o sobre el directorio "*/var*".

Para poder verificar la correcta instalación, en primera instancia, se debe verificar si se ha instalado Docker en el directorio especificado:

```
xavi@ubuntu:/var/lib/docker$ sudo ls
builder  containerd  containers  image  network  overlay2  plugins  runtimes  swarm  tmp  trust  volumes
xavi@ubuntu:/var/lib/docker$
```

ILUSTRACIÓN 8: DIRECTORIO INSTALACIÓN DOCKER

La segunda comprobación a realizar, si el directorio "*/var/lib/docker*", ubicado dentro de "*/dev/sda1*" se encuentra montado donde corresponde. En caso contrario, se debería gestionar con la herramienta para Linux *Logical Volume Manager* (LVM) para poder crear la partición correspondiente para docker:

```
xavi@ubuntu:/var/lib/docker$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            966M     0  966M   0% /dev
tmpfs           199M  1.6M  198M   1% /run
/dev/sda1        63G  7.4G   53G  13% /
tmpfs           994M     0  994M   0% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           994M     0  994M   0% /sys/fs/cgroup
```

ILUSTRACIÓN 9: VERIFICACIÓN UBICACIÓN DIRECTORIO DOCKER

4.1.1.2. Permisos de Usuario

Una vez instalado Docker en el directorio correcto, es necesario limitar que usuarios pueden controlar el demonio de Docker. Por defecto, el demonio solo puede ser controlado por el usuario *root* del sistema.

A continuación ejecutamos el contenedor básico de *Hello-World* para realizar la prueba que todo funciona correctamente.

```
xavi@ubuntu:/var/lib/docker$ docker run hello-world
docker: Got permission denied while trying to connect to the Docker daemon socke
t at unix:///var/run/docker.sock: Post http://%2Fvar%2Frun%2Fdocker.sock/v1.37/c
ontainers/create: dial unix /var/run/docker.sock: connect: permission denied.
See 'docker run --help'.
```

ILUSTRACIÓN 10: EJECUCIÓN CONTENEDOR HELLO-WORLD

Como se puede observar en la captura anterior, devuelve un error conforme el usuario no dispone de suficientes permisos para poder controlar el demonio.

Cuando Docker es instalado en una máquina, por defecto, se crea un grupo *docker*, donde únicamente los que sean integrados en este grupo, dispondrán de permisos para controlar el demonio. A la vez, Docker también puede ser controlado por todos aquellos usuarios que se encuentran en el grupo *sudo*.
Por tanto, es necesario verificar que usuarios disponen de permisos para controlar el demonio y añadir aquellos que sea necesario que puedan ejecutar acciones sobre Docker.

```
xavi@ubuntu:/var/lib/docker$ getent group docker
docker:x:999:
xavi@ubuntu:/var/lib/docker$ getent group sudo
sudo:x:27:xavi
```

ILUSTRACIÓN 11: PERMISOS DOCKER EN EL GRUPO DOCKER

Si a continuación ejecutamos el mismo comando pero con *superusuario* ya que el usuario *xavi* se encuentra dentro del grupo de sudo.

```
xavi@ubuntu:/var/lib/docker$ sudo docker run hello-world
[sudo] password for xavi:
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9bb5a5d4561a: Pull complete
Digest: sha256:f5233545e43561214ca4891fd1157e1c3c563316ed8e237750d59bde73361e77
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

ILUSTRACIÓN 12: EJECUCIÓN CORRECTA CONTENEDOR HELLO-WORLD

Para poder agilizar todo este proceso, es necesario añadir el usuario en cuestión al grupo *docker*. En este punto, es aconsejable llevar a cabo una buena custodia de las credenciales del usuario que se añadirá, debido a que dispondrá de permisos para realizar cualquier acción con el demonio de Docker.

```
xavi@ubuntu:/var/lib/docker$ sudo usermod -aG docker xavi
[sudo] password for xavi:
xavi@ubuntu:/var/lib/docker$ getent group docker
docker:x:999:xavi
```

ILUSTRACIÓN 13: AÑADIR USUARIO A GRUPO DOCKER

4.1.1.3. Auditoría Ficheros y Directorios

Como el demonio de Docker se ejecuta con privilegios de *root*, todos los directorios y ficheros deberían ser auditados constantemente para poder conocer todas sus actividades y sobretodo su uso. Para poder auditar de forma correcta todos los eventos, se debería usar el *framework* Linux Audit Daemon.

Con Linux Audit Daemon permite:

- Auditar accesos y modificaciones de ficheros.
- Monitorizar llamadas al sistema
- Detectar intrusiones
- Registrar comandos por los usuarios

Para configurar correctamente el demonio de auditoria, será necesario añadir nuevas reglas en el fichero "*/etc/audit/rules.d/audit.rules*". A continuación procederemos a añadir las reglas necesarias para poder auditar los directorios:

Open | audit.rules — /etc/audit/rules.d

```
## Rules
-w /usr/bin/docker -k docker
-w /var/lib/docker -k docker
-w /etc/docker -k docker
-w /usr/lib/systemd/system/docker.service -k docker
-w /usr/lib/systemd/system/docker.socket -k docker
-w /etc/default/docker -k docker
-w /etc/docker/daemon.json -k docker
-w /usr/bin/docker-containerd -k docker
-w /usr/bin/docker.runc -k docker
```

ILUSTRACIÓN 14: CONFIGURACIÓN REGLAS DE AUDITORIA

Una vez se han añadido las reglas, será necesario reiniciar el demonio de auditoria mediante el siguiente comando:

```
xavi@ubuntu:~$ sudo service auditd restart
xavi@ubuntu:~$
```

ILUSTRACIÓN 15: REINICIO DEMONIO DE AUDITORIA AUDITD

Finalmente, para poder revisar los logs generados durante la auditoria, se pueden encontrar en:

`/var/log/audit/audit.log`

4.1.2. Docker Daemon

El demonio de Docker es la parte más importante dado que es la pieza que se encarga de gestionar el ciclo de vida de las diferentes imágenes en la máquina host, tal y como hemos realizado la "*securización*" en el apartado anterior, como de administrar los recursos, ya sea de proporcionar como de limitar.

A continuación se analizaran diferentes "*Best Practices"* enfocadas principalmente al demonio de Docker.

4.1.2.1. Limitación Tráfico entre Contenedores

Una vez ya se está trabajando con contenedores desplegados, por defecto, Docker permite el tráfico entre contenedores desplegados en el mismo *Host*, por tanto, esto puede permitir que cada uno de los contenedores disponga de la capacidad de ver el tráfico de los otros contenedores, dado que todos ellos se encuentran en la misma red.

El hecho que pueda ver tráfico de otros contenedores, es una fuente indirecta de divulgación de datos hacia otros contenedores.

Para limitar el tráfico entre los diferentes contenedores desplegados y que por tanto, no se disponga de visibilidad entre ellos dentro de la misma máquina *Host*, existe un comando para deshabilitar esta opción.

Con el comando *dockerd* se está accediendo a las opciones que ofrece Docker para realizar modificaciones en el daemon de Docker.

```
$ dockerd COMMAND
```

Por tanto, para limitar el tráfico es necesario introducir el siguiente comando:

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

4.1.2.2. Autorización para el CLI

La autorización para el *Command Line Interface* se basa en los grupos de docker, tal y como se ha revisado en el punto *4.1.1.2. Permisos de Usuario*, donde es necesario añadir los usuarios que necesitan tener acceso al control general de Docker, principalmente para poder ejecutar órdenes mediante el demonio y de esa forma poder desplegar los servicios necesarios.

4.1.2.3. Gestión de Logs

La gestión de logs es una de las tareas más importantes dentro del mundo de la seguridad, ya que permite monitorizar que es lo que está pasando en todo momento en los contenedores.

En una mismo *Host* se pueden estar ejecutando diferentes contenedores de manera simultánea, donde cada uno de ellos, están generando sus propios *logs*, por tanto, es necesaria la gestión centralizada de los logs.

Existen varios comandos para la monitorización de los logs, como pueden ser:

```
$ docker logs <container_id>
```

```
$ docker service logs <SERVICE | TASK>
```

Los comandos anteriores son para los casos en los que la salida convencional (STDOUT y STDERR) se encuentra configurada de forma correcta.

En algunos casos, no será suficiente con el uso de los comandos anteriores dado que la información mostrada no será posible explotarla de forma idónea. En esos casos, hace falta seguir los siguientes pasos:

- En caso que dentro de un contenedor en ejecución se esté usando algún tipo de programa para tratar los *logs*, en ese caso, no sería aconsejable usar las instrucciones "*docker logs"* dado que no mostrará la información que se requiere.

- En caso que dentro del contenedor se esté ejecutando un proceso no interactivo como puede ser un servidor web podría disponer de un servicio que ya estuviera enviando los logs a algún fichero, por tanto, las salidas convencionales no se encontraran habilitadas, la solución reside en realizar una redirección de los logs convencionales.

  A continuación se muestran las opciones para poder redirigir y formatear los *logs* de forma que se puedan explotar de forma idónea. El comando y las diferentes opciones serán:

  ```
  $ dockerd COMMAND
  ```

Driver	Descripción
none	El comando *docker logs* no mostrara ninguna salida debido a que los *logs* no se encontraran disponibles.
json-file	Driver por defecto. Los *logs* serán formateados como JSON.
syslog	Los *logs* serán formateados como syslog. El demonio de syslog del sistema deberá estar ejecutándose.
journald	Los logs serán formateados como journald. El demonio de journald del sistema deberá estar ejecutándose.
gelf	Escribe los *logs* en un endpoint Graylog Extended Log Format (GELF) como podría ser Logstash.
fluentd	Los *logs* serán formateados como fluentd. El demonio de syslog del sistema deberá estar ejecutándose.
awslogs	Escribe los *logs* en Amazon CloudWatch Logs.
splunk	Escribe los *logs* en splunk.
gcplogs	Escribe los *logs* en Google Cloud Platform (GCP) Logging.

  Ejemplo:

  ```
  $ dockerd --log-driver syslog
  ```

4.1.2.4. Acceso Ficheros de Configuración Demonio

Tal y como se ha indicado en apartados anteriores, el demonio de Docker se ejecuta con permisos de root, por tanto, es muy importante limitar en todo momento los usuarios que disponen de control sobre el demonio.

En este apartado, se procederá a dar recomendaciones sobre cómo deberían configurar el acceso a los directorios y los ficheros del demonio a todos los usuarios que no sean los responsables de la gestión.

A continuación se indica cada uno de los ficheros que permisos deberían tener en un escenario por defecto (únicamente existe un usuario con permisos para controlar):

Fichero / Directorio	Usuario:Grupo	Permisos
docker.service	root:root	644 (rw-r--r--)
docker.socket	root:root	644 (rw-r--r--)
Socket de docker	root:docker	660 (rw-rw----)
/etc/docker	root:root	755 (rwxr-xr-x)
daemon.json	root:root	644 (rw-r--r--)
/etc/default/docker	root:root	644 (rw-r--r--)
Certificado del registro de docker	root:root	444 (r--r--r--)
Certificado de la CA del TLS	root:root	444 (r--r--r--)
Certificado de servidor de docker	root:root	444 (r--r--r--)
Clave privada del certificado de servidor de docker	root:root	400 (r--------)

Leyenda de Permisos:

- 7 es "*rwx*" (permiso de lectura, escritura y ejecución).
- 6 es "*rw*" (permiso de lectura y escritura)
- 5 es "*rx*" (permiso de lectura y ejecución)
- 4 es "*r*" (permiso de lectura)
- 0 es "-" (sin permisos).

4.1.3. Creación de Imágenes a Medida con Dockerfile

Una vez aplicadas buenas prácticas de seguridad tanto en la máquina host como al demonio así como a todos sus ficheros de configuración, a continuación se procede a identificar las buenas prácticas a la hora de crear imágenes de Docker a medida mediante el fichero Dockerfile.

Un fichero Dockerfile contiene todas las descripciones así como las instrucciones necesarias para poder construir una imagen de docker, por tanto, es necesario crear los ficheros Dockerfile de forma minimalista en cuanto a aplicaciones a ejecutar, así como usuarios que estarán presentes, que servicios se publicaran, etc. Es tan importante que depende de cómo los usuarios creen la imagen afectará a los contenedores que se ejecuten a posteriori.

4.1.3.1. Ejecución con Usuario no Root

Tal y como se ha indicado en apartados anteriores, por defecto, los contenedores se ejecutan con el usuario root, por tanto, se dispone de privilegios root dentro del contenedor.

La solución reside en especificar en la creación de la imagen, el usuario que se quiere que pueda ejecutar, además que una vez se ha ejecutado el contenedor, por defecto el usuario que lo ejecutará a partir de ese momento será el especificado en el Dockerfile.

Para añadir el usuario dentro del fichero Dockerfile, es necesario usar el comando:

```
RUN useradd <opciones>

USER <usuario>
```

A continuación se realiza una prueba para verificar el correcto funcionamiento de esta práctica:

En primer lugar, es necesario generar el fichero Dockerfile con la imagen base de Ubuntu (forzando última versión). A continuación se puede ver el fragmento del fichero, donde se encuentran comentadas las dos líneas importantes en este punto.

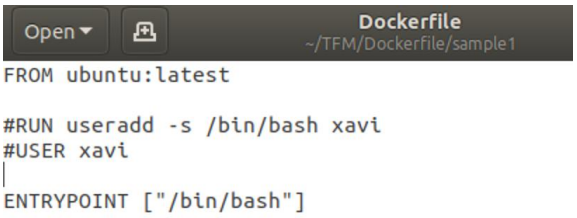

ILUSTRACIÓN 16: CREACIÓN IMAGEN UBUNTU SIN USUARIO

A continuación se realiza la ejecución del contenedor con la opción "*interactiva*" (*-i*), de esa forma resta a la espera de interacción con el usuario, además de la previa construcción de la imagen a partir del fichero Dockerfile.

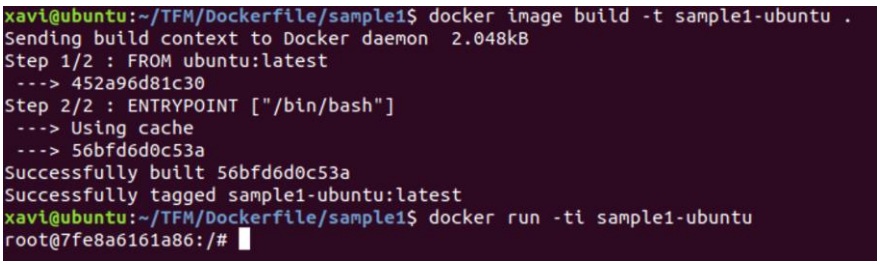

ILUSTRACIÓN 17: EJECUCIÓN IMAGEN UBUNTU CON USUARIO ROOT

Se observa que por defecto, se ejecuta dentro del contenedor ejecutado se está trabajando con el usuario root. A continuación se eliminan los comentarios del fichero Dockerfile, el resultado final será el siguiente:

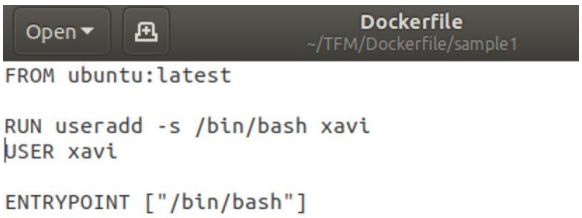

ILUSTRACIÓN 18: CREACIÓN DE IMAGEN DE UBUNTU CON USUARIO

Y por último, se vuelven a realizar las acciones anteriores, crear la imagen y ejecutar el contenedor. El resultado se puede observar que el contenedor se está ejecutando con el usuario xavi, por tanto, dentro del contenedor ya no hay un usuario con privilegios de *root*.

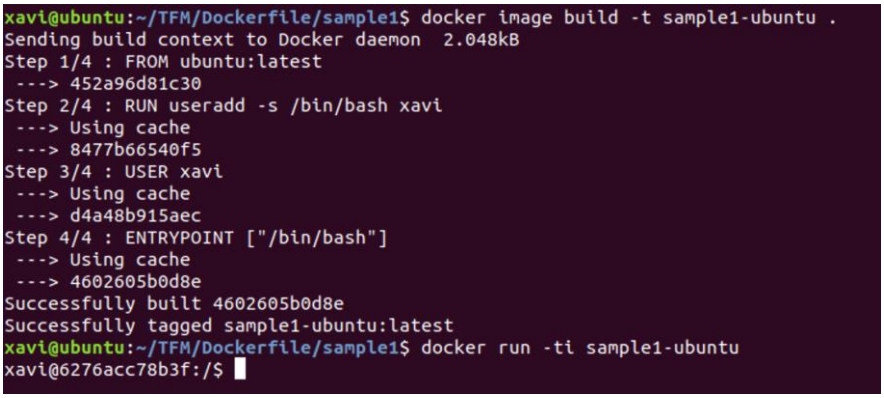

ILUSTRACIÓN 19: EJECUCIÓN IMAGEN UBUNTU CON USUARIO NO-ROOT

4.1.3.2. Content Trust

En este apartado, se quiere revisar la integridad de las imágenes como las diferentes capas que la componen. Por cada capa y por cada imagen disponen de un hash SHA256 que permite verificar si esa imagen ha sido modificada. Sin embargo, si alguna de estas imágenes son descargadas a través de alguna red insegura o sencillamente es publicada por un tercero o usuario malicioso, no es posible solo con la información de los hash para poder detectarlo.

Para ello, existen dos comandos que son introducidos en el fichero Dockerfile que permiten poder firmar la imagen a la hora de construirlas o de publicarlas en el propio registro de Docker, y por tanto, poder verificar dicha firma cuando se descargan del registro.

Para la firma de las imágenes, habrá que tener en cuenta las siguientes órdenes:

```
DOCKER_CONTENT_TRUST

DOCKER_CONTENT_TRUST_SERVER
```

La primera orden, permite habilitar y deshabilitar la verificación del *Content Trust* de Docker. En caso que se encuentre habilitado, docker realizará una verificación de la integridad de la imagen, ya sea público o privado el repositorio, apoyándose en todo momento a la máquina referida al servicio *Notary*. Esta comprobación la realizará el motor de docker cuando se realice un *push, build, create, pull o run* de una imagen de docker.

La segunda orden, permite definir la URL donde se encuentra el servidor *Notary.* En la gran mayoría de los casos, las empresas se nutren de las imágenes del repositorio de imágenes Docker Hub, por tanto, el servicio de *Notary* se llevará a cabo por la empresa Docker.

A continuación se indican como se introducirían los comandos anteriores en el fichero Dockerfile para poder firmar la imagen.

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

4.1.3.3. Eliminación Permisos setuid y setgid

Los permisos *Set User ID* (setuid) y *Set Group ID* (setgid) son permisos especiales que normalmente son usados para la granularidad de los accesos a directorios y ficheros del sistema operativo, para que usuarios que no dispongan de permisos root sean capaces de ejecutar esos ficheros y/o acceder a los directorios donde se encuentran ubicados los ficheros.

En el punto anterior, donde se ha definido los usuarios que podrían trabajar en el contenedor en ejecución (punto 4.1.3.1. Ejecución con Usuario no root), a su vez, es necesario definir que permisos se requieren para los directorios y ficheros dentro del contenedor en ejecución.

Existe un comando en los que se eliminan los permisos *setuid* y *setgid* durante la fase de construcción de la imagen mediante el fichero Dockerfile (preferiblemente al final del fichero).

```
RUN find / -perm +6000 -type f -exec chmod a-s {} \; || true
```

Este comando realiza una búsqueda de los ejecutables y elimina cualquier permiso de *setuid* y de *setgid* de cualquier usuario, incluso de los legítimos.

Es importante aplicar este comando de forma cuidadosa, dado que seguramente sea necesario que los programas legítimos requieran de dichos permisos y que por tanto, no se queden inservibles dentro del contenedor.

4.1.3.4. Uso de Instrucción Específica: Copy

A la hora de la creación de la imagen, nace la necesidad de poder copiar ficheros necesarios dentro den contenedor para poder ejecutar funciones que requieran de esos ficheros. Un ejemplo muy claro es cualquier servidor web que se quiera construir, será necesario añadir el contenido estático en el contenedor.

Para poder realizar dichas funciones, Docker proporciona dos instrucciones específicas:

`COPY`

`ADD`

La instrucción *COPY* únicamente es capaz de copiar ficheros de la máquina *Host* a la imagen que se está construyendo. A diferencia de la instrucción *ADD*, que aparte de poder copiar ficheros de la máquina *Host* a la imagen, además puede recuperar ficheros de URLs remotas y realizar operaciones con esos ficheros, como podrían ser desempaquetar o descomprimir ficheros de forma automática.

Por tanto, aunque sean muy similares las dos instrucciones, se debe priorizar a utilizar la instrucción *COPY* por encima de *ADD* debido a que se están añadiendo vulnerabilidades potenciales, dado que se dispone de la capacidad de descargar ficheros maliciosos de URLs que se desconoce su origen y no han sido escaneadas previamente, además de procedimientos de desempaquetado o descompresión automáticos.

4.1.3.5. No Almacenamiento de Secretos

Cuando se está generando una imagen mediante el fichero Dockerfile, es común usar este fichero y declarar diferentes secretos para poder realizar pruebas de conexión a esa Base de Datos o cualquier otro servicio, y una vez finalizadas las pruebas, se genera una estructura para almacenar estos secretos, pero en muchos casos no se elimina esa información de los ficheros de configuración, en este caso Dockerfile.

Cuando se habla de secretos, se referencia usuarios, passwords de servicios, claves privadas de certificados u otro tipo de información sensible en los que únicamente debería de conocer el administrador de los sistemas y que en muchos casos, salen a la luz.

Por tanto, la solución reside en realizar una revisión de los Dockerfile y que secretos se están exponiendo. Por otro lado, una de las soluciones reside en delegar está funcionalidad a orquestadores como podría ser *Docker Swarm* o el orquestador más popular que existe actualmente (ya integrado con Docker) *Kubernetes*.

A continuación se indica la información sobre donde se puede encontrar la solución para los secretos:

- Docker Swarm (https://docs.docker.com/engine/swarm/secrets)
- Kubernetes (https://kubernetes.io/docs/concepts/configuration/secret)

4.1.4. Contenedores

Una vez aplicadas buenas prácticas de seguridad tanto a la máquina *Host* sobre la que se van a ejecutar los contenedores, como al demonio de docker que va a gestionar todo el ciclo de vida de los diferentes contenedores que se ejecuten en la máquina *Host*, así como la creación a medida de las imágenes mediante el fichero Dockerfile, el próximo paso es realizar una revisión de buenas prácticas al *runtime* de los contenedores.

A continuación se profundizará en los aspectos más relevantes a la hora de aplicar buenas prácticas de seguridad para, limitar el uso de recursos de la máquina *Host*, como a nivel de mapear los puertos del contenedor.

4.1.4.1. Permisos de Linux Restringidos en los Contenedores

Los contenedores, por defecto, arrancan con una serie de privilegios de Linux limitados. Uno de los puntos fuertes, es la propia granularidad que dispone estos sistemas para poder proporcionar permisos, en caso que requiera la situación, y poder llevar a cabo la funcionalidad que se necesitaba ejecutar, sin llegar a proporcionar permisos de *root*.

Del mismo modo en que se pueden añadir diferentes privilegios, por defecto, lo ideal, en términos de seguridad es siempre aplicar lo más restringido o ser lo más minimalista posible. En otras palabras, no proporcionar permisos hasta que se demuestre que son necesarios para poder ejecutar las diferentes funcionalidades requeridas. Por tanto, aplicando este concepto, se consigue minimizar la exposición del contenedor.

Para poder proporcionar o eliminar permisos de Linux a los diferentes contenedores, docker proporciona el siguiente comando:

```
$ docker run <comando>
```

Donde en el comando podemos aplicar el añadir o simplemente eliminar privilegios mediante:

```
$ docker run --cap-add={OPCIÓN}
$ docker run --cap-drop={OPCIÓN}
```

Dentro de las opciones, se puede encontrar:

Privilegio	**Descripción**
SETPCAP	Si un fichero no dispone de capabilities, las añade o las elimina al conjunto de permitidas del invocante a/desde cualquier otro proceso.
MKNOD	Crea ficheros especiales usando el comando mknod.
AUDIT_WRITE	Escribe registros al log de auditoria del kernel.
CHOWN	Permite modificar propietario y grupo a ficheros.
NET_RAW	Permite el uso del socket RAW y del socket PACKET (permite el uso de herramientas como ping o tcpdump).
DAC_OVERRIDE	Bypass en las comprobaciones de los permisos de lectura, escritura y ejecución de un fichero.
FOWNER	Bypass en las comprobaciones de los permisos que requiere el UID del proceso coincida con el UID del fichero.
FSETID	No modifica los permisos setuid y setgid cuando se modifica un fichero
KILL	Bypass en las comprobaciones de permisos para el envío de señales.

SETGID	Permite añadir un mapeo GID a un namespace de usuario.
SETUID	Permite añadir un mapeo UID a un namespace de usuario.
NET_BIND_SERVICE	Permite enlazar el socket de un servicio a un puerto privilegiado (puertos < 1024).
SYS_CHROOT	Utiliza el comando chroot.
SETFCAP	Permite establecer capabilities de un fichero.

Un ejemplo podría ser el siguiente, donde se está limitando la opción *NET_DRAW y KILL*:

```
$ docker run --cap-drop={“NET_DRAW”, “KILL”} -d -p 27017:27017 mongo
```

4.1.4.2. Restricción de Ejecución de Contenedores Privilegiados

Siguiendo con las pautas del punto anterior, sobre los privilegios con los que se puede ejecutar un contenedor, en este punto, se trata con la posibilidad de ejecutar el contenedor con privilegios especiales.

Una de las funcionalidades donde por defecto no dispondría de privilegios, es la ejecución de docker dentro de docker.

Para poder añadir la funcionalidad de contenedor privilegiado, se añade el siguiente flag al comando *docker run*.

```
$ docker run --privileged <IMAGEN>
```

Mencionar que en ningún caso, se debería usar este tipo de acciones, a no ser que lo requiera la situación, dado que aplicando la mentalidad de seguridad, en todo momento es necesario ser minimalista y ser lo más restrictivo posible.

4.1.4.3. Restricción de Protocolos

En general, tal y como se ha venido indicando durante las buenas prácticas de seguridad, únicamente se deberían de abrir esos protocolos en los que son requeridos por el servicio en cuestión.

En este caso, únicamente se trabajará con el protocolo SSH. En ningún caso se debe habilitar el protocolo SSH en un contenedor, únicamente se debe permitir el protocolo SSH a la máquina *Host* dado que desde ésta, es donde se controlan todos los contenedores.

El desplegar un contenedor con el servicio SSH habilitado, la consecuencia es que se eleva de forma innecesaria la complejidad de mantener un entorno seguro, ya que dificultan las políticas de gestión de acceso, las claves, las contraseñas. Toda gestión de contenedores debería de realizarse de forma centralizada desde la máquina *Host* que es donde se encuentran hospedados.

En caso que el servicio SSH se encuentre ejecutándose dentro de un contenedor desplegado, la solución es desinstalar el servicio ya sea desde el propio contenedor en ejecución (en caso que tenga habilitada la consola) y en caso contrario, se debería gestionar desde el fichero Dockerfile, volver a construir la imagen y volver a desplegar la infraestructura. Para poder desinstalarlo, es necesario ejecutar el siguiente comando, en función de la distribución con la que se esté trabajando.

En distribuciones Red Hat / CentOS / Fedora

```
$ sudo dnf erase openssh-server
```

En distribuciones Debian / Ubuntu

```
$ sudo apt-get --purge remove openssh-server
```

4.1.4.4. Restricción de Puertos

En este apartado se trabajará con todos aquellos puertos denominados "*Well-known ports*", son todos aquellos puertos que están por debajo de 2014 y por tanto, son considerados como puertos privilegiados en cualquier sistema, esto significa que los usuarios y procesos que no sean *root* o tengan permisos *root*, no podrán atarse a ninguno de los puertos privilegiados.

Sin embargo, Docker permite que un contenedor pueda mapear un puerto del contenedor a un puerto privilegiado, porque el propio demonio de docker requiere privilegios de usuario *root* para ejecutarse y además, los contenedores disponen de la capacidad suficiente como para hacerlo debido a que se ejecutan de base con las capabilities del kernel de Linux "*NET_BIND_SERVICE*" habilitada, tal y como se ha visto en el punto 4.1.4.1 (Permisos de Linux restringidos en los contenedores).

Por defecto, en caso que no se especifique un puerto a mapear entre contenedor y host, docker automáticamente mapea un puerto disponible dentro del rango 49153 – 65535 de la máquina host.

Una buena práctica de seguridad consistiría en no permitir que los contenedores puedan mapear los puertos privilegiados de la máquina host. Pero existen excepciones, como podría ser el caso que en el contenedor este ejecutando un servicio HTTP o HTTPS, por tanto, en este caso, no habría ningún problema en mapear el puerto 80 o el puerto 443, respectivamente.

4.1.4.5. Limitación Recursos

El servicio de docker, por defecto, todos los contenedores comparten los recursos de la máquina *host* de forma equitativa, esto significa que no existe ninguna preferencia entre contenedores a la hora de consumir recursos.

Uno de los problemas que puede surgir, y por eso existen aplicaciones o software destinado a la monitorización de la infraestructura o clúster de contenedores para poder ver, de forma granulada, que contenedores pueden estar afectando a la estabilidad de toda la infraestructura, y a raíz de esto, provocar una denegación de servicio a la máquina host y por tanto, afectar a todo el eco-sistema de contenedores impidiendo ejecutar su operativa normal.

Una posible solución a los problemas de recursos que puedan surgir, es limitar cada uno de los contenedores mediante el comando:

```
$ docker run [OPTIONS] IMAGE [COMMAND] [ARG…]
```

El comando anterior, dispone de distintos parámetros de configuración que permiten tanto limitar el uso de recursos, como priorizar en caso que haga falta en un momento

en concreto para poder satisfacer en todo momento las necesidades de los diferentes usuarios.

A continuación se muestra las diferentes opciones que dispone el comando en cuestión para poder satisfacer las necesidades en cuanto a rendimiento, como podría ser *cpu, velocidad de lectura/escritura y la memoria.*

```
xavi@ubuntu:~/TFM/Dockerfile/sample1$ docker run --help | grep 'cpu\|device\|memory'
      --blkio-weight-device list       Block IO weight (relative device weight)
      --cpu-period int                 Limit CPU CFS (Completely Fair Scheduler) period
      --cpu-quota int                  Limit CPU CFS (Completely Fair Scheduler) quota
      --cpu-rt-period int              Limit CPU real-time period in microseconds
      --cpu-rt-runtime int             Limit CPU real-time runtime in microseconds
  -c, --cpu-shares int                 CPU shares (relative weight)
      --cpus decimal                   Number of CPUs
      --cpuset-cpus string             CPUs in which to allow execution (0-3, 0,1)
      --cpuset-mems string             MEMs in which to allow execution (0-3, 0,1)
      --device list                    Add a host device to the container
      --device-cgroup-rule list        Add a rule to the cgroup allowed devices list
      --device-read-bps list           Limit read rate (bytes per second) from a
                                       device (default [])
      --device-read-iops list          Limit read rate (IO per second) from a device
      --device-write-bps list          Limit write rate (bytes per second) to a
                                       device (default [])
      --device-write-iops list         Limit write rate (IO per second) to a device
      --kernel-memory bytes            Kernel memory limit
  -m, --memory bytes                   Memory limit
      --memory-reservation bytes       Memory soft limit
      --memory-swap bytes              Swap limit equal to memory plus swap: '-1' to
      --memory-swappiness int          Tune container memory swappiness (0 to 100)
```

ILUSTRACIÓN 20: USAGE DEL COMANDO DOCKER RUN

4.1.4.6. Aplicaciones Específicas

Una vez revisado todas las buenas prácticas aplicables para incrementar o mejorar la seguridad en el runtime de los contenedores, en este punto, introduciremos dos herramientas que permiten aislar aplicaciones de otras dentro de un sistema, permitiendo que en caso que un atacante malicioso consiga comprometer una de las aplicaciones y/o servicios de un contenedor, únicamente disponga de visión sobre lo comprometido quedando totalmente aislado del resto del sistema. Las aplicaciones responsables de poder realizar este tipo de aislado son:

- SELinux: es un proyecto de código abierto en el que dispone de un gran conjunto de reglas en las que permiten implantar los principales paradigmas de seguridad en los contenedores. Los principales paradigmas con los que trabaja son:
  - *MAC – Mandatory Access Control*
  - *MLS – Multi-Level Security (Bell-LaPadula o Biba)*
  - *RBAC – Role Based Access Control*
  - *TE – Type Enforcement*

- AppArmor: es otro proyecto en el que no usa paradigmas tan complejos como se ha visto en *SELinux*, sino que se basa en la ideología de denegar por defecto a la aplicación y por otro lado, a proteger y permitir el resto del sistema. Por tanto, se decide que aplicación se quiere proteger y el principio de denegación únicamente se aplica sobre la aplicación y no sobre el sistema, a diferencia de la aplicación anterior, en que sigue por defecto la aproximación de denegar todo por defecto.

En función de la distribución de Linux, se encuentra por defecto una aplicación u otra, o sencillamente, no se encuentra instalada y es necesario su implantación:

- En distribuciones Red Hat / CentOS / Fedora por defecto viene instalado SELinux.
- En distribuciones Suse / OpenSuse por defecto viene instalado AppArmor.
- En cualquier otra distribución, se puede instalar tanto SELinux como AppArmor.

Para poder aplicar las capas de seguridad ofrecidas por cada una de las aplicaciones, es necesario ejecutar los siguientes comandos en función de la aplicación que queremos que se ejecute con el contenedor.

En primer lugar, si queremos ejecutar por defecto AppArmor, es necesario introducir el siguiente flag (en negrita) cuando se ejecute el contenedor.

```
$ docker run --rm --security-opt apparmor=docker-default <imagen>
```

Por otro lado, en caso que la intención es trabajar con SELinux, en este caso, el funcionamiento es diferente a la anterior aplicación. Es necesario activar el demonio de docker antes de ejecutar el contenedor.

```
$ dockerd –selinux-enabled
```

Una vez activado el comando en el demonio, es necesario introducir el siguiente flag (en negrita) cuando se ejecute el contenedor.

```
$ docker run -i --tty --security-opt label=level:TopSecret <imagen>
```

4.1.5. Docker Bench Security

Por último, además de las buenas prácticas indicadas en todos los puntos anteriores, existen otras buenas prácticas aplicables, donde en ningún momento sustituyen a las anteriores.

En la versión 1.12.0 del 28 de julio de 2016, Docker añadió un nuevo servicio dentro de su ecosistema nombrado Docker Swarm. El principal objetivo de este servicio es el de la gestión u orquestación de contenedores dentro de una misma máquina host.

Debido a la aparición de este nuevo componente, aparecen nuevas prácticas de seguridad relacionadas con Docker Swarm que tienen como objetivo, verificar que existe el número mínimo de nodos maestros en el clúster de Docker Swarm, o que la comunicación entre los diferentes nodos es una comunicación cifrada.

Debido a este requisito de acciones a verificar antes de empezar a desplegar contenedores de forma segura en un entorno de producción, se decidió crear una herramienta para poder realizar una verificación de todas las buenas prácticas. Esta herramienta se llama Docker Bench Security, y es únicamente un script en el que se ejecuta y revisa cada una de las buenas prácticas recomendables a nivel de seguridad si realmente se encuentran implantadas y proporciona una foto del estado en el que se encuentra el ecosistema de Docker antes de empezar a desplegar contenedores.

Todas estas buenas prácticas, están inspirados en el *CIS Docker Community Edition Benchmark*, el cual proporciona una guía de buenas prácticas de seguridad aplicables a los diferentes entornos vistos en los puntos anteriores, máquina host, demonio de docker, prácticas de seguridad a las imágenes y a los contenedores de docker.

A continuación se muestra una ejecución completa de Docker Bench Security con sus resultados.

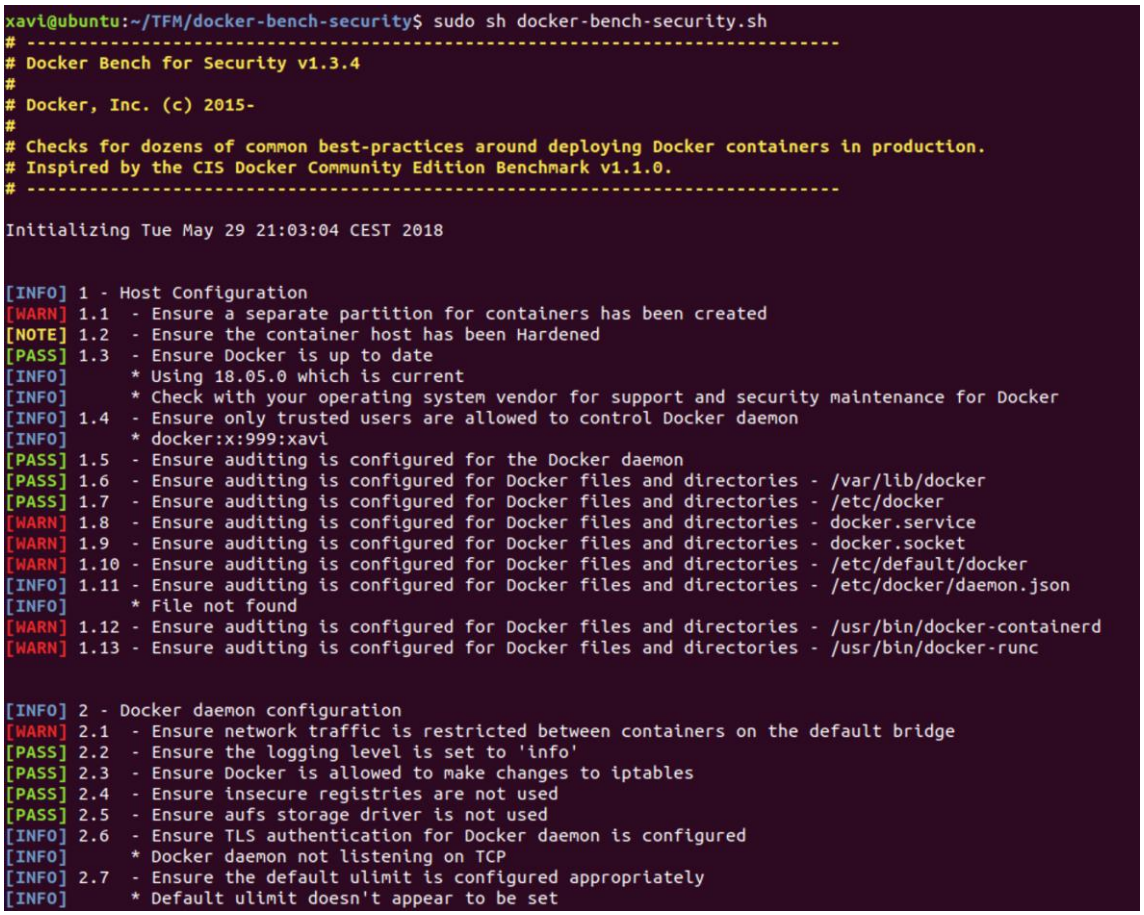

ILUSTRACIÓN 21: RESULTADO INFORME DOCKER BENCH SECURITY (PARTE 1)

```
[WARN] 2.8  - Enable user namespace support
[PASS] 2.9  - Ensure the default cgroup usage has been confirmed
[PASS] 2.10 - Ensure base device size is not changed until needed
[WARN] 2.11 - Ensure that authorization for Docker client commands is enabled
[WARN] 2.12 - Ensure centralized and remote logging is configured
[WARN] 2.13 - Ensure operations on legacy registry (v1) are Disabled
[WARN] 2.14 - Ensure live restore is Enabled
[WARN] 2.15 - Ensure Userland Proxy is Disabled
[PASS] 2.16 - Ensure daemon-wide custom seccomp profile is applied, if needed
[PASS] 2.17 - Ensure experimental features are avoided in production
[WARN] 2.18 - Ensure containers are restricted from acquiring new privileges


[INFO] 3 - Docker daemon configuration files
[PASS] 3.1  - Ensure that docker.service file ownership is set to root:root
[PASS] 3.2  - Ensure that docker.service file permissions are set to 644 or more restrictive
[PASS] 3.3  - Ensure that docker.socket file ownership is set to root:root
[PASS] 3.4  - Ensure that docker.socket file permissions are set to 644 or more restrictive
[PASS] 3.5  - Ensure that /etc/docker directory ownership is set to root:root
[PASS] 3.6  - Ensure that /etc/docker directory permissions are set to 755 or more restrictive
[INFO] 3.7  - Ensure that registry certificate file ownership is set to root:root
[INFO]      * Directory not found
[INFO] 3.8  - Ensure that registry certificate file permissions are set to 444 or more restrictive
[INFO]      * Directory not found
[INFO] 3.9  - Ensure that TLS CA certificate file ownership is set to root:root
[INFO]      * No TLS CA certificate found
[INFO] 3.10 - Ensure that TLS CA certificate file permissions are set to 444 or more restrictive
[INFO]      * No TLS CA certificate found
[INFO] 3.11 - Ensure that Docker server certificate file ownership is set to root:root
[INFO]      * No TLS Server certificate found
[INFO] 3.12 - Ensure that Docker server certificate file permissions are set to 444 or more restrictive
[INFO]      * No TLS Server certificate found
[INFO] 3.13 - Ensure that Docker server certificate key file ownership is set to root:root
[INFO]      * No TLS Key found
[INFO] 3.14 - Ensure that Docker server certificate key file permissions are set to 400
[INFO]      * No TLS Key found
[PASS] 3.15 - Ensure that Docker socket file ownership is set to root:docker
[PASS] 3.16 - Ensure that Docker socket file permissions are set to 660 or more restrictive
[INFO] 3.17 - Ensure that daemon.json file ownership is set to root:root
[INFO]      * File not found
[INFO] 3.18 - Ensure that daemon.json file permissions are set to 644 or more restrictive
[INFO]      * File not found
[PASS] 3.19 - Ensure that /etc/default/docker file ownership is set to root:root
[PASS] 3.20 - Ensure that /etc/default/docker file permissions are set to 644 or more restrictive


[INFO] 4 - Container Images and Build File
[INFO] 4.1  - Ensure a user for the container has been created
[INFO]      * No containers running
[NOTE] 4.2  - Ensure that containers use trusted base images
[NOTE] 4.3  - Ensure unnecessary packages are not installed in the container
[NOTE] 4.4  - Ensure images are scanned and rebuilt to include security patches
[WARN] 4.5  - Ensure Content trust for Docker is Enabled
[WARN] 4.6  - Ensure HEALTHCHECK instructions have been added to the container image
[WARN]      * No Healthcheck found: [sample1-nginx:latest sample1-ubuntu:latest]
[WARN]      * No Healthcheck found: [sample1-nginx:latest sample1-ubuntu:latest]
[WARN]      * No Healthcheck found: [xavifg/tfm-sample1-nginx:latest]
[WARN]      * No Healthcheck found: [nginx:latest]
[WARN]      * No Healthcheck found: [ubuntu:latest]
[WARN]      * No Healthcheck found: [hello-world:latest]
[INFO] 4.7  - Ensure update instructions are not use alone in the Dockerfile
[INFO]      * Update instruction found: [xavifg/tfm-sample1-nginx:latest]
[INFO]      * Update instruction found: [nginx:latest]
[NOTE] 4.8  - Ensure setuid and setgid permissions are removed in the images
[INFO] 4.9  - Ensure COPY is used instead of ADD in Dockerfile
[INFO]      * ADD in image history: [sample1-nginx:latest sample1-ubuntu:latest]
[INFO]      * ADD in image history: [sample1-nginx:latest sample1-ubuntu:latest]
[INFO]      * ADD in image history: [xavifg/tfm-sample1-nginx:latest]
[INFO]      * ADD in image history: [nginx:latest]
[INFO]      * ADD in image history: [ubuntu:latest]
[NOTE] 4.10 - Ensure secrets are not stored in Dockerfiles
[NOTE] 4.11 - Ensure verified packages are only Installed


[INFO] 5  - Container Runtime
[INFO]      * No containers running, skipping Section 5
```

ILUSTRACIÓN 22: RESULTADO INFORME DOCKER BENCH SECURITY (PARTE 2)

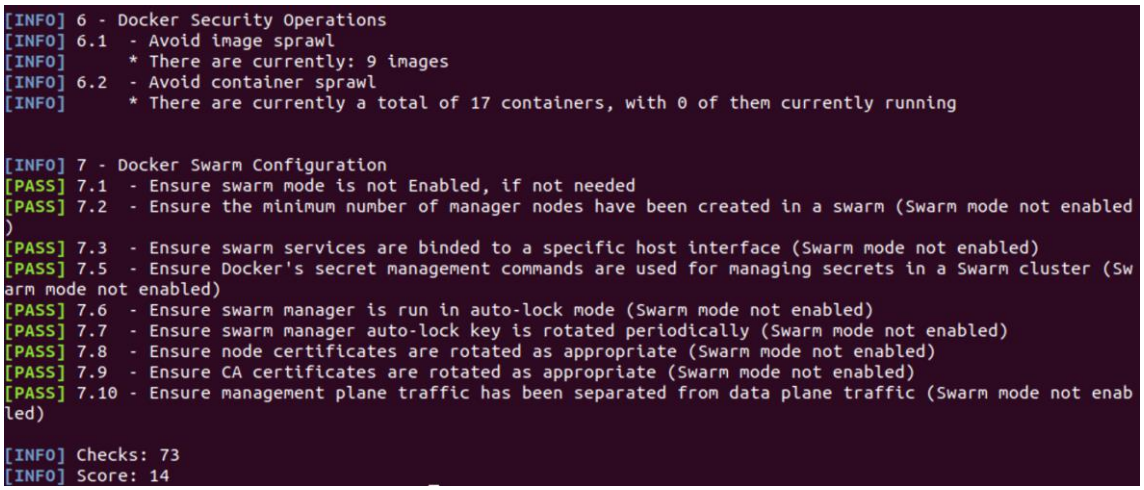

ILUSTRACIÓN 23: RESULTADO INFORME DOCKER BENCH SECURITY (PARTE 3)

4.2. Análisis Estático de Vulnerabilidades

El análisis estático de vulnerabilidades, normalmente se lleva a cabo por herramientas especializadas para esta función y se realiza de forma automática, en donde se analiza la imagen creada con su código fuente, de ahí el nombre de estático, dado que se analiza el contenido de esta antes de su despliegue a producción.

Esta es una etapa que actualmente, dentro de la cultura de DevOps, se está llevando a cabo cada vez más, ya que se ha integrado en todo el ciclo de vida del proyecto para poder aportar cada vez más información antes de que un producto salga a producción.

Esta etapa, si se entra en el mundo Docker en donde normalmente van de la mano con la cultura de DevOps, el análisis de vulnerabilidades de imágenes de Docker recae en la primera etapa de construcción de la solución. Este análisis sería el único análisis estático que caería dentro de las etapas incluidas en los *pipelines* de entrega y despliegue continuo y surge como nuevo requisito a nivel de seguridad debido a la generación de imágenes docker como solución de empaquetado para poder gestionar a posteriori la ejecución del aplicativo contenido en dichas imágenes como contenedores en los diferentes entornos.

Mencionar que aunque a nivel de Dockerfile también se pueden realizar buenas prácticas, como se ha visto en puntos anteriores, no es hasta el momento de la construcción de la imagen cuando se podría verificar que binarios y librerías se han incluido (siempre se puede forzar a la última versión donde en principio, debería estar parcheada a nivel de seguridad a diferencia de las obsoletas) en la propia imagen. Además, es la propiedad de inmutabilidad de las imágenes docker la que hace tan importante el análisis estático de vulnerabilidades sobre una imagen construida, ya que dicha imagen, tal y como se ha mencionado anteriormente, no podrá ser modificada a posteriori, sería necesario volver a modificar la imagen mediante fichero Dockerfile y volver a construir la imagen.

A continuación, se realizará una revisión de las diferentes herramientas o soluciones Open Source y con licencia gratuita para realizar de forma automática el análisis estático de vulnerabilidades.

4.2.1. Docker

En primer lugar, introduciremos una funcionalidad que dispone de Docker en su repositorio de imágenes. Docker facilita una funcionalidad dentro de Docker Hub y Docker Store en donde analiza todas aquellas imágenes que se han subido a los repositorios.

El usuario en cuestión, no tiene que realizar ninguna acción, Docker de forma periódica analiza todas las imágenes subidas en Docker Hub, y una vez finalizado el análisis, nos proporciona un resultado de las diferentes vulnerabilidades que han aparecido en esa imagen para cada uno de los componentes, así como el nivel de criticidad de éstas, además de informar del CVE de cada una de ellas.

Mencionar que durante la realización de este estudio, se realizaban escaneos en repositorios oficiales, como puede ser el repositorio oficial de Ubuntu, además de escaneos en repositorios privados o no oficiales. Desde el día 31 de Marzo de 2018, únicamente realiza escaneos estáticos de las imágenes a repositorios oficiales, siendo así, una funcionalidad perfecta si un usuario está creando su imagen a partir de una

oficial, en caso contrario, está funcionalidad no servirá de mucho para la gran mayoría de usuarios.

A continuación se indicará como se pueden consultar los diferentes resultados del escaneo automático de los diferentes repositorios oficiales. En nuestro caso, se ha seleccionado realizar una revisión al repositorio oficial de Ubuntu.

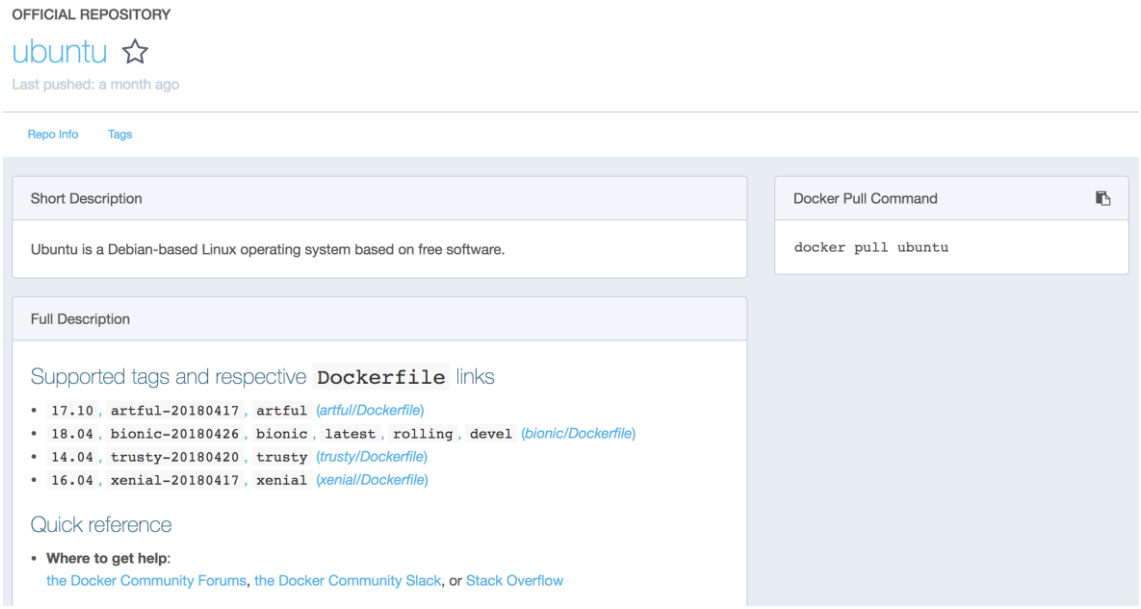

ILUSTRACIÓN 24: REPOSITORIO OFICIAL UBUNTU EN DOCKER HUB

A continuación, para acceder en el apartado de las revisiones de seguridad, se puede encontrar en la pestaña *Tags*, donde se puede observar el análisis realizado a cada una de las versiones que existen actualmente en el repositorio.

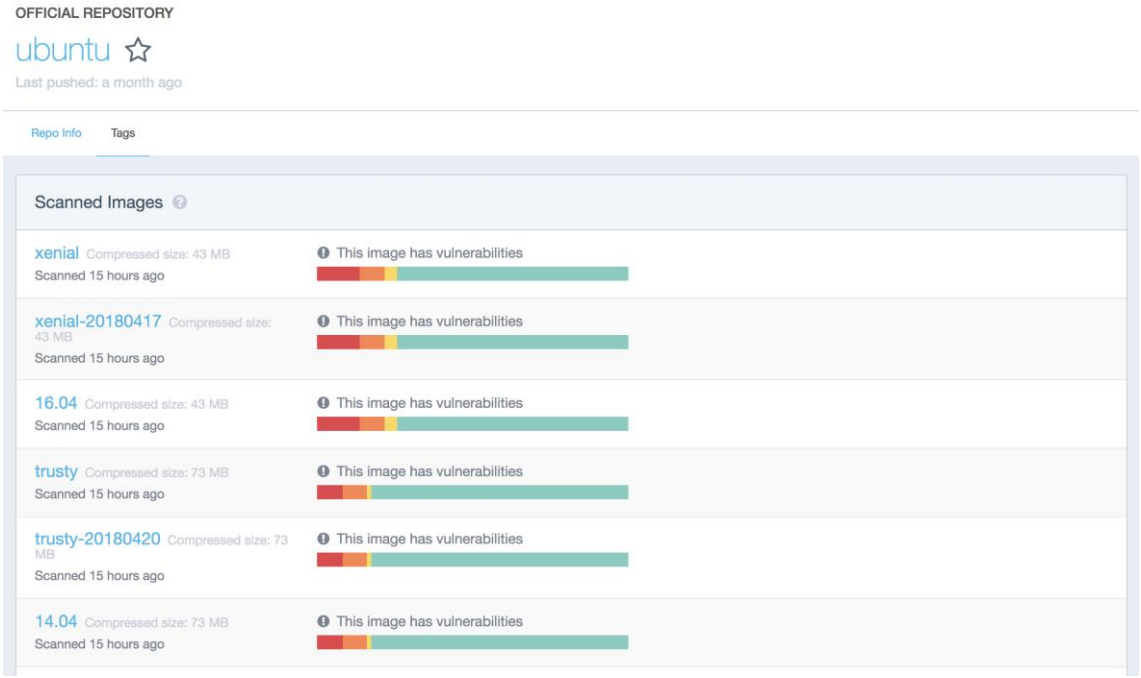

ILUSTRACIÓN 25: ANÁLISIS ESTÁTICO DE IMÁGENES DE UBUNTU

Se observa que el análisis para la imagen (o *tag)* de Xenial se ha realizado hace 15 horas, donde ha encontrado vulnerabilidades. Si se accede al *tag* en cuestión, docker nos proporciona toda la información necesaria del resultado.

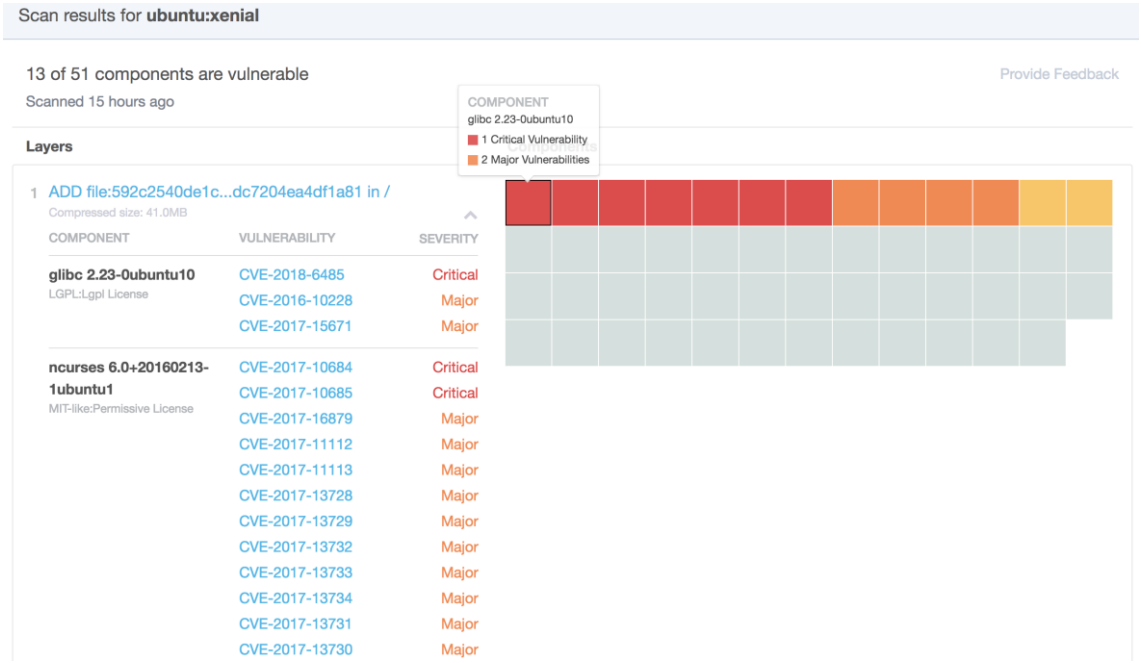

ILUSTRACIÓN 26: RESULTADO ANÁLISIS DE VULNERABILIDADES EN UBUNTU

Por tanto, se dispone de toda la información relativa a las vulnerabilidades encontradas en esa imagen. Los siguientes pasos serían identificar falsos positivos, si realmente se está expuesto a esa vulnerabilidad, y en caso que sí, corregirla de forma inmediata antes de realizar el despliegue del contenedor.

Como conclusión, es una buena herramienta que proporciona Docker para conocer un poco el estado de salud en cuanto a seguridad de las imágenes oficiales y disponer de conocimiento si se van aplicando cambios y parcheando esas vulnerabilidades.

4.2.2. Anchore.io

Anchore es una solución Open Source, disponible en versión on-premise o en versión de servicio proporcionado por terceros, donde su principal objetivo es el descubrimiento de imágenes docker en repositorios tanto públicos como privados, para realizar un análisis estático de vulnerabilidades conocidas.

Esta solución permite obtener el listado de paquetes instalados en el sistema operativo de la imagen docker, así como el listado completo de todos los ficheros y dependencias de Node.js y Ruby incluidas en dicha imagen docker. Es el conjunto de paquetes instalados a nivel de sistema operativo (imagen base) de la imagen docker sobre los que se realiza el análisis estático de vulnerabilidades conocidas.

Para la realización de los diferentes análisis, es necesario crearse una cuenta en Anchore.io, donde se accede a la aplicación por la URL https://anchore.io. En nuestro caso, se han realizado únicamente pruebas con la versión on-premise.

En primera instancia, se ha procedido a la creación de una imagen a partir del fichero Dockerfile, pero se ha forzado a que la versión de la imagen base sea obsoleta, y por otro lado, la misma imagen pero con el tag de *latest*. De esa forma se puede verificar si realmente el aplicativo Anchore está realizando correctamente el análisis estático. En un principio, con la imagen obsoleta, deberían existir muchas más vulnerabilidades que no a la que actualmente es la última versión de la imagen en cuestión.

El primer paso, es la creación de los diferentes Dockerfile. Se ha escogido la imagen base de Nginx para la creación de la imagen. Para la imagen *latest* de Nginx, se ha decidido realizar un *pull* de la imagen oficial, cambiar el tag, y subir esta imagen en el repositorio personal con el nombre de *test-tfm-nginx*, tal y como se puede apreciar en la siguiente imagen.

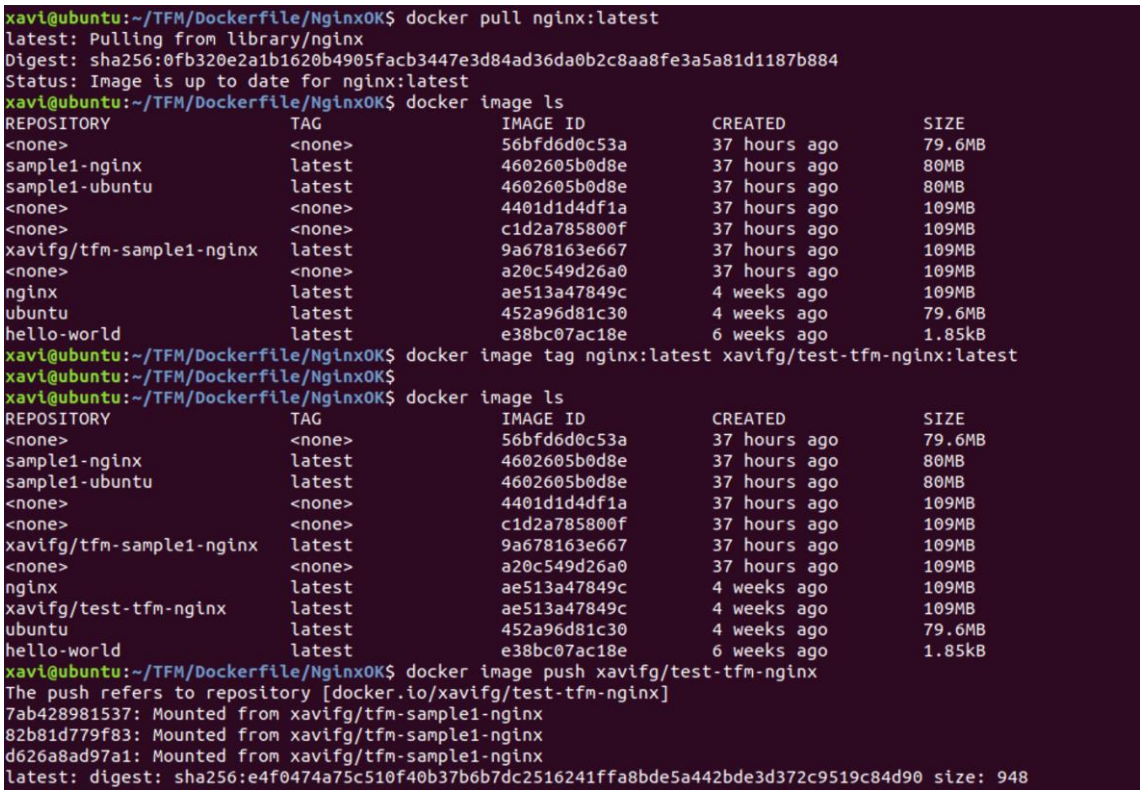

ILUSTRACIÓN 27: PULL Y RENOMBRAR IMAGEN OFICIAL NGINX

Verificando la imagen con tag *latest* en el repositorio oficial de Nginx, se aprecia como presenta algunas vulnerabilidades a partir del informe de Docker.

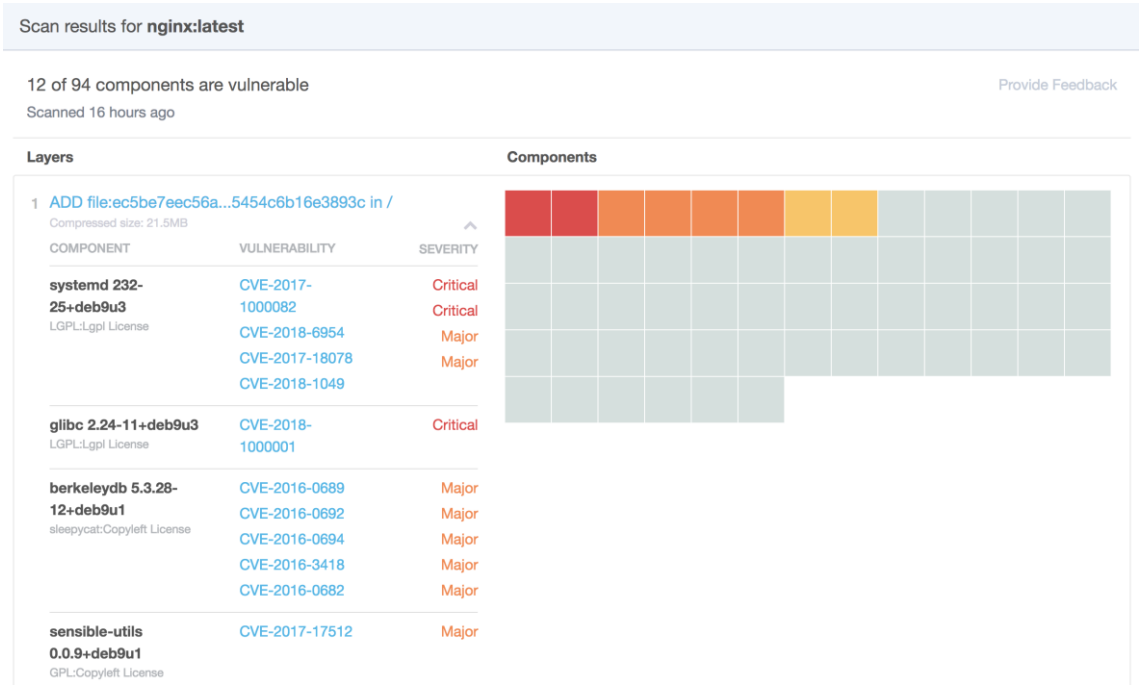

ILUSTRACIÓN 28: INFORME VULNERABILIDADES DE NGINX:LATEST

A continuación, enviamos a escanear la imagen de nuestro repositorio, para verificar que realmente se descubren las mismas vulnerabilidades.

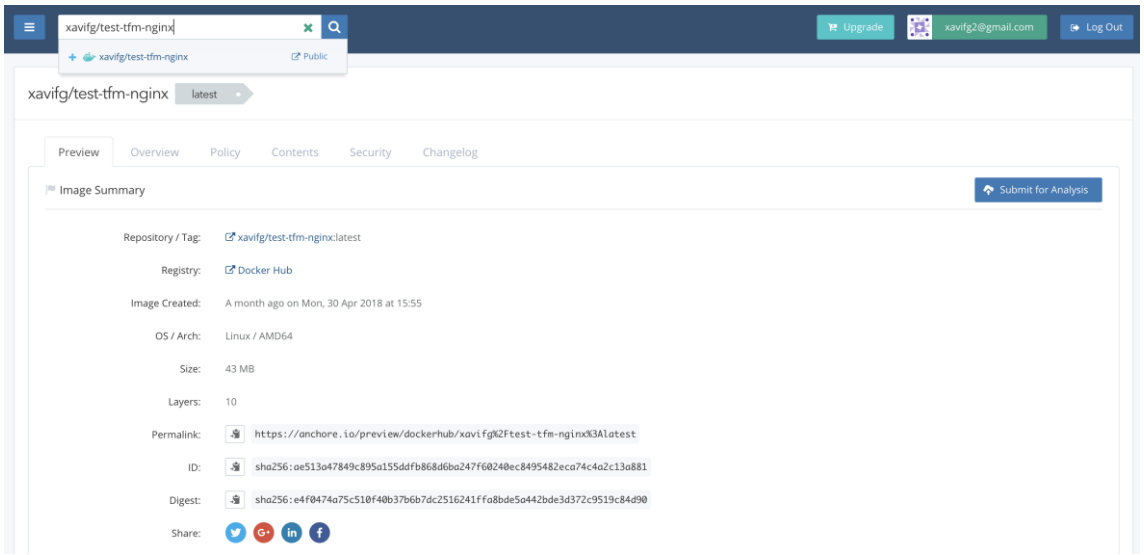

ILUSTRACIÓN 29: INFORMACIÓN IMAGEN XAVIFG/TEST-TFM-NGINX EN ANCHORE.IO

Una vez enviada a análisis, accedemos al espacio personal donde se encuentran todas las imágenes escaneadas, o en curso.

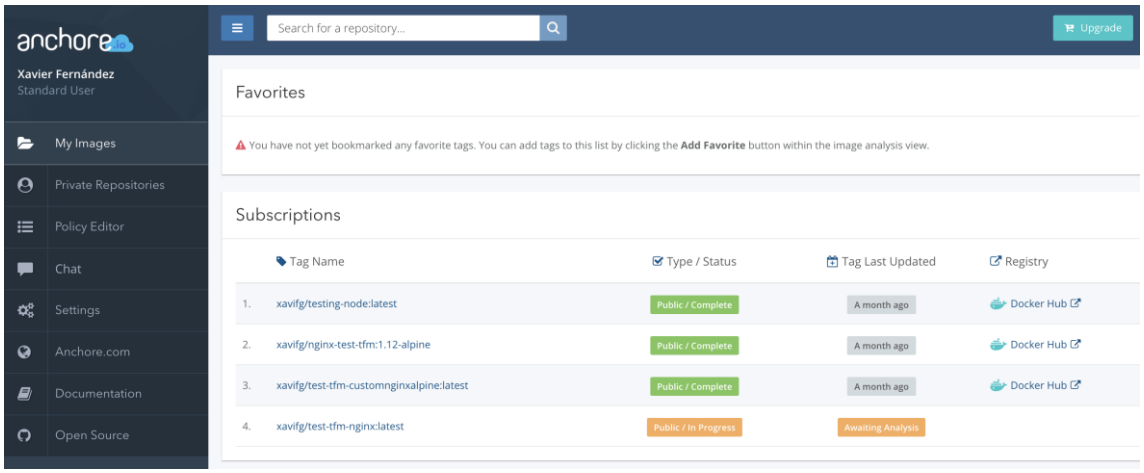

ILUSTRACIÓN 30: LISTADO DE MIS IMÁGENES EN ANCHORE.IO (PARTE 1)

En paralelo se ha procedido a generar una imagen mediante Dockerfile, en el que se intenta forzar que la versión se encuentre obsoleta. En el momento de la prueba, la última versión de Nginx era la 1.14. Con nuestra prueba, se fuerza a que se construya una imagen con la 1.5, con lo que, los propios binarios de la versión 1.5, deberían de tener algún tipo de vulnerabilidad, corregido en posteriores versiones.

ILUSTRACIÓN 31: DOCKERFILE NGINX OBSOLETO

```
xavi@ubuntu:~/TFM/Dockerfile/NginxNOK$ docker build -t xavifg/test-tfm-nginx-nok .
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM debian:stretch-slim
 ---> 26f0bb790e25
Step 2/4 : ENV NGINX_VERSION 1.5
 ---> Using cache
 ---> 362f44cf2f63
Step 3/4 : ENV NJS_VERSION   1.5
 ---> Using cache
 ---> 07fd6d14b70f
Step 4/4 : CMD ["/bin/sh"]
 ---> Running in 755bbd990583
Removing intermediate container 755bbd990583
 ---> 3708df6d9118
Successfully built 3708df6d9118
Successfully tagged xavifg/test-tfm-nginx-nok:latest
xavi@ubuntu:~/TFM/Dockerfile/NginxNOK$
xavi@ubuntu:~/TFM/Dockerfile/NginxNOK$ docker image ls
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
xavifg/test-tfm-nginx-nok   latest              3708df6d9118        13 seconds ago      55.3MB
<none>                      <none>              56bfd6d0c53a        38 hours ago        79.6MB
sample1-ubuntu              latest              4602605b0d8e        38 hours ago        80MB
sample1-nginx               latest              4602605b0d8e        38 hours ago        80MB
<none>                      <none>              4401d1d4df1a        38 hours ago        109MB
<none>                      <none>              c1d2a785800f        38 hours ago        109MB
xavifg/tfm-sample1-nginx    latest              9a678163e667        39 hours ago        109MB
<none>                      <none>              a20c549d26a0        39 hours ago        109MB
nginx                       latest              ae513a47849c        4 weeks ago         109MB
xavifg/test-tfm-nginx       latest              ae513a47849c        4 weeks ago         109MB
debian                      stretch-slim        26f0bb790e25        4 weeks ago         55.3MB
ubuntu                      latest              452a96d81c30        4 weeks ago         79.6MB
hello-world                 latest              e38bc07ac18e        6 weeks ago         1.85kB
xavi@ubuntu:~/TFM/Dockerfile/NginxNOK$
xavi@ubuntu:~/TFM/Dockerfile/NginxNOK$ docker image push xavifg/test-tfm-nginx-nok
The push refers to repository [docker.io/xavifg/test-tfm-nginx-nok]
d626a8ad97a1: Mounted from xavifg/test-tfm-nginx
latest: digest: sha256:885cb5ef5bbccc1aa42fb22ddf95302c7dc0974688abe9e5c6ca0834f34242d3 size: 529
xavi@ubuntu:~/TFM/Dockerfile/NginxNOK$
```

ILUSTRACIÓN 32: CREACIÓN Y PUSH IMAGEN OBSOLETA NGINX

Por último se ha procedido a analizar la imagen creada en Anchore.io. Como se puede apreciar a continuación, se observan las dos imágenes pendientes del resultado del análisis.

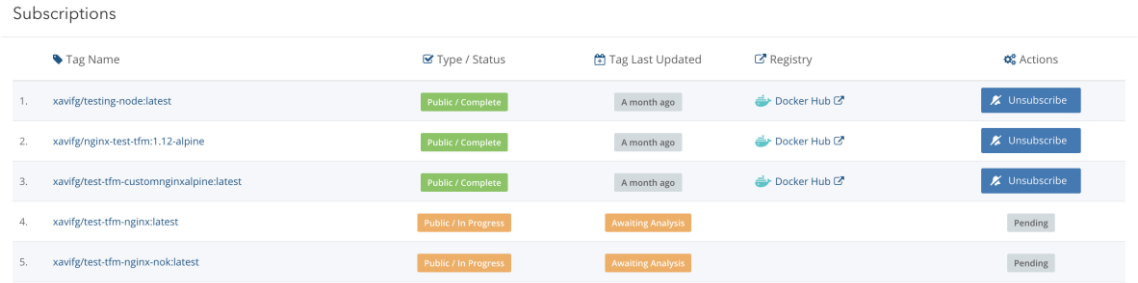

ILUSTRACIÓN 33: LISTADO DE MIS IMÁGENES EN ANCHORE.IO (PARTE 2)

Una vez finalizados los análisis de Anchore.io, podemos acceder a cada uno de ellos, y ver el resultado final de cada análisis, y de esa forma, aplicar posibles soluciones a las diferentes vulnerabilidades, en caso que existan, de esa forma poder mitigar posibles riesgos de seguridad.

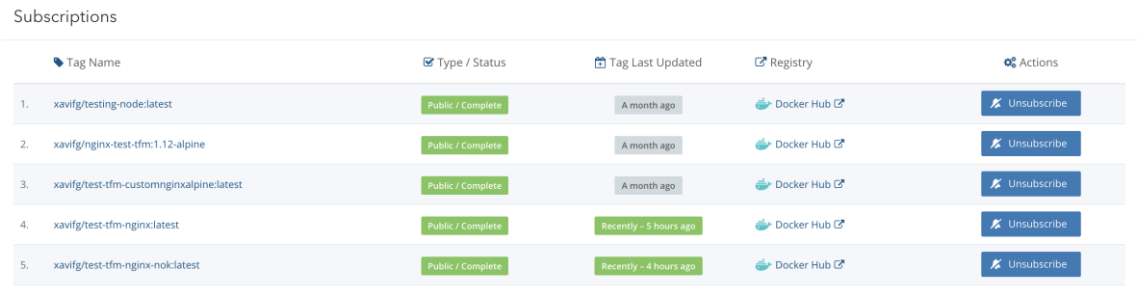

ILUSTRACIÓN 34: LISTADO DE MIS IMÁGENES FINALIZADAS EN ANCHORE.IO

Si accedemos a cada una de ellas, nos encontramos la siguiente información en la pestaña de seguridad.

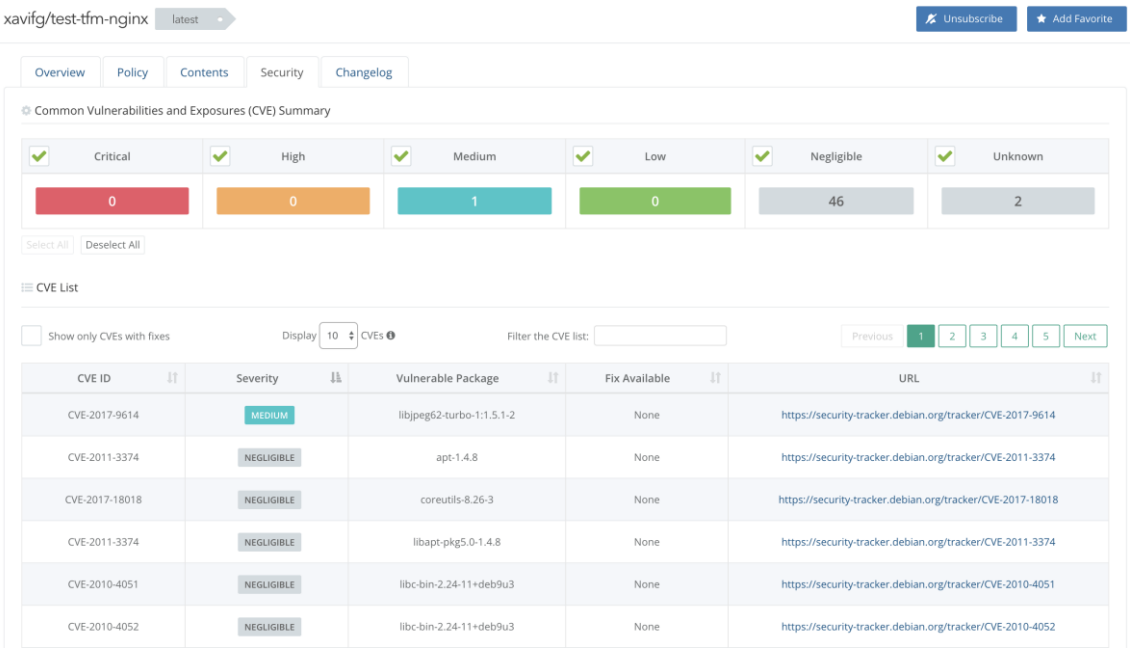

ILUSTRACIÓN 35: INFORME ANÁLISIS DE SEGURIDAD NGINX (NO OBSOLETO)

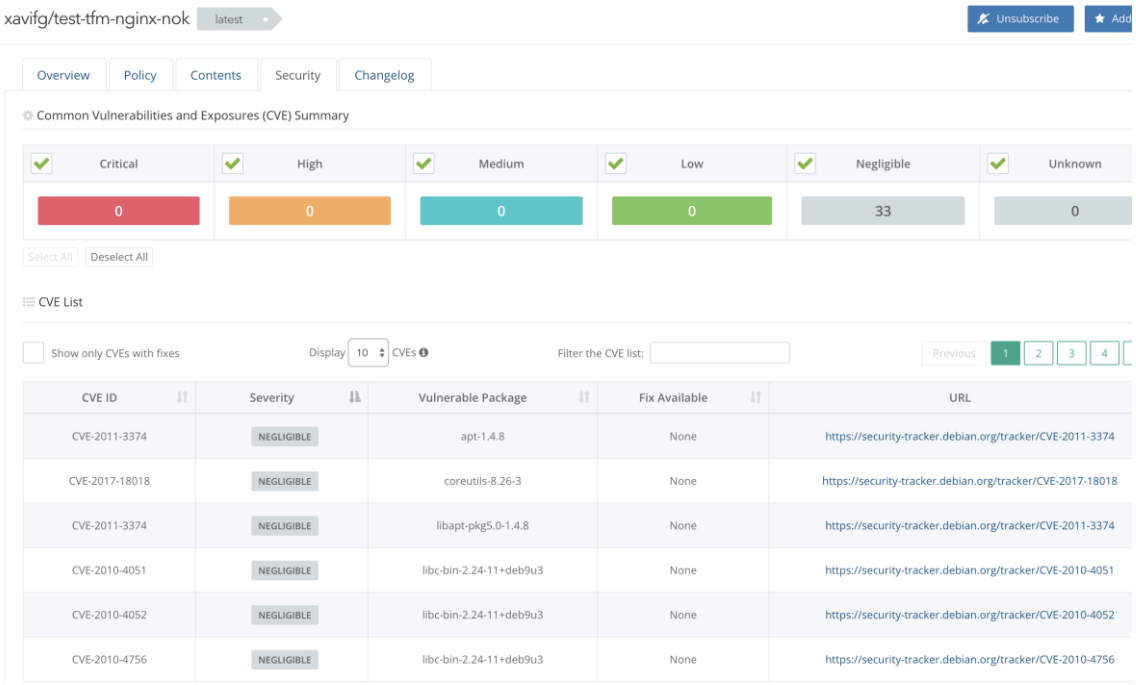

ILUSTRACIÓN 36: INFORME ANÁLISIS DE SEGURIDAD NGINX (OBSOLETO)

4.3. Generación y Persistencia de Logs

Dentro del mundo de la seguridad, uno de los conceptos más importantes es el de poder monitorizar en tiempo real que es lo que está pasando en los sistemas, o si ese sistema ha sido comprometido, poder realizar un análisis *post mortem* en el que se puedan extraer tota la información necesaria, como podría ser puerta de entrada, vector de ataque, etc., y de esa forma, poder adoptar medidas necesarias para prevenir cualquier tipo de ataque con las mismas características.

Docker ofrece funcionalidades nativas para obtener todas las trazas de los diferentes contenedores, y permite diferentes opciones, tal y como se ha podido evidenciar en el punto *4.1.2.3. Gestión de Logs*, donde se puede configurar la salida con el formato de *logs* o *trazas* que necesita para poder consumirlos y/o analizarlos.

A continuación se puede observar, que por defecto, los logs de docker, capturan todas las acciones que realiza en contenedor en cuestión. El problema viene relacionado con el tipo de comandos, dado que únicamente se puede realizar la búsqueda contenedor por contenedor. Además, no se dispone de opción de manipular esos logs mediante comandos dado que los comandos son los *nativos* de docker.

En primera instancia, arrancamos un contenedor con la imagen oficial de Ubuntu, donde especificamos que queremos interacción con el contenedor mediante el *flag* -i.

```
xavi@ubuntu:~$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
Digest: sha256:c8c275751219dadad8fa56b3ac41ca6cb22219ff117ca98fe82b42f24e1ba64e
Status: Image is up to date for ubuntu:latest
xavi@ubuntu:~$
xavi@ubuntu:~$ docker run -ti ubuntu
root@dd091cc40ed3:/#
root@dd091cc40ed3:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@dd091cc40ed3:/# cd home/
root@dd091cc40ed3:/home# ls
root@dd091cc40ed3:/home#
root@dd091cc40ed3:/home# cd ../dev/
root@dd091cc40ed3:/dev# ls
console  fd     mqueue  ptmx  random  stderr  stdout  urandom
core     full   null    pts   shm     stdin   tty     zero
root@dd091cc40ed3:/dev#
```

ILUSTRACIÓN 37: EJECUCIÓN CONTENEDOR UBUNTU

A continuación procedemos a revisar los logs generados por este contenedor mediante el comando *docker logs* y el identificador del contenedor generado cuando se ha realizado la puesta en marcha.

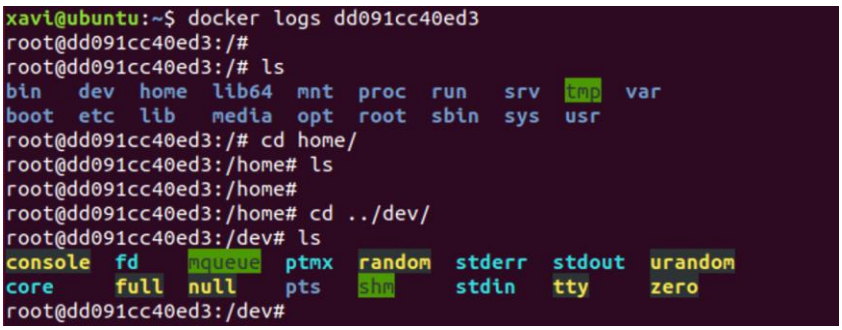

ILUSTRACIÓN 38: LOGS CONTENEDOR UBUNTU

Se observa que está reproduciendo todas las acciones que ha realizado el usuario, pero a nivel de seguridad, no se pueden explotar estos datos, dado que no sirve ver únicamente lo que ha realizado el usuario.

Para la realización de la siguiente prueba se ha decidido, dado que con la prueba anterior la información obtenida no nos sirve, activar en el demonio de docker el tipo de trazas *syslog*.

Ha sido necesario crear el fichero *daemon.json* en el directorio de docker */etc/docker*. Una vez creado el fichero, es necesario especificar el tipo de log de salida que se quiere, en nuestro caso, se ha definido el *syslog*.

ILUSTRACIÓN 39: CONTENIDO FICHERO CONFIGURACIÓN DAEMON DE DOCKER

Seguidamente, para que propague los cambios, es necesario reiniciar el servicio de Docker, con la siguiente instrucción, y una vez se haya vuelto a iniciar, ya dispondremos de los logs en *syslog*.

```
$ sudo service docker restart
```

La prueba que se realizará ahora es la de lanzar un contenedor con un servidor web, en este caso será Nginx, en el que se publicara una página web, y posteriormente nos conectaremos para evidenciar que realmente se está monitorizando cualquier acción que se realice en el servidor.

En primera instancia, se crea el servidor Nginx, mediante el fichero Dockerfile, donde principalmente lo que se hace es exponer el puerto 80 para que nos podamos contactar vía explorador, y una vez creado, se inspecciona el fichero *syslog*, que es donde estamos monitorizando cualquier acción.

```
xavi@ubuntu:~/TFM/Log/Dockerfile$ docker run -p 80:80 nginx-index
```

ILUSTRACIÓN 40: EJECUCIÓN CONTENEDOR SERVIDOR WEB (NGINX)

A continuación, procedemos a navegar a la página que estamos publicando y a su vez, también haremos una prueba de ejecutar el comando *curl* para evidenciar que cualquier acción queda registrada.

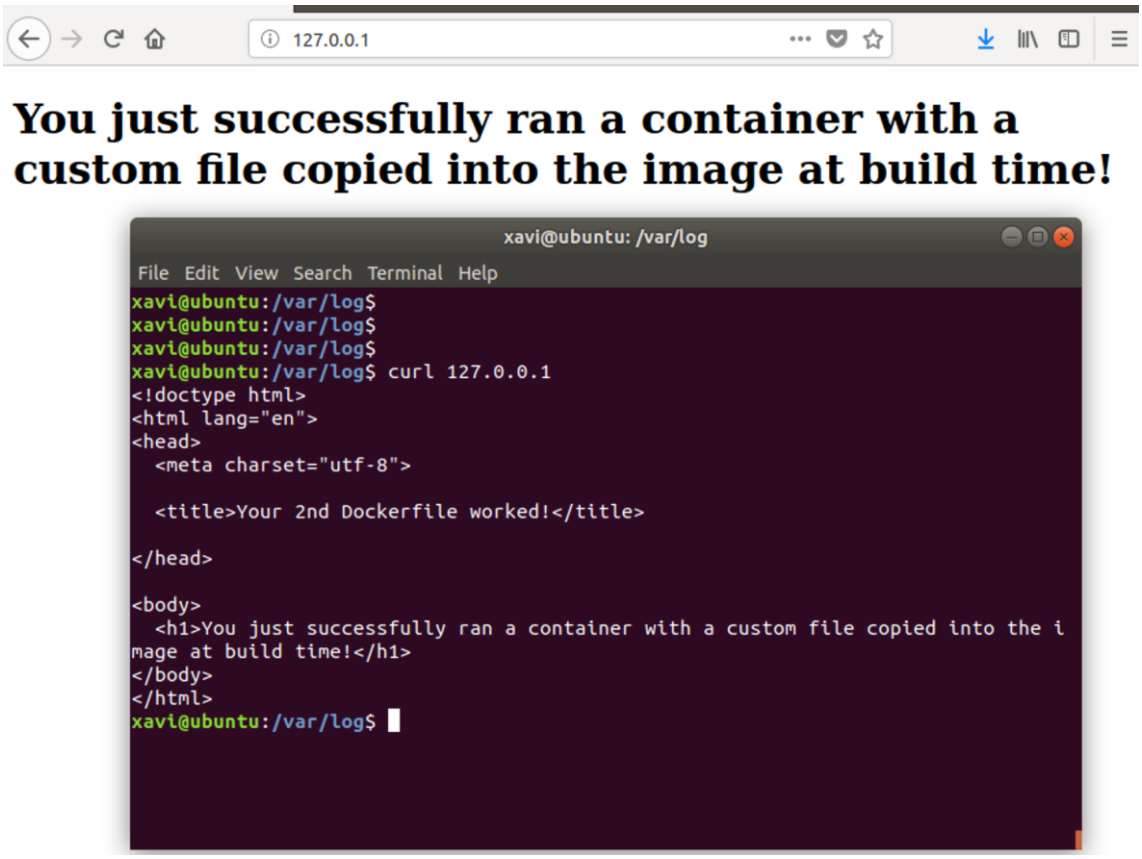

ILUSTRACIÓN 41: CONEXIÓN SERVIDOR WEB (NGINX)

Por último, procedemos a revisar el log *syslog* para evidenciar que ha quedado registrado.

```
Jun  1 19:19:15 ubuntu cc8b0a0c58f0[15107]: 172.17.0.1 - - [01/Jun/2018:17:19:15
 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:6
0.0) Gecko/20100101 Firefox/60.0" "-"
Jun  1 19:19:19 ubuntu cc8b0a0c58f0[15107]: 172.17.0.1 - - [01/Jun/2018:17:19:19
 +0000] "GET / HTTP/1.1" 200 249 "-" "curl/7.58.0" "-"
```

ILUSTRACIÓN 42: LOGS CONEXIÓN SERVIDOR WEB (NGINX)

A continuación, procedemos a parar el contenedor con el servidor web (Nginx) para evidenciar que disponemos de los logs aunque el servicio no este activado, de esa forma estamos cumpliendo con la persistencia de los logs, aunque no dispongamos del servicio.

ILUSTRACIÓN 43: PERSISTENCIA LOGS CONEXIÓN SERVIDOR WEB (NGINX)

Uno de los problemas existentes con este método usado, es que la salida de todos los contenedores, siempre se produce en el mismo destino, por tanto, siempre se produce al *syslog.* Únicamente se dispone del identificador de los contenedores para poder filtrar y ver las trazas específicas, tal y como se puede observar en la Ilustración anterior.

5. Conclusiones

Durante la realización de este proyecto, se había propuesto como objetivo el estudio de las diferentes medidas de seguridad aplicables a la tecnología de contenedores de Docker y poder proporcionar así una capa de seguridad en cada una de las partes que conforman el universo Docker.

Este objetivo se ha conseguido, ya que se han podido proporcionar buenas prácticas de seguridad para cada una de las partes de Docker, en su fase de instalación, configuración de los contenedores, diseño de las imágenes y en su última fase, una vez estos se encuentran desplegados.

Mencionar que tiempo atrás, muchas empresas al hacer el cambio a la nueva infraestructura de micro servicios, dejaron un poco de lado la seguridad, priorizando grandes temas como rendimiento, reducción de costes, versatilidad de la infraestructura, etc. Pero a día de hoy, donde cada vez las empresas se encuentran en el punto de mira de los diferentes atacantes, la seguridad es una prioridad para todas las empresas y se encuentra entre los grandes temas en los que se tiene que poner énfasis.

Como resultado final del proyecto, se ha evidenciado la gran cantidad de procesos o soluciones aplicables dentro del universo Docker, para que cualquier usuario administrador de los sistemas de la empresa, pueda aplicar cualquier de estas técnicas dentro del proceso de definición de la infraestructura así como definición de la arquitectura. Como aplicar buenas prácticas en la fase inicial, cuando se está realizando la elección y configuración del *Host*, la configuración de los diferentes procesos que conforman Docker antes de la puesta en producción, y una vez los contenedores se encuentran desplegados, poder revisar si los procesos instalados son vulnerables, y poder monitorizar en tiempo real si los contenedores sufren cualquier tipo de ataque, o simplemente, en caso necesario de tener cerrar uno de ellos por incidentes de seguridad, disponer de toda información necesaria vía trazas, para poder realizar un análisis *post mortem* del contenedor en cuestión, y poder aplicar las medidas de seguridad necesarias para evitar cualquier otro incidente.

Finalmente, este proyecto ha sido una experiencia muy buena debido a que en muchas de las buenas prácticas definidas en este proyecto pueden ser extrapoladas en cualquier otra situación, no únicamente en un entorno Docker. También me ha permitido adquirir conocimientos, no únicamente sobre Docker, donde era uno de mis principales objetivos, sino que también diferentes buenas prácticas de seguridad en este mundo. Añadir que he descubierto un mundo en el que hacía tiempo que me encontraba en la entrada, pero a día de hoy, después de la realización de este proyecto, he conseguido entrar en la inmensidad de este nuevo movimiento de infraestructura de micro servicios que se está forjando por méritos propios en todo el mundo. Existen conferencias de expertos sobre tecnologías en *Cloud Público* de cómo están usando Docker para poder modificar toda su infraestructura, o Kubernetes para poder orquestar todo el mundo de contenedores, etc, como pueden realizar un auto escalado de su infraestructura automáticamente en momentos oportunos, y como poder diseñar una arquitectura segura y resistente con la unión de todas estas tecnologías.

6. Glosario

Los acrónimos técnicas que se encuentran en este documento pertenecen a las siguientes palabras:

LXC: Linux Containers

CLI: Command Line Interface

NAT: Network address translation

CEO: Chief Executive Officer

VXLAN: Virtual Extensible Local Area Network

API: Application Programming Interface

HTTP: Hypertext Transfer Protocol

HTTPS: Hypertext Transfer Protocol Secure

CPU: Central Processing Unit

RAM: Random Access Memory

TCP: Transmission Control Protocol

IP: Internet Protocol

OpenSource: Software libre

Tags: Etiqueta

URL: Uniform Resource Locator

SSH: Secure Shell

7. Bibliografía

[1]. Docker Documentation, https://docs.docker.com

[2]. Docker Repository, https://hub.docker.com

[3]. Docker Store, https://store.docker.com

[4]. Docker Blog, https://blog.docker.com/author/diogo

[5]. Docker in Production, https://scaledocker.com

[6]. Securing Docker Containers, http://blog.wercker.com/securing-docker-containers

[7]. SANS Institute, A Checklist for Audit of Docker Containers, https://www.sans.org/reading-room/whitepapers/auditing/checklist-audit-docker-containers-37437

[8]. Fran Ramírez, Elías Grande, Rafael Troncoso. Docker: SecDevOps. 1ª Edición, Editorial 0xWord, Móstoles (Madrid), 2018

[9]. Docker Bench Security, https://github.com/docker/docker-bench-security

[10]. Anchore.io, https://anchore.io

[11]. AppArmor, http://wiki.apparmor.net/index.php

[12]. SELinux, https://github.com/SELinuxProject

[13]. Stackoverflow, https://stackoverflow.com

[14]. Ask Ubuntu, https://askubuntu.com

[15]. Github, https://github.com

[16]. Kubernetes, https://kubernetes.io/docs/concepts/configuration/secret

[17]. Server Fault, https://serverfault.com

[18]. CVE – Common Vulnerabilities and Exposures, https://cve.mitre.org/

[19]. Nginx, https://nginx.com/

[20]. DevOps, https://devops.com/

[21]. Atlassian, https://es.atlassian.com/devops