



IES Alixar

CURSO DE ESPECIALIZACIÓN EN CIBERSEGURIDAD

Docker Seguro

PROFESOR

MARIO ALFONSO LASSO MESA

PUESTA EN PRODUCCIÓN SEGURA

ALUMNO AUTOR

JOSÉ BENÍTEZ ROMERO

Año ACADÉMICO

2024-2025

“SI ES INTELIGENTE, ES VULNERABLE.”
— MYKKO HYPPONEN

Índice general

LISTADO DE CAPTURAS	VIII
LISTADO DE ACRÓNIMOS	XIII
I INTRODUCCIÓN A DOCKER	I
I.O.1 Comprobación de la instalación de Docker	I
I.O.2 Ejecución de un contenedor a partir de la imagen <code>hello-world</code> . . .	I
I.O.3 Ejecución y eliminación de un contenedor temporal	2
I.O.4 Comprobación del estado de los contenedores y eliminación de con- tenedores detenidos	2
I.O.5 Creación de un contenedor interactivo desde una imagen Debian . .	3
I.O.6 Conclusiones de esta sección	5
I.1 Creación de un contenedor con un servidor web	5
I.1.1 Creación del contenedor en segundo plano	5
I.1.2 Visualización de los logs del servidor web	6
I.1.3 Modificación de los archivos en el contenedor	6
I.1.4 Conclusiones de esta sección	8
I.2 Gestión de Imágenes Docker	8
I.2.1 Descarga de Imágenes	8
I.2.2 Creación de un Contenedor en Segundo Plano	9
I.2.3 Copia de Archivos a un Contenedor	9
I.2.4 Verificación del Espacio Ocupado	10
I.2.5 Inspección de la Imagen	10
I.2.6 Eliminación de Imágenes	11
I.2.7 Creación de Contenedores de MediaWiki	11
I.2.8 Conclusiones de esta sección	12
I.3 Creación y Uso de Volúmenes	12
I.3.1 Creación de Volúmenes Docker	12
I.3.2 Creación de Contenedores con Volúmenes Montados	13
I.3.3 Intento de Borrado del Volumen <code>volumen_datos</code>	13
I.3.4 Manipulación de Archivos en el Contenedor <code>c1</code>	14
I.3.5 Recreación del Contenedor <code>c1</code> como <code>c3</code>	14
I.3.6 Uso de <i>Bind Mount</i> para Compartir Datos	14
I.3.7 Modificación del Contenido del Archivo <code>index.html</code>	15
I.3.8 Comprobación de Directorio al Realizar un <i>Bind Mount</i>	16

1.3.9	Conclusiones de esta sección	16
1.4	Demostración (Caso Práctico LXC, usuario jbenrom)	17
2	IMAGENES DOCKER	27
2.1	Imagen personalizada de Docker	27
2.1.1	Preparación de Archivos	27
2.1.2	Construcción y Ejecución del Contenedor	30
2.1.3	Pruebas Avanzadas	30
2.1.4	Conclusión	31
2.2	Implementación de Nginx con Contenido Personalizado	31
2.2.1	Ejecución de la Imagen de Nginx desde DockerHub	31
2.2.2	Personalización del Contenido HTML Usando Bind Mounts	32
2.2.3	Creación de una Imagen Personalizada de Nginx	33
2.2.4	Subida de la Imagen Personalizada a DockerHub	33
2.2.5	Resumen de Resultados Interesantes	34
2.3	Demostración (Caso Práctico LXC, usuario jbenrom)	35
3	DOCKER COMPOSE Y SECRETS	41
3.1	Configuración de WordPress con MySQL	41
3.1.1	Preparación del proyecto	41
3.1.2	Definición del archivo <code>docker-compose.yml</code>	42
3.1.3	Ejecución y resultados	43
3.2	Despliegue de Redmine con <i>Secrets</i>	43
3.2.1	Configuración inicial	43
3.2.2	Definición de servicios y secretos	44
3.2.3	Ejecución y resultados	46
3.3	Creación de una pila LAMP con phpMyAdmin	47
3.3.1	Definición de servicios	47
3.3.2	Resultados y Consideraciones	48
3.3.3	Conclusiones de esta práctica	49
3.4	Demostración (Caso Práctico LXC, usuario jbenrom)	51
4	DOCKER SEGURO	59
4.1	Ejercicio de Seguridad con Docker: Asegurar Contenedores	59
4.1.1	Evitar Ataques DoS con Limitaciones de Recursos	59
4.1.2	Ejecutar un Contenedor con un Usuario Distinto de Root	60
4.1.3	Crear un Dockerfile para un Usuario Específico	60
4.1.4	Ejecutar un Contenedor Privilegiado (¡Con Precaución!)	61
4.1.5	Resumen	62
4.2	Imagen Docker Optimizada: Multi-Stage Builds	62
4.2.1	Paso 1: Imagen sin Multi-Stage Builds	62

4.2.2	Imagen con Multi-Stage Builds	64
4.2.3	Resumen	65
4.3	Escaneo de Vulnerabilidades con Trivy	65
4.3.1	Descarga de la Imagen de Trivy	65
4.3.2	Escaneo de las Imágenes <code>helloworld:single</code> y <code>helloworld:multi</code>	65
4.3.3	Escaneo de Diferentes Versiones de una Misma Imagen	66
4.3.4	Escaneo de Imágenes Base Populares	67
4.3.5	Interpretación de Resultados	68
4.4	Firma de Imágenes Docker y Docker Content Trust	68
4.4.1	Configuración Inicial	68
4.4.2	Gestión de Claves de Delegación	69
4.4.3	Firma y Subida de Imágenes Firmadas	70
4.4.4	Verificación de Firmas	70
4.4.5	Activación Global de Docker Content Trust	71
4.4.6	Conclusión	71
4.5	Demostración (Caso Práctico LXC, usuario jbenrom)	72

Índice de figuras

1.1	Respuesta del servidor nginx en el contenedor	17
1.2	Logs del contenedor	18
1.3	Agregamos un index.html al documentroot del contenedor (que esta enlazado a través de un bind mount)	18
1.4	Ahora se muestra el contenido de index.html	19
1.5	Copiamos un archivo previamente creado dentro de un contenedor activo, para mostrar <i>phpinfo()</i>	19
1.6	Salida de <i>phpinfo()</i>	19
1.7	Comprobamos las diferentes capas de una imagen de Docker con <code>docker inspect</code>	20
1.8	Tratamos de borrar una imagen de la cual depende un contenedor, y como vemos en la salida no nos deja	20
1.9	Una vez parado y borrado el contenedor si nos lo permite	20
1.10	Cuando descargamos distintas versiones de una misma imagen de Docker podemos comprobar que hay capas que ya existen de versiones anteriores y omite la descarga	22
1.11	Creamos volúmenes de Docker	23
1.12	Los listamos para comprobar que se han creado	23
1.13	Se lo asignamos a un contenedor	24
1.14	Y lo intentamos borrar mientras este esta en uso para comprobar que no nos lo permite el propio Docker	24
1.15	Una vez parado y borrado el contenedor que lo tenia asignado, si nos permite borrarlo	24
1.16	Entramos dentro del contenedor de apache para generar un archivo que se guardara en el volumen previamente asignado	25
1.17	Borramos el contenedor para poder asignar el volumen a otro contenedor	25
1.18	Al acceder a través del navegador al nuevo contenedor (<i>c3</i>) con el puerto 8081 mapeado, y el volumen_web asignado, vemos que se logrado la persistencia del archivo mediante el uso de volúmenes	25
1.19	También se puede lograr esta persistencia mediante el uso de bind mounts de directorios locales, lo que mapea el directorio local con un directorio del contenedor, pero en este caso habrá que tener en cuenta que el contenedor tenga los permisos necesarios en el directorio local mapeado	26
2.1	Creamos la imagen de nuestro apache personalizado.	35

2.2	También agregamos unas cuantas directivas a nuestro <code>php.conf</code> para hacer esta imagen mas segura, como <code>disable_functions</code> a la que le añadimos mas funciones peligrosas	36
2.3	Y lanzamos un contenedor a partir de esta con un <code>bind mount</code> , nótese que para que el contenedor pudiera escribir en <code>counter.txt</code> tuvimos que darle permisos <code>www-data</code>	36
2.4	Observamos que al visitar la dirección del contenedor, nuestro contador se va aumentando en <code>counter.txt</code>	37
2.5	Después descargamos y montamos un contenedor de <code>nginx</code> , con un <code>bind mount</code> a <code>/usr/share/nginx/html</code> para agregarle nuestra pagina de bienvenida . .	37
2.6	Pero para que este cambio sea persistente en la imagen, procedimos a crear nuestra propia imagen a partir de la de <code>nginx</code> en la que ya se incluyera nuestra pagina de bienvenida	38
2.7	Para poder subir nuestra imagen a DockerHub, nos logueamos en nuestra maquina, como <i>nanashi</i> (esto es porque ya que nos creamos una cuenta la creamos con un nombre de usuario que querramos usar en el futuro)	39
2.8	Para poder subir la imagen a DockerHub tuvimos que cambiarle el nombre ya que la habíamos creado anteriormente para el repositorio de el usuario <i>jbenrom</i>	40
3.1	Se crearon distintos Docker Compose, los cuales se utilizan para la gestión de aplicaciones multicontenedor, que por cierto Mario la clave <i>version</i> ya no se utiliza	51
3.2	Con <code>docker compose up -d</code> se construyen y se lanzan todos los servicios del Docker Compose, y como se puede observar la etiqueta <i>version</i> esta obsoleta	52
3.3	<code>docker compose ps</code> nos muestra los servicios del Docker Compose que están activos	52
3.4	Tambien nos permite el uso de <code>secrets</code> para no hardcodear credenciales u otra información sensible	53
3.5	Para esta Pila LAMP necesitabamos que en la carpeta del anfitrión mapeada con el <code>bind mount</code> , el usuario de <code>mariadb</code> de la imagen de <code>bitnami</code> (<code>uid 1001</code> tuviera los permisos necesarios	54
3.6	Agregamos un fichero <code>php</code> que básicamente nos muestra <code>phpinfo()</code> al directorio montado en el contenedor del <code>apache</code>	54
3.7	Al acceder a la dirección del <code>apache</code> observamos que nos muestra la información del entorno de <code>php</code> del contenedor	55
3.8	Para probar los <code>secrets</code> también accedimos a <i>phpmyadmin</i> con las credenciales asignadas en los <i>secrets</i>	56
4.1	Contenedor de <i>debian</i> con recursos limitados para prevenir ataques de denegación de servicios	72

4.2	Podemos ver que cuando accedemos como user, no tenemos permisos de super usuario, ademas me parece curioso que el uid 1001 con el que accedemos nos pone que no tiene nombre	72
4.3	Imagen personalizada a partir de <i>debian</i> a la que le añadimos un usuario y un grupo y hacemos que el contenedor se ejecute con este	72
4.4	Vemos que al entrar en la shell del contenedor y ejecutar <code>whoami</code> nos devuelve el usuario que creamos	73
4.5	También se realizaron pruebas de alcance de un contenedor privilegiado, y podemos ver que puede acceder (e incluso interactuar, aunque esto no se pudo comprobar dado que era un LXC, contenedor dentro de contenedor) al hardware del anfitrión	74
4.6	Se probó la eficiencia de las imágenes multistage de Docker, creando primero una imagen <i>single</i> de Golang, con todo el entorno de desarrollo para compilar, y después una multistage.	75
4.7	En la imagen multistage se utilizó la imagen de Golang para compilar nuestro script, y después se usó una imagen base de <i>debian</i> como entorno para el funcionamiento de nuestro servicio de helloworld	77
4.8	Al comparar las dos imágenes observamos que la multistage es mucho mas ligera ya que no incluye todo el entorno de desarrollo de <i>Golang</i>	78
4.9	Se utilizó la herramienta para Docker Trivy, para comprobar las vulnerabilidades clasificadas existentes en las imágenes de Docker	79
4.10	Se comprobó que la imagen single era vulnerable a mas <i>CVEs</i> al contener muchas mas herramientas y paquetes, incumpliendo así el principio de mínima exposición	79
4.11	También se probó con distintas versiones de la misma imagen para comprobar que las mas nuevas estaban expuestas a menos vulnerabilidades conocidas	80
4.12	Se compararon distintas imagenes de SO para así observar que la menos expuesta era la de <i>alpine</i> , al ser el entorno mas ligero y minimalista	81
4.15	Generamos claves y certificados para nuestro usuario de DockerHub	82
4.13	Se procedió a usar Notary para la gestión de certificados y la firma de imágenes propias de Docker	82
4.14	Se activo la directiva de Docker Content Trust para la comprobación de imágenes firmadas	82
4.16	Se firmó la imagen de helloworld:multi	83
4.17	Y se probó a hacer <code>docker pull</code> a una imagen firmada y una no firmada después de haber activado la directiva, el resultado fue un error por parte de Notary de la imagen no firmada	83

Listado de acrónimos

LXC	LinuX Containers
LAMP	Linux, Apache, MySQL, PHP/Perl/Python
CMS	Content Management System
DCT	Docker Content Trust
UID	User Identifier
GID	Group Identifier
CVE	Common Vulnerabilities and Exposures


1

Introducción a Docker

En esta sección se ejecutarán varios comandos básicos de Docker para comprobar el estado de los contenedores y realizar operaciones fundamentales. A continuación, se detallan los pasos a seguir y los resultados obtenidos en cada uno.

1.0.1. COMPROBACIÓN DE LA INSTALACIÓN DE DOCKER

Para verificar que Docker Engine esté correctamente instalado en el sistema Ubuntu, ejecutamos el siguiente comando para comprobar la versión:



```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker --version
```

Este comando mostrará la versión de Docker instalada, confirmando que el servicio está disponible para la creación y manejo de contenedores.

1.0.2. EJECUCIÓN DE UN CONTENEDOR A PARTIR DE LA IMAGEN `hello-world`

Para comprobar el funcionamiento básico de Docker, ejecutaremos un contenedor desde la imagen `hello-world`. Al no estar disponible localmente, Docker descargará la imagen auto-

máticamente desde Docker Hub y luego ejecutará el contenedor. Esto se hace con el siguiente comando:

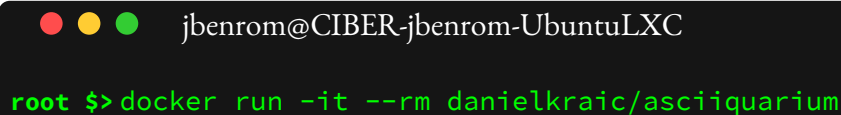


```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run hello-world
```

El resultado esperado es un mensaje de bienvenida de Docker que confirma que el contenedor se ejecutó correctamente. La primera vez que se ejecute, la imagen `hello-world` será descargada desde el repositorio oficial en Docker Hub.

1.0.3. EJECUCIÓN Y ELIMINACIÓN DE UN CONTENEDOR TEMPORAL

Ahora, ejecutaremos un contenedor temporal que se eliminará automáticamente al detenerlo, usando la imagen `danielkraic/asciiquarium`, la cual genera un entorno interactivo con una animación ASCII en la terminal. Esto se logra con el siguiente comando:



```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run -it --rm danielkraic/asciiquarium
```

El parámetro `--rm` asegura que el contenedor se elimine automáticamente una vez detenido. Para salir, presiona `Ctrl+C`. Este comando demuestra cómo un contenedor puede crearse para tareas temporales sin dejar residuos.

1.0.4. COMPROBACIÓN DEL ESTADO DE LOS CONTENEDORES Y ELIMINACIÓN DE CONTENEDORES DETENIDOS

Después de que un contenedor finaliza su tarea, también se detiene. Para verificar que no quedan contenedores en ejecución, utilizamos:



```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker ps
```

Este comando lista los contenedores en ejecución. Para ver todos los contenedores, tanto en ejecución como detenidos, usamos:


```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker ps -a
```

Para eliminar los contenedores detenidos y liberar espacio, se ejecuta el siguiente comando:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker rm <container_id>
```

Donde `<container_id>` es el identificador del contenedor que queremos eliminar. Si hay varios contenedores detenidos, podemos especificar múltiples IDs o usar el comando `docker rm $(docker ps -a -q)` para eliminarlos todos.

1.0.5. CREACIÓN DE UN CONTENEDOR INTERACTIVO DESDE UNA IMAGEN DEBIAN

A continuación, crearemos un contenedor interactivo a partir de la imagen `debian` y dentro de él instalaremos el editor de texto `nano`:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run -it debian
```

Una vez dentro del contenedor, instalamos el paquete `nano`:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> apt update && apt install -y nano
```

Luego, salimos de la terminal del contenedor. En este punto, el contenedor se detendrá porque no hay procesos activos en ejecución. Para verificarlo, ejecutamos:

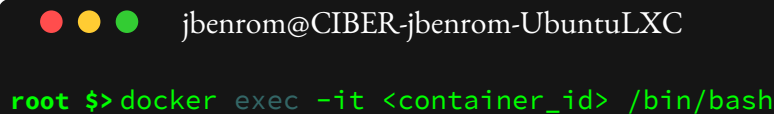
```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker ps
```

Como el contenedor se ha detenido, usamos el comando `docker start` para volver a iniciarlo:



```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker start <container_id>
```

A continuación, accedemos al contenedor en modo interactivo:



```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker exec -it <container_id> /bin/bash
```

Verificamos si nano sigue instalado ejecutando el comando `nano`. Después de confirmar la instalación, salimos del contenedor y lo detenemos:



```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker stop <container_id>
```

Finalmente, eliminamos el contenedor:



```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker rm <container_id>
```

Para confirmar el comportamiento de los contenedores al reiniciar desde una imagen, crearemos un nuevo contenedor interactivo a partir de `debian`:



```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker run -it debian
```

Al revisar, se observa que nano ya no está instalado, ya que los contenedores no persisten los cambios realizados en ellos al crearse desde cero.

1.0.6. CONCLUSIONES DE ESTA SECCIÓN

Este ejercicio muestra cómo Docker descarga y ejecuta imágenes desde Docker Hub automáticamente, y cómo los contenedores se comportan como instancias efímeras que desaparecen al detenerse, a menos que se gestionen específicamente para persistir cambios. Los resultados clave incluyen:

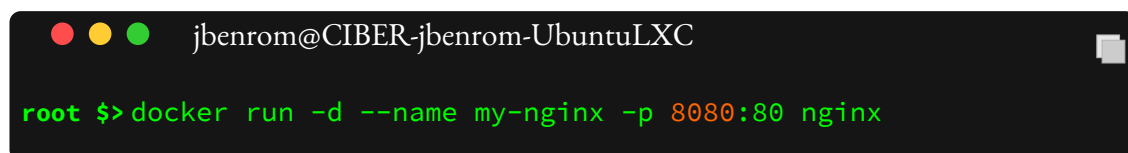
- Docker descarga automáticamente las imágenes al no estar disponibles localmente.
- Los contenedores efímeros pueden ser eliminados automáticamente con el parámetro `--rm`.
- Al detener un contenedor interactivo, este se detiene debido a la ausencia de un proceso en ejecución.
- Cambios como la instalación de paquetes dentro de un contenedor no se mantienen en nuevas instancias creadas desde la misma imagen base.

1.1. CREACIÓN DE UN CONTENEDOR CON UN SERVIDOR WEB

En esta sección, se creará un contenedor basado en la imagen oficial de `nginx`, configurando el puerto de acceso y manipulando los archivos del servidor web. Esto es útil en desarrollo para probar aplicaciones web en entornos aislados y en ciberseguridad para asegurar el correcto control de accesos y la integridad de los archivos en el servidor.

1.1.1. CREACIÓN DEL CONTENEDOR EN SEGUNDO PLANO

Para iniciar un servidor web usando `nginx` en segundo plano, se ejecuta el siguiente comando. Docker descargará automáticamente la imagen desde Docker Hub la primera vez que se ejecute.

A terminal window with a dark background. The prompt is 'jbenrom@CIBER-jbenrom-UbuntuLXC'. The command entered is 'root \$> docker run -d --name my-nginx -p 8080:80 nginx'. The command is highlighted in green.

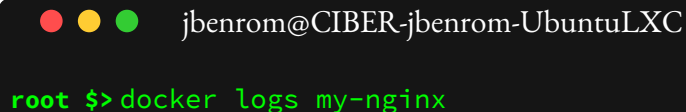
```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run -d --name my-nginx -p 8080:80 nginx
```

Este comando asigna el nombre `my-nginx` al contenedor y mapea el puerto 80 del contenedor con el puerto 8080 en la red del host, permitiendo acceder al servidor web desde

`http://<IP_DEL_HOST>:8080`. Al abrir esta URL en un navegador, mostrará la página de bienvenida predeterminada de `nginx`.

1.1.2. VISUALIZACIÓN DE LOS LOGS DEL SERVIDOR WEB

Para visualizar los registros generados por el servidor web `nginx`, ejecutamos:



```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker logs my-nginx
```


Este comando muestra los accesos y actividades recientes en el servidor. Es útil en desarrollo para depurar peticiones al servidor y en ciberseguridad para monitorear actividades potencialmente sospechosas.

1.1.3. MODIFICACIÓN DE LOS ARCHIVOS EN EL CONTENEDOR

La imagen `nginx` guarda sus archivos web en el directorio `/usr/share/nginx/html` dentro del contenedor. Para cambiar la página web que sirve el servidor, se puede actualizar el archivo `index.html` de diferentes maneras.

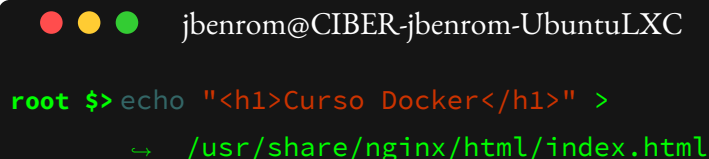
MÉTODO 1: ACCESO INTERACTIVO AL CONTENEDOR

Podemos acceder de manera interactiva al contenedor con el siguiente comando:



```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker exec -it my-nginx bash
```

Dentro del contenedor, navegamos a `/usr/share/nginx/html` y creamos un archivo `index.html` personalizado. Por ejemplo:



```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> echo "<h1>Curso Docker</h1>" >
      /usr/share/nginx/html/index.html
```

Luego, al refrescar el navegador, se mostrará el nuevo contenido.

MÉTODO 2: CREACIÓN DEL ARCHIVO MEDIANTE UN COMANDO EXTERNO

Otra opción para modificar el contenido de `index.html` es ejecutar el comando de creación desde el exterior del contenedor:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker exec my-nginx bash -c 'echo "<h1>Curso Docker</h1>"
    ↪ > /usr/share/nginx/html/index.html'
```

Este comando realiza la misma operación sin necesidad de acceso interactivo. Luego, comprobamos en el navegador que la página muestre el nuevo contenido.

MÉTODO 3: VINCULACIÓN DE UN DIRECTORIO DEL HOST MEDIANTE UN *BIND MOUNT*

Para hacer el contenido del servidor web persistente y editable desde el host, podemos vincular un directorio local. Primero, creamos el subdirectorio en el host donde colocaremos el contenido de la web:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> mkdir -p ~/nginx/contenido
root $> echo "<h1>Curso Docker</h1>" >
    ↪ ~/nginx/contenido/index.html
```

Luego, iniciamos el contenedor `nginx` con el siguiente comando, vinculando el directorio local:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker run --rm -d -p 8080:80 --name web -v
    ↪ ~/nginx/contenido:/usr/share/nginx/html nginx
```

Este comando permite que el contenido de `index.html` en `/nginx/contenido` se refleje automáticamente en el servidor web del contenedor. Al actualizar el archivo `index.html` en el host y refrescar el navegador, los cambios serán visibles de inmediato en el servidor.

I.1.4. CONCLUSIONES DE ESTA SECCIÓN

Este ejercicio demuestra el despliegue de un servidor web en Docker y los métodos para gestionar y modificar el contenido en desarrollo y ciberseguridad. Algunos puntos interesantes incluyen:

- Docker gestiona el despliegue de aplicaciones de forma aislada, lo cual es ideal para probar entornos de servidor.
- Los métodos de acceso y modificación del contenido (interactivo, comandos externos, *bind mounts*) permiten una alta flexibilidad según los requisitos.
- Los logs del contenedor son esenciales en ciberseguridad para monitorear accesos y detectar posibles actividades anómalas.
- Los *bind mounts* permiten una edición de contenido persistente y controlada, sin necesidad de reconstruir la imagen, lo cual es útil en desarrollo.

I.2. GESTIÓN DE IMÁGENES DOCKER

Esta sección aborda la gestión de imágenes en Docker, lo cual es fundamental para entender cómo funcionan las capas de imágenes y el almacenamiento en contenedores. Esto es relevante tanto en desarrollo, para optimizar el uso de recursos y gestionar versiones de aplicaciones, como en ciberseguridad, para garantizar que las imágenes descargadas provienen de fuentes confiables y no contienen capas con vulnerabilidades.

I.2.1. DESCARGA DE IMÁGENES

Se descargan varias imágenes desde Docker Hub utilizando el comando `docker pull`. Este comando permite traer las imágenes al registro local y facilita el acceso a diferentes versiones de sistemas operativos y servicios, útiles en ambientes de desarrollo y pruebas de seguridad.

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker pull ubuntu:22.04
root $> docker pull tomcat:9.0.70-jdk11
root $> docker pull jenkins/jenkins:lts
root $> docker pull php:8.3-apache
```

Para visualizar todas las imágenes descargadas en el sistema local, usamos:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker images
```

1.2.2. CREACIÓN DE UN CONTENEDOR EN SEGUNDO PLANO

Creamos un contenedor en segundo plano a partir de la imagen `php:8.3-apache`:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker run -d --name php-apache php:8.3-apache
```

A continuación, comprobamos el tamaño que ocupa el contenedor en disco:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker ps -a -s
```

Observamos que el tamaño real del contenedor es `0B`, ya que Docker comparte las capas de la imagen. Este proceso optimiza el almacenamiento, importante en entornos de desarrollo con múltiples contenedores y para reducir la superficie de ataque en seguridad, manteniendo el sistema lo más liviano posible.

1.2.3. COPIA DE ARCHIVOS A UN CONTENEDOR

Creamos un archivo en el sistema anfitrión con el siguiente contenido para verificar la configuración de PHP en el contenedor:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> echo "<?php echo phpinfo(); ?>" > info.php
```

Copiamos este archivo al contenedor con `docker cp`:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker cp info.php php-apache:/var/www/html
```

Luego, accedemos a `http://<IP_DEL_HOST>:8080/info.php` en el navegador para ver la configuración de PHP en ejecución dentro del contenedor.

1.2.4. VERIFICACIÓN DEL ESPACIO OCUPADO

Tras modificar el contenido del contenedor, verificamos nuevamente el espacio ocupado por el contenedor para observar si ha habido cambios en su tamaño:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker ps -a -s
```

Esta operación es útil para verificar cómo afecta el contenido dinámico al tamaño de un contenedor en entornos de desarrollo y para evaluar el uso de almacenamiento en seguridad, donde minimizar datos persistentes es crucial.

1.2.5. INSPECCIÓN DE LA IMAGEN

Para inspeccionar la imagen `php:8.3-apache` y ver las capas que la componen, utilizamos:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker inspect php:8.3-apache
```

Este comando permite ver detalles de la imagen y cada capa, lo cual es útil en desarrollo para verificar las dependencias y en ciberseguridad para entender el contenido y configuración exacta de la imagen.

1.2.6. ELIMINACIÓN DE IMÁGENES

Intentamos eliminar la imagen `php:8.3-apache` del sistema:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker rmi php:8.3-apache
```

Al no poder eliminar una imagen en uso, primero eliminamos el contenedor que la utiliza y luego intentamos nuevamente. Alternativamente, podemos utilizar `docker image prune -a` para limpiar todas las imágenes no utilizadas:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker rm php-apache
root $> docker rmi php:8.3-apache
root $> docker image prune -a
```

1.2.7. CREACIÓN DE CONTENEDORES DE MEDIAWIKI

Ahora, creamos tres contenedores basados en distintas versiones de `MediaWiki`. Estas versiones permiten comparar la compatibilidad de diferentes lanzamientos y probar configuraciones de desarrollo o análisis de seguridad en entornos idénticos.

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run -d -p 8080:80 --name mediawiki1 mediawiki
root $> docker run -d -p 8081:80 --name mediawiki2
    ↪ mediawiki:1.36.3
root $> docker run -d -p 8082:80 --name mediawiki3
    ↪ mediawiki:1.35.5
```

Al acceder a cada contenedor en los puertos 8080, 8081 y 8082 respectivamente, verificamos que cada instancia esté ejecutando una versión diferente de `MediaWiki`. Docker sólo descarga las capas necesarias en la primera versión, ahorrando ancho de banda y almacenamiento.

1.2.8. CONCLUSIONES DE ESTA SECCIÓN

Este ejercicio demuestra cómo Docker optimiza la gestión de imágenes, facilitando el uso de múltiples versiones de aplicaciones en contenedores sin duplicación innecesaria. Entre los aspectos destacados se encuentran:

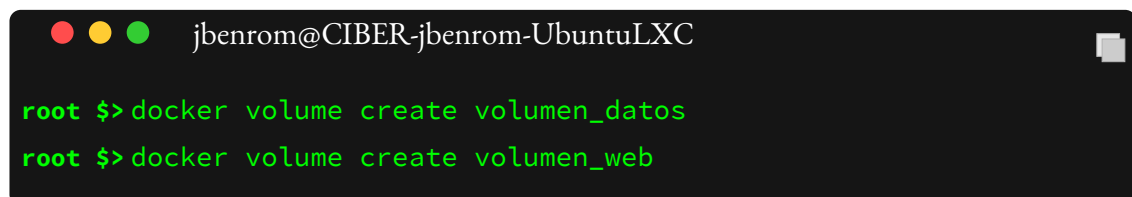
- Las capas de imagen se comparten entre contenedores, minimizando el uso de almacenamiento, lo cual es útil en desarrollo y en seguridad para reducir la superficie de ataque.
- Los contenedores mantienen el almacenamiento en cero hasta que se modifican, asegurando que no ocupan espacio innecesario en entornos de producción y en seguridad.
- Las diferentes versiones de una imagen comparten capas comunes, optimizando el almacenamiento y simplificando la gestión de versiones en el desarrollo de aplicaciones.
- El comando `docker image prune -a` facilita la limpieza de imágenes no utilizadas, ideal para optimizar el uso de recursos en máquinas de desarrollo o producción.

1.3. CREACIÓN Y USO DE VOLÚMENES

En esta sección trabajamos con volúmenes y *bind mounts*, herramientas fundamentales en Docker para la persistencia de datos y la compartición de archivos entre el contenedor y el sistema anfitrión. Esto es esencial en entornos de desarrollo, donde los datos deben persistir más allá del ciclo de vida del contenedor, y en ciberseguridad, donde es importante gestionar cuidadosamente los datos compartidos para evitar fugas o acceso no autorizado.

1.3.1. CREACIÓN DE VOLÚMENES DOCKER

Creamos los volúmenes `volumen_datos` y `volumen_web` mediante el comando:

A terminal window with a dark background. At the top, it shows the prompt 'jbenrom@CIBER-jbenrom-UbuntuLXC'. Below, two commands are entered in green text: 'root \$> docker volume create volumen_datos' and 'root \$> docker volume create volumen_web'.

```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker volume create volumen_datos  
root $> docker volume create volumen_web
```

Para listar los volúmenes creados:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker volume ls
```

1.3.2. CREACIÓN DE CONTENEDORES CON VOLÚMENES MONTADOS

Primero, lanzamos un contenedor llamado c1 que utiliza la imagen php:8.3-apache y monta el volumen volumen_web en la ruta /var/www/html. Exponemos este contenedor en el puerto 8080.

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker run -d --name c1 -p 8080:80 --mount
    ↪ source=volumen_web,target=/var/www/html php:8.3-apache
```

Luego, creamos un segundo contenedor c2 basado en la imagen mariadb, montando volumen_datos en /var/lib/mysql y estableciendo la contraseña de root mediante la variable de entorno MYSQL_ROOT_PASSWORD:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker run -d --name c2 -e MYSQL_ROOT_PASSWORD=admin
    ↪ --mount source=volumen_datos,target=/var/lib/mysql
    ↪ mariadb
```

1.3.3. INTENTO DE BORRADO DEL VOLUMEN volumen_datos

Intentamos borrar el volumen volumen_datos con:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker volume rm volumen_datos
```

Al recibir un mensaje de error, paramos y eliminamos el contenedor c2 para liberar el volumen:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker stop c2
root $> docker rm c2
root $> docker volume rm volumen_datos
```

1.3.4. MANIPULACIÓN DE ARCHIVOS EN EL CONTENEDOR c1

Creamos un archivo `index.html` en el contenedor `c1` con el siguiente contenido:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> echo "<h1>Bienvenido a Docker</h1>" > index.html
root $> docker cp index.html c1:/var/www/html/
```

Accedemos al archivo a través de `http://localhost:8080/index.html` para confirmar que se sirve correctamente.

1.3.5. RECREACIÓN DEL CONTENEDOR c1 COMO c3

Eliminamos el contenedor `c1` y creamos un nuevo contenedor `c3` con características similares, pero mapeado al puerto 8081:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker stop c1
root $> docker rm c1
root $> docker run -d --name c3 -p 8081:80 --mount
    ↪ source=volumen_web,target=/var/www/html php:8.3-apache
```

Accedemos a `http://localhost:8081/index.html` para verificar el acceso.

1.3.6. USO DE *BIND MOUNT* PARA COMPARTIR DATOS

Creamos un directorio en el sistema anfitrión con un archivo `index.html`:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> mkdir saludo
root $> echo "<h1>HOLA SOY XXXXXX</h1>" > saludo/index.html
```

Iniciamos dos contenedores basados en `php:8.3-apache` que hagan un *bind mount* del directorio `saludo` en `/var/www/html` del contenedor, mapeados a los puertos 8181 y 8282:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker run -d --name c1 -p 8181:80 -v
    ↪ ./saludo:/var/www/html php:8.3-apache
root $> docker run -d --name c2 -p 8282:80 -v
    ↪ ./saludo:/var/www/html php:8.3-apache
```

Para confirmar el contenido, utilizamos `curl` en cada contenedor:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> curl http://localhost:8181
root $> curl http://localhost:8282
```

1.3.7. MODIFICACIÓN DEL CONTENIDO DEL ARCHIVO `index.html`

Modificamos el archivo `index.html` en `saludo` y accedemos a ambos contenedores para verificar la actualización inmediata:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> echo "<h1>HOLA SOY XXXXXX, AHORA MODIFICADO</h1>" >
    ↪ saludo/index.html
root $> curl http://localhost:8181
root $> curl http://localhost:8282
```

1.3.8. COMPROBACIÓN DE DIRECTORIO AL REALIZAR UN *BIND MOUNT*

Creamos un contenedor con la imagen `httpd:2.4` utilizando la opción `-v` para realizar un *bind mount* en un directorio inexistente en el anfitrión:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run -d --name apache -p 8383:80 -v
      ↪ ./nuevo_directorio:/usr/local/apache2/htdocs httpd:2.4
```

Al verificar el sistema anfitrión, observamos que Docker crea automáticamente el directorio vacío. Para probar el acceso, copiamos `index.html` en el directorio montado y accedemos a `http://localhost:8383/index.html`.

Si repetimos el proceso con `--mount` en lugar de `-v`, el contenedor no se ejecutará si el directorio no existe en el anfitrión, ya que `--mount` no crea directorios automáticamente.

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run -d --name apache -p 8383:80 --mount
      ↪ source=./nuevo_directorio,target=/usr/local/apache2/htdocs
      ↪ httpd:2.4
```

1.3.9. CONCLUSIONES DE ESTA SECCIÓN

Este ejercicio ilustra la gestión de volúmenes y *bind mounts* en Docker, con las siguientes conclusiones destacadas:

- Los volúmenes permiten persistencia de datos en Docker, útil para almacenar datos de servicios como bases de datos, sin perder información al reiniciar contenedores.
- La capacidad de reutilizar volúmenes en diferentes contenedores es esencial en entornos de desarrollo para la persistencia de datos y en ciberseguridad para manejar datos sensibles.
- Los *bind mounts* ofrecen acceso a archivos en tiempo real entre el sistema anfitrión y los contenedores, útil en desarrollo para probar cambios sin reconstruir contenedores.
- El uso de `-v` frente a `--mount` permite control sobre la creación de directorios, algo a considerar en la configuración de contenedores y la seguridad de datos.

I.4. DEMOSTRACIÓN (CASO PRÁCTICO LXC, USUARIO JBEN-ROM)

En esta sección se muestra, en capturas de pantalla, las partes del proceso y los resultados de este, más interesantes.

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
302e3ee49805: Pull complete
cd986b3703ae: Extracting [=====>] 17.04MB/41.88MB
34a52cbc3961: Download complete
d1875670ac8a: Download complete
af17adb1bdcc: Download complete
97182578e5ec: Download complete
67b9310357e1: Download complete
```

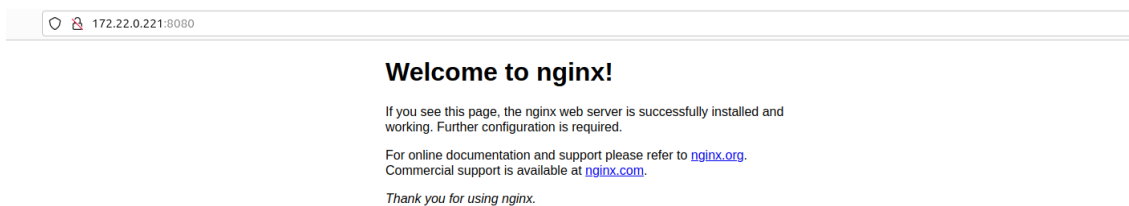


Figura 1.1: Respuesta del servidor nginx en el contenedor

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker logs web
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2024/10/01 13:43:58 [notice] 1#1: using the "epoll" event method
2024/10/01 13:43:58 [notice] 1#1: nginx/1.27.1
2024/10/01 13:43:58 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2024/10/01 13:43:58 [notice] 1#1: OS: Linux 6.8.12-1-pve
2024/10/01 13:43:58 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 524288:524288
2024/10/01 13:43:58 [notice] 1#1: start worker processes
2024/10/01 13:43:58 [notice] 1#1: start worker process 29
2024/10/01 13:43:58 [notice] 1#1: start worker process 30
2024/10/01 13:43:58 [notice] 1#1: start worker process 31
2024/10/01 13:43:58 [notice] 1#1: start worker process 32
2024/10/01 13:43:58 [notice] 1#1: start worker process 33
2024/10/01 13:43:58 [notice] 1#1: start worker process 34
2024/10/01 13:43:58 [notice] 1#1: start worker process 35
2024/10/01 13:43:58 [notice] 1#1: start worker process 36
2024/10/01 13:43:58 [notice] 1#1: start worker process 37
2024/10/01 13:43:58 [notice] 1#1: start worker process 38
2024/10/01 13:43:58 [notice] 1#1: start worker process 39
2024/10/01 13:43:58 [notice] 1#1: start worker process 40
172.22.0.187 -- [01/Oct/2024:13:56:52 +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/117.0"
--
172.22.0.187 -- [01/Oct/2024:13:56:52 +0000] "GET /favicon.ico HTTP/1.1" 404 153 "http://172.22.0.221:8080/" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/117.0"
--
2024/10/01 13:56:52 [error] 29#29: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 172.22.0.187, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "172.22.0.221:8080", referer: "http://172.22.0.221:8080/"
```

Figura 1.2: Logs del contenedor

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ cd nginx/contenido/
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nginx/contenido$ echo "<h1>Curso Docker</h1>" > index.html
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nginx/contenido$ cd ../..
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ ls -lR
.:
total 4
drwxr-xr-x 3 jbenrom docker 4096 Oct  1 14:15 nginx

./nginx:
total 4
drwxr-xr-x 2 jbenrom docker 4096 Oct  1 14:29 contenido

./nginx/contenido:
total 4
-rw-rw-r-- 1 jbenrom jbenrom 22 Oct  1 14:29 index.html
jbenrom@CIBER-jbenrom-UbuntuLXC:~$
```

Figura 1.3: Agregamos un index.html al documentroot del contenedor (que esta enlazado a través de un bind mount)




Curso Docker

Figura 1.4: Ahora se muestra el contenido de index.html

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
fe40be9e96c0   php:8.3-apache "docker-php-entrypoi..." About a minute ago Up About a minute 0.0.0.0:8080->80/tcp, [::]:8080->80/tcp web
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker cp index.php $(docker ps -q):/var/www/html
Successfully copied 2.05kB to fe40be9e96c0:/var/www/html
jbenrom@CIBER-jbenrom-UbuntuLXC:~$
```

Figura 1.5: Copiamos un archivo previamente creado dentro de un contenedor activo, para mostrar `phpinfo()`

PHP Version 8.3.12

System	Linux fe40be9e96c0 6.8.12-1-pve #1 SMP PREEMPT_DYNAMIC PMX 6.8.12-1 (2024-08-05T16:17Z) x86_64
Build Date	Sep 27 2024 06:16:57
Build System	Linux - Docker
Build Provider	https://github.com/docker-library/php
Configure Command	"/configure" "-build=x86_64-linux-gnu" "-with-config-file-path=/usr/local/etc/php" "-with-config-file-scan-dir=/usr/local/etc/php/conf.d" "--enable-option-checking=fatal" "--with-mhash" "--with-pic" "--enable-mbstring" "--enable-mysqlnd" "--with-password-argon2" "--with-sodium=shared" "--with-pdo-sqlite=static" "--with-sockets" "--with-curl" "--with-iconv" "--with-openssl" "--with-readline" "--with-zlib" "--disable-phdbg" "--with-pear" "--with-libdir=lib/x86_64-linux-gnu" "--disable-cgi" "--with-apxs2" "build_alias=x86_64-linux-gnu"
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/etc/php
Loaded Configuration File	(none)
Scan this dir for additional .ini files	/usr/local/etc/php/conf.d
Additional .ini files parsed	/usr/local/etc/php/conf.d/docker-php-ext-sodium.ini
PHP API	20230831
PHP Extension	20230831
Zend Extension	420230831
Zend Extension Build	API420230831,NTS
PHP Extension Build	API20230831,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	enabled
Zend Memory Manager	enabled
Zend Multibyte Support	provided by mbstring

Figura 1.6: Salida de `phpinfo()`

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker inspect php:8.3-apache | awk '/layers/,0'
  "Type": "layers",
  "Layers": [
    "sha256:8d853c8add5d1e7b0aa4c4b68a3d9fb8e7a0da27970c2acf831fe63be4a0cd2c",
    "sha256:599a2026a8f66696b36b8ef53c962013d3353a88e7c69cbdfc7e14b53ecca477",
    "sha256:60f0b6ccdbac85dc6451105f31d75ccb0427265e0badd0cd7cf4ae4f6574854b",
    "sha256:dfd6300bd525aa29559079b12b0201adb0515dcfc414c80a613713a768052d8a",
    "sha256:6025fd56312dc58ed5263e29be0f2f5aca4aaea359217ff96a6464a22ba7a3dc",
    "sha256:5c8595af30bed2edf5c7cc6d31fa78cba059c2be3552dbbcaca80ffb3ba41542",
    "sha256:3c776fff1cc44f68b307398771dc5ef4577bac97df33f017ad5a6accc187f7b9",
    "sha256:17da03c83dee6d3c44dd60720bbedbac668c20faa726b841d388728d93590720",
    "sha256:c08243f5fb7ee131f8aa8592250f6c12f59e1033e86bb98cb48981338b8b00dd",
    "sha256:e31294ebff217c65fe2a73402b58cc17e37847645069add8f12fb64609434143",
    "sha256:c89404bd5086418dfa7803d269de22f8a1d7978cb88818c8a60ec32cac137a3a",
    "sha256:7e144c2aed00026d1abf94c9831ca2fde37bbdc8cffda3827ee915a5a2102929",
    "sha256:dcfff08eb727dcc961698c31ac64a7f8b8cadf8714837e544c79e420a25fcbad"
  ]
},
  "Metadata": {
    "LastTagTime": "0001-01-01T00:00:00Z"
  }
}
```

Figura 1.7: Comprobamos las diferentes capas de una imagen de Docker con `docker inspect`

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker rmi php:8.3-apache
Error response from daemon: conflict: unable to remove repository reference "php:8.3-apache" (must force) - container fe40be9e96c0 is using its referenced image 2fa865df359d
jbenrom@CIBER-jbenrom-UbuntuLXC:~$
```

Figura 1.8: Tratamos de borrar una imagen de la cual depende un contenedor, y como vemos en la salida no nos deja

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker stop $(docker ps -q)
fe40be9e96c0
```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker rm $(docker ps -q -a)
fe40be9e96c0
jbenrom@CIBER-jbenrom-UbuntuLXC:~$
```

Figura 1.9: Una vez parado y borrado el contenedor si nos lo permite

```

Unable to find image 'mediawiki:latest' locally
latest: Pulling from library/mediawiki
302e3ee49805: Pull complete
07fc0890b857: Pull complete
141aa7d58c57: Pull complete
2720d4bca8b3: Pull complete
82deca51468c: Pull complete
dec741dfa526: Pull complete
e204b0efab94: Pull complete
da3427d4ab01: Pull complete
75c32fd90f5c: Pull complete
ea57467a10b9: Pull complete
56c9d963ab40: Pull complete
7f7723130213: Pull complete
c2f5f2697bee: Pull complete
d21b227ee3ec: Pull complete
414288eddc8: Pull complete
ebfef0b5d063: Pull complete
16bde7818295: Pull complete
55dbe9558230: Pull complete
c4772fcfe567: Pull complete
2fcad4f56dd9: Pull complete
Digest: sha256:8d266b2bf6d47158f67965f2468d49dc983719279b92da44b38b3d2426f888a6
Status: Downloaded newer image for mediawiki:latest
f7f70b1ee1b20fc00410fda0ff2882bb37d9a86eb561eaff140bb07c14fada3a

```

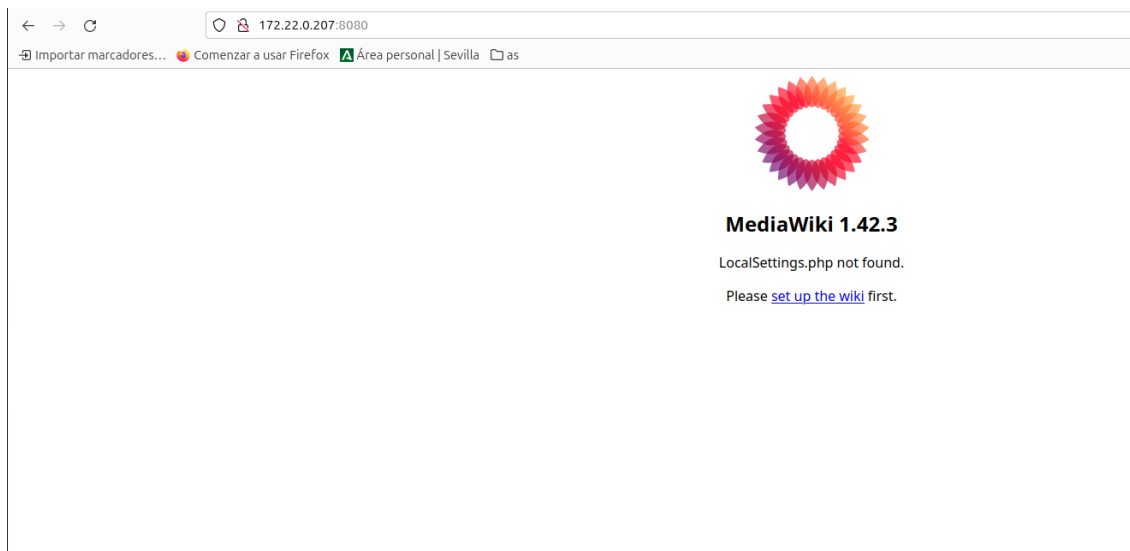
```

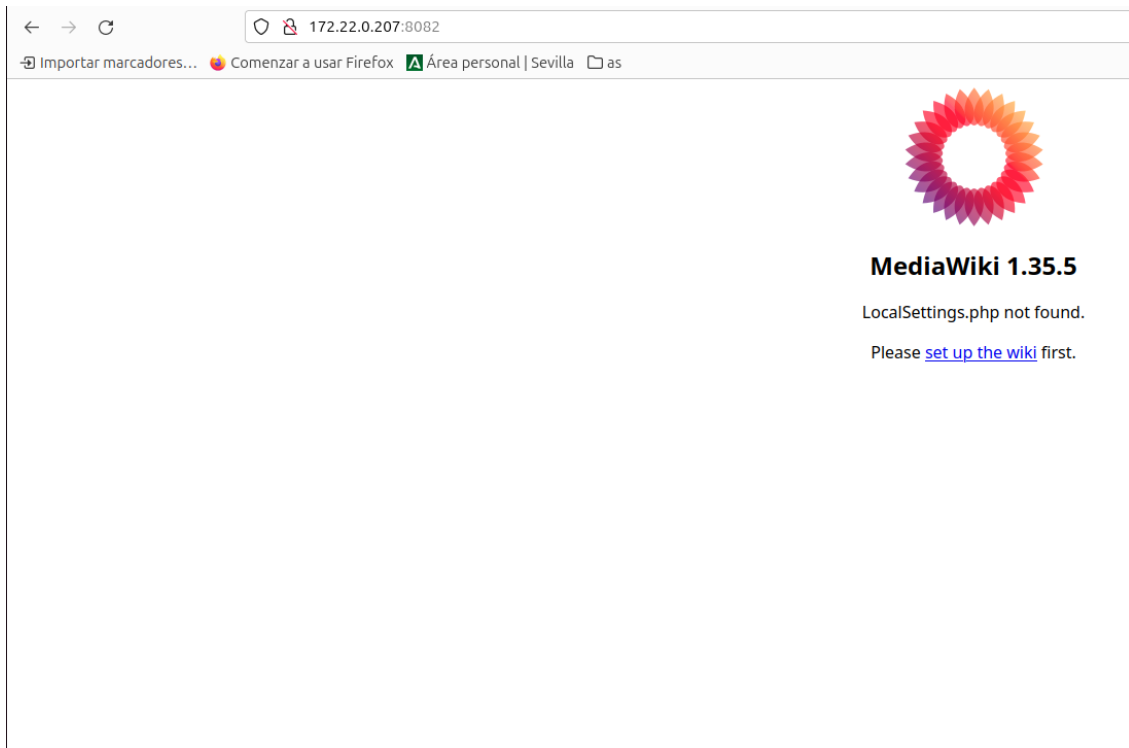
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker run -d -p 8081:80 --name mediawiki2 mediawiki:1.36.3
Unable to find image 'mediawiki:1.36.3' locally
1.36.3: Pulling from library/mediawiki
c229119241af: Pull complete
47e86af584f1: Pull complete
e1bd55b3ae5f: Pull complete
1f3a70af964a: Pull complete
0f5086159710: Pull complete
7d9c764dc190: Pull complete
ec2bb7a6eead: Pull complete
9d9132470f34: Pull complete
fb23ab197126: Pull complete
cbdd566be443: Pull complete
be224cc1ae0f: Pull complete
629912c3cae4: Pull complete
f1bae9b2bf5b: Pull complete
add6e63a3419: Pull complete
97428b23ad3a: Pull complete
8853f302be86: Pull complete
cb2c69ee8e82: Pull complete
cedb956544ab: Pull complete
051cc1d997ba: Pull complete
0291284fc5dd: Pull complete
Digest: sha256:3a4458511c5a556c34d7d8df1c1a447df8b0bf35d296d564220c9c07e5e4bb89
Status: Downloaded newer image for mediawiki:1.36.3
40ac1a9733c4d137019731alc2fc1fc12cccd83395b1fe4c0f07fb7eb3516c02

```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker run -d -p 8081:80 --name mediawiki2 mediawiki:1.36.3
Unable to find image 'mediawiki:1.36.3' locally
1.36.3: Pulling from library/mediawiki
c229119241af: Pull complete
47e86af584f1: Pull complete
e1bd55b3ae5f: Pull complete
1f3a70af964a: Pull complete
0f5086159710: Pull complete
7d9c764dc190: Pull complete
ec2bb7a6eead: Pull complete
9d9132470f34: Pull complete
fb23ab197126: Pull complete
cbdd566be443: Pull complete
be224cc1ae0f: Pull complete
629912c3cae4: Pull complete
f1bae9b2bf5b: Pull complete
add6e63a3419: Pull complete
97428b23ad3a: Pull complete
8853f302be86: Pull complete
cb2c69ee8e82: Pull complete
cedb956544ab: Pull complete
051cc1d997ba: Pull complete
0291284fc5dd: Pull complete
Digest: sha256:3a4458511c5a5556c34d7d8df1c1a447df8b0bf35d296d564220c9c07e5e4bb89
Status: Downloaded newer image for mediawiki:1.36.3
40ac1a9733c4d137019731a1c2fc1fc12cccd83395b1fe4c0f07fb7eb3516c02
```

Figura 1.10: Cuando descargamos distintas versiones de una misma imagen de Docker podemos comprobar que hay capas que ya existen de versiones anteriores y omite la descarga





```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ for volume in volumen_datos volumen_web; do docker volume create $volume; done
volumen_datos
volumen_web
```

Figura 1.11: Creamos volúmenes de Docker

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker volume list
DRIVER      VOLUME NAME
local       volumen_datos
local       volumen_web
```

Figura 1.12: Los listamos para comprobar que se han creado

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker run -d --volume volumen_datos:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=admin --name c2 mariadb
Unable to find image 'mariadb:latest' locally
latest: Pulling from library/mariadb
eda6120e237e: Pull complete
3bea7484bf1d: Pull complete
97768484d3db: Pull complete
f7c4ca00d7c4: Pull complete
200fec9a56a: Pull complete
4c3b42e2cd08: Pull complete
e287e4cde285: Pull complete
c19c18c0f9a2: Pull complete
Digest: sha256:9e7695800ab8fa72d75053fe536b090d0c9373465b32a073c73bc7940a2e8dbe
Status: Downloaded newer image for mariadb:latest
444e417c51e90dd31d899679aa87ca261f48daec0b7fddc1a34fe7426531c3902
```

Figura 1.13: Se lo asignamos a un contenedor

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker volume rm volumen_datos
Error response from daemon: remove volumen_datos: volume is in use - [444e417c51e90dd31d899679aa87ca261f48daec0b7fddc1a34fe7426531c3902]
```

Figura 1.14: Y lo intentamos borrar mientras este esta en uso para comprobar que no nos lo permite el propio Docker

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker stop c2
c2
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker rm c2
c2
jbenrom@CIBER-jbenrom-UbuntuLXC:~$
```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker volume rm volumen_datos
volumen_datos
jbenrom@CIBER-jbenrom-UbuntuLXC:~$
```

Figura 1.15: Una vez parado y borrado el contenedor que lo tenia asignado, si nos permite borrarlo

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker run -d --volume volumen_web:/var/www/html -p 8080:80 --name c1 php:8.3-apache
Unable to find image 'php:8.3-apache' locally
8.3-apache: Pulling from library/php
302e3ee49805: Already exists
07fc0890b857: Already exists
141aa7d58c57: Already exists
2720d4bca0b3: Already exists
82deca51468c: Already exists
dec741dfa526: Already exists
e204b0efab94: Already exists
87046c2c35e7: Pull complete
3842de6108cf: Pull complete
9a17887140a6: Pull complete
9208c52a9c8e: Pull complete
8d11d91a387d: Pull complete
4ce4dac50170: Pull complete
Digest: sha256:6d553ea70429a19fcl790cfd3796b4d456dbe48095e5c600756b31094fd44072
Status: Downloaded newer image for php:8.3-apache
75b85a25e1069932eac8437a33278b1ec5b60159aa2df0d3af02ac478fd3c844
docker: Error response from daemon: driver failed programming external connectivity on endpoint c1 (23fc36326181db9ce5abb41f6e160aa3a5b7d853b89ab25cb2216259e6b27f221): Bind for 0.0.0.0:8080 failed: port is already allocated.
```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker exec -it c1 bash
root@75b85a25e106:/var/www/html# echo '<h1> Hola </h1>' > index.html
root@75b85a25e106:/var/www/html# ls -l
total 4
-rw-r--r-- 1 root root 16 Oct  7 17:06 index.html
root@75b85a25e106:/var/www/html#
```

Figura 1.16: Entramos dentro del contenedor de apache para generar un archivo que se guardara en el volumen previamente asignado

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker stop c1
c1
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker rm c1
c1
```

Figura 1.17: Borramos el contenedor para poder asignar el volumen a otro contenedor

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker start c3
c3
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
31d6139ad398	php:8.3-apache	"docker-php-entrypoi..."	33 seconds ago	Up 2 seconds	0.0.0.0:8081->80/tcp, [::]:8081->80/tcp	c3
352f2396f49e	mediawiki:1.35.5	"docker-php-entrypoi..."	About an hour ago	Up About an hour	0.0.0.0:8082->80/tcp, [::]:8082->80/tcp	mediawiki3

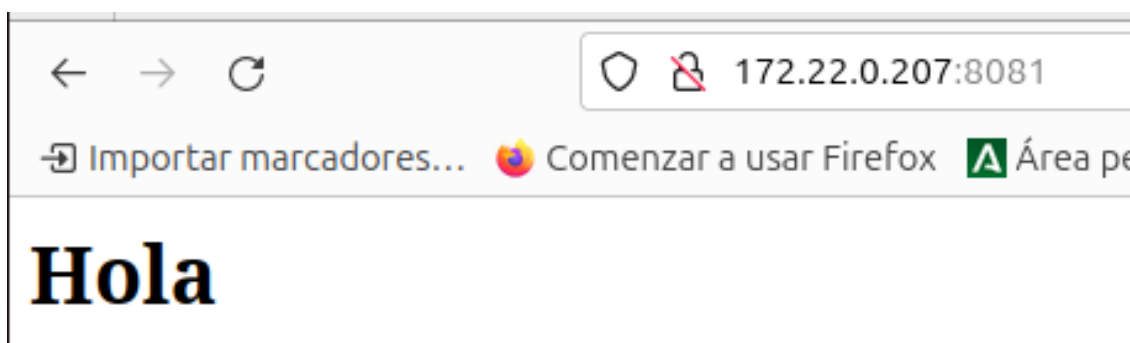


Figura 1.18: Al acceder a través del navegador al nuevo contenedor (c3) con el puerto 8081 mapeado, y el volumen_web asignado, vemos que se logro la persistencia del archivo mediante el uso de volúmenes

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ pwd
/home/jbenrom
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ mkdir saludo
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ ls
index.php  nginx  saludo
jbenrom@CIBER-jbenrom-UbuntuLXC:~$
```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ echo '<h1> HOLA SOY JOSE </h1>' > saludo/index.html
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ ls -l saludo
total 4
-rw-rw-r-- 1 jbenrom jbenrom 25 Oct  7 17:19 index.html
jbenrom@CIBER-jbenrom-UbuntuLXC:~$
```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker run -d -v ~/saludo:/var/www/html -p 8181:80 --name c1 php:8.3-apache
31253cd03bbcc63b68ad4bd423798989edcb11158360533a081b7a3b1a69b97b
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
31253cd03bbcc63b68ad4bd423798989edcb11158360533a081b7a3b1a69b97b	php:8.3-apache	"docker-php-entrypoi..."	11 seconds ago	Up 10 seconds	0.0.0.0:8181->80/tcp, [::]:8181->80/tcp	c1

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker run -d -v ~/saludo:/var/www/html -p 8282:80 --name c2 php:8.3-apache
2af0ff3438f49d7847f89c43bfea85e694b0de64367ee189442ea6f946f7cf8c
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2af0ff3438f49d7847f89c43bfea85e694b0de64367ee189442ea6f946f7cf8c	php:8.3-apache	"docker-php-entrypoi..."	5 seconds ago	Up 5 seconds	0.0.0.0:8282->80/tcp, [::]:8282->80/tcp	c2
31253cd03bbcc63b68ad4bd423798989edcb11158360533a081b7a3b1a69b97b	php:8.3-apache	"docker-php-entrypoi..."	About a minute ago	Up About a minute	0.0.0.0:8181->80/tcp, [::]:8181->80/tcp	c1

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ curl http://localhost:8181
<h1> HOLA SOY JOSE </h1>
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ curl http://localhost:8282
<h1> HOLA SOY JOSE </h1>
```

Figura 1.19: También se puede lograr esta persistencia mediante el uso de bind mounts de directorios locales, lo que mapea el directorio local con un directorio del contenedor, pero en este caso habrá que tener en cuenta que el contenedor tenga los permisos necesarios en el directorio local mapeado

2

Imágenes Docker

2.1. IMAGEN PERSONALIZADA DE DOCKER

En esta práctica, implementaremos una aplicación PHP dentro de un contenedor Docker. A continuación, detallamos los pasos necesarios para configurar el contenedor y las pruebas correspondientes.

2.1.1. PREPARACIÓN DE ARCHIVOS

DOCKERFILE

El archivo `Dockerfile` define la configuración del contenedor. A continuación, se presenta su contenido:

```
1 # 1. Utilizamos Debian como imagen base
2 FROM debian:latest
3
4 # 2. Establecemos la zona horaria
5 RUN apt-get update && apt-get install -y tzdata && \
6     ln -sf /usr/share/zoneinfo/$(timedatectl | grep "Time
7     zone" | awk '{print $3}') /etc/localtime && \
    dpkg-reconfigure -f noninteractive tzdata
```

```

8
9 # 3. Instalamos Apache, PHP 8.3 y otras utilidades
10 RUN apt-get install -y apache2 php8.3 curl nano
11
12 # 4. Definimos el directorio de trabajo y copiamos los
    archivos desde el host al contenedor
13 WORKDIR /var/www/html
14 COPY . /var/www/html
15
16 # 5. Creamos el directorio /data y damos permisos al
    usuario www-data de Apache
17 RUN mkdir /data && chown -R www-data:www-data /data
18
19 # 6. Configuramos Apache para utilizar PHP 8.3 y
    habilitamos el módulo PHP
20 RUN a2enmod php8.3
21
22 # 7. Copiamos un archivo de configuración de PHP
    personalizado (si es necesario)
23 COPY custom-php.ini /etc/php/8.3/apache2/conf.d/
24
25 # 8. Copiamos el script de inicio y le damos permisos de
    ejecución
26 COPY entrypoint.sh /var/www/html/entrypoint.sh
27 RUN chmod +x /var/www/html/entrypoint.sh
28
29 # 9. Definimos el script de inicio como el entrypoint
30 ENTRYPOINT ["/var/www/html/entrypoint.sh"]

```

SCRIPT entrypoint.sh

Este script se utiliza para iniciar el servidor Apache. A continuación, se presenta su contenido:

```

#!/bin/bash
# Iniciamos Apache en segundo plano

```

```
apache2ctl -D FOREGROUND
```

SCRIPT index.php

El archivo PHP implementa un contador que almacena su estado en /data/counter.txt.

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> <?php
root $> $filename = '/data/counter.txt';
root $>
root $> // Creamos el archivo si no existe
root $> if (!file_exists($filename)) {
root $>     file_put_contents($filename, '0');
root $> }
root $>
root $> // Leemos el valor actual del contador
root $> $counter = (int) file_get_contents($filename);
root $> $counter++;
root $>
root $> // Escribimos el nuevo valor
root $> file_put_contents($filename, $counter);
root $>
root $> // Mostramos el contador en la página
root $> echo "<h1>Contador: $counter</h1>";
root $> ?>
```

ARCHIVO custom-php.ini

Para personalizar la configuración de PHP, usamos el archivo custom-php.ini:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> memory_limit = 128M
root $> max_execution_time = 30
```

2.1.2. CONSTRUCCIÓN Y EJECUCIÓN DEL CONTENEDOR

CONSTRUIR LA IMAGEN

Ejecutamos el siguiente comando para construir la imagen Docker:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker build -t custom_apache_php:1.0 .
```

EJECUTAR EL CONTENEDOR CON BIND MOUNT

Lanzamos el contenedor y mapeamos el directorio ./data del host:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run -d --rm -p 8086:80 --name
    ↪ my_apache_php_container -v $(pwd)/data:/data
    ↪ custom_apache_php:1.0
```

ACCESO A LA APLICACIÓN

Accedemos a <http://localhost:8086/index.php> para probar el contador.

2.1.3. PRUEBAS AVANZADAS

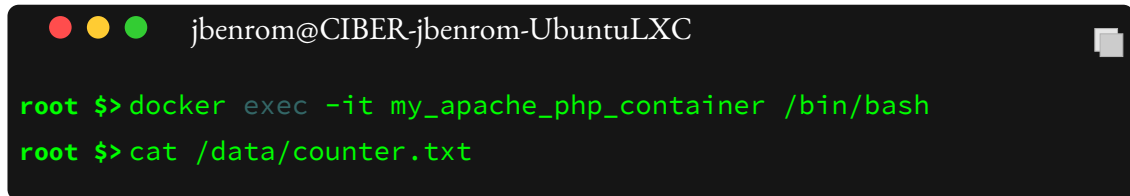
PERSISTENCIA CON VOLUMEN

Creamos un volumen Docker para mantener los datos persistentes:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker volume create my_volume
root $> docker run -d --rm -p 8086:80 --name
    ↪ my_apache_php_container -v my_volume:/data
    ↪ custom_apache_php:1.0
```

ACCESO AL CONTENEDOR

Ingresamos al contenedor para verificar el archivo `counter.txt`:



```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker exec -it my_apache_php_container /bin/bash  
root $> cat /data/counter.txt
```

2.1.4. CONCLUSIÓN

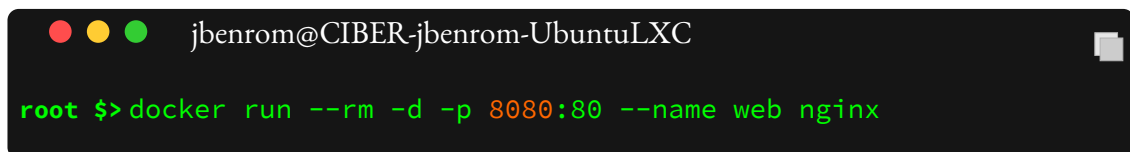
Esta práctica demuestra cómo configurar una aplicación PHP con Apache dentro de un contenedor Docker, implementando mecanismos de persistencia mediante bind mounts y volúmenes.

2.2. IMPLEMENTACIÓN DE NGINX CON CONTENIDO PERSONALIZADO

En esta práctica, trabajaremos con la imagen oficial de Nginx desde DockerHub, personalizaremos su contenido HTML usando *bind mounts*, crearemos una imagen personalizada y la subiremos a DockerHub.

2.2.1. EJECUCIÓN DE LA IMAGEN DE NGINX DESDE DOCKERHUB

Ejecutamos la imagen de Nginx en modo demonio, mapeando el puerto 80 del contenedor al puerto 8080 en el host:



```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker run --rm -d -p 8080:80 --name web nginx
```

Este comando realiza lo siguiente:

- `--rm`: elimina el contenedor cuando se detiene.
- `-d`: ejecuta el contenedor en segundo plano.
- `-p 8080:80`: mapea el puerto 80 en el contenedor al puerto 8080 en el host.

- `--name web`: asigna el nombre web al contenedor.

Podemos verificar el estado del contenedor con:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker ps
```

Para probar el servidor, accedemos desde el navegador a <http://localhost:8080> o ejecutamos:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> curl http://localhost:8080
```

2.2.2. PERSONALIZACIÓN DEL CONTENIDO HTML USANDO BIND MOUNTS

Creamos un directorio en el host para almacenar el contenido HTML personalizado:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> mkdir -p ~/nginx/contenido
```

Luego, creamos una página HTML en este directorio:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> echo "<!DOCTYPE html><html><head><title>My Custom
    ↳ Page</title>
root $> </head><body><h1>Hello from Nginx!</h1></body></html>" >
    ↳ ~/nginx/contenido/index.html
```

Ejecutamos el contenedor de Nginx con el directorio montado:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker run --rm -d -p 8080:80 --name web -v
    ↪ ~/nginx/contenido:/usr/share/nginx/html nginx
```

Ahora, accedemos a <http://localhost:8080> para verificar que el contenido personalizado se muestra correctamente.

2.2.3. CREACIÓN DE UNA IMAGEN PERSONALIZADA DE NGINX

Para incluir el archivo HTML de forma permanente en una imagen, creamos un Dockerfile:

```
1 # Usa la última versión de Nginx como base
2 FROM nginx:latest
3
4 # Copia el archivo index.html al directorio de Nginx
5 COPY ./contenido/index.html /usr/share/nginx/html/index.
    html
```

Construimos la imagen personalizada:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker build -t webserver .
```

Ejecutamos un contenedor basado en la imagen personalizada:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker run --rm -d -p 8080:80 --name web webserver
```

Accedemos nuevamente a <http://localhost:8080> para verificar que el archivo HTML personalizado se muestra correctamente.

2.2.4. SUBIDA DE LA IMAGEN PERSONALIZADA A DOCKERHUB

Para compartir la imagen, la subimos a DockerHub. Primero, iniciamos sesión en DockerHub:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker login
```

Renombramos la imagen para incluir nuestro nombre de usuario:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker tag webserver <tu_usuario>/webserver:v1
```

Subimos la imagen:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker push <tu_usuario>/webserver:v1
```

Ahora, la imagen está disponible públicamente y puede descargarse en otros hosts:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker pull <tu_usuario>/webserver:v1
```

2.2.5. RESUMEN DE RESULTADOS INTERESANTES

- **Cache de la imagen:** Docker reutiliza imágenes descargadas previamente para acelerar los despliegues.
- **Bind Mount:** Permite editar archivos en el host y ver cambios inmediatos en el contenedor sin reconstruir la imagen.
- **Imagen personalizada:** Los archivos incluidos en la imagen permanecen disponibles incluso si se elimina el contenedor.
- **DockerHub:** Facilita la distribución de imágenes personalizadas en diferentes entornos.

2.3. DEMOSTRACIÓN (CASO PRÁCTICO LXC, USUARIO J BENROM)

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ docker build -t jbenrom/custom_apache_php:1.0 .
[+] Building 14.6s (5/11)
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 864B
=> [internal] load metadata for docker.io/library/debian:latest
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/7] FROM docker.io/library/debian:latest@sha256:e11072c1614c08bf88b543fcfe09d75a0426d90896408e926454e88078274fcb
=> => resolve docker.io/library/debian:latest@sha256:e11072c1614c08bf88b543fcfe09d75a0426d90896408e926454e88078274fcb
=> => sha256:e11072c1614c08bf88b543fcfe09d75a0426d90896408e926454e88078274fcb 1.85kB / 1.85kB
=> => sha256:bfee693abf500131d9c2aea2e9780a4797dc3641644bac1660b5eb9e1f1e3306 529B / 529B
=> => sha256:617f2e89852eb97cfe806541696430978f98c5e88832aef30bdecca520dba85b 1.46kB / 1.46kB
=> => sha256:7d98d813d54f6207a57721008a4081378343ad8f1b2db66c121406019171805b 49.56MB / 49.56MB
=> => extracting sha256:7d98d813d54f6207a57721008a4081378343ad8f1b2db66c121406019171805b
=> [internal] load build context
=> => transferring context: 1.40kB
=> [2/7] RUN apt-get update -qq >/dev/null && apt-get install -y -qq procps ssh apache2 php -qq >/dev/null
=> => # debconf: delaying package configuration, since apt-utils is not installed
```

Figura 2.1: Creamos la imagen de nuestro apache personalizado.

```

; The unserialize_max_depth specifies the default depth limit for unserialized
; structures. Setting the depth limit too high may result in stack overflows
; during unserialization. The unserialize_max_depth ini setting can be
; overridden by the max_depth option on individual unserialize() calls.
; A value of 0 disables the depth limit.
unserialize_max_depth = 4096

; When floats & doubles are serialized, store serialize_precision significant
; digits after the floating point. The default value ensures that when floats
; are decoded with unserialize, the data will remain the same.
; The value is also used for json_encode when encoding double values.
; If -1 is used, then dtoa mode 0 is used which automatically select the best
; precision.
serialize_precision = -1

; open_basedir, if set, limits all file operations to the defined directory
; and below. This directive makes most sense if used in a per-directory
; or per-virtualhost web server configuration file.
; Note: disables the realpath cache
; https://php.net/open-basedir
open_basedir =

; This directive allows you to disable certain functions.
; It receives a comma-delimited list of function names.
; https://php.net/disable-functions
disable_functions = exec,passthru,shell_exec,system,proc_open

; This directive allows you to disable certain classes.
; It receives a comma-delimited list of class names.
; https://php.net/disable-classes
disable_classes =

; Colors for Syntax Highlighting mode. Anything that's acceptable in
; <span style="color: ?"> would work.
; https://php.net/syntax-highlighting
highlight.string = #DD0000
highlight.comment = #FF9900
highlight.keyword = #007700
highlight.default = #0000BB
highlight.html = #000000

; If enabled, the request will be allowed to complete even if the user aborts
; the request. Consider enabling it if executing long requests, which may end up
; being interrupted by the user or a browser timing out. PHP's default behavior
; is to disable this feature.
; https://php.net/ignore-user-abort
ignore_user_abort = On

; Determines the size of the realpath cache to be used by PHP. This value should
; be increased on systems where PHP opens many files to reflect the quantity of
; the file operations performed.

```

Figura 2.2: También agregamos unas cuantas directivas a nuestro php.conf para hacer esta imagen mas segura, como `disable_functions` a la que le añadimos mas funciones peligrosas

```

jbenrom@cIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ docker run -d --rm -p 8086:80 --name servidor_apache -v ./data:/data jbenrom/custom_apache_php:1.0
c9be949e9d4fcc0ce266776f82146ec1b1afa6e28ca1a88411b0770c94d76a58
jbenrom@cIBER-jbenrom-UbuntuLXC:~/nueva_imagen$

```

Figura 2.3: Y lanzamos un contenedor a partir de esta con un bind mount, nótese que para que el contenedor pudiera escribir en `counter.txt` tuvimos que darle permisos `www-data`

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ curl localhost:8086/index.php
<pre>Hello, world /data/counter.txt
Counterx: 10
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$
```

Figura 2.4: Observamos que al visitar la dirección del contenedor, nuestro contador se va aumentando en counter.txt

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ docker run --rm -d -p 8080:80 --name web nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
a480a496ba95: Pull complete
f3ace1b8ce45: Pull complete
11d6fdd0e8a7: Pull complete
f1091da6fd5c: Pull complete
40eea07b53d8: Pull complete
6476794e50f4: Pull complete
70850b3ec6b2: Pull complete
Digest: sha256:28402db69fec7c17e179ea87882667f1e054391138f77ffaf0c3eb388efc3ffb
Status: Downloaded newer image for nginx:latest
f7a7018980c0fa1872aa35e156ac9f80299d3d3ee70499454f38bad1a8e7ead5
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS                               NAMES
f7a7018980c0   nginx     "/docker-entrypoint. ..."  4 seconds ago Up 3 seconds  0.0.0.0:8080->80/tcp, [::]:8080->80/tcp   web
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ curl http://localhost:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color:scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$
```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ mkdir -p ~/nginx/contenido
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ echo "<!DOCTYPE html><html><head><title>Prueba</title></head><body><h1>Hello World! Jose</h1></body></html>" > ~/nginx/contenido/index.html
bash: !DOCTYPE: event not found
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ echo "<html><head><title>Prueba</title></head><body><h1>Hello World! Jose</h1></body></html>" > ~/nginx/contenido/index.html
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ docker run --rm -d -p 8080:80 --name web -v ~/nginx/contenido:/usr/share/nginx/html nginx
7c
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ docker stop $(docker ps -q)
7c7018980c0
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ docker run --rm -d -p 8080:80 --name web -v ~/nginx/contenido:/usr/share/nginx/html nginx
bb3675cf34a4bf8a3ac00b442717cd2c6804ef61cc1404bf6fe5771eb7dba4b
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$ curl http://localhost:8080
<html><head><title>Prueba</title></head><body><h1>Hello World! Jose</h1></body></html>
jbenrom@CIBER-jbenrom-UbuntuLXC:~/nueva_imagen$
```

Figura 2.5: Después descargamos y montamos un contenedor de nginx, con un bind mount a /usr / share / nginx / html para agregarle nuestra pagina de bienvenida

```

GNU nano 6.2
# Usa la última versión de Nginx como base
FROM nginx:latest

# Copia el archivo index.html al directorio de Nginx
COPY /home/jbenrom/contenido/index.html /usr/share/nginx/html/index.html

```

Figura 2.6: Pero para que este cambio sea persistente en la imagen, procedimos a crear nuestra propia imagen a partir de la de nginx en la que ya se incluyera nuestra pagina de bienvenida

```

jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$ docker build -t jbenrom/webserver:1.0 .
[+] Building 0.3s (7/7) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 207B
=> [internal] load metadata for docker.io/library/nginx:latest
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 124B
=> CACHED [1/2] FROM docker.io/library/nginx:latest
=> [2/2] COPY ./index.html /usr/share/nginx/html/index.html
=> exporting to image
=> => exporting layers
=> => writing image sha256:34b99ed10326cf4ce602c019343b876197e80f475a016e68135cfe3ee15fe15d
=> => naming to docker.io/jbenrom/webserver:1.0
jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$

```

```

jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$ docker run --rm -d -p 8080:80 --name web jbenrom/webserver:1.0
89db4e01dc988fe5f809f1b627d6ac1f110a73a0c3cff6e94b82142d6088c8bd
jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$ curl http://localhost:8080
<html><head><title>Prueba</title></head><body><h1>Hello World! Jose</h1></body></html>
jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$

```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$ docker login -u nanash1
Password:
WARNING! Your password will be stored unencrypted in /home/jbenrom/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credential-stores

Login Succeeded
jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$ docker system info
Client: Docker Engine - Community
Version: 27.3.1
Context: default
Debug Mode: false
Plugins:
  buildx: Docker Buildx (Docker Inc.)
    Version: v0.17.1
    Path: /usr/libexec/docker/cli-plugins/docker-buildx
  compose: Docker Compose (Docker Inc.)
    Version: v2.29.7
    Path: /usr/libexec/docker/cli-plugins/docker-compose

Server:
Containers: 1
  Running: 1
  Paused: 0
  Stopped: 0
Images: 3
Server Version: 27.3.1
Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Using metacopy: false
  Native Overlay Diff: true
  userxattr: true
Logging Driver: json-file
Cgroup Driver: systemd
Cgroup Version: 2
Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local splunk syslog
Swarm: inactive
Runtimes: io.containerd.runc.v2 runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 7f7fdf5fed64eb6a7caf99b3e12efcf9d60e311c
runc version: v1.1.14-0-g2c9f560
init version: de40ad0
Security Options:
  apparmor
  seccomp
    Profile: builtin
  cgroupns
Kernel Version: 6.8.12-2-pve
Operating System: Ubuntu 22.04.5 LTS
```

Figura 2.7: Para poder subir nuestra imagen a DockerHub, nos logueamos en nuestra maquina, como *nanash1* (esto es porque ya que nos creamos una cuenta la creamos con un nombre de usuario que querramos usar en el futuro)

```

CPUs: 4
Total Memory: 2GiB
Name: CIBER-jbenrom-UbuntuLXC
ID: a7ff2b3b-3e54-4a42-88ae-d100a86b210a
Docker Root Dir: /var/lib/docker
Debug Mode: false
Username: nanash1
Experimental: false
Insecure Registries:
  127.0.0.0/8
Live Restore Enabled: false

jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$ docker system info | grep nanash1
Username: nanash1
jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$

```

```

jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$ docker tag jbenrom/webserver:1.0 nanash1/webserver:1.0
jbenrom@CIBER-jbenrom-UbuntuLXC:~/imagen_nginx$ docker push nanash1/webserver:1.0
The push refers to repository [docker.io/nanash1/webserver]
5bae26143590: Pushed
e4e9e9ad93c2: Pushed
6ac729401225: Pushed
8ce189049cb5: Pushed
296af1bd2844: Pushed
63d7ce983cd5: Pushing [=====>] 3.584kB
b33db0c3c3a8: Pushing [====>] 7.082MB/116.9MB
98b5f35ea9d3: Pushing [=====>] 8.11MB/74.78MB

```

Figura 2.8: Para poder subir la imagen a DockerHub tuvimos que cambiarle el nombre ya que la habíamos creado anteriormente para el repositorio de el usuario *jbenrom*

3

Docker Compose y Secrets

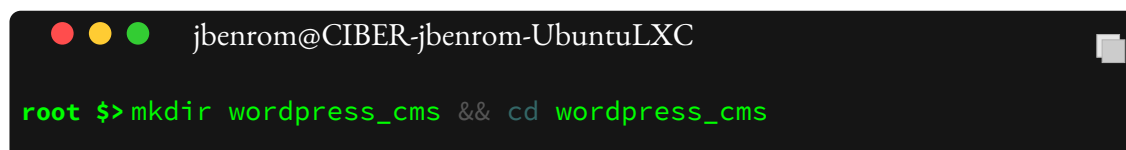
En esta sección se detalla el uso de Docker Compose para desplegar entornos multi-contenedor, mostrando cómo simplifica la configuración y gestión de servicios interdependientes. A través de tres casos prácticos—WordPress con MySQL, un entorno seguro para Redmine usando *secrets*, y una pila LAMP con **phpMyAdmin**—se exploran sus funcionalidades y ventajas.

3.1. CONFIGURACIÓN DE WORDPRESS CON MYSQL

Se comenzó configurando un entorno básico para WordPress respaldado por una base de datos MySQL. Este ejercicio ilustró cómo Docker Compose automatiza la creación de servicios que necesitan interactuar entre sí.

3.1.1. PREPARACIÓN DEL PROYECTO

Primero, se creó un directorio para organizar los archivos necesarios:

A terminal window with a dark background. The title bar shows three colored circles (red, yellow, green) and the text 'jbenrom@CIBER-jbenrom-UbuntuLXC'. The terminal content shows the command 'root \$> mkdir wordpress_cms && cd wordpress_cms' in green text.

```
root $> mkdir wordpress_cms && cd wordpress_cms
```

Dentro de este directorio, se definió un archivo `docker-compose.yml` que especificaba los servicios de WordPress y MySQL, sus volúmenes y variables de entorno.

3.1.2. DEFINICIÓN DEL ARCHIVO `docker-compose.yml`

El archivo incluía la configuración de dos servicios principales:

```
version: '3.9'

services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    volumes:
      - wordpress_data:/var/www/html
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
      WORDPRESS_DB_NAME: wordpress

volumes:
  db_data:
  wordpress_data:
```

- **Servicio db:** Proporciona una base de datos MySQL 5.7. Se utiliza un volumen llamado `db_data` para garantizar la persistencia de datos.
- **Servicio wordpress:** Aloja WordPress y está configurado para conectarse al servicio db. Se expone en el puerto 8000.

3.1.3. EJECUCIÓN Y RESULTADOS

Para desplegar los servicios, se utilizó el siguiente comando:

```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker-compose up -d
```

Esto inició los contenedores en segundo plano. Su estado se verificó con:

```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker-compose ps
```

Finalmente, accediendo a <http://localhost:8000> desde un navegador, se confirmó el correcto funcionamiento de WordPress. Este ejercicio resaltó cómo Docker Compose permite orquestar servicios de manera eficiente.

3.2. DESPLIEGUE DE REDMINE CON *SECRETS*

En este caso, se configuró un entorno multi-contenedor para Redmine y su base de datos MariaDB, incorporando *secrets* para manejar información sensible de forma segura.

3.2.1. CONFIGURACIÓN INICIAL

Como primer paso, se creó un directorio para los archivos del proyecto:

```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> mkdir redmine_project && cd redmine_project
```

3.2.2. DEFINICIÓN DE SERVICIOS Y SECRETOS

El archivo `docker-compose.yml` se diseñó para incluir el uso de *secrets*, asegurando que las contraseñas no estuvieran expuestas directamente en el archivo:

```
version: '3.9'

services:
  mariadb:
    image: bitnami/mariadb:latest
    environment:
      - MARIADB_ROOT_PASSWORD_FILE=/run/secrets/db_root_password
      - MARIADB_USER=bn_redmine
      - MARIADB_PASSWORD_FILE=/run/secrets/db_password
      - MARIADB_DATABASE=bitnami_redmine
    volumes:
      - mariadb_data:/bitnami/mariadb
    secrets:
      - db_root_password
      - db_password
  redmine:
    image: bitnami/redmine:latest
    ports:
      - "8081:3000"
    environment:
      - REDMINE_DB_MYSQL=mariadb
      - REDMINE_DB_USERNAME=bn_redmine
      - REDMINE_DB_PASSWORD_FILE=/run/secrets/db_password
      - REDMINE_DB_DATABASE=bitnami_redmine
    depends_on:
      - mariadb
    volumes:
      - redmine_data:/bitnami/redmine
    secrets:
      - db_password
```

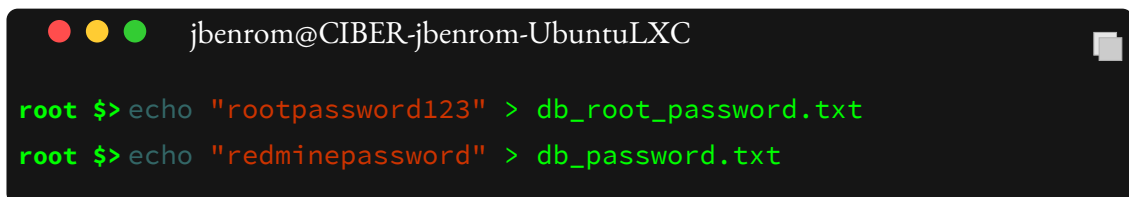
volumes:

mariadb_data:
redmine_data:

secrets:

db_root_password:
file: ./db_root_password.txt
db_password:
file: ./db_password.txt

Se crearon dos archivos para almacenar los secretos necesarios:



```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> echo "rootpassword123" > db_root_password.txt  
root $> echo "redminepassword" > db_password.txt
```

Creamos el Dockerfile para Apache y PHP

Se creó un archivo php.Dockerfile para configurar un contenedor que ejecute Apache y PHP. El contenido del archivo es el siguiente:

```
1 # php.Dockerfile  
2 FROM php:8.2-apache  
3 RUN apt update -qq > /dev/null && apt install -y -qq nano  
    git > /dev/null
```

Este Dockerfile utiliza una imagen base de PHP con Apache e instala las herramientas nano y git para facilitar la administración dentro del contenedor.

Configuramos el Directorio de la Base de Datos y cambiamos el Propietario

Se creó un directorio destinado a los archivos de la base de datos y se modificó su propietario al usuario con UID 1001, siguiendo las recomendaciones de Bitnami. Los comandos utilizados fueron:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> mkdir mariadb
root $> chown -R 1001:1001 mariadb
```

Esta configuración garantiza que el servicio de base de datos tenga los permisos adecuados para operar correctamente.

Creamos el Archivo de Prueba en PHP

En el directorio `html`, que se encuentra mapeado al directorio del servidor Apache, se creó un archivo llamado `prueba.php` para verificar que el servidor web esté ejecutando correctamente PHP.

Primero, se creó el directorio `html`:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> mkdir html
```

Luego, se creó el archivo `prueba.php` con el siguiente contenido:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> <?php
root $> phpinfo();
root $> ?>
```

Este archivo genera una página que muestra la configuración de PHP en el servidor, confirmando su correcto funcionamiento.

3.2.3. EJECUCIÓN Y RESULTADOS

Al ejecutar el entorno con:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker-compose up -d
```

Para comprobar el estado de los contenedores y el mapeo de puertos, se utilizó:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker-compose ps
```

los servicios quedaron operativos. Redmine estuvo disponible en <http://localhost:8081>. Este ejercicio subrayó la utilidad de *secrets* para proteger datos sensibles.

3.3. CREACIÓN DE UNA PILA LAMP CON phpMyAdmin

Finalmente, se desplegó un entorno LAMP (Linux, Apache, MySQL, PHP) que incluyó **phpMyAdmin** para gestionar bases de datos de manera visual.

3.3.1. DEFINICIÓN DE SERVICIOS

El archivo `docker-compose.yml` especificó los servicios principales:

```
version: "3.7"

services:
  web-server:
    build:
      dockerfile: php.Dockerfile
      context: .
    restart: always
    volumes:
      - "./html:/var/www/html/"
    ports:
      - "8080:80"
  mariadb:
    image: docker.io/bitnami/mariadb:10.6
    environment:
      - MARIADB_DATABASE=miDB
      - MARIADB_USER=miUsuario
      - MARIADB_PASSWORD_FILE=/run/secrets/db_password
      - MARIADB_ROOT_PASSWORD_FILE=/run/secrets/root_password
```

```

ports:
  - "3306:3306"
volumes:
  - ./mariadb:/bitnami/mariadb
secrets:
  - db_password
  - root_password
restart: always

phpmyadmin:
  image: phpmyadmin/phpmyadmin
  environment:
    - PMA_ARBITRARY=1
    - PMA_HOST=mariadb
  ports:
    - "8181:80"
  volumes:
    - /sessions

```

3.3.2. RESULTADOS Y CONSIDERACIONES

Tras desplegar los servicios, los siguientes recursos quedaron accesibles:

- <http://localhost:8080>: Servidor Apache alojando las aplicaciones web.
- <http://localhost:8181>: Interfaz gráfica de **phpMyAdmin** para la administración de bases de datos.

Adicionalmente, se realizaron las siguientes comprobaciones para evaluar el comportamiento del entorno:

COMPROBACIÓN DEL VOLUMEN DE LA BASE DE DATOS

Para verificar que el volumen `mariadb_data` fue creado correctamente y que los datos están siendo almacenados de forma persistente, se ejecutó:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker volume ls
```

Este comando listó los volúmenes disponibles, confirmando la existencia de `mariaadb_data`, lo cual garantiza que los datos de MariaDB persisten incluso tras reinicios o eliminaciones del contenedor.

GESTIÓN DE LOS CONTENEDORES

Para detener los contenedores sin eliminarlos, se utilizó el siguiente comando:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker-compose stop
```

Si era necesario limpiar completamente los recursos, se empleó:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker-compose down -v
```

El parámetro `-v` asegura que también se eliminen los volúmenes asociados, liberando espacio y recursos del sistema.

3.3.3. CONCLUSIONES DE ESTA PRÁCTICA

Esta configuración demostró varias ventajas del uso de Docker Compose y *secrets* en entornos multi-contenedor:

- **Persistencia de datos:** El volumen `mariaadb_data` asegura que los datos de la base de datos permanecen incluso tras detener o eliminar el contenedor.
- **Gestión segura de credenciales:** La incorporación de *secrets* evita la exposición de contraseñas en los archivos de configuración, mejorando la seguridad.
- **Administración simplificada con phpMyAdmin:** La integración de **phpMyAdmin** en un contenedor separado permite gestionar bases de datos sin necesidad de instalaciones adicionales en el sistema anfitrión.

En conjunto, este ejercicio destacó la facilidad con la que Docker Compose permite crear, gestionar y asegurar entornos complejos, optimizando tanto la configuración inicial como el mantenimiento de los servicios.

3.4. DEMOSTRACIÓN (CASO PRÁCTICO LXC, USUARIO JBENROM)

```
GNU nano 6.2
version: '3.9'

services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    volumes:
      - wordpress_data:/var/www/html
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
      WORDPRESS_DB_NAME: wordpress

volumes:
  db_data:
  wordpress_data:
```

Figura 3.1: Se crearon distintos Docker Compose, los cuales se utilizan para la gestión de aplicaciones multicontenedor, que por cierto Mario la clave *version* ya no se utiliza

```

jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms$ docker-compose up -d
command 'docker-compose' not found, but can be installed with:
sudo apt install docker-compose
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms$ docker compose up -d
WARN[0000] /home/jbenrom/wordpress_cms/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion
[+] Running 11/35
  db [#####] 121.6MB / 137.9MB Pulling
    20e4dcae4c69 Extracting [=====] 45.61MB/50.5MB
    ✓ 1c56c3d4ce74 Download complete
    ✓ e9f03a1c24ce Download complete
    ✓ 68c3898c2015 Download complete
    ✓ 6b95a940e7b6 Download complete
    ✓ 90986bb8de6e Download complete
    ✓ ae71319cb779 Download complete
    ✓ ffc89e9dfd88 Download complete
    43d05e938198 Downloading [=====] 48.13MB/56.29MB
    ✓ 064b2d298fba Download complete
    ✓ df9a4d85569b Download complete
  wordpress [#####] Pulling
    ✓ a480a496ba95 Already exists
    ✓ f79e4c25acfd Pull complete
    3cdb90068a52 Downloading [==>] 3.717MB/104.3MB
    8a7ce5258fc9 Waiting
    4c3f92bad452 Waiting
    ec2e1a686039 Waiting
    3eadf64c8bac Waiting
    f3c0c509ac91 Waiting
    68f29586bcad Waiting
    b5fe5cc54219 Waiting
    50d05b87f75c Waiting
    aa0dfe201c6c Waiting
    84359bb2faf0 Waiting
    4f4fb700ef54 Waiting
    cab2d0ebc76d Waiting
    27f0363b4ad4 Pulling fs layer
    54c8d9b377c1 Waiting
    25b784f87492 Waiting
    6cf5da888841 Waiting
    0e9c975c2d7b Waiting
    cb365401e647 Waiting
    3ee972f158cd Waiting

```

Figura 3.2: Con `docker compose up -d` se construyen y se lanzan todos los servicios del Docker Compose, y como se puede observar la etiqueta `version` esta obsoleta

```

jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms$ docker compose ps
WARN[0000] /home/jbenrom/wordpress_cms/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion

```

NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS	PORTS
wordpress_cms-db-1	mysql:5.7	"docker-entrypoint.s"	db	2 minutes ago	Up 2 minutes	3306/tcp, 33060/tcp
wordpress_cms-wordpress-1	wordpress:latest	"docker-entrypoint.s"	wordpress	2 minutes ago	Up 2 minutes	0.0.0.0:8000->80/tcp, [::]:8000->80/tcp

Figura 3.3: `docker compose ps` nos muestra los servicios del Docker Compose que están activos

```

GNU nano 6.2
services:
  web-server:
    build:
      dockerfile: php.Dockerfile
      context: .
    restart: always
    volumes:
      - "./html:/var/www/html/"
    ports:
      - "8080:80"

  mariadb:
    image: docker.io/bitnami/mariadb:10.6
    environment:
      - MARIADB_DATABASE=miDB
      - MARIADB_USER=miUsuario
      - MARIADB_PASSWORD_FILE=/run/secrets/db_password
      - MARIADB_ROOT_PASSWORD_FILE=/run/secrets/root_password
    ports:
      - "3306:3306"
    volumes:
      - ./mariadb:/bitnami/mariadb
    secrets:
      - db_password
      - root_password
    restart: always

  phpmyadmin:
    image: phpmyadmin/phpmyadmin
    environment:
      - PMA_ARBITRARY=1
      - PMA_HOST=mariadb
    ports:
      - "8181:80"
    volumes:
      - /sessions

volumes:
  mariadb_data:

secrets:
  db_password:
    file: ./db_password.txt
  root_password:
    file: ./root_password.txt

```

Figura 3.4: Tambien nos permite el uso de secrets para no hardcodear credenciales u otra información sensible

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms/lamp_stack$ echo "miPassword" > db_password.txt
echo "passwRoot" > root_password.txt
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms/lamp_stack$ nano php.Dockerfile
```

```
GNU nano 6.2 php.Dockerfile
FROM php:8.2-apache
RUN apt update -qq > /dev/null && apt install -y -qq nano git > /dev/null
```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms/lamp_stack$ mkdir mariadb
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms/lamp_stack$ chown -R 1001:1001 mariadb
chown: changing ownership of 'mariadb': Operation not permitted
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms/lamp_stack$ sudo chown -R 1001:1001 mariadb
[sudo] password for jbenrom:
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms/lamp_stack$
```

Figura 3.5: Para esta Pila LAMP necesitabamos que en la carpeta del anfitrión mapeada con el bind mount, el usuario de mariadb de la imagen de bitnami(uid 1001 tuviera los permisos necesarios

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms/lamp_stack$ mkdir html
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms/lamp_stack$ echo '<?php
phpinfo();
?>' > html/prueba.php
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms/lamp_stack$ ls html/
prueba.php
jbenrom@CIBER-jbenrom-UbuntuLXC:~/wordpress_cms/lamp_stack$
```

Figura 3.6: Agregamos un fichero php que básicamente nos muestra `phpinfo()` al directorio montado en el contenedor del apache

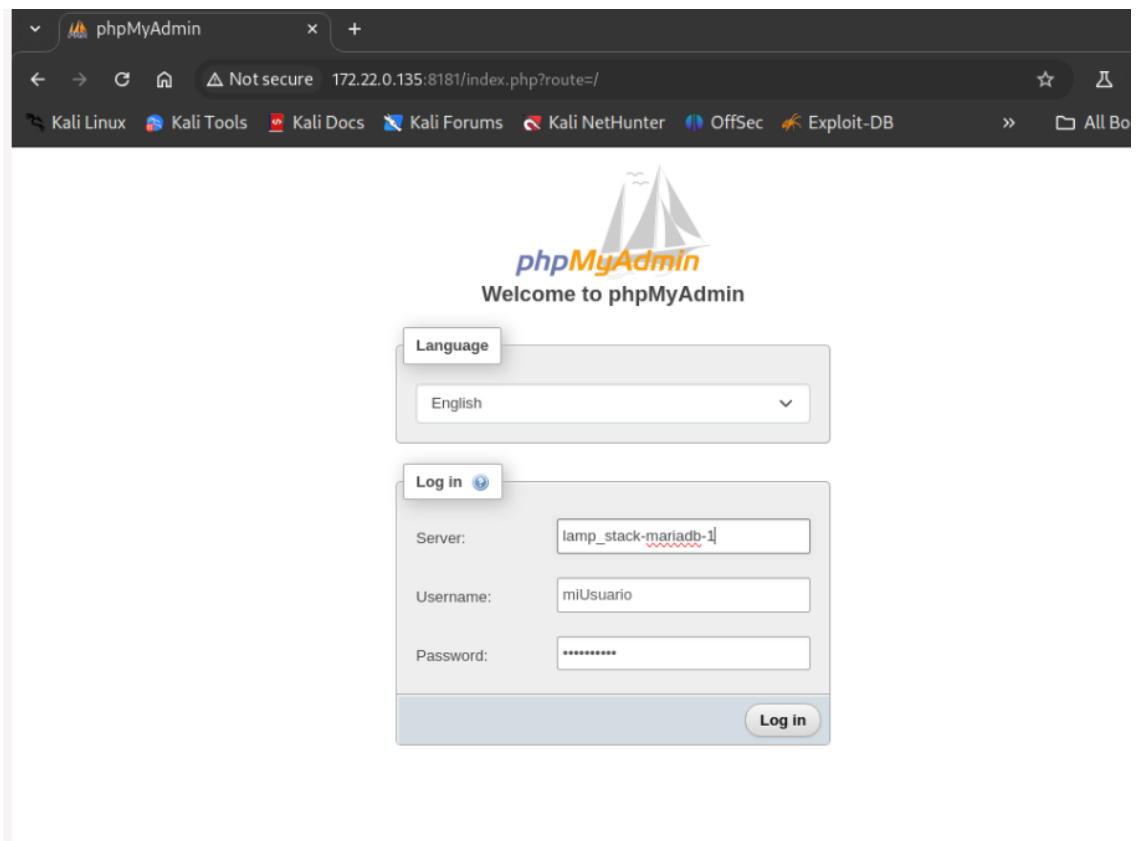
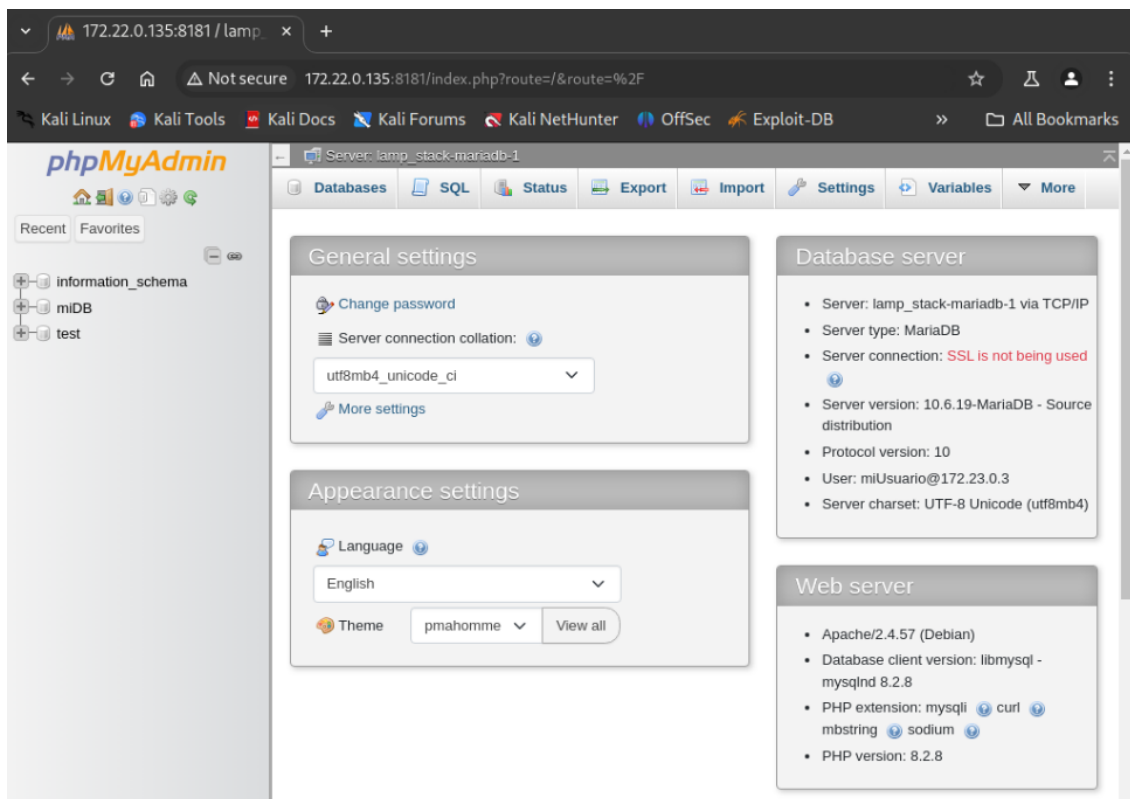


Figura 3.8: Para probar los secrets también accedimos a *phpmyadmin* con las credenciales asignadas en los *secrets*



4

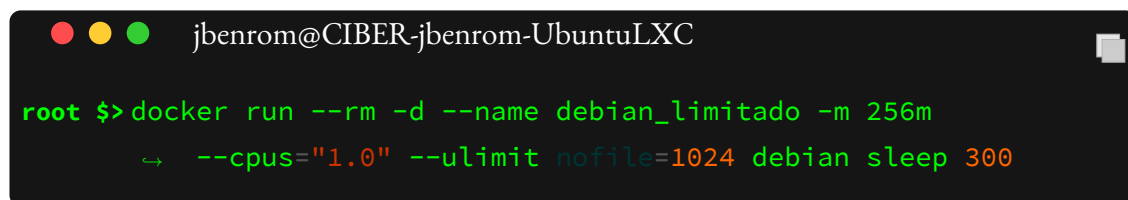
Docker Seguro

4.1. EJERCICIO DE SEGURIDAD CON DOCKER: ASEGURAR CONTENEDORES

En esta práctica se implementan diversas medidas de seguridad para los contenedores de Docker, enfocándonos en limitar el uso de recursos, ejecutar contenedores con un usuario no-root y analizar los riesgos asociados a contenedores privilegiados.

4.1.1. EVITAR ATAQUES DOS CON LIMITACIONES DE RECURSOS

Ejecutar un contenedor Debian con limitaciones de memoria y CPU utilizando los parámetros `-m`, `--cpus` y `--ulimit`.

A terminal window with a dark background. The prompt is 'jbenrom@CIBER-jbenrom-UbuntuLXC'. The command entered is 'root \$> docker run --rm -d --name debian_limitado -m 256m --cpus="1.0" --ulimit nofile=1024 debian sleep 300'. The output is not visible.

```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker run --rm -d --name debian_limitado -m 256m  
--cpus="1.0" --ulimit nofile=1024 debian sleep 300
```

Explicación de los parámetros:

- `-m 256m`: Limita el uso de memoria a 256 MB.
- `--cpus="1.0"`: Limita el uso de CPU a un núcleo.

- `--ulimit nofile=1024`: Restringe el número de archivos abiertos a 1024.
- `sleep 300`: Mantiene el contenedor ejecutándose durante 5 minutos para verificar el uso de recursos.

Para comprobar los recursos utilizados, ejecuta el siguiente comando:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker stats debian_limitado
```

Este comando muestra estadísticas de uso de memoria, CPU y otros recursos del contenedor.

4.1.2. EJECUTAR UN CONTENEDOR CON UN USUARIO DISTINTO DE ROOT

Ejecución de un contenedor con un usuario no-root utilizando el parámetro `--user`.

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run --rm -it --user 1001:1001 debian
```

Explicación del parámetro:

- `--user 1001:1001`: Asigna el UID y GID 1001 al usuario y grupo del contenedor.

Dentro del contenedor, intenta listar el contenido del directorio `/root` para comprobar permisos:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> ls /root
```

Un error de permisos indica que el usuario no-root no tiene acceso al directorio `/root`.

4.1.3. CREAR UN DOCKERFILE PARA UN USUARIO ESPECÍFICO

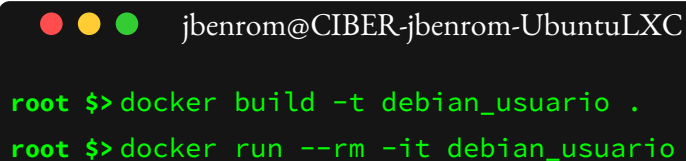
Se crea un `Dockerfile` que añade un usuario y grupo específicos y lo establece como usuario predeterminado.

```
1 FROM debian
2 RUN groupadd -g 1001 jbenrom && useradd -u 1001 -g jbenrom
   jbenrom
3 USER jbenrom
```

Explicación de los comandos:

- `groupadd -g 1001 jbenrom`: Crea un grupo `jbenrom` con GID `1001`.
- `useradd -u 1001 -g jbenrom jbenrom`: Crea un usuario `jbenrom` con UID `1001` y lo asigna al grupo.
- `USER jbenrom`: Configura el contenedor para que se ejecute como `jbenrom`.

Construir la imagen y verificar su ejecución:



```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker build -t debian_usuario .
root $> docker run --rm -it debian_usuario
```

Dentro del contenedor, utiliza los comandos `id` y `whoami` para verificar el usuario en ejecución:

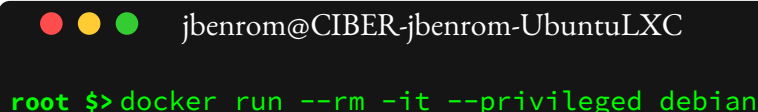


```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> id
root $> whoami
```

4.1.4. EJECUTAR UN CONTENEDOR PRIVILEGIADO (¡CON PRECAUCIÓN!)

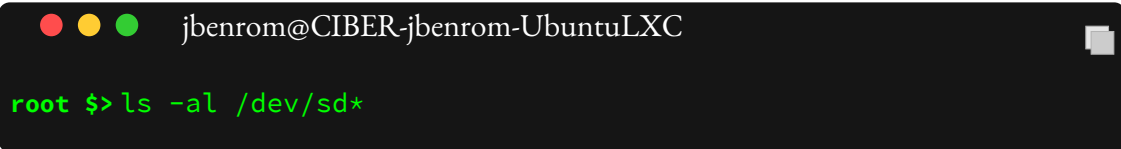
Ejecución de un contenedor con el parámetro `--privileged`, que otorga permisos elevados.



```
jbenrom@CIBER-jbenrom-UbuntuLXC

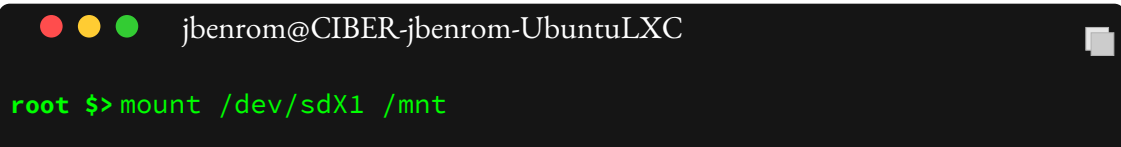
root $> docker run --rm -it --privileged debian
```

Dentro del contenedor, intenta listar los discos del sistema host:

A terminal window with a dark background. The prompt is 'jbenrom@CIBER-jbenrom-UbuntuLXC'. The command 'root \$> ls -al /dev/sd*' is entered in green text.

```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> ls -al /dev/sd*
```

También puedes montar particiones del sistema host (solo para entornos de prueba):

A terminal window with a dark background. The prompt is 'jbenrom@CIBER-jbenrom-UbuntuLXC'. The command 'root \$> mount /dev/sdX1 /mnt' is entered in green text.

```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> mount /dev/sdX1 /mnt
```

Esto demuestra los riesgos de seguridad asociados con contenedores privilegiados, ya que pueden acceder y modificar recursos críticos del host.

4.1.5. RESUMEN

- **Limitación de Recursos:** Establecer límites de CPU, memoria y número de archivos abiertos ayuda a evitar abusos de recursos y posibles ataques DoS.
- **Ejecución sin Privilegios:** Usar un usuario no-root minimiza el impacto potencial de una vulnerabilidad en el contenedor.
- **Contenedores Privilegiados:** Son útiles para pruebas avanzadas, pero deben evitarse en producción debido a los riesgos de seguridad.

4.2. IMAGEN DOCKER OPTIMIZADA: MULTI-STAGE BUILDS

En esta sección se realiza un análisis comparativo entre la creación de una imagen Docker tradicional y el uso de la técnica de *multi-stage builds*. Se demuestra cómo esta última permite reducir el tamaño de la imagen final al excluir herramientas de desarrollo innecesarias en entornos de producción.

4.2.1. PASO 1: IMAGEN SIN MULTI-STAGE BUILDS

En primera instancia, se desarrolló una imagen Docker convencional, que incluye tanto el binario de la aplicación como las herramientas de desarrollo.

CÓDIGO FUENTE DE LA APLICACIÓN GO

El archivo `main.go` fue creado dentro del directorio `src`. Este archivo contiene el siguiente código, que representa una aplicación básica en Go:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello_world_from_Go!")
}
```

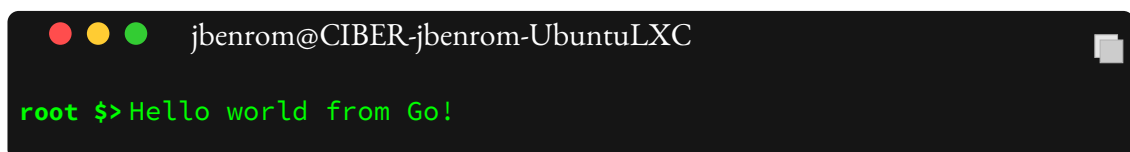
DOCKERFILE SIN MULTI-STAGE BUILDS

El primer `Dockerfile` diseñado utiliza la imagen base `golang:1.14.2-alpine` para compilar la aplicación Go directamente dentro del contenedor. Este diseño incorpora tanto el proceso de compilación como las herramientas de desarrollo en la misma imagen.

```
1 FROM golang:1.14.2-alpine
2 WORKDIR /src
3 COPY src .
4 RUN go build -o /out/helloworld .
5 ENTRYPOINT ["/out/helloworld"]
```

CONSTRUCCIÓN Y VERIFICACIÓN DE LA IMAGEN

La imagen resultante fue construida y ejecutada. La salida del contenedor confirmó que la aplicación Go funcionaba correctamente, mostrando el mensaje:

A terminal window with a dark background. The title bar shows three colored circles (red, yellow, green) and the text 'jbenrom@CIBER-jbenrom-UbuntuLXC'. The terminal content shows a green prompt 'root \$>' followed by the output 'Hello world from Go!' in green text.

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $>Hello world from Go!
```

El tamaño de la imagen, comprobado mediante el comando `docker images`, fue relativamente grande debido a la inclusión de las herramientas de desarrollo utilizadas durante la compilación.

4.2.2. IMAGEN CON MULTI-STAGE BUILDS

Para optimizar el tamaño de la imagen, se implementó un enfoque basado en *multi-stage builds*. Este método separa el proceso de compilación en una etapa intermedia y genera una imagen final más ligera, adecuada para entornos de producción.

DOCKERFILE CON MULTI-STAGE BUILDS

El Dockerfile modificado consta de dos etapas: una etapa de `builder`, que contiene las herramientas de desarrollo necesarias para compilar la aplicación, y una etapa de producción basada en `alpine:3.12`, diseñada para ser más eficiente y ligera.

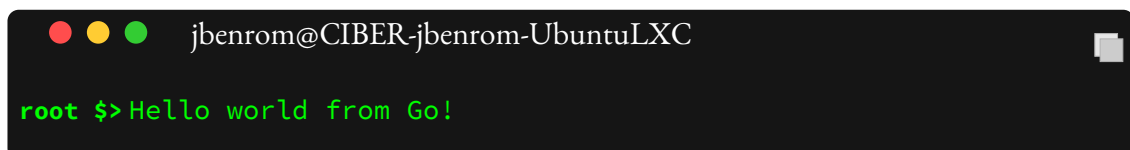
```
1 FROM golang:1.14.2 - alpine AS builder
2 WORKDIR /src
3 COPY src .
4 RUN go build -o /out/helloworld .
5
6 FROM alpine:3.12 AS bin
7 COPY --from=builder /out/helloworld /
8 ENTRYPOINT ["/helloworld"]
```

CONSTRUCCIÓN Y COMPARACIÓN DE TAMAÑOS

Tras construir la imagen multi-stage, se realizó una comparación de tamaños entre las imágenes `helloworld:single` y `helloworld:multi`. Como era de esperar, la imagen `helloworld:multi` resultó ser considerablemente más pequeña al contener únicamente el binario de la aplicación.

VERIFICACIÓN DE LA IMAGEN OPTIMIZADA

Al ejecutar la imagen optimizada, se verificó que la aplicación continuaba funcionando correctamente, mostrando el mensaje:



```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $>Hello world from Go!
```

4.2.3. RESUMEN

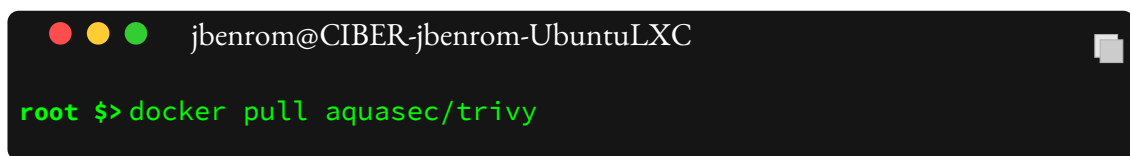
- La imagen sin *multi-stage builds* incluye tanto el binario de la aplicación como las herramientas de desarrollo, lo que incrementa significativamente su tamaño.
- La imagen con *multi-stage builds* separa las etapas de compilación y producción, resultando en una imagen final más ligera y adecuada para entornos de producción.
- Este enfoque optimiza el uso de recursos y minimiza riesgos al evitar herramientas innecesarias en producción.

4.3. ESCANEADO DE VULNERABILIDADES CON TRIVY

En esta sección se documenta el uso de *Trivy*, una herramienta de escaneo de vulnerabilidades en imágenes de contenedores Docker. El objetivo es analizar imágenes previamente construidas, comparar su nivel de seguridad y evaluar diferentes imágenes base utilizadas en entornos de contenedores.

4.3.1. DESCARGA DE LA IMAGEN DE TRIVY

Para realizar el escaneo, se utilizó la imagen oficial de Trivy disponible en Docker Hub. La descarga de la imagen se llevó a cabo con el siguiente comando:



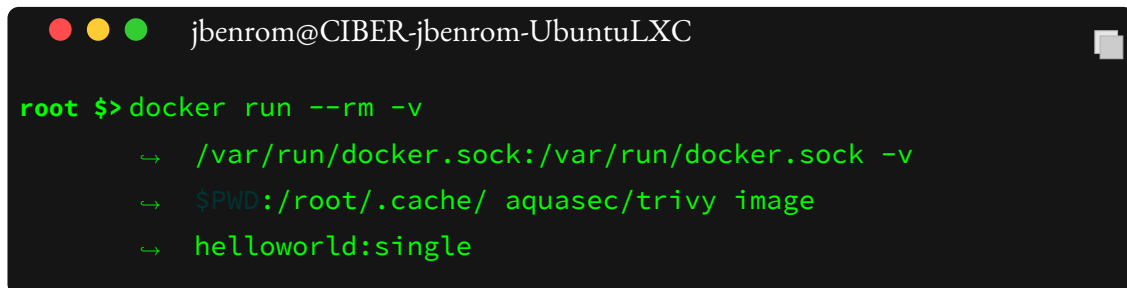
```
jbenrom@CIBER-jbenrom-UbuntuLXC  
root $> docker pull aquasec/trivy
```

4.3.2. ESCANEADO DE LAS IMÁGENES `helloworld:single` y `helloworld:multi`

El análisis incluyó dos imágenes creadas previamente: `helloworld:single` y `helloworld:multi`. Ambas imágenes fueron escaneadas utilizando Trivy, montando el directorio de caché para evitar descargas repetidas de la base de datos de vulnerabilidades, así como el *socket* de Docker para acceder a las imágenes locales.

ESCANEEO DE `helloworld:single`

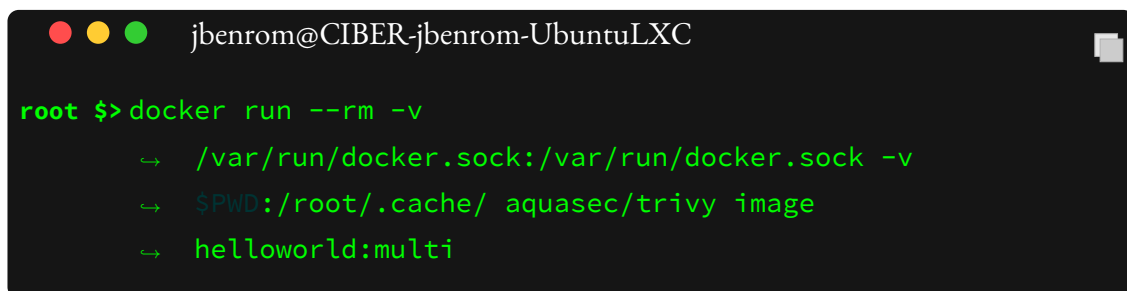
El comando ejecutado para escanear esta imagen fue el siguiente:

A terminal window with a dark background and green text. The prompt is 'root \$' and the command is 'docker run --rm -v /var/run/docker.sock:/var/run/docker.sock -v \$PWD:/root/.cache/ aquasec/trivy image helloworld:single'. The window title is 'jbenrom@CIBER-jbenrom-UbuntuLXC'.

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run --rm -v
    ↪ /var/run/docker.sock:/var/run/docker.sock -v
    ↪ $PWD:/root/.cache/ aquasec/trivy image
    ↪ helloworld:single
```

ESCANEEO DE `helloworld:multi`

El escaneo de la imagen optimizada se llevó a cabo con el comando:

A terminal window with a dark background and green text. The prompt is 'root \$' and the command is 'docker run --rm -v /var/run/docker.sock:/var/run/docker.sock -v \$PWD:/root/.cache/ aquasec/trivy image helloworld:multi'. The window title is 'jbenrom@CIBER-jbenrom-UbuntuLXC'.

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker run --rm -v
    ↪ /var/run/docker.sock:/var/run/docker.sock -v
    ↪ $PWD:/root/.cache/ aquasec/trivy image
    ↪ helloworld:multi
```

RESULTADOS COMPARATIVOS

La comparación de los resultados de ambos escaneos indicó que la imagen `helloworld:single` contenía más vulnerabilidades. Esto se debe a la inclusión de herramientas de desarrollo y dependencias adicionales, que incrementan la superficie de ataque. En contraste, la imagen `helloworld:multi`, optimizada mediante *multi-stage builds*, mostró menos vulnerabilidades al excluir componentes innecesarios para entornos de producción.

4.3.3. ESCANEEO DE DIFERENTES VERSIONES DE UNA MISMA IMAGEN

Se exploró cómo las vulnerabilidades varían entre diferentes versiones de una misma imagen. Como ejemplo, se analizaron las versiones `nginx:latest` y `nginx:1.18` con los siguientes comandos:


```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker run --rm -v
    => /var/run/docker.sock:/var/run/docker.sock -v
    => $PWD:/root/.cache/ aquasec/trivy image nginx:latest
root $> docker run --rm -v
    => /var/run/docker.sock:/var/run/docker.sock -v
    => $PWD:/root/.cache/ aquasec/trivy image nginx:1.18
```

El análisis reveló que las versiones más antiguas suelen presentar un mayor número de vulnerabilidades debido a la inclusión de paquetes desactualizados. Esto pone de manifiesto la importancia de mantener actualizadas las imágenes utilizadas en producción.

4.3.4. ESCANEADO DE IMÁGENES BASE POPULARES

Se realizó un escaneo de vulnerabilidades en imágenes base comunes como `debian:latest`, `ubuntu:latest` y `alpine:latest`. Los comandos ejecutados fueron los siguientes:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker run --rm -v
    => /var/run/docker.sock:/var/run/docker.sock -v
    => $PWD:/root/.cache/ aquasec/trivy image debian:latest
root $> docker run --rm -v
    => /var/run/docker.sock:/var/run/docker.sock -v
    => $PWD:/root/.cache/ aquasec/trivy image ubuntu:latest
root $> docker run --rm -v
    => /var/run/docker.sock:/var/run/docker.sock -v
    => $PWD:/root/.cache/ aquasec/trivy image alpine:latest
```

COMPARACIÓN DE RESULTADOS

Las imágenes basadas en Alpine presentaron un menor número de vulnerabilidades debido a su reducido tamaño y la ausencia de dependencias innecesarias. Por otro lado, Debian y Ubuntu mostraron más vulnerabilidades, relacionadas con su base de paquetes más amplia y compleja.

4.3.5. INTERPRETACIÓN DE RESULTADOS

Los resultados obtenidos de Trivy permitieron identificar lo siguiente:

- **Localización de vulnerabilidades:** Trivy señaló los paquetes específicos afectados y las versiones en las que se corrigieron.
- **CVE:** Cada vulnerabilidad se acompañó de su identificador *Common Vulnerabilities and Exposures* (CVE), proporcionando enlaces para obtener más detalles sobre los riesgos asociados.
- **Versiones más antiguas:** Imágenes y sistemas base desactualizados, como *debian* o *ubuntu*, mostraron más vulnerabilidades debido a la inclusión de software obsoleto.

Este análisis destaca la importancia de elegir imágenes base ligeras y actualizadas, como *Alpine*, y de optimizar las imágenes personalizadas para entornos de producción.

4.4. FIRMA DE IMÁGENES DOCKER Y DOCKER CONTENT TRUST

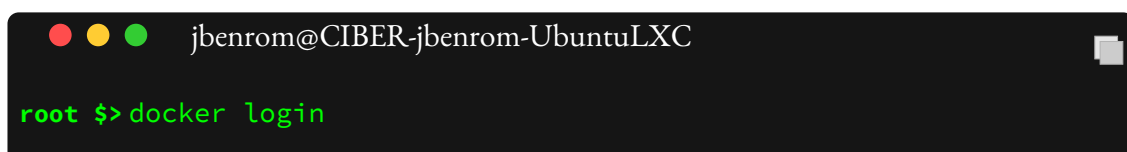
En esta sección se describe el proceso de firma de imágenes Docker utilizando *Docker Content Trust* (DCT). Esta funcionalidad permite garantizar la integridad y autenticidad de las imágenes mediante la firma criptográfica, asegurando que no han sido manipuladas durante su transferencia.

4.4.1. CONFIGURACIÓN INICIAL

Para implementar DCT, es necesario realizar algunas configuraciones iniciales tanto en el cliente de Docker como en Notary, la herramienta encargada de administrar claves y firmas.

INICIO DE SESIÓN EN DOCKER HUB

Se comenzó realizando un inicio de sesión en Docker Hub para habilitar la interacción con los repositorios personales:

A terminal window with a dark background. The title bar shows three colored circles (red, yellow, green) and the text 'jbenrom@CIBER-jbenrom-UbuntuLXC'. The terminal content shows 'root \$> docker login' in green text.

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker login
```

La verificación del usuario autenticado se realizó con el siguiente comando:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker system info
```

CONFIGURACIÓN DE NOTARY

Se confirmó que el cliente Notary estaba correctamente configurado. El archivo `~/.notary/config.json` contenía la siguiente estructura:

```
1 {
2   "trust_dir": "~/.docker/trust"
3 }
```

4.4.2. GESTIÓN DE CLAVES DE DELEGACIÓN

Para firmar imágenes, se generaron claves de delegación con Docker Trust. Esto incluye un par de claves pública y privada asociadas al usuario firmante:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker trust key generate nanash1
```

La clave pública se guardó en el directorio de trabajo actual como `<nombre_usuario>.pub`, mientras que la clave privada se almacenó en `~/.docker/trust/private`. Las claves creadas fueron verificadas con:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> notary key list
```

CREACIÓN DE REGLAS DE DELEGACIÓN

Para asociar un firmante al repositorio, se utilizó el siguiente comando:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker trust signer add --key nanash1.pub nanash1
      ↪ nanash1/primer_repositorio
```

Esto inicializó el repositorio generando una *clave raíz* (root key) y una *clave de repositorio* (repository key), que se verificaron con el cliente Notary.

4.4.3. FIRMA Y SUBIDA DE IMÁGENES FIRMADAS

Para probar la firma, se etiquetó la imagen `helloworld:multi` con el nombre completo del repositorio en Docker Hub:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker tag helloworld:multi
      ↪ <nombre\_usuario>/helloworld:2.0
```

La firma y la subida de la imagen al repositorio se llevaron a cabo con:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker trust sign nanash1/helloworld:2.0
```

Alternativamente, se habilitó el uso explícito de DCT durante el comando `docker push`:

```
jbenrom@CIBER-jbenrom-UbuntuLXC

root $> docker push --disable-content-trust=false
      ↪ <nombre\_usuario>/helloworld:2.0
```

4.4.4. VERIFICACIÓN DE FIRMAS

Se eliminó la imagen localmente para comprobar la firma al volver a descargarla desde Docker Hub:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker rmi nanash1/helloworld:2.0
```

La descarga de la imagen firmada desde Docker Hub validó automáticamente su integridad:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> docker pull nanash1/helloworld:2.0
```

Cualquier intento de descargar imágenes no firmadas resultó en un error, confirmando la eficacia de Docker Content Trust.

4.4.5. ACTIVACIÓN GLOBAL DE DOCKER CONTENT TRUST

Para garantizar que todas las operaciones futuras de `pull`, `run` y `build` verificaran las firmas, se habilitó Docker Content Trust globalmente:

```
jbenrom@CIBER-jbenrom-UbuntuLXC
root $> echo "export DOCKER_CONTENT_TRUST=1" >> ~/.bashrc
root $> source ~/.bashrc
```

4.4.6. CONCLUSIÓN

La integración de Docker Content Trust permite validar la integridad y autenticidad de las imágenes utilizadas en un entorno de producción. Esto asegura que las imágenes no han sido manipuladas durante su distribución, fortaleciendo la seguridad general del ecosistema de contenedores.

4.5. DEMOSTRACIÓN (CASO PRÁCTICO LXC, USUARIO JBENROM)

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker run --rm -d --name debian_limitado -m 256m --cpus="1.0" --ulimit nofile=1024 debian sleep 300
Unable to find image 'debian:latest' locally

latest: Pulling from library/debian
7d98d813d54f: Already exists
Digest: sha256:e11072c1614c08bf88b543cfe09d75a0426d90896408e926454e88078274fcb
Status: Downloaded newer image for debian:latest
e19f16b3c46a03a1679947ed85482afbb279084289a8dbb8ad46b72564e529a7
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS                               NAMES
e19f16b3c46a   debian    "sleep 300"              5 seconds ago Up 5 seconds                                debian_limitado
e70496be12c8   lamp_stack-web-server  "docker-php-entrypoi-"  14 hours ago  Up 5 minutes  0.0.0.0:8080->80/tcp, [::]:8080->80/tcp  lamp_stack-web-server-1
bc15b10ab5bd   bitnami/mariadb:10.6  "/opt/bitnami/script-"  14 hours ago  Up 5 minutes  0.0.0.0:3306->3306/tcp, :::3306->3306/tcp  lamp_stack-mariadb-1
```

Figura 4.1: Contenedor de *debian* con recursos limitados para prevenir ataques de denegación de servicios

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
e19f16b3c46a	debian_limitado	0.00%	2.355MiB / 256MiB	0.92%	1.32kB / 0B	9.9MB / 0B	1

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker run --rm -it --user 1001:1001 debian
I have no name!@bc08a90b4403:/$ ls /root
ls: cannot open directory '/root': Permission denied
I have no name!@bc08a90b4403:/$
```

Figura 4.2: Podemos ver que cuando accedemos como user, no tenemos permisos de super usuario, además me parece curioso que el uid 1001 con el que accedemos nos pone que no tiene nombre

```
GNU nano 6.2 Dockerfile
FROM debian
RUN groupadd -g 1001 grupo_jbenrom && useradd -u 1001 -g grupo_jbenrom jbenrom
USER jbenrom
```

Figura 4.3: Imagen personalizada a partir de *debian* a la que le añadimos un usuario y un grupo y hacemos que el contenedor se ejecute con este

```

jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker$ docker build -t jbenrom/debian:1.0 . && docker run --rm -it jbenrom/debian:1.0
[+] Building 0.7s (6/6) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 141B
=> [internal] load metadata for docker.io/library/debian:latest
=> [internal] load .dockerignore
=> => transferring context: 2B
=> CACHED [1/2] FROM docker.io/library/debian:latest
=> [2/2] RUN groupadd -g 1001 grupo_jbenrom && useradd -u 1001 -g grupo_jbenrom jbenrom
=> exporting to image
=> => exporting layers
=> => writing image sha256:93a9cde32a4067c2312cc687ec4261afcc0b5255b763068bd866f3efc139234b
=> => naming to docker.io/jbenrom/debian:1.0
jbenrom@88998cc6e9d8f:/$ id
uid=1001(jbenrom) gid=1001(grupo_jbenrom) groups=1001(grupo_jbenrom)
jbenrom@88998cc6e9d8f:/$ whoami
jbenrom
jbenrom@88998cc6e9d8f:/$ █

```

Figura 4.4: Vemos que al entrar en la shell del contenedor y ejecutar whoami nos devuelve el usuario que creamos

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker$ docker run --rm -it --privileged debian
root@fdcd0d0268c97:/# lsblk
```

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPOINTS
sda	8:0	0	931.5G	0	disk	
`-sda1	8:1	0	931.5G	0	part	
sdb	8:16	0	1.6T	0	disk	
-sdb1	8:17	0	1007K	0	part	
-sdb2	8:18	0	1G	0	part	
`-sdb3	8:19	0	1.6T	0	part	
zd0	230:0	0	40G	0	disk	
-zd0p1	230:1	0	487M	0	part	
-zd0p2	230:2	0	1K	0	part	
-zd0p3	230:3	0	8G	0	part	
`-zd0p5	230:5	0	31.5G	0	part	
zd16	230:16	0	10G	0	disk	
`-zd16p1	230:17	0	10G	0	part	
zd32	230:32	0	32G	0	disk	
`-zd32p1	230:33	0	32G	0	part	
zd48	230:48	0	10G	0	disk	
`-zd48p1	230:49	0	10G	0	part	
zd64	230:64	0	32G	0	disk	
`-zd64p1	230:65	0	32G	0	part	
zd80	230:80	0	19.5G	0	disk	
-zd80p1	230:81	0	1M	0	part	
`-zd80p2	230:82	0	19.5G	0	part	
zd96	230:96	0	40G	0	disk	
-zd96p1	230:97	0	487M	0	part	
-zd96p2	230:98	0	1K	0	part	
-zd96p3	230:99	0	8G	0	part	
`-zd96p5	230:101	0	31.5G	0	part	
zd112	230:112	0	10G	0	disk	
`-zd112p1	230:113	0	10G	0	part	
zd128	230:128	0	10G	0	disk	
`-zd128p1	230:129	0	10G	0	part	
zd144	230:144	0	19.5G	0	disk	
-zd144p1	230:145	0	1M	0	part	
`-zd144p2	230:146	0	19.5G	0	part	
zd160	230:160	0	19.5G	0	disk	
-zd160p1	230:161	0	1M	0	part	
`-zd160p2	230:162	0	19.5G	0	part	
zd176	230:176	0	19.5G	0	disk	
-zd176p1	230:177	0	1M	0	part	
`-zd176p2	230:178	0	19.5G	0	part	
zd192	230:192	0	19.5G	0	disk	
-zd192p1	230:193	0	1M	0	part	
`-zd192p2	230:194	0	19.5G	0	part	
zd208	230:208	0	19.5G	0	disk	
-zd208p1	230:209	0	1M	0	part	
`-zd208p2	230:210	0	19.5G	0	part	
zd224	230:224	0	19.5G	0	disk	
-zd224p1	230:225	0	1M	0	part	
`-zd224p2	230:226	0	19.5G	0	part	
zd240	230:240	0	19.5G	0	disk	
-zd240p1	230:241	0	1M	0	part	
`-zd240p2	230:242	0	19.5G	0	part	

Figura 4.5: También se realizaron pruebas de alcance de un contenedor privilegiado, y podemos ver que puede acceder (e incluso interactuar, aunque esto no se pudo comprobar dado que era un LXC, contenedor dentro de contenedor) al hardware del anfitrión


```

jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker$ cd ..
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ mkdir -p multi_stage
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ cd multi_stage/
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ echo '// main.go
package main

import "fmt"

func main() {
    fmt.Println("Hello world from Go!")
}
' > main.go
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ ls
main.go
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ █

```

```

GNU nano 6.2 Dockerfile
↑ Dockerfile sin multi-stage
FROM golang:1.14.2-alpine
WORKDIR /src
COPY src .
RUN go build -o /out/helloworld .
ENTRYPOINT ["/out/helloworld"]
█

```

Figura 4.6: Se probó la eficiencia de las imágenes multistage de Docker, creando primero una imagen *single* de Golang, con todo el entorno de desarrollo para compilar, y después una multistage.

```

jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ mkdir -p src
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ cp ./main.go src
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ ls -lr
total 12
drwxrwxr-x 2 jbenrom jbenrom 4096 Nov  4 14:32 src
-rw-rw-r-- 1 jbenrom jbenrom   96 Nov  4 14:26 main.go
-rw-rw-r-- 1 jbenrom jbenrom  144 Nov  4 14:28 Dockerfile
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ tree
.
|-- Dockerfile
|-- main.go
'-- src
    '-- main.go

1 directory, 3 files
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$

```

```

jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ docker build -t helloworld:single .
[+] Building 10.9s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 183B
=> [internal] load metadata for docker.io/library/golang:1.14.2-alpine
=> [auth] library/golang:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/4] FROM docker.io/library/golang:1.14.2-alpine@sha256:9b3ad7928626126b72b916609ad081cfb6c0149f6e60cef7fcl9e15a0d1e973
=> => resolve docker.io/library/golang:1.14.2-alpine@sha256:9b3ad7928626126b72b916609ad081cfb6c0149f6e60cef7fcl9e15a0d1e973
=> => sha256:cbdbe7a5bc2a134ca8ec91be58565ec07d037386d1f1d8385412d224deafca08 2.81MB / 2.81MB
=> => sha256:408f875501273f3af2a9cbff2a23e736585641e73da80dd81712518b28e7843c 301.28kB / 301.28kB
=> => sha256:fe522b08c9798748151fec9b7a908aca712cd102cfcb8ed79d57405b71e6cc4 153B / 153B
=> => sha256:9b3ad7928626126b72b916609ad081cfb6c0149f6e60cef7fcl9e15a0d1e973 1.65kB / 1.65kB
=> => sha256:b0678825431fd5e27a211e0d7581d5f24ced6b4d25ac1411416fa8044fa6c51 1.36kB / 1.36kB
=> => sha256:dda4232b2bd580bbf633be12d62e8d0e00f6b7bd60ea6faee157bad1809c53c4 3.83kB / 3.83kB
=> => sha256:618fff1cf170e1785ab64028237182717bc1e1287d03cf0888e424b7563ae5df 132.01MB / 132.01MB
=> => sha256:0d8b518583db0dc830a3a43c739d6cc91b7610c09d9eba918ae54b20a1dcd18c 126B / 126B
=> => extracting sha256:cbdbe7a5bc2a134ca8ec91be58565ec07d037386d1f1d8385412d224deafca08
=> => extracting sha256:408f875501273f3af2a9cbff2a23e736585641e73da80dd81712518b28e7843c
=> => extracting sha256:fe522b08c9798748151fec9b7a908aca712cd102cfcb8ed79d57405b71e6cc4
=> => extracting sha256:618fff1cf170e1785ab64028237182717bc1e1287d03cf0888e424b7563ae5df
=> => extracting sha256:0d8b518583db0dc830a3a43c739d6cc91b7610c09d9eba918ae54b20a1dcd18c
=> [internal] load build context
=> => transferring context: 161B
=> [2/4] WORKDIR /src
=> [3/4] COPY src .
=> [4/4] RUN go build -o /out/helloworld .
=> exporting to image
=> => writing image sha256:d64177cb5b719888887167a8c4cbd909c0cd5af7d8c242456a83d5f1ae7b8085
=> => naming to docker.io/library/helloworld:single
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ docker run --rm helloworld:single
Hello world from Go!
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$

```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
helloworld	single	d64177cb5b71	7 days ago	372MB
jbenrom/debian	1.0	93a9cde32a40	7 days ago	117MB
lamp_stack-web-server	latest	2d3fb50b80f7	8 days ago	566MB
jbenrom/webserver	1.0	34b99ed10326	8 days ago	192MB
nanash1/webserver	1.0	34b99ed10326	8 days ago	192MB
jbenrom/custom_apache_php	1.0	b9f109c76230	8 days ago	322MB
bitnami/mariadb	10.6	9132c90dbf8f	2 weeks ago	390MB
debian	latest	617f2e89852e	3 weeks ago	117MB
wordpress	latest	4c9b15c9a8ae	3 weeks ago	697MB
nginx	latest	3b25b682ea82	5 weeks ago	192MB
mysql	5.7	5107333e08a8	11 months ago	501MB
phpmyadmin/phpmyadmin	latest	933569f3a9f6	16 months ago	562MB

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~$
```

```
# Dockerfile con multi-stage builds
FROM golang:1.14.2-alpine AS builder
WORKDIR /src
COPY src .
RUN go build -o /out/helloworld .

FROM alpine:3.12 AS bin
COPY --from=builder /out/helloworld /
ENTRYPOINT ["/helloworld"]
```

Figura 4.7: En la imagen multistage se utilizó la imagen de GoLang para compilar nuestro script, y después se usó una imagen base de `debian` como entorno para el funcionamiento de nuestro servicio de `helloworld`

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ docker build -t helloworld:multi -f multi.Dockerfile .
[+] Building 2.6s (14/14) FINISHED
=> [internal] load build definition from multi.Dockerfile
=> => transferring dockerfile: 266B
=> [internal] load metadata for docker.io/library/alpine:3.12
=> [internal] load metadata for docker.io/library/golang:1.14.2-alpine
=> [auth] library/golang:pull token for registry-1.docker.io
=> [auth] library/alpine:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [builder 1/4] FROM docker.io/library/golang:1.14.2-alpine@sha256:9b3ad7928626126b72b916609ad081cfb6c0149f6e60cef7fc1e9e15a0d1e973
=> [internal] load build context
=> => transferring context: 53B
=> [bin 1/2] FROM docker.io/library/alpine:3.12@sha256:c75ac27b49326926b803b9ed43bf088bc220d22556de1bc5f72d742c91398f69
=> => resolve docker.io/library/alpine:3.12@sha256:c75ac27b49326926b803b9ed43bf088bc220d22556de1bc5f72d742c91398f69
=> => sha256:1b7ca6aa1ddfe716f3694edb811ab35114db3e93f3ee38d7dab6b4d9270-b0c 2.81MB / 2.81MB
=> => sha256:c75ac27b49326926b803b9ed43bf088bc220d22556de1bc5f72d742c91398f69 1.64kB / 1.64kB
=> => sha256:cb64bbe7fa61366c234e1090e91427314ee13ec6420e9426cfe7f314056813 528B / 528B
=> => sha256:24c8ece58a1aa807c0d8ea121f91cee2efba99624d0a8aed732155fb31f28993 1.47kB / 1.47kB
=> => extracting sha256:1b7ca6aa1ddfe716f3694edb811ab35114db3e93f3ee38d7dab6b4d9270cb0c
=> CACHED [builder 2/4] WORKDIR /src
=> CACHED [builder 3/4] COPY src .
=> CACHED [builder 4/4] RUN go build -o /out/helloworld .
=> [bin 2/2] COPY --from=builder /out/helloworld /
=> exporting to image
=> => exporting layers
=> => writing image sha256:146b98bcb75737482939e4bb043f8979d7f09ab3b6d81b40fb4b6a4bf700cbf9
=> => naming to docker.io/library/helloworld:multi
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
helloworld	multi	146b98bcb757	51 seconds ago	7.65MB
helloworld	single	d64177cb5b71	7 days ago	372MB
jbenrom/debian	1.0	93a9cde32a40	7 days ago	117MB
lamp_stack-web-server	latest	2d3fb50b80f7	8 days ago	566MB
jbenrom/webserver	1.0	34b99ed10326	8 days ago	192MB
nanash1/webserver	1.0	34b99ed10326	8 days ago	192MB
jbenrom/custom_apache_php	1.0	b9f109c76230	8 days ago	322MB
bitnami/mariadb	10.6	9132c90dbf8f	2 weeks ago	390MB
debian	latest	617f2e89852e	3 weeks ago	117MB
wordpress	latest	4c9b15c9a8ae	3 weeks ago	697MB
nginx	latest	3b25b682ea82	5 weeks ago	192MB
mysql	5.7	5107333e08a8	11 months ago	501MB
phpmyadmin/phpmyadmin	latest	933569f3a9f6	16 months ago	562MB

Figura 4.8: Al comparar las dos imágenes observamos que la multistage es mucho mas ligera ya que no incluye todo el entorno de desarrollo de Golang

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/multi_stage$ docker run --rm helloworld:multi
Hello world from Go!
```

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
zlib	CVE-2022-37434	CRITICAL	fixed	1.2.12-r0	1.2.12-r2	zlib: heap-based buffer over-read and overflow in inflate() in inflate.c via a... https://avd.aquasec.com/nvd/cve-2022-37434

79

jenrom@CIRES-jenrom-UbuntuXG:~/trivy\$ docker run --rm -v /var/run/docker.sock:/var/run/docker.sock -v \$PWD:/root/.cache/ aquasec/trivy image mariadb:latest

2024-11-11T22:52:21Z INFO [vuln] Vulnerability scanning is enabled

2024-11-11T22:52:21Z INFO [secret] Secret scanning is enabled

2024-11-11T22:52:21Z INFO [secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning

2024-11-11T22:52:21Z INFO [secret] Please see also https://aquasecurity.github.io/trivy/v0.57/docs/scanner/secret#recommendation for faster secret detection

2024-11-11T22:52:30Z INFO Detected OS family="ubuntu" version="24.04"

2024-11-11T22:52:30Z INFO (ubuntu) Detecting vulnerabilities... os_version="24.04" pkg_num=146

2024-11-11T22:52:30Z INFO Number of language-specific files num=1

2024-11-11T22:52:30Z INFO (gobinary) Detecting vulnerabilities...

2024-11-11T22:52:30Z WARN Using severities from other vendors for some vulnerabilities. Read https://aquasecurity.github.io/trivy/v0.57/docs/scanner/vulnerability#severity-selection for details.

mariadb:latest (ubuntu 24.04)

Total: 21 (UNKNOWN: 0, LOW: 16, MEDIUM: 5, HIGH: 0, CRITICAL: 0)

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title	
coreutils	CVE-2016-2781	LOW	affected	9.4-3ubuntu6		coreutils: Non-privileged session can escape to the parent session in chroot https://avd.aquasec.com/nvd/cve-2016-2781	
gpg	CVE-2022-3219			2.4.4-2ubuntu17		gpg: denial of service issue (resource consumption) using compressed packets https://avd.aquasec.com/nvd/cve-2022-3219	
gpgconf							
gpgv							
libc-bin	CVE-2016-20013			2.39-0ubuntu8.3		sha256crypt and sha512crypt through 0.6 allow attackers to cause a denial of...	
libc6						https://avd.aquasec.com/nvd/cve-2016-20013	
libgcrypt20	CVE-2024-2236	MEDIUM		1.10.3-2build1		libgcrypt: vulnerable to Marvin Attack https://avd.aquasec.com/nvd/cve-2024-2236	
libgsasapi-krb5-2	CVE-2024-26462	LOW		1.20.1-6ubuntu2.1		krb5: Memory leak at /krb5/src/kdc/ndr.c https://avd.aquasec.com/nvd/cve-2024-26462	
	CVE-2024-26458					krb5: Memory leak at /krb5/src/lib/tpc/pmap_rmt.c https://avd.aquasec.com/nvd/cve-2024-26458	
	CVE-2024-26461					krb5: Memory leak at /krb5/src/lib/gsasapi/krb5/k5sealv3.c https://avd.aquasec.com/nvd/cve-2024-26461	

Figura 4.11: También se probó con distintas versiones de la misma imagen para comprobar que las mas nuevas estaban expuestas a menos vulnerabilidades conocidas

80

```
jbenrom@CIBER-jbenrom-Ubuntu18:~/trivy$ docker run --rm -v /var/run/docker.sock:/var/run/docker.sock -v $PWD:/root/.cache/ aquasec/trivy image debian:latest
2024-11-11T22:54:57Z INFO [vuln] Vulnerability scanning is enabled
2024-11-11T22:54:57Z INFO [secret] Secret scanning is enabled
2024-11-11T22:54:57Z INFO [secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2024-11-11T22:54:57Z INFO [secret] Please see also https://aquasecurity.github.io/trivy/v0.57/docs/scanner/secret#recommendation for faster secret detection
2024-11-11T22:55:03Z INFO Detected OS family="debian" version="12.7"
2024-11-11T22:55:03Z INFO [debian] Detecting vulnerabilities... os_version="12" pkg_num=88
2024-11-11T22:55:03Z INFO Number of language-specific files num=0
2024-11-11T22:55:03Z WARN Using severities from other vendors for some vulnerabilities. Read https://aquasecurity.github.io/trivy/v0.57/docs/scanner/vulnerability#severity-selection for details.

debian:latest (debian 12.7)
=====
Total: 78 (UNKNOWN: 0, LOW: 58, MEDIUM: 14, HIGH: 5, CRITICAL: 1)
```

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
apt	CVE-2011-3374	LOW	affected	2.6.1		It was found that apt-key in apt, all versions, do not correctly... https://avd.aquasec.com/nvd/cve-2011-3374
bash	TEMP-0841056-B18BAF			5.2.15-2+b7		[Privilege escalation possible to other user than root] https://security-tracker.debian.org/tracker/TEMP-0841056-B1-8BAF
bsdutils	CVE-2022-0563			1:2.38.1-5+deb12u1		util-linux: partial disclosure of arbitrary files in chfn and chsh when compiled... https://avd.aquasec.com/nvd/cve-2022-0563
coreutils	CVE-2016-2781		will_not_fix	9.1-1		coreutils: Non-privileged session can escape to the parent session in chroot https://avd.aquasec.com/nvd/cve-2016-2781
	CVE-2017-18018		affected			coreutils: race condition vulnerability in chown and chgrp https://avd.aquasec.com/nvd/cve-2017-18018

```
jbenrom@CIBER-jbenrom-Ubuntu18:~/trivy$ docker run --rm -v /var/run/docker.sock:/var/run/docker.sock -v $PWD:/root/.cache/ aquasec/trivy image ubuntu:latest
2024-11-11T22:55:52Z INFO [vuln] Vulnerability scanning is enabled
2024-11-11T22:55:52Z INFO [secret] Secret scanning is enabled
2024-11-11T22:55:52Z INFO [secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2024-11-11T22:55:52Z INFO [secret] Please see also https://aquasecurity.github.io/trivy/v0.57/docs/scanner/secret#recommendation for faster secret detection
2024-11-11T22:55:54Z INFO Detected OS family="ubuntu" version="24.04"
2024-11-11T22:55:54Z INFO [ubuntu] Detecting vulnerabilities... os_version="24.04" pkg_num=91
2024-11-11T22:55:54Z INFO Number of language-specific files num=0

ubuntu:latest (ubuntu 24.04)
=====
Total: 6 (UNKNOWN: 0, LOW: 5, MEDIUM: 1, HIGH: 0, CRITICAL: 0)
```

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
coreutils	CVE-2016-2781	LOW	affected	9.4-3ubuntu6		coreutils: Non-privileged session can escape to the parent session in chroot https://avd.aquasec.com/nvd/cve-2016-2781
gpgv	CVE-2022-3219			2.4.4-2ubuntu17		gnupg: denial of service issue (resource consumption) using compressed packets https://avd.aquasec.com/nvd/cve-2022-3219
libc-bin	CVE-2016-20013			2.39-0ubuntu8.3		sha256crypt and sha512crypt through 0.6 allow attackers to cause a denial of... https://avd.aquasec.com/nvd/cve-2016-20013
libc6						
libcrypt20	CVE-2024-2236	MEDIUM		1:10.3-2build1		libcrypt: vulnerable to Marvin Attack https://avd.aquasec.com/nvd/cve-2024-2236
libssl3t64	CVE-2024-41996	LOW		3.0.13-0ubuntu3.4		openssl: remote attackers (from the client side) to trigger unnecessarily expensive server-side... https://avd.aquasec.com/nvd/cve-2024-41996

```
jbenrom@CIBER-jbenrom-Ubuntu18:~/trivy$ docker run --rm -v /var/run/docker.sock:/var/run/docker.sock -v $PWD:/root/.cache/ aquasec/trivy image alpine:latest
2024-11-11T22:56:30Z INFO [vuln] Vulnerability scanning is enabled
2024-11-11T22:56:30Z INFO [secret] Secret scanning is enabled
2024-11-11T22:56:30Z INFO [secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2024-11-11T22:56:30Z INFO [secret] Please see also https://aquasecurity.github.io/trivy/v0.57/docs/scanner/secret#recommendation for faster secret detection
2024-11-11T22:56:32Z INFO Detected OS family="alpine" version="3.20.3"
2024-11-11T22:56:32Z INFO [alpine] Detecting vulnerabilities... os_version="3.20" repository="3.20" pkg_num=14
2024-11-11T22:56:32Z INFO Number of language-specific files num=0
2024-11-11T22:56:32Z WARN Using severities from other vendors for some vulnerabilities. Read https://aquasecurity.github.io/trivy/v0.57/docs/scanner/vulnerability#severity-selection for details.

alpine:latest (alpine 3.20.3)
=====
Total: 2 (UNKNOWN: 0, LOW: 2, MEDIUM: 0, HIGH: 0, CRITICAL: 0)
```

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
libcrypto3	CVE-2024-9143	LOW	fixed	3.3.2-r0	3.3.2-r1	openssl: Low-level invalid GF(2^m) parameters lead to OOB memory access https://avd.aquasec.com/nvd/cve-2024-9143
libssl3						

Figura 4.12: Se compararon distintas imagenes de SO para asi observar que la menos expuesta era la de *alpine*, al ser el entorno mas ligero y minimalista

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ docker trust signer add --key nanash1.pub nanash1 nanash1/primer_repositorio
Adding signer "nanash1" to nanash1/primer_repositorio...
Initializing signed repository for nanash1/primer_repositorio...
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with ID 551bc16f:
Repeat passphrase for new root key with ID 551bc16f:
Enter passphrase for new repository key with ID 7223afd:
Repeat passphrase for new repository key with ID 7223afd:
Successfully initialized "nanash1/primer_repositorio"
Successfully added signer: nanash1 to nanash1/primer_repositorio
```

Figura 4.15: Generamos claves y certificados para nuestro usuario de DockerHub

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ notary key list
```

ROLE	GUN	KEY ID	LOCATION
root		551bc1632cdb2a8aa81c3bb14f33e9f7050bc4cc39f2b5895168cc3eac91d79e	/home/jbenrom/.docker/trust/private
nanash1		6673eb260ca7747962f7b155de33174fe87ff47342a19b952e64acc66a76b5ab	/home/jbenrom/.docker/trust/private
targets	...sh1/primer_repositorio	7223afd17cd40787493286472a63d7d913932996f699713f9a0a9c89b8bf7a08	/home/jbenrom/.docker/trust/private

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ docker system info | grep Username
Username: nanash1
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ curl -o notary https://github.com/notaryproject/notary/releases/download/v0.6.1/notary-Linux-amd64
% Total % Received % Xferd Average Speed Time Time Current
Dload Upload Total Spent Left Speed
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ wget -O notary https://github.com/notaryproject/notary/releases/download/v0.6.1/notary-Linux-amd64
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ ls
notary notary-Linux-amd64
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ rm notary-Linux-amd64
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ ls
notary
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ chmod +x notary
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ sudo mv -2 notary /usr/bin/
[sudo] password for jbenrom:
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ ls
```

Figura 4.13: Se procedió a usar Notary para la gestión de certificados y la firma de imágenes propias de Docker

```
GNU nano 6.2 /home/jbenrom/.notary/config.json *
{
  "trust_dir": "~/.docker/trust"
}
```

Figura 4.14: Se activo la directiva de Docker Content Trust para la comprobación de imágenes firmadas


```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ docker trust inspect --pretty nanash1/primer_repositorio
No signatures for nanash1/primer_repositorio

List of signers and their keys for nanash1/primer_repositorio

SIGNER    KEYS
nanash1    6673eb260ca7

Administrative keys for nanash1/primer_repositorio

Repository Key:    7223afd17cd40787493286472a63d7d913932996f699713f9a0a9c89b8bf7a08
Root Key:          0f3e278e948288c3f75efadfc200a47d548736e6c32e5a8e3b3f9469b55ce501
```

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ docker tag helloworld:multi nanash1/helloworld:2.0
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ docker trust sign nanash1/helloworld:2.0
Enter passphrase for root key with ID 551bc16:
Enter passphrase for new repository key with ID 46f8baa:
Repeat passphrase for new repository key with ID 46f8baa:
Enter passphrase for nanash1 key with ID 6673eb2:
Created signer: nanash1
Finished initializing signed repository for nanash1/helloworld:2.0
Signing and pushing trust data for local image nanash1/helloworld:2.0, may overwrite remote trust data
The push refers to repository [docker.io/nanash1/helloworld]
6dce60511929: Pushed
1ad27bdd166b: Mounted from library/alpine
2.0: digest: sha256:07eccce2868415beb9c342b5d985276a1af6db27625bbf755c7876d79f8a3721 size: 738
Signing and pushing trust metadata
Enter passphrase for nanash1 key with ID 6673eb2:
Successfully signed docker.io/nanash1/helloworld:2.0
```

Figura 4.16: Se firmo la imagen de helloworld:multi

```
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ docker rmi nanash1/helloworld:2.0
Untagged: nanash1/helloworld:2.0
Untagged: nanash1/helloworld@sha256:07eccce2868415beb9c342b5d985276a1af6db27625bbf755c7876d79f8a3721
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ docker pull nanash1/helloworld:2.0
2.0: Pulling from nanash1/helloworld
Digest: sha256:07eccce2868415beb9c342b5d985276a1af6db27625bbf755c7876d79f8a3721
Status: Downloaded newer image for nanash1/helloworld:2.0
docker.io/nanash1/helloworld:2.0
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ echo "export DOCKER_CONTENT_TRUST=1" >> ~/.bashrc
source ~/.bashrc
(arg: 1) docker pull nanash1/webserver
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ docker pull nanash1/webserver
Using default tag: latest
Error: remote trust data does not exist for docker.io/nanash1/webserver: notary.docker.io does not have trust data for docker.io/nanash1/webserver
jbenrom@CIBER-jbenrom-UbuntuLXC:~/docker_content_trust$ docker trust inspect --pretty nanash1/helloworld:2.0

Signatures for nanash1/helloworld:2.0

SIGNED TAG    DIGEST                                SIGNERS
2.0           07eccce2868415beb9c342b5d985276a1af6db27625bbf755c7876d79f8a3721    nanash1

List of signers and their keys for nanash1/helloworld:2.0

SIGNER    KEYS
nanash1    6673eb260ca7

Administrative keys for nanash1/helloworld:2.0

Repository Key:    46f8baa4eb63571225df7e89ee623492f7e7a0027547fd9c0138cdf2bcefeddb
Root Key:          ff6465d301cc75b31dc4cae17df94e357b4f629752b4d859061c3b3a39fc7478
```

Figura 4.17: Y se probó a hacer `docker pull` a una imagen firmada y una no firmada después de haber activado la directiva, el resultado fue un error por parte de Notary de la imagen no firmada