

Seguridad en docker

Víctor Ponz



Institut Educació Secundària

El Caminàs

Curso de especialización
en Ciberseguridad

Contenidos

Seguridad host	3
Seguridad en el demonio	3
Seguridad en contenedores	4
Privilegios	4
Seguridad en imágenes	5
Escaneo pasivo de vulnerabilidades	7

Seguridad host

Práctica 1

Realiza los comandos y configuraciones del documento.

Como comparte el mismo kernel de la máquina host, todo el software del sistema debe estar actualizado a la máxima versión estable.

Antes de aplicar seguridad en el host, se debería usar alguna tipo de seguridad en el host mediante `ip_tables`, SELinux, apparmor, defensa en profundidad, perímetro etc.

Algo también muy importante es controlar los permisos de usuarios. Sólo deben acceder a docker aquellos usuarios con permiso de root, ya que docker debe acceder al kernel. Lo aconsejable es crear un grupo de docker e ir añadiendo ahí los usuarios que puedan lanzar contenedores.

```
sudo usermod -aG docker $USER
```

y recargar los permisos (o reiniciar sesión)

```
newgrp docker
```

Seguridad en el demonio

El demonio corre como superusuario, así que debemos impedir que los usuarios puedan tocar la configuración y el socket no deberían poder verlo.

El archivo es `/etc/docker/daemon.json` (si no existe, se debe crear para introducir las siguientes configuraciones)

Poner `debug: false` y es muy interesante configurar `ulimits` que permiten definir los archivos que pueden cargar los contenedores y es interesante dejar unos límites por defecto.

También es interesante `icc` que impide que haya conectividad entre los contenedores ya que todos están en la misma red por defecto de tal manera que no se verán entre sí y sólo aquellos que estén linkados explícitamente en su configuración. Por ejemplo `wordpress` y `mysql`

Ejemplo de archivo `daemon.json`

```
{
  "debug": false
}
```

Otro archivo es `key.json` y ningún usuario que no sea root no debería acceder porque ahí se almacena en base64 la key para conectarse por TLS (por ejemplo con el Registry).

Seguridad en contenedores

Son uno de los ejes principales del hardening pues donde hay más configuraciones afectadas.

Por ejemplo asignando un `ulimit` al contenedor:

```
docker run --ulimit nofile=512:512 --rm debian sh -c "unlimit -n"
```

Si se retira el flag de `ulimits`, daría los de por defecto. Se puede modificar el archivo de configuración para fijar unos límites pequeños y luego modificarlo en tiempo de ejecución.

También se pueden meter límites para otro tipo de recursos. Por ejemplo para limitar el alcance de una ataque un DDoS en el que nuestro contenedor puede quedarse sin recursos y esto afectaría al resto de contenedores (incluso a la máquina host) pues se quedarían sin recursos

Por ejemplo:

```
docker run -it --cpus=".5" ubuntu /bin/bash
```

De esta forma nunca gastaría más de 0.5 CPUs

Otra configuración interesante es reiniciar en el fallo:

en el comando `--restart=on-failure`

Privilegios

Por defecto el usuario es `root`

Se puede cambiar el usuario por defecto al correr la imagen y podemos forzar a ello con el flag `-u` `uuid`

Por ejemplo, si corremos el siguiente comando con el usuario 4400

```
docker run -u 4400 alpine ls /root
```

Nos devolverá que no tengo accesos

```
ls: can't open '/root': Permission denied
```

Otro flag relacionado con permisos es `--privileged`:

```
docker run -it --privileged ubuntu
```

```
mount -t tmpfs none /mnt
```

Luego correr:

```
df -h
```

Y nos muestra que ha sido capaz de montarlo, pero si no le añadimos privilegios devuelve:

```
mount: /mnt: permission denied.
```

El que sí tiene privilegios hereda todas las [capabilities](#) de linux.

Otra cosa es montar el socket de docker en un contenedor. Por ejemplo, levantar un docker dentro de docker. Esto se ve mucho en CD/CI.

Por ejemplo:

```
docker run -v /var/run/docker.sock:/var/run/docker.sock -it docker
```

Cuando se lanza tengo un docker dentro de docker es la misma ejecución de docker porque comparten el socket. Si lanzo dentro un contenedor ubuntu, también lo podré listar si hago docker ps en el host

Seguridad en imágenes

Es más fiable confiar en una imagen de la que tengo acceso al Dockerfile que una que ya viene construida. Puede tener malware si la imagen no es confiable.

Con docker commit se crea una imagen, aunque su uso es desaconsejable, ya que no se puede auditar. Sólo confiar en las imágenes de Docker, o de fuente confiable como Google Cloud o Microsoft Azure.

Por ejemplo, si buscamos un imagen de Wodpress encontraremos unas 8000 imágenes. La más popular es la imagen oficial que es la generada por Docker. En el caso de que la haya generado otro desarrollador estará indicado en la imagen. Por ejemplo Multicontainer WordPress de **Microsoft**

Se puede hacer una imagen a partir de un repositorio en GitHub como por ejemplo, esta [imagen](#) alojada en <https://github.com/irespaldiza/whoami> por lo que se puede clonar o crearlo tú a partir del Dockerfile.

El servicio que lo soporta es Notary que es una de las aplicaciones a las que da soporte Docker.

Esta variable de entorno es muy aconsejable tenerla a true. Por ejemplo se puede incluir en el bash.rc.

A partir de una imagen es parecido a compilar a partir de la definición para generar las capas y guardarla en un registry como docker.hub

La imagen de alpine sólo pesa 5.61 MB porque comparte el kernel con el host.

Los comandos se ejecutan en tiempo de compilación en la preparación del entorno y ENTRYPOINT y CMD en tiempo de ejecución.

Lo normal es poner en ENTRYPOINT un comando y en CMD los parámetros (que se pueden sobreescibir)

También existe el comando ADD que es muy parecido a COPY. La diferencia es que COPY sólo permite copiar desde el equipo y ADD desde una url.

Y otra diferencia es que ADD puede copiar y descomprimir archivos, pero la capa no se podrá cachear.

Es una mala praxis no usar la versión al utilizar un paquete. Por ejemplo, alpine para buscar imágenes se usa docker search alpine, pero no nos muestra tag

De esta forma, se puede comprobar si tiene vulnerabilidades o si dentro de un tiempo la vuelvo a generar puede dar problemas de compatibilidad.

Es bastante normal que el código deba estar compilado lo que añade más superficie a ser atacada porque no interesa tener un compilador en la imagen ya que si nuestro contenedor es atacado el atacante podría compilar programas en nuestro sistema, cosa que está totalmente prohibida.

por ejemplo

```
FROM golang:alpine AS builder
ENV GO111MODULE=auto
COPY whoami.go /app/
WORKDIR /app
RUN go build -o whoami

FROM alpine
WORKDIR /app
COPY --from=builder /app/whoami /app/
ENTRYPOINT ./whoami
```

El concepto pot es de **Kubernetes** que son un grupo de contenedores que comparten en el mismo espacio de puertos

RETO

Haz un docker file en GitHub y conéctalo con Docker Hub para que genere el build cada vez que se modifica.

Escaneo pasivo de vulnerabilidades

Se puede hacer con alternativas gratuitas pues en DockerHub es Pro.

Una de ellas es `trivy`. Por ejemplo

```
trivy image python:3.4-alpine
```

Práctica 2

Realiza un testeo de una imagen de Wordpress (no uses tags recientes, pues no tendrán tantas vulnerabilidades)

Más información en [Docker Security Cheat Sheet](#)