# Avance de proyecto: Análisis de seguridad de Docker

Maximiliano Osorio
mosorio@inf.utfsm.cl

Valparaíso, 10 de mayo de 2015

## 1. Descripción del problema

Como se vio en la propuesta, *container-based virtualization* presenta mejor rendimiento que *hypervisor-based virtualization* [11], despliegue rapido de aplicaciones, portabilidad a través de host, control de versiones entre otras características[2], pero las soluciones basadas en *hypervisors* presentan un mejor aislamiento debido a que VM no puede comunicarse con el kernel de host, solo se puede comunicar con su mismo kernel. Por otra parte *container-based virtualization* necesita comunicarse con el kernel del host, por lo tanto permite que un atacante puede atacar al host u otro *container* .
Este aspecto es importante de analizar para validar a *container-based virtualization* como una solución en el producción en sistemas cloud u otros.

## 2. Trabajo realizado hasta la fecha

### 2.1. Investigación

Al ser un tema nuevo y no conocido completamente por el autor ha sido necesario utilizar gran parte del tiempo en investigar y documentar como es el funcionamiento de Docker y las herramientas que utiliza para crear ambientes virtuales seguros. Es por ello que el marco teórico es extenso con el fin de poder diseñar posibles ataques y proponer soluciones o mitigaciones a los ataques.

#### 2.1.1. Docker

Docker es un proyecto open-source que utiliza la tecnología de los containers (libcontainer) para "construir, migrar y correr aplicaciones distribuidas". Actualmente utilizado por Yelp, Spotify, entre otros [4] [9] Docker es una solución que simplifica el uso de los *containers* que han estado presente durante más de una década. Primero provee una interfaz simple y segura para crear y controlar *containers* [1], segundo permite a los desarrolladores empaquetar y correr sus aplicaciones de manera sencilla y ademas se integra con herramientas terceras que permiten administración y despliegue como Puppet, Ansible y Vagrant. Además que existen diversas herramientas de orquestación como Mesos, Shipyard, Kubernetes, RancherOS y Docker Swarm. Docker puede separarse en dos grandes componentes:

Dentro de Docker engine existen componentes:

- Docker Engine
- Docker Images
- Docker Containers
- Docker Registries

Docker engine es una herramienta liviana y portable para manejar *container-based virtualization* utilizando la arquitectura de la figura . Los containers corren encima del servicio de Docker que se encarga de ejecutar y manejar los *containers* . Docker Client, provee una interfaz para interactuar con los containers con los usuarios a través de RESTful APIs. [1]

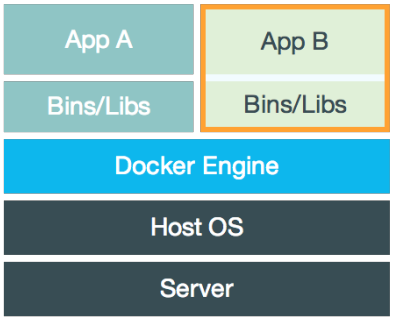


Figura 1: Docker

Docker utiliza una arquitectura de cliente-servidor, Docker Client habla con Docker daemon y este construye, maneja y corre los *containers*. Docker Client y Docker daemon pueden correr en el mismo host o se puede conectar el cliente desde un host remoto. El cliente y el daemon se comunican en forma de sockets o RESTful API. [3]

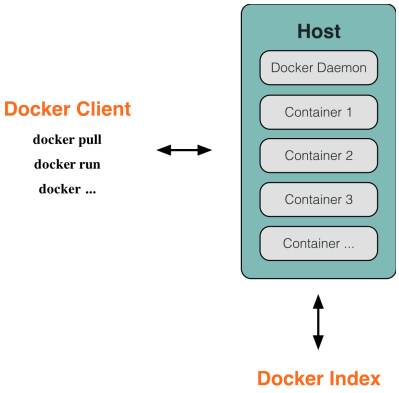


Figura 2: Arquitectura

Una imagen de Docker (*Docker Images*) es una plantilla de solo lectura *read-only template*. Por ejemplo, una imagen puede contener un sistema operativo de Ubuntu con Apache y una aplicación web instalada o simplemente el sistema operativo. Las imágenes son usadas para construir *Docker containers*. Cuando el usuario crea cambios en el *container* , este cambio no ser realiza en la imagen. Docker añade una capa adicional con los cambios de la imagen. [1] Por ejemplo, si el usuario utiliza una imagen base de Debian, luego añade el paquete emacs y luego añade el paquete apache, el estado de capas estaría representado por la figura 3 [4]. Esto permite tener un proceso de distribución de imágenes más eficiente dado que solo es necesario distribuir las actualizaciones. [1]. Lo *filesystem* descrito anteriormente se denomina *Union File System (UnionFS)*.

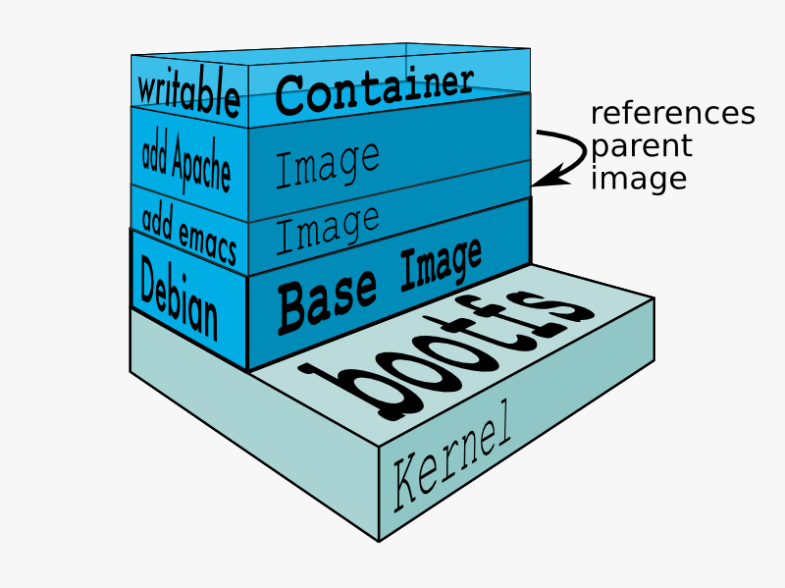


Figura 3: Representación *union file system*

Docker *Registries* son repositorios de imágenes estos pueden ser públicos y privados, vía a estos los usuarios pueden compartir las imágenes personalizadas o base. Docker cuenta con un repositorio publico llamado Docker Hub, Docker Hub es un repositorio central donde se pueden compartir y obtener imágenes, estas imágenes pueden ser verificadas tanto en la autenticidad y integridad a través un firmado y verificación de datos.[1]. Los *registries* son parte fundamental de eco sistema de Docker como una herramienta de desarrollo y despliegue de aplicaciones. Un caso de uso de un eco sistema típico que utiliza Docker está descrito por los siguientes eventos y la figura 4:

- Devs (desarrolladores) utiliza una imagen lista para usar, ya sea por ejemplo Java, PHP o Rails, está puede ser obtenida de repositorios públicos o privados.
- Paralelamente, ops (operaciones) pueden contribuir añadiendo patrones de seguridad o de despliegue.
- Devs y ops envían y traen sus cambios a través de Docker con comandos como commit, push, pull, tag hasta llegar a una versión de producción
- Ops traen los cambios a los servidores de producción, staging o quality assurance

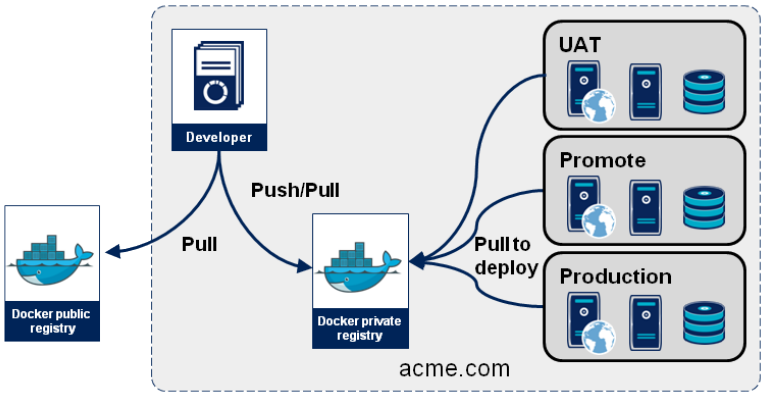


Figura 4: Representación *Dinámica de un Docker registry*

Docker *container* consiste de un sistema operativo, con archivos de usuarios y metadata, como se mencionó anteriormente cada *container* es construido a partir de una imagen. Esa imagen indica a Docker lo que contiene el *container* , cual es proceso que debe correr cuando el *container* es creado y una variedad de configuraciones [3]. Para describir los pasos incluidos en la creación de un *container*, se ejemplificará con la creación de uno.

```
docker run -i -t ubuntu /bin/bash
```

- *Docker client* le informa al Docker que debe correr un *container*.
- El comando será el *init 0* del container en este caso en */bin/bash*
- **Traer la imagen:** Docker verifica la existencia de la imagen ubuntu y sino existe en el host, entonces Docker descarga de un *registry* ya sea privado o publico. Si la imagen existe entonces crea el container.
- **Asignar un *filesystem* y montar una capa *read-write*:** El *container* es creado en el *filesystem* y se añade una capa en modo *read-write* a la imagen.
- **Crear la red y conectar con el *bridge interface*:** Crea la interfaz de red que permite que el Docker *container* pueda hablar con el host a través del bridge (docker0).
- **Asignar un IP:** Asigna una ip del pool al *container*
- **Capturar el *output*, *input* y errores**.

Docker utiliza dos funcionalidades de Linux *namespaces* y *cgroups* para crear ambientes virtuales para los *containers* . *cgroups* o *control groups* proveen un mecanismo de contabilidad y limites de recursos que pueden utilizar los *containers* . [1] Los *namespaces* proveen del aislamiento que es llamado *container* . Cuando sea crea un *container* , Docker crea un conjunto de *namespaces* para el *container* . Los *namespaces* utilizados por Docker: mount (mnt) para el manejo del montaje, pid para el aislamiento de los procesos, net para el manejo de interfaces y ipc para acceder a recursos de IPC. Con el objetivo de detectar posibles brechas de seguridad se analizará cada una de esas.

### 2.1.2. Process isolation

Docker logra el aislamiento de los procesos separando los procesos en *namespaces* y limitando los permisos y la visibilidad de los procesos a otros *containers*. *PID namespaces* (añadido en el kernel $\geq$ 2,6,3,2) es el mecanismo utilizado, logrando que un proceso que se encuentra en el *container* solo pueda ver procesos que se encuentra en ese *container* . Por lo tanto un atacante no puede observar procesos de otro *container* , lo que hace difícil un ataque [1] [8].

### 2.1.3. Filesystem isolation

Docker usa *mount namespaces* o *filesystem namespaces* para aislar los *filesystems* asociados a los *containers* . De la misma forma que ocurre con los procesos, los eventos del *filesystem* que ocurre en el *container* afectan a ese *container* . Pero existen *Linux kernel file systems* que deben ser montados desde host para que el *container* pueda funcionar correctamente, esto permite a los *containers* acceso directo al host. Docker utiliza dos técnicas para proteger el host: primero montar los *file-systems* en modo lectura para evitar que puedan escribir en ellos y eliminar la opción de montar *file-systems* en modo escritura y lectura. [13]

### 2.1.4. Device isolation

Docker utiliza *cgroups* que permite especificar que *device* puede ser utilizado con el *container* . Esto bloquea la posibilidad de crear y usar *device nodes* que puedan ser utilizados para atacar el host. Los *device nodes* que son creados para cada *container* por defecto: son: /dev/console, /dev/null, /dev/zero, /dev/full, /dev/tty*, /dev/urandom, /dev/random, /dev/fuse [13]

#### 2.1.5. IPC isolation

IPC (*inter-process communication* es un conjunto de objetos para el intercambio de data a través de los procesos, como semáforos, colas de mensajes, segmentos de memoria compartida. Los procesos corriendo en los *containers* utilizan *IPC namespaces* que permite la creación de un *IPC* separado y independiente para cada *container* , con esto se previene que procesos en un *container* interfieran con otros *containers* o el host.

#### 2.1.6. Network isolation

Para cada *container* , Docker crea una red independiente usando *network namespaces*, compuesta de su propia IP, rutas, *network devices* Esto permite que el *container* pueda interactuar con otro host a través de su propia interfaz.

Por defecto, la conexión se realiza gracias al host que provee un *Virtual Ethernet bridge* en la máquina host, llamado docker0 que automaticamente realiza un *forward* de los paquetes entre las interfaces. Cuando Docker crea un nuevo *container* , esto establece una interfaz de red virtual con un nombre único que se conecta con el *bridge (docker0)* y con la interfaz *eth0* del *container* . [1]

#### 2.1.7. Limite de recursos

*Cgroups* controlan la cantidad de recursos como CPU, memoria, *disk I/O* que el *container* puede utilizar. A partir de esto se protege contra ataques de DoS.

### 2.2. Implementación

Actualmente la implementación base se encuentra lista. Utilizando un nodo Centos con project-atomic y Docker 1.5.0. Se habilitó SElinux para realizar pruebas con un *Mandatory Access Control*

```
Docker version 1.5.0, build a8a31ef/1.5.0
```

### 2.3. Discusión y posibles ataques

#### 2.3.1. Secure image repository

Basado en la idea de Fernandez, Monge y Hashizume de *Secure virtual machine image repository* [5] se investigó sobre si Docker hub cumple con los patrones especificados. Fernandez, Monge y Hashizume definen defensas para un repositorio de VMI entre ellas se encuentran: Authenticator-Authorizer, Secure Channel, Security Logger/Auditor y filter [5], basado en el changelog de Docker 1.3 se muestra que implementó un proceso de verificación de las imágenes llamado *signed images* completando la lista de defensas nombradas, Rudenberg [12] y Jay [6] alertan que el proceso de traer un imagen no es seguro debido a que:

- Docker nunca verifica *checksum* de la imagen permitiendo un ataque.
- Utilización de xz, escrito en C (un lenguaje *memory-safe* potencialmente se podría ejecutar código malicioso y escalar en privilegios

Al probar realizar las pruebas, se determina que no afectan a la última versión de Docker. Para más detalles consultar a CVE-2014-9356, CVE-2014-9357, CVE-2014-9358 [7]

#### 2.3.2. Network

El modelo de red de Docker conecta la interfaz de *container* con el *bridge docker0* esto permite que exista comunicación de layer-2 entre los *containers* y el host. Actualmente un *container* es capaz de comunicarse con otro *container* por defecto en Docker, en caso de querer aislar un *container* se debe utilizar la opción –icc. En este

caso no es útil debido que a la opción –icc actua a nivel de iptables o sea *layer 3* y los *containers* están conectados por *layer 2* por lo tanto es posible realizar un ARP-spoofing. nyatec propone dos soluciones para esto: eliminar la capacidad de NET_RAW para evitar que el *container* puede crear *PF_PACKET sockets* que son necesarios para el ARP spoofing attack o la utilización de etables para filtrar los *Ethernet frame* con direcciones de destino incorrecto. [10]. La utilización de SElinux también es una solución comprobado en la prueba de concepto. **Prueba de concepto**: Se crean dos *containers* , la imagen de atacante contiene las programa arpspoof.

```
docker run --name=atacante -ti ubuntu_arp bash
docker run --name=victima --ti ubuntu bash
```

El atacante hace arp-spoofing, a la victima (172.17.0.3)

```
for i in $(seq 1 5); do  arpspoof -t 172.17.0.3 172.17.42.1 > /dev/null 2>&1 &; done;
```

Y la victima sufre el ataque

```
? (172.17.0.4) at 02:42:ac:11:00:04 [ether] on eth0
? (172.17.42.1) at 02:42:ac:11:00:04 [ether] on eth0
```

# 3. Conclusiones preliminares

Docker es un sistema seguro siendo utilizado con la configuración por defecto aunque si existen vulnerabilidades como se nombro anteriormente. La utilización de herramientas complementarías como SElinux aumentan el grado de seguridad es por ello que en la siguiente entrega el informe se centrará en la utilización de Docker con SElinux y resolver la pregunta ¿podemos asegurar el host u otros *containers* si el atacante escapó del *container* ?

# 4. Trabajo restante

- Completar investigación con *Linux Capabilities* y *Mandatory Access Control*
- Alcances de ataques en un ambiente de Docker con SElinux (MAC) integrado.
- Es posible asegurar el host o los *containers* si el atacante salió del *container* .

## 4.1. Trabajos posibles:

Análisis de una solución de orquestación de *containers*, por ejemplo Kubernetes.

# Referencias

[1] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[2] Docker. 7.2. advantages of using docker, 2015.

[3] Docker. Understanding docker, May 2015.

[4] Docker. Use cases, 2015.

[5] EduardoB. Fernandez, Raul Monge, and Keiko Hashizume. Building a security reference architecture for cloud systems. *Requirements Engineering*, pages 1–25, 2015.

[6] Trevor Jay. Before you initiate a "docker pull", Dic 2014.

[7] LVM. docker-io: multiple vulnerabilities, Dic 2014.

[8] LVM. Pid namespaces in the 2.6.24 kernel, 2015.

[9] Victor Marmol, Rohit Jnagal, and Tim Hockin. Networking in containers and container clusters.

[10] nyantec. Docker networking considered harmful, Mar 2015.

[11] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Kang G Shin, et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 2007.

[12] Jonanthan Rudenberg. Docker image insecurity, Dic 2014.

[13] Daniel Walsh. Bringing new security features to docker, sep 2014.