KIPS Tr. Comp. and Comm. Sys.
Vol.6, No.2 pp.75~86 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2017.6.2.75

Method of Digital Forensic Investigation of Docker-Based Host

Kim Hyeon Seung[†] · Sang Jin Lee[††]

ABSTRACT

Docker, which is one of the various virtualization technology in server systems, is getting popular as it provides more lightweight environment for service operation than existing virtualization technology. It supports easy way of establishment, update, and migration of server environment with the help of image and container concept. As the adoption of docker technology increases, the attack motive for the server for the distribution of docker images and the incident case of attacking docker-based hosts would also increase. Therefore, the method and procedure of digital forensic investigation of docker-based host including the way to extract the filesystem of containers when docker daemon is inactive are presented in this paper.

Keywords : Docker, Image, Container, Inactive State

도커 기반 호스트에 대한 디지털 포렌식 조사 기법

김 현 승[†] · 이 상 진[††]

요 약

오늘날 다양한 서버 내 가상화 기술 중 도커(Docker)는 기존의 방식보다 경량화된 서비스 운영 환경을 제공함으로써 많은 기업 환경에 도입되고 있다. 도커는 이미지, 컨테이너 개념을 통해 서버 환경 구축, 업데이트, 이동을 효율적으로 할 수 있게 지원한다. 도커가 많이 보급될수록 도커 이미지를 배포하는 서버나 도커 기반의 호스트에 대한 공격 유인이 증가할 것이다. 이에 본 논문에서 도커 데몬이 비활성화된 상태에서도 컨테이너의 파일 시스템을 추출할 수 있는 방안을 포함하여 도커를 사용하는 호스트에 대한 포렌식 조사 기법과 그 절차를 제시하였다.

키워드 : 도커, 이미지, 컨테이너, 비활성화 상태

1. 서 론

오늘날 다양한 가상화 기술이 서버 운영에 활용되고 있다. 그 중 도커(Docker)는 2013년에 등장한 가상화 도구이다. 이는 Xen, KVM 등과 같은 하이퍼바이저를 이용한 가상 머신 생성 기술과 다른 원리로 경량화된 가상화 환경을 제공하여 큰 인기를 끌고 있으며 AWS, Google Cloud Platform, MS Azure 등의 클라우드 서비스에서 공식 지원하고 있다. Right Scale 사의 클라우드 트렌드 조사에 따르면 도커 사용률은 지난 1년간 2배 이상 증가했으며[1], Data Dog사는 지난 1년간 자사 고객들의 도커 도입 증가율이 30% 이상인 것으로 발표했다[2]. 이처럼 급성장 중인 도커는 서버 환경의 구축 및 버전 관리에 최적화된 시스템을 제공함으로써 최근 클라우드 시스템의 보급과 함께 등장한 개념인 Immutable Infrastructure을 실현하기 위해 적합한 서비스로 평가받고 있다. 즉, 호스트 OS와 서버 운영 환경(서버 프로그램 등)을 분리된 것으로 보는 관점에서 도커는 운영 환경 관리에 최적화된 서비스이다. 그러나 도커가 실제 서비스 환경에 많이 보급될수록 도커 이미지 배포 서버에 악성코드가 삽입된 이미지가 업로드될 가능성이 높고, 도커 기반의 호스트가 공격받는 사례 또한 증가할 것이다. 따라서 도커를 사용하는 환경에 대한 포렌식 조사 기법이 필요하다.

2. 관련 연구

도커에 관한 기존 연구들은 하이퍼바이저를 기반으로 한 가상화 기술과 도커 기술를 대조하여 후자의 우수성을 강조하였다[3-5]. 그리고 도커 기술의 현황을 살펴보고 그 보급을 촉진하기 위해 해결되어야 할 문제들을 제시한 연구가 있고[6], 도커 기반 서비스의 성능 평가에 관한 실증적인 연

† 준 회 원 : 고려대학교 정보보호대학원 정보보호학과 석사과정
†† 종신회원 : 고려대학교 정보보호대학원 교수
Manuscript Received : October 10, 2016
Accepted : November 11, 2016
* Corresponding Author : Sang Jin Lee(sangjin@korea.ac.kr)

구 또한 존재한다[7]. 이밖에 도커 기반의 새로운 아키텍처를 설계하려는 연구가 있었고[8-10], 도커의 보안성에 대한 연구도 이루어졌다[11]. 특히 기존에 Vmware 등의 가상화 툴을 사용해 악성코드를 분석하던 것과 유사하게 도커를 활용하여 악성코드 조사를 위한 환경을 구축하는 방법 또한 논의되었다[12].

그러나 도커 서비스에 대해 디지털 포렌식 조사 관점에서 논의된 바는 없다. 도커를 기반으로 한 호스트에서 어떠한 일이 발생했는지를 파악하기 위해서는 기존의 호스트 조사 기법들 이외에도 도커 환경에 특화된 방법들이 필요하다. 즉, 도커 데몬이 제공하는 명령어를 사용하고 도커 데몬의 구동 결과 남는 아티팩트들의 의미를 이해해야 한다. 따라서 본 논문에서는 도커 기반의 리눅스 호스트 조사 기법을 제시하고자 하며 이는 도커 버전 1.12.0을 기준으로 한다.

3. 도커 기술의 구성 요소

호스트 OS 위에서 별도의 격리된 실행 환경을 제공하는 도커는 아래와 같은 개념들로 이루어져 있다.

3.1 도커의 기반 기술

도커는 Fig. 1과 같이 기존의 가상화 기술처럼 호스트의 OS 위에 하이퍼바이저와 게스트 OS를 구동하는 방식이 아니라 OS 위의 도커 엔진 상에서 컨테이너라는 격리된 실행 환경을 제공하는 형태로 기능한다.

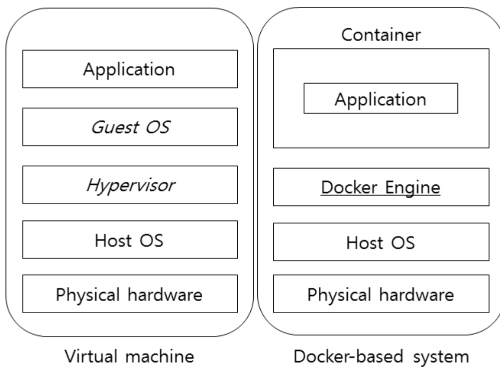

Fig. 1. Comparison of Docker with Virtual Machine

즉, 도커 엔진 위에는 별도의 게스트 OS를 설치하지 않고 서버 운영을 위한 프로그램과 라이브러리만 설치하며 도커 엔진은 호스트와 OS 자원을 공유한다. 이를 위해 리눅스 커널의 cgroups와 namespaces라는 기능을 사용하는데 도커의 초기 버전은 LXC(Linux Container)를 기반으로, 버전 0.9부터는 LXC를 대신하는 libcontainer가 사용되고 있다.

3.2 이미지와 컨테이너

이미지는 서비스 운영에 필요한 서버 프로그램, 소스 코드, 컴파일된 실행 파일 등이 있는 파일 시스템과 그것이 컨테이너화될 경우 반영될 설정사항까지를 포괄하여 의미하고 컨테이너는 도커 데몬이 특정 이미지를 대상으로 구성한 격리된 실행 환경을 말한다. 이미지는 복수의 레이어(layer)가 계층화된 형태로 이루어져 있다. 도커 데몬이 어떤 저장 드라이버를 사용하는지에 따라 각 레이어로 이미지를 구성하는 방식이 다르고 이미지의 각 레이어들은 읽기만 가능하다. 이때 저장 드라이버는 사용자가 도커 데몬 실행 시에 옵션으로 설정 가능하다.

특정 이미지에서 컨테이너를 구성하기 위해 보통 ‘docker run -i -t --name hello ubuntu /bin/bash’ 형식의 명령어가 쓰이는데 docker run은 이미지를 기반으로 새 컨테이너를 생성하기 위한 명령어, -i(interactive)와 -t(pseudo-tty)는 컨테이너에 터미널을 할당하여 사용자의 입력을 받도록 하는 옵션, hello는 새로 지정할 컨테이너명, ubuntu는 새 컨테이너의 기반이 되는 이미지명, /bin/bash는 컨테이너가 생성되면서 컨테이너 내에서 가장 먼저 실행될 프로그램을 나타낸다. 이밖에도 다양한 옵션을 지정하여 여러 특성을 가진 컨테이너를 생성할 수 있다.

도커 데몬이 특정 이미지를 대상으로 컨테이너를 구성하게 되면 Fig. 2와 같이 해당 이미지의 레이어 중 최상단에 읽기 쓰기가 가능한 레이어인 ‘컨테이너 레이어’가 추가되고 해당 컨테이너 내에서 발생하는 파일 시스템 상의 모든 변경 사항(쓰기, 변경, 삭제 등)은 CoW(Copy-on-write) 기법을 통해 이 컨테이너 레이어에만 반영된다. 즉, 컨테이너 상에서 어떠한 변경이 일어나도 해당 변경은 이미지 내의 레이어의 내용을 컨테이너 레이어로 복사한 후 이를 대상으로 이루어지기 때문에 이미지 자체에는 아무런 변화가 발생하지 않는다. 이러한 원리로 인해 한 이미지에서 여러 컨테이너를 생성할 수 있고 특정 컨테이너가 삭제되어도 그 기반 이미지에는 아무런 영향이 없다. 이때 컨테이너 내의 각 레이어를 구성하는 구체적인 방식은 도커 데몬이 사용하는 저장 드라이버에 따라 다르다.

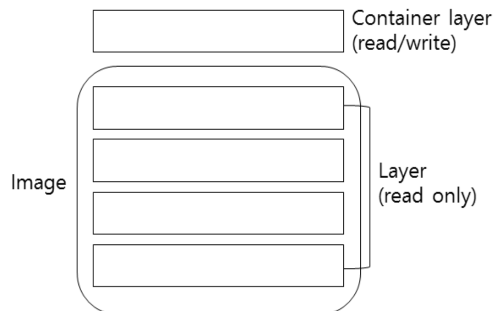

Fig. 2. Layers in a Container

도커 서비스는 결국 이 이미지와 컨테이너 개념을 중심으로 하여 운용되는데 이때 이미지를 생성하는 방법에는 다음과 같은 것들이 있다.

1) 도커 허브의 저장소 활용

이는 ‘docker pull 이미지명’ 명령어를 통해 도커 회사가 제공하는 공개 저장소에서 네트워크를 통해 이미지를 가져오는 방법이다. 도커 허브로 불리는 공개 저장소에는 도커 회사가 제공하는 공식 이미지뿐만 아니라 전세계인들이 본인의 필요에 맞게 커스터마이징한 유/무료의 다양한 이미지들이 존재한다. 이밖에도 프라이빗 계정이나 회사 계정을 생성해 조직 내 직원들과 같은 특정인들끼리만 사용할 수 있는 저장소의 이미지를 사용할 수도 있다.

2) Dockerfile 작성 후 빌드

Dockerfile이란 도커 이미지의 생성을 자동화할 수 있도록 하는 파일이다. Dockerfile의 작성 규칙에 맞게 파일을 생성한 후 해당 파일이 속한 경로에서 ‘docker build –t khs/webserver:v2.(현재 경로를 의미)’ 형식의 명령어를 사용하면 이미지를 생성(빌드)할 수 있다. 이때 –t 옵션에는 해당 이미지의 이름을 설정할 수 있는데 / 앞부분은 사용자의 이름, / 과 : 사이는 해당 이미지의 이름, : 뒤에는 같은 이미지를 구별할 수 있는 태그가 설정된다. 즉 같은 이미지라도 태그에 따라 구별이 된다. Dockerfile의 작성에 사용되는 대표적인 옵션들과 그 설명은 Table 1과 같고 실제 사용 예는 Fig. 3과 같다.

Table 1. Options used in a Dockerfile

Option	Meaning
FROM	Base image to use
MAINTAINER	Person who maintain this image
END	Environment variables of the image
LABEL	Metadata of the image
ADD	Files or directories to add to the image
VOLUME	Data volume of the container which would be based on this image
USER	The account that will execute the script of “RUN, CMD”
WORKDIR	The directory in which the script of “RUN, CMD” will be executed
RUN	Shell script to execute for generating an image like installing new packages
CMD	Program that would be started on the completion of setting a container
EXPOSE	Port number to expose to another container in the same host or to the host

```
FROM ubuntu
MAINTAINER khs <khs@korea.ac.kr>
LABEL Description="This is the version 1 server" Version="1.0"
RUN apt-get update && apt-get install -y inotify-tools nginx apache2 openssh-server
```

Fig. 3. Example of Dockerfile Contents

3) 컨테이너의 파일 시스템 상태를 새 이미지로 구성

도커는 컨테이너의 파일 시스템 상태를 그대로 새 이미지로 만들 수 있다. 즉, 원본 이미지를 컨테이너로 구성한 후 컨테이너에서 여러 가지 변경 작업을 한 후에 해당 파일 시스템 자체를 다시 새 이미지로 만들 수 있는데 이 때문에 도커를 통해 서버 운영 환경을 지속적으로 업데이트 할 수 있는 것이다. 이를 위한 명령어의 예로 ‘docker commit ubuntucontainer –m “added new packages” –a “khs” khs/myubuntu:v2’는 ‘ubuntucontainer’라는 이름을 가진 컨테이너 내의 파일 시스템을 ‘khs/myubuntu:v2’라는 이미지로 만든다. 이때 –m 옵션으로 해당 변경사항을 커밋 메시지로 남기고 –a 옵션으로 해당 커밋 수행자의 이름을 남길 수 있다. 이렇게 만들어진 새 이미지를 배포하여 다시 컨테이너로 구성하는 작업을 통해 전체 서버의 업데이트를 손쉽게 할 수 있다. 만약 변경 작업 도중의 오류 발생 등으로 인해 원래의 상태로 돌아가고 싶다면 단지 업그레이드 이전의 이미지를 갖고 컨테이너를 구성하면 된다.

3.3 데이터 볼륨

컨테이너 레이어 상의 변경사항은 컨테이너가 사라지면 함께 삭제되기 때문에 컨테이너의 존재 여부와는 관계없이 영구적으로 보관해야할 데이터나 다른 컨테이너 및 호스트와 공유가 필요한 데이터를 저장할 공간이 필요하다. 이를 위해 도커 데몬은 데이터 볼륨이라는 기능을 지원한다. 데이터 볼륨은 컨테이너에 마운트된 호스트 파일 시스템 상의 디렉토리나 파일을 말한다. 컨테이너 구성 시 “-v 호스트 내 경로:컨테이너 내 경로” 옵션을 주면 컨테이너 내부의 해당 경로에 접근하여 호스트 내의 해당 경로를 데이터 볼륨으로 사용할 수 있다. 데이터 볼륨에 저장하는 파일이나 디렉토리는 컨테이너 내의 파일 시스템이 아니라 곧바로 호스트의 해당 경로에 저장된다. 즉, 컨테이너 내에서 데이터 볼륨에 접근하여 이루어지는 행위는 컨테이너 자체의 파일 시스템과는 아무런 상관이 없으며 호스트 파일 시스템에만 영향이 있다. 따라서 상기한 ‘docker commit’ 명령을 통해 컨테이너의 현 상태를 이미지로 생성해도 데이터 볼륨 내의 내용은 이미지에 포함되지 않는다. 또한 데이터 볼륨 내 파일과 디렉토리들은 컨테이너가 삭제되더라도 존재한다. 데이터 볼륨은 한 컨테이너에서 여러 개의 지정이 가능하고 특정 컨테이너의 데이터 볼륨을 다른 컨테이너에서도 사용할 수 있다.

3.4 네트워크 기능

도커는 컨테이너 간의 다양한 네트워크를 지원하는데 다음과 같이 단일 호스트 내의 네트워크와 복수 호스트 간의 네트워크로 분류할 수 있다.

1) 단일 호스트 내의 네트워크

a) 브리지 모드(Bridge mode)

이미지를 컨테이너로 구성하면 해당 컨테이너는 기본적으

로 'docker0'이라는 이름을 가진 bridge 네트워크에 연결된다. 만약 사용자가 새로운 브리지 모드 네트워크를 생성하기 위해서는 'docker network create -d bridge 지정할 네트워크명'을 실행하면 된다. 그리고 이후에 컨테이너 구성 시 '--network=네트워크명' 옵션을 지정하여 해당 컨테이너를 특정 네트워크에 연결할 수 있다. 하나의 컨테이너는 복수의 네트워크에 연결될 수 있으나 컨테이너 간의 통신은 동일 네트워크 내에서만 가능하다. 동일한 브리지 모드 네트워크 내에 속한 컨테이너끼리는 IP 주소로 서로 통신할 수 있는데 만약 IP 주소가 아닌 방식으로 접근하기 위해서는 컨테이너 간의 연결을 명시해야 한다. 이는 컨테이너 구성 시 '--link 컨테이너명 혹은 id:연결 시 사용할 별칭' 옵션을 지정하면 되고 이렇게 구성된 컨테이너에서는 IP 주소뿐만 아니라 지정한 별칭으로도 해당 컨테이너에 접근할 수 있다. 이는 link 옵션을 사용하면 관련 DNS 정보가 해당 컨테이너 내에 반영되기 때문에 가능하다.

b) 호스트와의 포트 매핑(Port mapping)

컨테이너가 서비스를 제공하기 위해서는 외부와의 통신이 가능해야 한다. 이때 컨테이너가 외부에 접근하는 것은 가능하지만 외부에서 컨테이너에 접근하기 위해서는 컨테이너와 호스트와의 포트 매핑(port mapping)이 필요하다. 이는 호스트의 특정 포트로 들어오는 패킷을 컨테이너의 특정 포트에 그대로 전달하는 것으로 도커를 서버 운영에 활용할 수 있도록 하는 핵심 기술이다. 컨테이너 구성 시 -P(랜덤 포트 지정), -p(특정 포트 지정) 옵션을 사용해 컨테이너와 호스트 사이에 포트 매핑을 할 수 있다. 그 예로 'docker run -d -p 80:5000 training/webapp python app.py' 명령의 경우 호스트의 80번 포트를 컨테이너의 5000번 포트와 매핑하여 서비스를 제공할 수 있다.

2) 복수 호스트 간의 네트워크

a) 앰배서더(ambassador) 컨테이너

여러 호스트 간의 연결방식은 서로 다른 호스트 내의 도커 서비스 상의 애플리케이션이 어떤 방식으로 통신하느냐에 따라 다양할 수 있다. 그 중 하나로 앰배서더 컨테이너를 이용한 방식이 있다.

이는 Fig. 4와 같이 'svendowideit/ambassador'라는 이미지로 구성된 컨테이너를 매개로 하여 서로 다른 호스트 상의 컨테이너끼리 연결하는 방법이다. 앰배서더 컨테이너는

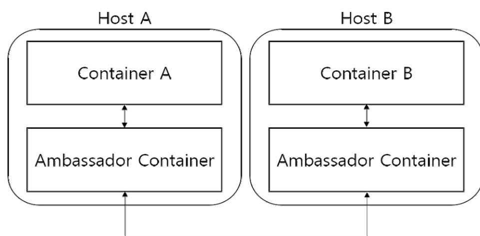

Fig. 4. Ambassador Network Pattern

socat이라는 프로그램을 이용하여 TCP 연결을 다른 곳으로 전달하도록 구성되어 있는데 그 사용 예는 다음과 같다.

Step 1. 호스트에서 포트번호 x를 사용하는 서버 컨테이너(A)를 구성한다.

Step 2. 'docker run -d --link A:servercontainer --name ambassadorforserver -p x:x svendowideit/ambassador' 명령어로 A를 위한 앰배서더 컨테이너를 구성한다.

Step 3. 다른 호스트에서 'docker run -d --name ambassadorforclient --expose y(=클라이언트 컨테이너가 접속할 포트번호) -e SERVER_PORT_TCP=tcp://A가 속한 호스트의 IP 주소:x svendowideit/ambassador' 명령어로 클라이언트 컨테이너(B)를 위한 앰배서더 컨테이너를 구성한다.

Step 4. 클라이언트 컨테이너(B)를 구성하면서 '--link ambassadorforclient:별칭' 옵션을 준다.

Step 5. B에서 y번 포트를 통해 A에 접근할 수 있다.

Step 3.을 수행하면 클라이언트의 앰배서더 컨테이너가 '-e' 옵션으로 할당된 환경변수의 내용 중 "_TCP="라는 문자열에 연결된 IP 주소와 포트 번호를 추출해 socat 명령을 실행한다. socat명령은 로컬의 y번 포트에서 서버 컨테이너가 있는 호스트의 x번 포트로 TCP 프로토콜 상의 데이터를 전달하도록 설정한다.

접속하고자 하는 컨테이너의 포트를 노출시키고 해당 포트로 직접 연결하는 방식이 아닌 이러한 패턴이 존재하는 이유는 만약 특정 서버 서비스를 제공하는 컨테이너가 실행되고 있는 호스트의 IP 주소가 변경되면 원래 접속하고 있던 클라이언트 컨테이너에서 소스 레벨의 수정이 이루어져야 하는데 이 패턴을 사용하면 클라이언트 컨테이너를 위한 앰배서더 컨테이너 실행 시 환경 변수 옵션을 변경하는 것만으로 이에 대응할 수 있기 때문이다.

b) Overlay Network

이는 서로 다른 호스트 간의 컨테이너끼리 통신할 수 있는 네트워크로 도커가 정식으로 지원하는 네트워크 기능이다. 도커 상에서 이를 사용하려면 1.12.0 버전부터 지원되는 Swarm mode를 사용하거나 Consul, Etcd, ZooKeeper와 같은 별도의 Key-value Store를 사용하여야 한다. Swarm mode는 복수 호스트에서 작동되는 도커 엔진들을 클러스터로 묶어서 서비스를 제공하는 형태이다. Swarm 내의 호스트들은 manager node 또는 worker node로 동작하며 전자는 클러스터가 수행할 서비스의 태스크 배분, 후자는 실제로 태스크를 수행하는 역할을 하게 된다. 별도의 설정을 지정하지 않으면 manager node 또한 worker node처럼 태스크를 수행한다. Swarm mode에는 서비스의 태스크 개수를 지정할 수 있는 replica 모드와 모든 노드에서 태스크를 수행하도록 하는 global 모드가 있다. Swarm mode는 다음과 같은 방식으로 사용할 수 있다.

Step 1. manager node로 사용할 호스트에서 'docker swarm init --advertise-addr 호스트 IP 주소' 명령어로

swarm을 구성하고 이때 출력되는 토큰을 보관한다.

Step 2. worker node로 사용할 각 호스트들에서 'docker swarm join −token Step 1에서 획득한 토큰 호스트 IP 주소:2377' 명령어를 실행해 swarm에 worker node들을 포함시킨다. 포트번호 2377은 도커의 클러스터 관리를 위한 통신에 필요하기 때문에 지정한다.

Step 3. Swarm을 구성한 뒤에는 manager node에서 'docker service create --name webserver --replicas 3 --publish 8080:80 nginx' 형식의 명령어로 서비스 제공을 시작할 수 있는데 이는 'nginx 이미지를 3개의 태스크로 만들어(컨테이너를 생성함을 의미) webserver라는 서비스 이름으로 운영하며 이때 각 노드의 8080번 포트를 해당 노드 내의 컨테이너의 80번 포트에 연결해 공개한다는 뜻이다. 이때 Swarm 내에 있는 노드의 공개된 포트라면 해당 노드가 현재 태스크를 수행 중이지 않다고 하더라도 Swarm의 라우팅 기술을 통해 서비스 제공을 받을 수 있다.

4. 도커 기반 호스트 포렌식 조사

도커 기반 호스트를 조사해야 하는 상황은 다음과 같이 발생할 수 있다. 예를 들어, 기업 내에서 서비스 업데이트를 위해 일정한 주기마다 업데이트된 도커 이미지를 제공하는 서버가 있는데 해당 서버가 공격을 받아 악성 프로그램이 있는 이미지가 배포될 수 있다. 이후 각 서버에서 이러한 이미지를 받아 서비스 업데이트를 위해 컨테이너화한다면 공격자의 의도에 따라 각 서버가 C&C 서버 등의 용도로 이용될 수 있다. 또 도커 기반으로 서비스를 제공하던 호스트 자체가 공격을 받을 수도 있다. 이외에도 특정 문제로 인해 도커 기반 호스트가 제공하던 서비스가 중단되면 그 원인을 파악해야 한다. 이러한 경우에는 해당 호스트 자체의 아티팩트뿐만 아니라 도커 서비스 상의 아티팩트도 조사해야 한다.

도커 기반 호스트를 조사할 때 도커 데몬이 작동되고 있거나 그렇지 않을 때의 2가지 상황이 있다. 전자의 경우 도커의 기본 명령어를 사용하여 조사할 수 있다. 후자의 경우 도커 서비스를 다시 실행한 후 도커의 기본 명령어를 사용할 수도 있으나 만약 아래와 같은 경우라면 도커 관련 아티팩트를 갖고 정보를 추출해야 한다.

(1) 도커 기반 호스트의 하드디스크를 복제한 후 복제본 하드디스크로 도커 서비스를 다시 실행하더라도 당시 각 컨테이너의 실행 상황이 그대로 재현되지 않아 당시 상황에 대한 정보가 필요할 때

(2) 특정 컨테이너의 파일 시스템을 도커 명령어가 아닌 포렌식 툴로 조사하거나 삭제된 파일을 복구하기 위해 이를 하나의 파일로 추출해야 할 필요가 있을 때

(3) 조사를 위해 호스트의 파일 시스템 전체를 복제하는 것이 아닌 컨테이너의 파일 시스템만을 추출하여 신속한 조사를 하고자 할 때

이러한 경우에는 도커 서비스가 남기는 아티팩트를 수집하고 그 의미를 파악할 수 있어야 한다. 도커 서비스의 활성화 여부와 관계없이 도커 기반 호스트는 Fig. 5와 같은 순서로 조사를 수행해야 한다. 조사 대상 범위 내에서 도커를 사용하는 호스트들을 파악하고 이들 간의 네트워크를 파악한 후, 각 호스트 내의 컨테이너의 실행 이력을 파악해야 한다. 이하에서는 Fig. 5의 절차에 따라 도커 서비스가 실행 중인 상태와 그렇지 않은 상태에서 각 단계별로 조사하는 기법들을 제시하고자 한다. 후술하는 명령어들의 인자 중 '컨테이너명'과 '이미지명'은 각각 '컨테이너ID', '이미지ID'로 대체해도 무방하다.

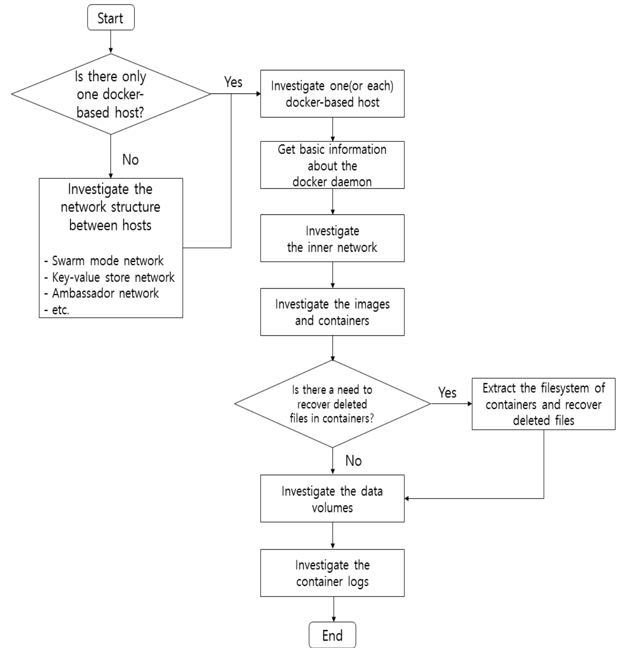

Fig. 5. Procedure for the Investigation of Docker-Based Host(s)

4.1 호스트 간 네트워크 파악

일단 복수의 호스트가 도커 기반 서비스를 제공 중인지 단일 호스트에서만 도커 기반 서비스를 운영 중인지를 파악해야 한다. 전자의 경우에는 일단 도커 기반 호스트 간의 네트워크 구조를 파악한 후, 개별 호스트 조사로 진행해야 한다. 즉, 각 호스트 내의 도커 엔진들이 클러스터로 묶여 하나의 서비스를 수행 중인지 또는 특정 호스트 내의 컨테이너가 웹 서버, 또다른 호스트 내의 컨테이너가 DB 서버의 역할을 하는 중인지 등을 파악해야 한다. 도커 기반의 호스트끼리 연결되는 방식은 서로 다른 호스트에 있는 각 컨테이너 간의 통신 방식에 따라 다양할 것이나 도커에서는 기본적으로 overlay network라는 명칭으로 서로 다른 호스트의 컨테이너 간 네트워크를 지원한다. 따라서 조사 대상 호스트들에서 'docker network ls' 명령의 결과 중 driver명이 "overlay"인 네트워크의 ID를 수집하면 같은 overlay network id를 가진 서로 다른 호스트 내의 컨테이너들이 같은 네트워크에 속해 있음을 알 수 있다.

만약 모든 도커 기반 호스트들이 swarm mode로 서비스를 수행 중이라면 네트워크 정보 중 scope명이 swarm인 네

트워크가 존재하고 ‘docker info’ 명령의 결과 중 swarm 키의 값이 “active”로 나타난다.

Swarm mode인 경우 해당 클러스터에서 어떤 서비스를 제공하고 있는지를 파악해야 하는데 이를 위해서는 manager node를 찾아야 한다. 이는 swarm mode의 정보 조회에 관한 명령어가 manager node에서만 동작하기 때문이다. manager node에서 ‘docker service ls’ 명령으로 swarm이 수행 중인 서비스 내용을, ‘docker service ps 서비스명’ 명령으로 해당 서비스 제공을 위해 수행되고 있는 태스크 목록을, ‘docker service inspect 서비스명’ 명령으로 Fig. 6과 같이 해당 서비스의 상세 내용을 파악해야 한다. 이밖에도 manager node에서 ‘docker node ls’ 명령으로 해당 swarm 내의 모든 node의 현황을, ‘docker node inspect 노드명’ 명령으로 특정 노드의 상세 정보를, ‘docker node ps 노드명‘ 명령으로 해당 노드에서 수행되고 있는 서비스 태스크를 확인할 수 있다.

도커 서비스가 비활성화된 상태라면 ‘/var/lib/docker/containers/’의 각 디렉토리 내 config.v2.json 파일을 조회하여 해당 컨테이너가 속한 overlay network 정보를 알 수 있고 만약 swarm mode로 동작 중이었다면 해당 호스트가 속한 swarm에서 수행되고 있던 서비스의 정보와 해당 노드에서 수행 중이던 태스크 정보를 확인할 수 있다. 각 호스트의 ‘/var/lib/docker/swarm/docker-state.json’ 파일 내에 해당 호스트가 manager node인 경우 local address에 자신의 IP 주소를 가지고 있고, worker node인 경우 local address는 비워져 있으며 Remote Address에 manager node의 IP 주소를, AdvertiseAddr에 자신의 IP 주소를 갖고 있다. 이를 통해 도커 서비스가 활성화된 당시의 네트워크 상황을 알 수 있다.

```
root@ubuntu:/# docker service inspect webserver
[
    {
        "ID": "9aadrdi4gm6j2ucl87q5kcz1i",
        "Version": {
            "Index": 57
        },
        "CreatedAt": "2016-09-
07T03:36:54.237113489Z",
        "UpdatedAt": "2016-09-
07T03:36:54.241113308Z",
        "Spec": {
            "Name": "webserver",
            "TaskTemplate": {
                "ContainerSpec": {
                    "Image": "nginx"
                },
                "Resources": {
                    "Limits": {},
                    "Reservations": {}
                },
                "RestartPolicy": {
                    "Condition": "any",
                    "MaxAttempts": 0
                },
                "Placement": {}
            },
            "Mode": {
                "Replicated": {
                    "Replicas": 3
                }
            },
            "UpdateConfig": {
                "Parallelism": 1,
                "FailureAction": "pause"
            },
            "EndpointSpec": {
                "Mode": "vip",
                "Ports": [
                    {
                        "Protocol": "tcp",
                        "TargetPort": 80,
                        "PublishedPort": 8080
                    }
                ]
            }
        },
```

Fig. 6. Information of the Service in a Swarm

4.2 단일 호스트 조사

호스트 간의 연결 구조를 파악했다면 단일 호스트 조사로 진입해야 한다. 단일 호스트 조사는 다음과 같은 순서로 진행한다.

1) 도커 기본 정보 수집

먼저 해당 호스트 내 도커 서비스의 기본 정보를 수집하여 전반적인 도커 운영 상황을 판단해야 한다. ‘docker version’명령으로 도커 서비스의 버전 관련 정보를 알 수 있고, ‘docker info’ 명령으로 도커 엔진 상의 이미지 수, 컨테이너 수, 저장 드라이버, 파일 시스템, swarm mode 작동 여부 등과 같은 기본적인 정보를 조회할 수 있다. 또 ‘docker stats’으로 각 컨테이너의 현재 호스트 리소스 사용 통계를 볼 수 있다. 특히, 도커 서비스가 활성화된 상태에서 ‘docker info’ 명령은 Fig. 7과 같이 도커 서비스의 전반적인 정보를 알려주기 때문에 사고 대응의 초동 조치 시에 필수적인 명령어라고 할 수 있다.

```
root@ubuntu:/# docker info
Containers: 2
 Running: 0
 Paused: 0
 Stopped: 2
Images: 2
Server Version: 1.12.1
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 10
 Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge overlay host null
Swarm: inactive
```

Fig. 7. Result of ‘docker info’ Command

2) 호스트 내 네트워크 조사

단일 호스트 내에서도 여러 컨테이너들 간의 네트워크를 구성하여 서비스를 제공할 수 있으므로 그 현황을 조사해야 한다. “4.1 호스트 간 네트워크 파악” 부분과 유사하게 ‘docker network ls’ 명령으로 파악된 네트워크 중 driver명이 “bridge”인 네트워크들을 ‘docker network inspect 네트워크명’ 명령으로 조사하면 해당 네트워크에 연결되어 실행 중인 컨테이너들의 정보를 알 수 있다. 컨테이너를 생성하면

기본적으로 연결되는 docker0 네트워크와는 다른 이름의 브리지 네트워크가 존재한다면 이는 서비스 제공을 위해 의도적으로 구축한 네트워크일 가능성이 높기 때문에 해당 네트워크로 연결된 컨테이너들이 어떤 기능을 하는지 살펴보아야 한다.

이에 대하여 Fig. 8과 같이 'docker port 컨테이너명 혹은 docker ps' 명령을 이용해 컨테이너와 그 호스트와의 포트 매핑 현황을 파악해야 한다. 이는 특정 컨테이너가 외부에 어떠한 서비스를 제공하려면 반드시 호스트의 포트포워딩 서비스를 사용할 수밖에 없는데 어떤 번호의 포트 번호로 어느 컨테이너가 연결되었는지를 보고 해당 컨테이너의 역할을 파악할 수 있기 때문이다. 예를 들어 호스트의 80포트에 연결된 컨테이너는 웹 서버 역할을 하는 것으로 파악하거나 호스트에 3306 포트를 열어 연결된 컨테이너는 Mysql DB 서버 역할을 하는 것으로 추측할 수 있다.

```
root@ubuntu:/# docker port testcontainer
80/tcp -> 0.0.0.0:9000
root@ubuntu:/# docker ps
CONTAINER ID ~  PORTS                   NAMES
731c3ce94d94     0.0.0.0:9000->80/tcp  testcontainer
```

Fig. 8. Port Mapping of Containers with Its Host

도커 서비스가 비활성화된 경우라면 '/var/lib/docker/containers'의 각 디렉토리 내의 config.v2.json 파일을 조회하여 해당 컨테이너가 속한 bridge network 정보를 알 수 있고 해당 컨테이너와 그 호스트와의 포트 매핑 현황을 알 수 있다. 동일 경로에 있는 hosts 파일 내에는 컨테이너 구성 당시 '--link 옵션'으로 명시적으로 연결한 컨테이너들의 정보가 'Links:[연결한 컨테이너 이름:지정한 별칭]' 형식으로 존재하는데 이는 서비스 제공을 위해 특별히 별칭으로 다른 컨테이너에 접근할 수 있도록 지정한 유의미한 정보이므로 파악해야 한다.

3) 이미지 및 컨테이너 조사

이는 도커 기반 호스트 조사에서 핵심적인 부분으로 존재하는 이미지들과 서비스를 제공 중이던 컨테이너의 현황을 파악함으로써 상기한 네트워크 조사 결과와 결합하여 해당 호스트의 역할을 규명할 수 있을 뿐만 아니라 해당 컨테이너 내의 파일 시스템에서 보다 직접적인 정보를 수집할 수 있다.

a) 이미지 및 컨테이너 조회

도커가 활성화된 상태에서는 'docker images' 명령으로 존재하는 이미지를, 'docker ps' 명령으로 활성화된 상태의 컨테이너들을 조회할 수 있다. 만약 존재하지만 정지된 상태의 컨테이너들까지 조회하려면 'docker ps -a' 명령을 입력해야 한다. 각 이미지나 컨테이너의 상세 정보를 수집하려면 'docker inspect 컨테이너명 혹은 이미지명' 명령을 사용한다.

b) 특정 컨테이너 조사

도커 기반 호스트 조사 시에는 컨테이너가 어떤 행위를 수행하는 중인지 파악하는 것이 중요하다. 'docker top 컨테이너명' 명령으로 특정 컨테이너 내에서 실행 중인 프로세스를 파악할 수 있고, 'docker attach 컨테이너명' 명령으로 현재 컨네이너에서 출력되고 있는 내용을 확인할 수 있다. 또 기존 이미지에서 컨테이너화된 후 파일 시스템 상 달라진 부분을 보기 위해 'docker diff 컨테이너명' 명령을 사용할 수 있는데 그 실행 결과의 예는 Fig. 9와 같다. A는 추가, C는 변경, D는 삭제된 파일 혹은 디렉토리를 나타낸다. 만약 추가된 파일 중에 심층적으로 분석해야 할 파일이 있다면 'docker copy 컨테이너명:컨테이너경로 호스트경로' 명령으로 해당 디렉토리나 파일을 호스트로 복사하여 확인할 수 있다.

```
root@ubuntu:/# docker diff networktest
D /run
C /bin
C /bin/uname
A /hello
```

Fig. 9. Result of 'docker diff' Command

이처럼 컨테이너 내의 프로세스와 파일 시스템 상 변경사항을 조사함으로써 해당 컨테이너의 실행 흐름에 이상한 점이 있지 않은지 살펴볼 수 있다. 만약 문제가 있어보이는 컨테이너 내의 모든 프로세스의 동작을 멈추고자 하면 'docker pause 컨테이너명' 명령을 실행하고 다시 프로세스들을 실행하고자 하면 'docker unpause 컨테이너명' 명령을 사용한다.

c) 파일 시스템 추출하기

도커 서비스가 활성화되어 있을 때 이미지나 컨테이너의 파일 시스템을 파일 단위로 추출할 수 있다. 이를 위해 'docker save -o 파일명.tar 이미지명'으로 특정 이미지를 tar 파일로 추출한 후 다른 호스트의 도커 엔진 위에서 'docker load < 해당 파일명.tar'으로 다시 이미지로 로드할 수 있다. 또 특정 컨테이너의 파일 시스템을 그대로 추출하고 싶다면 'docker export 컨테이너명 > 파일명.tar'으로 추출한 후 'cat 파일명.tar | docker import - 지정할 이름'으로 다시 이미지로 생성할 수 있다.

그러나 이같이 컨테이너의 파일 시스템을 도커 엔진에서 다시 불러들여도 그 상태에서는 이미지 내에서 존재하다가 삭제된 파일이나 컨테이너 내에서 생성된 후 삭제된 파일을 복구할 수는 없다. 따라서 특정 컨테이너 내에서 삭제된 파일을 복구하기 위해서는 아래와 같이 별도로 파일 시스템을 추출하는 방법이 필요하다.

d) 컨테이너 내 삭제된 파일 복구

컨테이너 내에서 삭제된 파일을 복구하기 위해서는 별도의 처리가 필요하다. 이 때 그 방법은 도커에서 사용하는 저장 드라이버에 따라 달라진다. 도커 엔진은 저장 드라이버

를 통해 이미지와 컨테이너의 파일 시스템을 관리하고 그 드라이버 종류는 도커 실행 시 옵션으로 설정할 수 있다. 이하에서는 도커 메뉴얼이 실제 서비스 프로덕션(production) 모드에서 사용하도록 권장하는 AUFS와 Devicemapper 저장 드라이버를 사용한 도커 서비스 내의 파일 시스템을 추출하기 위한 작업 과정을 제시한다.

d-1) AUFS 저장 드라이버

이는 'advanced multi-layered unification filesystem'의 약자로 AUFS 저장 드라이버는 서로 다른 디렉토리를 병합 마운트(union mount)하여 마치 하나의 파일 시스템처럼 보이게 하는 방식을 사용한다. 이때 마운트되는 각 디렉토리를 브랜치(branch)라고 하며 모든 브랜치는 호스트의 '/var/lib/docker/aufs/diff/' 경로에 존재한다.

특정 이미지가 컨테이너로 구성되면 해당 컨테이너의 읽기/쓰기를 위한 branch도 해당 경로에 생성되며 최종적으로 '/var/lib/docker/aufs/mnt/' 경로의 각 디렉토리에 해당 컨테이너의 브랜치들이 병합 마운트되어 마치 하나의 파일 시스템인 것처럼 보이게 된다.

파일을 삭제할 경우 AUFS 저장 드라이버는 해당 파일이 존재하는 브랜치에서 파일을 직접 삭제하는 것이 아니라 Fig. 10과 같이 최상단의 컨테이너 브랜치에서 '.wh.삭제된 파일명(whiteout 파일)'의 숨김 파일을 생성함으로써 병합 마운트의 결과로 보이는 파일 시스템(unified view)에서는 마치 해당 파일이 삭제된 것처럼 보이게 한다. 따라서 각 브랜치 즉, 디렉토리를 전수조사하면 삭제된 파일을 발견할 수 있다.

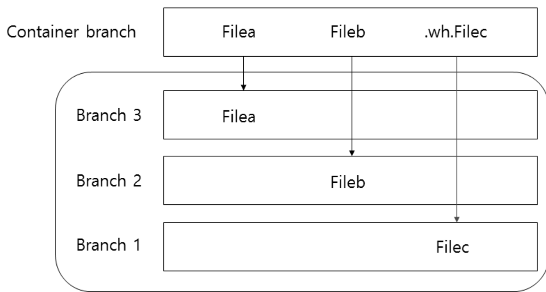

Fig. 10. How a File is Deleted in AUFS

실험 결과, AUFS 저장 드라이버를 사용하는 도커 데몬에서 특정 이미지의 각 브랜치를 조사할 수 있는 과정은 다음과 같다.

Step 1. '/var/lib/docker/image/aufs/repositories.json' 파일을 열람하여 원하는 이미지 이름의 sha256 값을 찾는다.

Step 2. '/var/lib/docker/image/aufs/imagedb/content/sha256' 경로에서 Step 1에서 구한 값을 이름으로 가진 파일을 찾는다. 그리고 해당 파일의 내용 중 rootfs 키의 diff_ids 키의 값으로 지정된 배열을 찾는다. Fig. 11과 같은 이 배열이 해당 이미지를 이루는 각 레이어 즉, 브랜치들의 sha256 값의 집합이고 오래된 브랜치일수록 가장 먼저 등장한다. 이 중에서 파일 시스템을 확인하고자 하는 브랜치의 sha256 값을

```
~
"rootfs":{"type":"layers","diff_ids":
["sha256:c8a75145fcc4e1a66cd86b3cbbe14da1a37894129005e461a43875a094b93412",
"sha256:c6f2b330b60c7c32642c47871b28aab110a7214ed6aac305dd03f70b95cdc610",
"sha256:055757a19384c8afff0e79db7bb84fd481d3a9565d78962c7f368d5ac5984998",
"sha256:48373480614b79e5c1b0a080807fa8ffaea12695f548406ea77feb5074e195e3",
"sha256:0cad5e07ba339f87eb58850252a0ad00e104bae4cfc66b376265e16c32a0aae9",
"sha256:ff7ff7f37694165bb1e0e2ce194ef0bab3114874f341197ba381bbf3249d4d0f"]}}
~
```

Fig. 11. SHA256 Values of Layers in an Image

선택한다.

Step 3. '/var/lib/docker/image/aufs/layerdb/sha256/'에 존재하는 여러 디렉토리들을 획득한다. 각 디렉토리 내에는 cache-id 파일과 diff 파일이 존재한다. 이때 그 diff 파일의 내용이 Step 2에서 선택한 값과 같은 디렉토리를 찾는다. 이후 해당 디렉토리 내의 cache-id 파일 내용으로 있는 값을 찾는다.

Step 4. '/var/lib/docker/aufs/diff/'에서 Step 3에서 찾은 값을 이름으로 가진 디렉토리가 해당 브랜치의 파일 시스템을 갖고 있다.

이와 같은 실험 결과를 보면 도커 데몬은 각 레이어의 SHA256 값을 해당 브랜치의 이름으로 바로 사용하지 않고 대신 cache-id라는 것을 브랜치의 이름으로 지정한 후 이 두 값을 매핑하는 방식을 사용함을 알 수 있다. 이미지가 아닌 특정 컨테이너의 컨테이너 브랜치(최상단 브랜치)를 찾을 수 있는 방법은 다음과 같다.

Step 1. '/var/lib/docker/containers/' 내에서 컨테이너 ID와 같은 디렉토리의 이름을 획득한다. 만약 컨테이너 이름을 기준으로 찾고자하는 경우 모든 디렉토리 내의 config.v2.json 파일 내에서 Name 키의 값을 조사해 원하는 컨테이너의 이름과 일치하는 파일을 가진 디렉토리의 이름을 획득한다.

Step 2. '/var/lib/docker/image/aufs/layerdb/mounts' 경로에서 Step 1에서 획득한 이름을 가진 디렉토리 내에서 mount-id 파일 내용의 값을 확인한다. 이 값을 이름으로 가진 '/var/lib/docker/aufs/diff/' 내의 디렉토리가 해당 컨테이너의 컨테이너 브랜치에 해당한다. 그리고 '/var/lib/docker/aufs/mnt/' 내에서 이 값을 이름으로 갖는 디렉토리가 해당 컨테이너의 모든 레이어가 병합 마운트되는 지점이다.

상기한 방식을 통해 도커 데몬을 통하지 않고서도 원하는 이미지나 컨테이너의 실제 내용물에 직접 접근할 수 있다. 만약 특정 컨테이너에서 기존의 이미지 내용 중 삭제된 파일이나 디렉토리를 찾고 싶다면 Step 2의 결과로 찾은 컨테이너 브랜치 내에서 'find . -name ".wh.*"'을 실행하여 삭제된 것들을 파악한 후 각 브랜치를 전수 조사하면 된다.

이 방법으로 원래 이미지에는 존재하였으나 컨테이너 상에서 삭제된 파일이나 디렉토리들을 복구할 수 있다. 그러나 만약 컨테이너 상에서 특정 파일이나 디렉토리가 생성되었다가 다시 삭제되었을 때에는 AUFS 저장 드라이버의 경우 컨테이너 브랜치 또한 결국 하나의 디렉토리이기 때문에 해당 호스트의 전체 파일 시스템을 복제하여 [13, 14]와 같이 기존 연구들에서 수행되었던 각 파일 시스템 별 파일 복

구 기법을 적용해야 한다. 즉, 호스트의 파일 시스템을 복제한 후 컨테이너 브랜치에 해당하는 디렉토리 내에 존재하다가 삭제된 파일이나 디렉토리가 있는지를 조사해야 한다.

d-2) Devicemapper 저장 드라이버

Devicemapper는 커널에 기반하여 여러 고급 볼륨 관리 기술들을 제공하는 프레임워크로 도커는 Devicemapper의 씬 프로비저닝(thin-provisioning)과 스냅샷(snapshotting) 기술을 사용한다. Devicemapper 저장 드라이버와 AUFS 저장 드라이버와의 가장 큰 차이점은 파일 단위로 파일 시스템을 다루는 후자와 달리 전자는 블록 단위로 파일 시스템을 관리한다는 점이다.

Devicemapper 저장 드라이버를 활용하는 도커 데몬 상에서 모든 이미지와 컨테이너는 하나의 논리 디바이스(thin device, virtual device)이다. Devicemapper를 저장 드라이버로 쓰는 도커 데몬은 두 개의 블록 디바이스를 가지고 씬 풀(thin-pool)을 형성한 후 이를 기반으로 최초의 논리 디바이스인 base device를 만든다. 이후 생성되는 모든 이미지와 컨테이너는 결국 이 base device의 스냅샷으로 새롭게 생성되는 논리 디바이스이다. 씬 풀은 Fig. 12와 같이 실제 데이터를 갖고 있는 블록들이 담겨 있는 데이터 디바이스와 각 논리 디바이스가 데이터 디바이스 내 어떤 블록들을 갖고 있는지에 대한 포인터 정보를 담고 있는 메타데이터 디바이스로 구성된다.

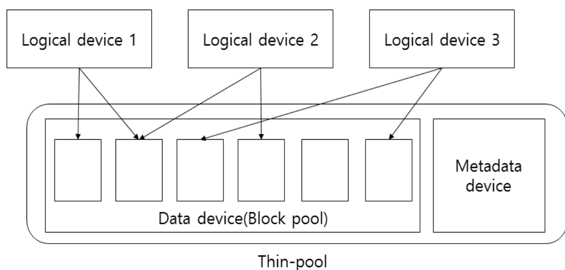

Fig. 12. How a Thin-Pool is Composed

Devicemapper 저장 드라이버를 사용하는 방법에는 loop-lvm mode와 direct-lvm mode 두 가지가 있다. 전자는 호스트 내의 2개의 sparse file을 각각 루프 디바이스로 마운트한 후 이를 가지고 씬 풀을 구성하는 방식이고 후자는 실제 블록 디바이스에 논리 볼륨 2개를 생성하여 이를 가지고 씬 풀을 구성하는 방식이다. 한 호스트에서 Loop-lvm mode로 씬 풀을 구성하고 외장 저장매체의 볼륨을 이용해 Direct-lvm mode로 또다른 씬 풀을 구성한 후 모든 블록 디바이스를 조회한 결과는 Fig. 13과 같다.

조사 시에 loop-lvm mode로 작동하던 씬 풀을 재구성하는 방법은 다음과 같다.

Step 1. 활성화된 호스트에서 'lsblk' 명령을 통해 씬 풀의 용량을 구한다.(도커 데몬의 loop-lvm mode에서의 기본 설정 값은 100GB이므로 별도의 설정이 없으면 100GB이다.)

```
[root@localhost /]# lsblk
NAME                          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
                                   ~

Direct-lvm mode
sdc                              8:32   1 14.5G  0 disk
└sdc1                            8:33   1 14.5G  0 part
  ├fortest-datapool_tmeta 253:3    0  148M  0 lvm
  │ └fortest-datapool     253:5    0 13.8G  0 lvm
  └fortest-datapool_tdata 253:4    0 13.8G  0 lvm
    └fortest-datapool     253:5    0 13.8G  0 lvm

                                   ~

Loop-lvm mode
loop0                            7:0    0  100G  0 loop
└docker-253:0-657959-pool 253:2    0  100G  0 dm
loop1                            7:1    0    2G  0 loop
└docker-253:0-657959-pool 253:2    0  100G  0 dm
```

Fig. 13. Direct-lvm Mode and Loop-Lvm Mode

Step 2. 'losetup /dev/loop0 /var/lib/docker/devicemapper/devicemapper/data'와 'losetup /dev/loop1 /var/lib/docker/devicemapper/devicemapper/metadata' 명령으로 두 파일을 루프 마운트한다.

Step 3. 'dmsetup create examplethinpool −table "0 x thin-pool /dev/loop1 /dev/loop0 128 0 0" 명령으로 하나의 씬 풀을 구성한다. 이 때 x의 값은 Step 1에서 구한 용량을 512byte로 나눈 몫을 입력한다. 해당 명령은 씬 풀의 이름을 examplethinpool로, 시작 블록을 0 블록부터, x * 512 byte 만큼의 용량으로, 데이터 디바이스는 /dev/loop0, 메타데이터 디바이스는 /dev/loop1로 설정하며, 데이터 블록 할당 단위를 64KB(128*512byte)로 하여 씬 풀을 생성한다는 의미이다. 그 결과로 '/dev/maper/examplethinpool'이라는 씬 풀이 생성된다.

direct-lvm mode의 경우 파일을 루프 마운트하는 것이 아니라 실제 물리 볼륨에 씬 풀이 구성되어 있어 해당 물리 볼륨이 있는 하드디스크를 복제하여 복제본을 조사용 PC에 마운트한 후 다음의 순서에 따라 존재하는 thin-pool을 활성화시켜주어야 한다.

Step 1. lvdisplay 명령으로 논리 볼륨들을 조회한 후 그 중 Fig. 14와 같은 pool이 존재하는 볼륨을 찾는다.

Step 2. 'lvchange −activate y 볼륨그룹명(Fig. 14에서 VG Name 값)'으로 해당 볼륨들을 다시 활성화시킨다.

두 모드에서 위의 방법으로 thin-pool을 구성한 후 특정 이미지의 파일 시스템은 다음과 같은 과정을 통해 찾을 수 있다.

```
root@ubuntu:~# lvdisplay
  --- Logical volume ---
  LV Name                dataforthinpool
  VG Name                fortest
  LV UUID                CUo2fC-oUhI-bNf0-qXNb-unMi-KZcr-Np9d67
  LV Write Access        read/write
  LV Creation host, time ubuntu, 2016-09-07 21:04:13 -0700
  LV Pool metadata       dataforthinpool_tmeta
  LV Pool data           dataforthinpool_tdata
  LV Status              NOT available
  LV Size                13.75 GiB
  Current LE             3520
  Segments               1
  Allocation             inherit
  Read ahead sectors     auto
```

Fig. 14. Inactive Logical Volume which has the Pool

Step 1. AUFS에서 이미지를 찾는 과정인 Step 1~3까지 동일하게 수행하여 파일 시스템을 보기를 원하는 레이어의 cache-id 값을 구한다. Devicemapper의 경우 Step 1~3 내의 경로 정보 중 aufs 부분만 devicemapper로 대체하여 그대로 수행하면 된다.

Step 2. '/var/lib/docker/devicemapper/metadata'에서 Step 1의 cache-id 값을 이름으로 갖는 파일 내용의 값 중 device-id 값과 size 값을 획득한다. 별도의 설정이 없으면 도커의 각 이미지와 컨테이너 내 파일 시스템은 10GB 용량의 논리 디바이스로 설정되므로 size 값은 10GB인 경우가 일반적이다.

Step 3. 'dmsetup create forinvestigation –table "0 20971520 thin /dev/mapper/examplethinpool device-id 값"' 명령을 실행한다. 이는 '/dev/mapper/examplethinpool'이라는 씬 풀에서, 시작 블록을 0 블록부터, 20971520 * 512byte (10GB)만큼의 용량으로 해당 디바이스 아이디를 가진 논리 디바이스를, forinvestigation이라는 이름으로 생성하는 것으로 이를 통해 해당 이미지의 파일 시스템을 재구성할 수 있다.

Step 4. 이후 '/dev/mapper/forinvestigation'이라는 디바이스가 생성되는데 이를 그대로 dd 명령 등으로 복제하여 하나의 파일 시스템 파일을 생성하거나 'mount /dev/mapper/forinvestigation /마운트할 경로'로 마운트하여 내부의 파일 시스템을 살펴보면 된다.

만약 특정 컨테이너의 레이어를 찾고자 한다면 AUFS에서 특정 컨테이너의 브랜치를 찾는 과정과 동일하게 진행하여 mount-id 값을 획득한 후 '/var/lib/docker/devicemapper/metadata' 디렉토리에서 그 값을 이름으로 가진 파일의 내용 중 device-id 값을 획득한다. 이후는 이미지의 경우와 동일하게 해당 디바이스 아이디를 가진 논리 디바이스를 생성하여 파일 시스템을 조사하면 된다.

만약 씬 풀을 구성하고 특정 이미지나 컨테이너의 파일 시스템을 추출하는 것을 넘어 도커 데몬이 재구성된 씬 풀을 기반으로 동작할 수 있도록 하려면 'dockerd –storage-driver=devicemapper –storage-opt=dm.thinpooldev=/dev/mapper/해당 thin-pool의 이름 &' 명령을 실행하여 도커 데몬을 실행하면 된다. 그 후 존재하는 컨테이너를 재시작해 동적 분석을 할 수 있다.

기반 이미지에 있던 파일이 컨테이너 상에서 삭제된 경우에 컨테이너와 그 기반 이미지의 각 논리 디바이스를 상기한 방식으로 그대로 추출해서 비교함으로써 대응할 수 있다. 만약 컨테이너 내에서 새로 생성되었다가 삭제된 파일의 경우에는 상기한 방법대로 특정 컨테이너의 논리 디바이스를 추출한 후 해당 파일 시스템 파일을 가지고 기존의 파일 복구 기법을 적용하면 된다.

도커 데몬이 활성화된 상태에서는 도커 명령어를 사용해 즉석에서 컨테이너 내부의 파일 시스템을 파악하는 것이 용이하지만 삭제된 파일을 복구하거나 특정 시점의 파일 시스템을 향후에도 분석하고자 한다면 이처럼 컨테이너 내의 파일 시스템을 별도로 추출해야 한다.

4.3 데이터 볼륨 조사

이미지와 컨테이너의 파일 시스템을 조사한 후에는 컨테이너의 파일시스템에 저장되지 않지만 컨테이너가 생성한 파일들이 담겨 있는 데이터 볼륨 조사를 진행해야 한다. 이러한 데이터 볼륨에는 컨테이너 간의 공유를 위한 데이터들이 담겨 있을 확률이 높고 데이터 볼륨 자체가 컨테이너의 삭제 여부와는 상관없이 계속 존재하고 있으며 이미지나 컨테이너의 파일 시스템을 복제하더라도 데이터 볼륨의 내용은 포함되지 않기 때문에 별도의 수집이 필요하다.

도커가 활성화된 상태에서는 'docker volume ls' 명령과 'docker volume inspect 볼륨명' 명령으로 볼륨에 관한 정보들을 얻을 수 있다.

도커 데몬이 비활성화된 상태에서는 특정 컨테이너가 사용한 데이터 볼륨 내의 파일들을 확인하려면 '/var/lib/docker/containers/' 내의 각 디렉토리의 config.v2.json 파일 내의 Name 키의 값으로 컨테이너의 이름을 확인하여 원하는 컨테이너 관련 파일인지를 파악한 후, 동일 파일 내용 중 Mountpoints 키의 Name 키의 값을 조회하면 된다. 이후 '/var/lib/docker/volumes/'에서 이 값을 이름으로 갖는 디렉토리 안에서 해당 컨테이너가 데이터 볼륨에 남긴 파일이나 디렉토리들을 볼 수 있다.

4.4 컨테이너 로그 조사

상기한 호스트 내 아티팩트들은 컨테이너 내의 각종 실행 흔적에 대한 정보를 시간대별로, 직접적으로 알기에는 큰 단서가 되기 힘들다. 따라서 시간대에 따른 컨테이너 내의 실행 흔적을 알기 위해서는 반드시 그 로그를 분석하여야 한다. 컨테이너는 구성 당시 '--log-driver' 옵션으로 로그 방식을 지정하여 컨테이너들이 실행 도중 출력한 데이터들을 기록할 수 있다. 별도의 옵션을 주지 않으면 json-file 로그 드라이버를 사용하며 이 경우 'docker logs 컨테이너명'을 통해 Fig. 15와 같이 특정 컨테이너의 로그를 조회할 수 있다.

```
root@ubuntu:~# docker logs logtest
root@5b917c8fce8c:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@5b917c8fce8c:/# mkdir malwares
root@5b917c8fce8c:/# wget https://github.com/ytisf/theZoo/tarball/master
--2016-10-19 09:24:58--  https://github.com/ytisf/theZoo/tarball/master
Resolving github.com (github.com)... 192.30.253.112
Connecting to github.com (github.com)|192.30.253.112|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://codeload.github.com/ytisf/theZoo/legacy.tar.gz/master [following]
--2016-10-19 09:24:58--  https://codeload.github.com/ytisf/theZoo/legacy.tar.gz/master
Resolving codeload.github.com (codeload.github.com)... 192.30.253.120
Connecting to codeload.github.com (codeload.github.com)|192.30.253.120|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 385149596 (367M) [application/x-gzip]
Saving to: 'master'

master              100%[===================>] 367.31M  4.85MB/s    in 69s

2016-10-19 09:26:08 (5.36 MB/s) - 'master' saved [385149596/385149596]
```

Fig. 15. Result of "docker logs" Command

도커가 비활성화된 경우 '/var/lib/docker/containers/컨테이너ID-json.log' 형식의 파일을 조회하면 동일한 정보를 시간별로 확인할 수 있다.

만약 로그 드라이버로 기본 옵션이 아닌 다른 호스트로 로그를 전송하는 옵션이 지정된 컨테이너라면 'docker logs' 명령을 사용할 수 없고 로그가 저장된 별도의 호스트를 조사하여야 한다. 로그 드라이버 옵션으로 "gelf, fluentd, awslogs, splunk, etwlogs, gcplogs"를 사용할 수 있는데 이는 외부의 호스트로 로그를 전송하는 옵션이다. 이러한 경우 로그 수집용 호스트의 위치와 운용 원리를 파악하여 해당 컨테이너의 로그를 추출할 수 있어야 한다. 이러한 로그는 컨테이너에 대한 표준 입출력 기록을 갖고 있기 때문에 이를 조사하면 시스템 관리자가 컨테이너 내부에 접속하여 무슨 행위를 했는지, 원격에서 어떤 악성 행위가 이루어졌는지, 어떠한 애플리케이션 오류가 발생했는지 등에 관한 단서를 얻을 수 있다.

4.5 아티팩트 정리

상기한 순서대로 도커 서비스에 관한 아티팩트를 조사하면 호스트 자체의 조사만으로는 알 수 없는 추가적인 정보들을 알 수 있다. 특히 도커 기반으로 서비스를 제공하고 있는 호스트라면 제공되고 있는 서비스와 관련된 각종 설정 파일, 시스템 로그, 애플리케이션 로그와 같은 중요 정보들이 모두 컨테이너 내의 파일 시스템 상에 존재할 것이므로 이를 추출하고 복원하는 데 중점을 두어야 한다.

5. 결론 및 향후 과제

본 논문에서는 도커 기반 리눅스 호스트를 디지털 포렌식 조사하는 기법과 절차에 대해 살펴보았다. 도커 데몬이 활성화된 호스트에서는 도커 데몬이 지원하는 명령어로 도커 서비스에 대한 조사를 용이하게 수행할 수 있다. 그러나 본문의 내용과 같이 당시의 네트워크 상황을 재구성하거나 삭제된 파일 복구 등을 위해 컨테이너의 파일 시스템을 추출해야 하는 경우 또한 존재한다. 따라서 도커 데몬이 비활성화 되어있을 때의 아티팩트 조사 기법이 계속 연구되어야 한다.

가상화 방식의 두 가지 방식 중 컨테이너 방식의 가상화 방식으로 점점 그 인지도를 넓혀가고 있는 도커는 Docker Inc.의 지원으로 인해 보다 더 다양한 기술과 인터페이스를 선보일 것으로 예상된다. 향후에는 아직 실제 서비스 모드에서 사용하도록 권장되지 않는 저장 드라이버가 안정화 단계에 돌입하여 실제 서비스에 적용되는 경우 각각 어떻게 파일시스템을 재구성할 수 있을지에 대해 연구할 것이다. 최근 윈도우와 맥 OS에서도 도커를 사용할 수 있도록 관련 리눅스 커널 기능을 가상화한 서비스인 'Docker for Windows', 'Docker for Mac'도 등장하였다. 해당 서비스들은 리눅스 호스트와는 상이한 아티팩트를 남길 것이므로 이에 대해서도 연구를 지속할 예정이다.

References

[1] Data dog article, "8 surprising facts about real docker adoption"[Internet], https://www.datadoghq.com/docker-adoption/.

[2] Right Scale Survey, "New DevOps Trends: 2016 State of the Cloud Survey" [Internet], http://www.rightscale.com/blog/cloud-industry-insights/new-devops-trends-2016-state-cloud-survey.

[3] Yu-mi Bae, Sung-jae Jung, and Woo-young Soh, "Comparative Analysis of the Virtual Machine and Containers Methods through the Web Server Configuration," *Journal of the Korea Institute of Information and Communication Engineering*, vol.18, No.11, pp.2670-2677, 2014.

[4] Jung-Yeon Hwang and Ho-Yong Ryu, "Performance Comparison and Forecast Analysis between KVM and Docker," *Journal of KIIT*, Vol.13, No.11, pp.127-136, 2015.

[5] Ann Mary Joy, "Performance comparison between Linux containers and virtual machines," *Computer Engineering and Applications (ICACEA), 2015 International Conference*, pp.342-346, 2015.

[6] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio, "Container-based orchestration in cloud:state of the art and challenges," *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pp.70-75, 2015.

[7] P. China Venkanna Varma, Venkata Kalyan Chakravarthy K., V. Valli Kumari, and S. Viswanadha Raju, "Analysis of a Network IO Bottleneck in Big Data Environments Based on Docker Containers," *Big Data Research*, Vol.3, pp.24‐28, 2016.

[8] B. R. Cha and E. J. Kang, "Global Network Verification Test for Docker-based Secured mobile VoIP," *Smart Media Journal*, Vol.4, no.4, pp.47-55, 2015

[9] Y. J. Lee and S. R. Rim1, "A scheme of Docker-based Version Control for Open Source Project," *Journal of the Korea Academia-Industrial Cooperation Society*, Vol.17, No.2, pp.8-14, 2016.

[10] J. W. Park and Jaegyoon Hahm, "Container-based Cluster Management System for User-driven Distributed Computing," *KIISE Transactions on Computing Practices*, Vol.21, No.9, pp.587-595, 2015.

[11] Thanh Bui, "Analysis of Docker Security" [Internet], https://pdfs.semanticscholar.org/ab69/38ec199280213fc092b45abd6170ec95abda.pdf.

[12] Lenny Zeltser, "Running Malware Analysis Apps as Docker Containers"[Internet] https://digital-forensics.sans.org/blog/2014/12/10/running-malware-analysis-apps-as-docker-containers.

[13] Dohyun Kim, Jungheum Park, and Sangjin Lee, "File Carving for Ext4 File Systemon Android OS," *Journal of the Korea Institute of Information Security & Cryptology(JKIISC)* Vol.23, No.3, pp.417-429, 2013.

[14] Jae-hyoung Ahn, Jung-heum Park, and Sang-jin Lee, "The Rese arch on the Recovery Techniques of Deleted Files in the XFS Filesystem," *Journal of the Korea Institute of Information Security & Cryptology*, Vol.24, No.5, pp.885-896, 2014.

김 현 승

e-mail : dongkid321@naver.com
2015년 경찰대학 법학과(학사)
2015년~현　재 고려대학교 정보보호대학원 정보보호학과 석사과정
관심분야 : 디지털 포렌식

이 상 진

e-mail : sangjin@korea.ac.kr
1987년 고려대학교 수학과(학사)
1989년 고려대학교 수학과(석사)
1994년 고려대학교 수학과(박사)
1989년~1999년 ETRI 선임연구원
1999년~현　재 고려대학교 정보보호대학원 교수
2008년~현　재 고려대학교 디지털포렌식연구센터 센터장
관심분야 : Digital Forensic, Steganography, Hash Function