ERNW WHITE PAPER 64/ (02, 2018)

INCIDENT ANALYSIS AND FORENSICS IN DOCKER ENVIRONMENTS

Version:	1.0
Date:	06.02.2018
Classification:	Public
Authors:	Andreas Dewald, Matthias Luft, Julian Suleder

Matthias Luft

Digital unterschrieben von Matthias Luft
Datum: 2018.02.13 18:32:09 +01'00'

ERNW Enno Rey Netzwerke GmbH
Carl-Bosch-Str. 4
69115 Heidelberg

www.ernw.de
www.troopers.de
www.insinuator.net

ERNW
providing security.

TABLE OF CONTENT

ERNW
providing security.

LIST OF FIGURES

1 Abstract

In this article, we describe the impact of the increased use of *Docker* in corporate environments on forensic investigations and incident analysis. Even though Docker is being used more and more (Portworx, Inc., 2017), the implications of the changed runtime environment for forensic processes and tools have barely been considered. We describe the technological basics of Docker and, based on them, outline the differences that occur with respect to digital evidence and previously used methods for evidence acquisition. Specifically, we look at digital evidence within a Docker container which are lost or need to be acquired in different ways compared to a classical virtual machine, and what new traces and opportunities arise from Docker itself.

2 Introduction

Docker[1] is still a comparatively new technology that has already found widespread adoption in Agile software development due to the flexible and fast deployment model (Datadog Inc, 2016) and the ability for rapid software development based on a large ecosystem. This high prevalence in combination with a high adaptation rate means that many servers are already being operated as Docker hosts on which many (micro-) services run in so-called *containers* and thus significantly influence the IT landscape. This influence has already been highlighted from different perspectives of IT security (Jayanth Gummarajul, 2015) (Docker Security Team, 2016) (Theo Combe, 2016) (Jeeva Chelladhurai, 2016). However, no consideration has yet been made in the context of incident analysis and forensic investigations, which are becoming increasingly relevant due to the steadily growing number of existing containers.

2.1 Motivation

This article deals with the question of which changes result from the use of Docker containers for forensic analysis and incident analysis. We divide this question into two aspects:

1) What changes with respect to known methods and evidence?
2) Which new evidence is added by Docker itself?

Previous forensic methods focus predominantly on physical or virtual machines, which do not fundamentally differ in the actual analysis steps. However, it has not yet been considered how traditional processes must be changed to assimilate the use of Docker containers. This article tries to contribute to close this gap.

2.2 Related Work

The Docker ecosystem and its effects have already been explored from different angles of IT security. Gummarajul et al. (Jayanth Gummarajul, 2015) tested publicly available Docker images for known vulnerabilities and identified relevant vulnerabilities in the clear majority of images. The Docker Security Team (Docker Security Team, 2016) went about how to avoid such vulnerabilities in the future and how to

[1] *The term* Docker *in this article refers to the components* Docker Engine *and* containerd *in combination. Although Docker correctly stands for a sum of open-source projects, the mentioned use corresponds to the more common used terminology.*

integrate corresponding measures into an enterprise supply chain. Various papers have already discussed and evaluated Docker's capabilities and limitations for isolating processes (Theo Combe, 2016) (Grattafiori, 2016). From a forensic point of view, the subject has so far been little considered (Jiang Du, 2016).

2.3 Overview

The rest of this article is structured as follows: In Section 3, we explain Docker's basic techniques as they are relevant for the understanding of this article. Section 4 then discusses relevant aspects of Docker in the context of forensic investigations. Section 5 summarizes the content of this work, describes its conclusion and gives an outlook on open aspects and problems for future work.

3 Technological Basics of Docker

Docker is an open-source software for the administration and deployment of software containers. The term software container is not clearly defined, but there is a common understanding of the term across operating system boundaries. This understanding defines software containers as a runtime environment within an operating system that isolates processes or process groups from each other while using a single common kernel. Among other things, this isolation relates to the separation of process spaces, mechanisms for inter-process communication, network usage, file system access, and resource utilization and allocation. Software containers in Linux, the original and still the most popular platform for Docker, are based on the following features of the Linux kernel to implement the isolation: cgroups, namespaces and file system drivers that support different layers. Docker implements an open specification for containers created by the Open Container Iniative (Open Container Initiative, 2018).

3.1 Containers and Images

In Docker's common parlance, the term *container* refers to the runtime instantiation of an *image*. A Docker image contains all necessary data and information needed to create a group of processes with defined properties. For example, the image contains information about which network ports are offered by the contained applications and which program should be executed during instantiation. The image is - apart from the kernel syscall interface and kernel devices - completely independent from the host system on which the image is instantiated. All entities in the Docker environment (such as containers and images, but also network segments) are referenced by a unique identifier (in the following: containerID / imageID). However, the creation of this identifier takes place in different ways: In the case of images, the identifier is a hash over certain parts of the image while containerIDs are randomly generated.

3.2 Cgroups

Cgroups (abbreviation for *control groups*) are a mechanism of the Linux kernel for limiting and measuring resource utilization of process groups. Cgroups can be controlled by various means (such as direct access to an overlay file system in /proc or abstracting user interfaces). In the context of forensic analysis, it is only relevant that containers can be assigned to one or more cgroups.

3.3 Namespaces

The Linux kernel currently supports the following namespaces: Mount, Process ID, Network, Interprocess Communication, UTS and User ID. With the Linux kernel, it is possible to create different namespaces that result in the virtualization of the respective resources. For example, the same file system mount points can be used in different namespaces: Thus, the mount-point /mnt can be used both within all containers as well as on the host system; Similar behaviors arise for other namespaces.

3.4 Layered Filesystem

A layered file system is based on a file system driver, which offers the possibility to build a single file system from different layers to present it in a uniform and abstract manner to a process. This concept is illustrated in Figure 1.

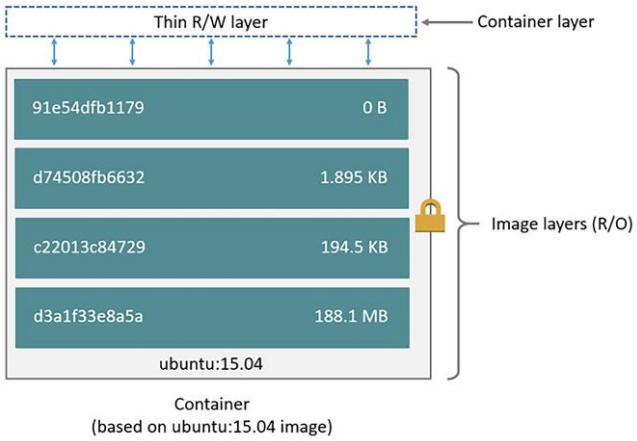

Figure 1: Docker Illustration of the Layered Filesystem Model (Docker Inc, 2018).

A Docker image consists of one or more layers; individual layers can derive from different sources and can be created/ provided by different persons or parties. When a container is instantiated, an r/w layer is created, which is set as the top layer. All write accesses within a container are executed only in this layer; underlying layers remain unchanged.

4 Forensic Analysis on Docker Hosts

In principle, a forensic analysis gets started on a Docker host just like on a regular system: A dump of the hard disk and ideally the main memory is created. However, the analysis of the dumps may provide incomplete results, unless the specifics of (Docker) containers are taken into account. A typical analysis of disk and memory dumps would still result in a list of files and processes, but the mapping to containers, or even information about whether certain files are relevant for the reconstruction of the actual file system view of the live system would not be included. The following sections represent various aspects to consider, when analyzing Docker hosts to provide a comprehensive forensic analysis.

4.1 File recovery and accountability

Docker containers access files through specific file system drivers (Docker Inc, 2018). The file system is not mapped to block devices as in traditional (virtualized) operating system environments, but is based on layered file system-based structures. This type of file system access is shown in Figure 2 and results in various characteristics that must be considered in forensic analysis.

Figure 2: Layered File Systems (Docker Inc, 2018).

For example, searching for files on a disk dump of the host system will also find files from Docker Images. However, the following additional questions must be answered on a Docker host:

1) Which image provided the given file?
2) Which containers used the given file?
3) Was the file deleted at container level? (This operation is potentially different from deletions on regular file systems)

To answer these questions, it is relevant that an image can be used by several containers by using the r/w layer. An association between files and containers is (only) possible via runtime information and, if available, specific configuration files. At runtime, the `docker ps` command issues a list of all started

containers, `docker ps -a` a list of all not yet terminated containers. The first line of the list contains the beginning of the ContainerID (hereafter ContainerIDShort). The complete ContainerID can be determined using the `docker inspect ContainerID` command. If a runtime analysis is not possible, various configuration files contain information about containers on the system. The default directory for Docker's configuration is `/var/lib/docker`. Container configuration is stored in `/var/lib/docker/container/ContainerID`. Containers that are currently being executed can be identified by two characteristics: On the one hand, the container configuration `config.v2.json` contains the attribute `Running: true` and on the other hand, the Linux file system permissions of the container subdirectory `shm` are set to `1777`. There is also a dedicated directory for the r/w layer that indicates whether a container has been started in the past (see Section 4.2.2).

Furthermore, it must be identified whether the file originates from a container (i.e. the r/w layer) or an image. This information is relevant for analyzing the visibility of the file at runtime. If a file is deleted within a container, there are two possibilities for the deletion:

1) The file originates from the r/w layer: In this case the file is deleted with normal operating system mechanisms on the underlying file system layer.
2) The file originates from an image layer: In this case, a deletion reference is left in the r/w layer, but the file remains present in the image layer.

Accordingly, different methods must be used to recover deleted files, which we explain in the following sections.

4.2 Recovery of Files of the R/W Layer

If a file which was stored in the r/w layer of the container is deleted, this file (reminder: which is stored as a regular file in the host file system) is deleted in the file system of the host. Which metadata is retained in the file system and how it can be recovered depends on the actual host file system (such as ext3, ext4, zfs, ...). It would be beyond the scope of this article to address the specificities, but these are not specific to Docker, but common forensic file system analysis conditions and are well known and well documented (Carrier, File System Forensic Analysis, 2005).

This means that the available methods for recovering a deleted file from the r/w layer correspond to those used in the forensic analysis of a physical disk or virtual disk file (for example in vmdk or vdi format) of a virtual machine (VM). It should be noted that with a classic virtual machine, it is also possible to directly analyze the hard disk device (such as `/dev/sda`) from within the virtual machine with forensic software

tools with root privileges. This option does not exist in Docker containers (even with root privileges) because the device can not be opened (unless the container was started – against the default behavior – with elevated privileges).

Basically, there are two common file recovery methods available:

1) File Carving
2) Filesystem Analysis

The specifics of interpreting the results of these two well-known methods in the Docker context are discussed in more detail in the following sections.

4.2.1 File Carving

The term "file carving" (Anandabrata Pal, 2009) is typically used to describe the method of linearly searching a volume, disk image, or file for characteristic patterns (magic bytes) for the beginning and/or end of files. Since the file system is not considered, this approach can recover both allocated and deleted files that have not yet been overwritten. However, fragmented files (with a few exceptions from specialized carvers for certain single file types) can only be reconstructed incompletely. Furthermore, any meta-information about the files, such as their filename, path, timestamp, or similar is missing.

Due to this limitation, the usage for forensic investigations in the Docker environment is subject to problems: Assigning a previously deleted file recovered by file carving to a specific container, or even just distinguishing whether the file belonged to a Docker container or the host system itself, is only possible with metadata. However, as discussed above, files that have been restored by carving usually lack such information, so that a reliable assignment is no longer possible. Only if the content of the file itself provides information about its context (for example, if it contains a ContainerID), an association can be made.

One possible approach for obtaining at least a rough idea of the origin of such a file is to exclude all files restored by file carving, which are still stored in an allocated form in the file system, or that can be restored with file system information (as described below). Furthermore, it can then be checked for the remaining previously unknown carved files, to which block group the individual data blocks in which the file was found belong. Since at least ext file systems allocate new files, if possible, in the same block group in which the parent directory is located, this information can provide a rough context. However, this information is very vague and may also be inaccurate so that it may at best be considered helpful in case of incident analysis, but will be too uncertain for use in forensic investigations.

4.2.2 File System Analysis

In contrast to file carving, the file system analysis uses management structures stored in the file system, such as the MFT (master file table) in the case of NTFS or the inode tables of the group descriptors in the case of ext file systems. Depending on the file system and circumstance, deleted files can also be recovered based on this information, as described in detail by Brian Carrier (Carrier, The sleuth kit, 2007) and implemented in the Sleuthkit toolset (Carrier, The sleuth kit, 2007). Although a deleted file can not always be recovered using this method, it does provide some benefits if applicable: On the one hand, fragmentation of deleted files is not a problem (as opposed to file carving), and on the other, metadata such as file names, entire paths and timestamps can be reconstructed. This information (especially file names and paths) enables to determine the origin of such a file to the host system or a container (and to which particular container). This assignment essentially is accomplished via container/image IDs which are located in the path of a file belonging to a container. Here we differentiate between Docker's two predominantly used layer file systems, the older *AUFS* and the in the current version of Docker used *Overlay2*, in which the assignment is slightly different. We address them separately in the next sections.

4.2.2.1 AUFS

In Section 4.1, we described how the ContainerID of existing Docker containers can be determined. Now we assume that we could recover a deleted file from a container using file system analysis. Therefore, the full original path of the file is known. We now explain how the container to which the file belonged can be derived. In the case of AUFS, files of a container are stored in the `/var/lib/docker/aufs/` directory.

Now the recovered file may stem from different layers. The individual layers can be found in individual directories under their AufsID: `/var/lib/docker/aufs/layers/$AufsID/`. So first, the AufsID is required be extracted from the recovered path. To infer an association to a container with the AufsID, for all existing containers, as described above, the ContainerID and the file `/var/lib/Docker/image/aufs/layerdb/mounts/$ContainerID/mount-id` must be read out. The ID stored in this file can then be compared with the AufsID. In case of a match, the container that originally contained the file was successfully identified.

4.2.2.2 Overlay2

Compared to AUFS, Overlay2 uses another ID called MountID. The MountID of a container is held in the container configuration file `/var/lib/docker/containers/ContainerID/config.v2.json` and identifies the r/w layer of the container stored in the following folder:

`/var/lib/docker/overlay2/$MountID`. The folder `/var/lib/docker/overlay2/$MountID-init` is created during initial startup of the container and thereby allows fast indexing of all r/w layers within `/var/lib/Docker/overlay2/`. The deeper layers are stored in the file `/var/lib/Docker/overlay2/$MountID/lower`[2] in an abbreviated form (the LayerIDShort) in descending order. The LayerIDShort is used as the name of a symbolic link in the folder `/var/lib/Docker/overlay2/l/` which points at `/var/lib/Docker/overlay2/$LayerID`. Also relevant for further forensic analysis are the subfolders `diff` and `merged` in `/var/lib/docker/containers/$ContainerID/`. The `diff` folder contains all files that were created and not yet deleted in the r/w layer. The `merged` folder contains the view of the entire file system across all layers provided by Overlay2. When deleting a file in the r/w layer provided by a lower layer, an inode is allocated, flagged as a Linux character device, thereby instructing Overlay2 to ignore this file for the overall view. Files that were created in the r/w layer and subsequently deleted are deleted with normal operating system/file system functions.

Section 7 contains a script that outputs all relevant IDs to a provided ContainerIDShort when using Overlay2.

4.3 Recovery of Files of an Image Layer

In this case, a deletion reference is stored in the r/w layer as shown in Figure 3, but the file remains in the image layer.

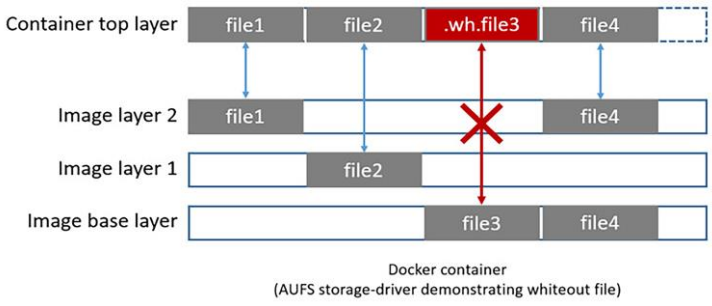

Figure 3: Deletion in LayerFS (Docker Inc, 2018)

[2] *More examples on*

Overlay2 allocates an inode in the r/w layer, which bears the name of the deleted file and is marked as a character device by a file system flag. Thus, all files deleted in a lower layer can be identified by the following command: `find /var/lib/Docker/overlay2/$ContainerID/diff -type c`. File recovery is then possible by means of iterating through the layers and checking whether the corresponding file is existent.

We would like to note that the procedure described here works analogously in the post-mortem analysis of the entire host system. Equally, corresponding inodes can be identified and, if appropriate, the associated files can be extracted from the directory of the underlying layer containing the original file.

4.4 Namespaces

Linux namespaces result in various effects on forensic analysis. The UTS namespace allows the configuration of container-specific time zones. Potential time differences must be considered in the live analysis of containers, but do not affect a post-mortem analysis. Overlay2 always uses the time of the host system when modifying files and dynamically adjusts timestamps for each container at runtime.

The PID namespace may cause a process on the host and a process in one (or more) containers to receive the same process ID (PID). The PIDs on the host are always unique, only the PID within the container can be displayed identically to a PID on the host. This fact becomes relevant when log files contain PIDs and should be used for post-mortem analysis. A translation from container PID to host PID is not possible without runtime information.

Similarly to PID namespaces, user namespaces allow to map a user ID (UID) or group ID (GID) from one container to another UID on the host. For example, a process can run inside a container with UID 0, but the corresponding process on the host runs with UID 65000. Like in PID namespaces, this results in problems in the analysis if log files contain UIDs, but in this case an assignment of container UID/GID to host UID/GID via the `/etc/subuid` and `/etc/subgid` is possible.

4.5 cgroups

Cgroups have the least impact on forensic analysis and are not exclusively relevant to the analysis of Docker hosts. Generally speaking, cgroups can have descriptive names that can be used as additional evidence (such as for processes running on the host and their intended use). cgroups can be created dynamically at runtime or through background services and based on configuration files. At runtime, existing cgroups can be identified through the sysfs virtual kernel file system `/sys/fs/cgroup` A

persistent configuration of cgroups is possible via the file `/etc/cgconfig.conf`; the file contains the cgroup names and can potentially provide further evidence in the context of an analysis.

4.6 Container Management

The Docker administration components (so-called `containerd`) optionally offer network access. `containerd` exposes a network port that provides access to Docker's entire management functionality. The access to this network interface does not require authentication by default and therefore leads to the possibility of unauthorized starts of containers. Accordingly, it is no longer possible to tell which user was responsible for starting a specific container.

The network exposure of `containerd` can be defined during runtime with the command line call which contains the option `-l` or `–listen`. A persistent configuration depends on the operating system; typical locations for the corresponding configuration files are `/etc/default/Docker` and `/etc/Docker/daemon.json`.

5 Summary and Outlook

This article discusses the impact of using Docker containers on incident analysis and forensic processes. In the following we summarize the work briefly, give restrictions, give an outlook on future work and close the article with a conclusion.

5.1 Summary

In Section 3, we discussed Docker's basic techniques before going in Section 4 for Docker's specifics in the context of forensic investigations. Specifically, in Section 4.1 we discussed the possibilities, difficulties, and approaches for the reconstruction of deleted files and their association with Docker containers, notably the differences in the layer from which a file was deleted (Section 4.2 and 4.3), and the differences in the assignment between AUFS and Overlay2. We also discussed the possibilities and limitations of file recovery with file carving. Section 4.4 dealt with forensically relevant aspects of Docker namespaces, while Section 4.5 dealt with the so-called cgroups and the topic of container management (Section 4.6).

5.2 Limitations and Future Work

This paper introduces Docker forensics and highlights specifics of forensics or incident analysis of a Docker environment compared to physical hosts and classic virtualization techniques. The consideration of artifacts, which are specifically caused by Docker, is not yet exhaustive and must be continued in detail in the future. So far, the topic of live forensics of a Docker container and a Docker host has not been considered, as well as difficulties and necessary adjustments in the memory analysis of a Docker host. Another unresolved topic is the investigation of possibilities for the reconstruction and assignment of artifacts from completely deleted containers, instead of single deleted files of a still existing container. Special features of Docker Swarms must also be considered in the future. For all these aspects, adjustments to existing forensic tools or the development of specialized plug-ins for the Docker context may be necessary, for example, for main memory analysis tools such as Volatility and Rekall.

5.3 Conclusion

With this article, we reviewed Docker's specifics in forensic investigations and incident analysis, and discussed how deleting files in Docker containers affects their recoverability. We differentiated and explained the deletion of a file in the r/w layer compared to a deletion in an underlying layer, the use of the

older AUFS compared to the currently used Overlay2 by Docker, as well as the reconstruction of data by file carving compared to a reconstruction by file system analysis. Furthermore, problems and limitations of these methods were discussed, as well as limitations of this paper and an outlook on further subjects to be considered in this field.

6 References

Anandabrata Pal, N. M. (2009). *The evolution of file carving.* IEEE Signal Processing Magazine.

Carrier, B. (2005). *File System Forensic Analysis.* Pearson Education.

Carrier, B. (2007). *The sleuth kit*. Retrieved January 7, 2018, from http://www.sleuthkit.org/sleuthkit/desc.php

Datadog Inc. (2016). *8 surprising facts about real docker adoption*. Retrieved March 14, 2017, from https://www.datadoghq.com/docker-adoption/

Docker Inc. (2018). *Docker and aufs in practice*. Retrieved January 7, 2018, from https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/#container-reads-and-writes-with-aufs

Docker Inc. (2018). *Docker and overlayfs in practice*. Retrieved April 2, 2018, from https://docs.docker.com/engine/userguide/storagedriver/overlayfs-driver/

Docker Inc. (2018). *Docker storage drivers*. Retrieved April 2, 2018, from https://docs.docker.com/engine/userguide/storagedriver/

Docker Inc. (2018). *Understand images, containers, and storage drivers*. Retrieved April 2, 2018, from https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/#images-and-layers

Docker Security Team. (2016). *Securing the enterprise software supply chain using*. Retrieved March 14, 2017, from https://blog.docker.com/2016/08/securing-enterprise-software-supply-chain-using-docker/

Grattafiori, A. (2016). *Understanding and hardening linux containers.* NCC Group.

Jayanth Gummarajul, T. D. (2015). *Over 30 security vulnerabilities*. Retrieved April 2, 2017, from https://www.banyanops.com/pdf/BanyanOps-AnalyzingDockerHub-WhitePaper.pdf

Jeeva Chelladhurai, P. R. (2016). *Securing docker containers from denial of service (dos) attacks.* Services Computing.

Jiang Du, S. W. (2016). *Dcff: a container forensics framework based on docker.* 3rd International Conference on Materials Engineering, Manufacturing.

Open Container Initiative. (2018). *Open Container Initiative*. Retrieved April 2, 2018, from https://www.opencontainers.org/about

Portworx, Inc. (2017). *Portworx Annual Container Adoption Survey 2017*. Retrieved February 07, 2018, from https://portworx.com/wp-content/uploads/2017/04/Portworx_Annual_Container_Adoption_Survey_2017_Report.pdf

Theo Combe, A. M. (2016). *To docker or not to docker: A security perspective.* IEEE Cloud Computing.

7 Appendix A: Script Container-Information

The following listing represents a script that outputs all relevant IDs of a container in Overlay2:

```
#!/usr/bin/env bash
set -e
if [ -e $1 ];
then
     echo "Please provide container ID as argument."
     exit
fi

short_id="$1"
docker_lib="/var/lib/Docker"
docker_containers="$docker_lib/containers"
docker_overlay="$docker_lib/overlay2"

tmp_dir=$(echo $docker_containers/$short_id*)

if [ ! -d $tmp_dir ];
then
     echo "No container matched the provided container ID."
     exit
fi

long_id=$(basename $tmp_dir)

image_id=$(grep -Po 'Image":.*?[^\\]",' \
     $docker_containers/$long_id/config.v2.json | \
     grep sha256 | cut -d ":" -f "3" | cut -d '"' -f 1)
image=$(grep -Po 'Image":.*?[^\\]",' \
     $docker_containers/$long_id/config.v2.json | \
     grep -v sha256 | cut -d '"' -f "3")

mount_id=$(cat $docker_lib/image/overlay2/layerdb/mounts/$long_id/mount-id)
path_to_rw_layer="$docker_overlay/$mount_id/diff"
path_to_live_mount="$docker_overlay/$mount_id/merge"
# list is ordered from highest layer to lowest layer
layer_list=$(cat $docker_overlay/$mount_id/lower)
```

```
IFS=':' read -r -a layer_array <<< "$layer_list"
echo "=========== Container $long_id ======================="
echo "|"
echo "| Image:
$image"
echo "| ImageID:
$image_id"
echo "| MountID:
$mount_id"
echo "| Container Config: $docker_containers/$long_id"
echo "|"
echo "|======================================================"
echo "| Layers:"

for index in "${!layer_array[@]}"; do
     if [ $index -eq 0 ];
     then
            continue
     fi
     ll=$(readlink $docker_overlay/${layer_array[$index]})
     layer=$(echo $ll | cut -d '/' -f 2)
     echo "| $docker_overlay/$layer"
     echo "|-----------------------------------------------"
done
```