Docker Containers for Legal Professionals

Armijn Hemel, MSc – version 2020-04-01-1

www.linuxfoundation.org

Table of Contents

About the Author

Armijn Hemel, MSc, is the owner of Tjaldur Software Governance Solutions. He is an active researcher of and internationally recognized expert in open source license compliance and supply chain management. He studied computer science at Utrecht University in The Netherlands, where he pioneered reproducible builds with NixOS. In the past he served on the board of NLUUG and was a member of the coreteam of gpl-violations.org. Currently he is a board member at NixOS Foundation and creates tools for and documentation about license compliance.

Table of contents 2

Introduction 3

Historical perspective on docker 4

Separation 4
Hardware virtualization/emulation 5
Hypervisors 5
Hybrids 5

Docker container technology deep dive 5

Containers and images: what is the difference? 6
Docker layers 6
Process for deploying a docker container 7
  Creating or downloading a docker image 7
    Loading a docker image 7
    Creating a docker image using a “dockerfile” 7
      “Hello world!” Example 8
      Creating a docker image 8
    Searching and downloading docker images 9
  Creating a docker container 10
  Running a docker container 11
Docker repositories and registries 11
  Configuring repositories to search 12
  Fetching images from a repository 12
  Publishing images in a repository 12
Docker on-disk representation 12
  Full docker image 15

License compliance questions for docker containers 16

What software is distributed? 16
  Distributing a full image 17
  Distributing a recipe to build a container image 18
Who distributes the software? 18
The license of dockerfiles versus software inside containers 19
Compliance for all layers, not just the final layer 19
How to collect and publish the required source code? 21

License compliance checklist for docker 21

Is any software distributed? 21
What software is distributed? 21
How is the software distributed? 21
Who is distributing the software? 21

Existing tools and other research 22

Conclusion 23

Introduction

Deployment, distribution, and execution of software and especially services have significantly changed in the last few years. A few years ago, a person had to install a Linux® based distribution with the necessary software and dependencies — these days, it is now much more common to "spin up a container" and run a service. A container is basically nothing more than one or more applications with all dependencies, data, and configuration in a single isolated environment that can be deployed without the need to buy a new system or create a virtual machine. A typical use case would be to isolate services from each other.

There are many tools that make it easy to deploy many containers at once for very short-lived or very long-lived tasks. These tools have simplified the task of installing and configuring software. Containers take away the burden of preparing a machine and installing, configuring, and maintaining software packages.
The goal of this article is to shed light on one of these technologies, namely the docker run-time container engine popularized by Docker®, Inc.

Docker has had a significant impact on the popularity of containers and made it much simpler on the technological side of things, but on the legal side, there are potentially increased complexities. For example, using containers makes it easier for developers to deploy software, but it also makes it easier to deploy (sometimes inadvertently) the wrong thing. Docker hides many of the implementation details, and developers might end up unknowingly shipping all kinds of software without knowing the license compliance issues that occur as a result.

By making it easy to obtain, build on top of, and deploy containerized software, distribution of software can bypass compliance processes that a company might have in place for more traditionally-distributed products and services. The focus of this article will be on analyzing what the compliance challenges are and how to prepare for and remedy these challenges. The article will touch on how to manage, start, or run a container, to the extent it is relevant for understanding the compliance implications.

This is not meant as the definitive guide about license compliance for containers. Instead, it is meant as a starting point for a more informed discussion about how container compliance should be achieved. There are gaps and likely also errors in this document.

There are many tools that make it easy to deploy many containers at once for very short-lived or very long-lived tasks. These tools have simplified the task of installing and configuring software. Containers take away the burden of preparing a machine and installing, configuring, and maintaining software packages.
The goal of this article is to shed light on one of these technologies, namely the docker run-time container engine popularized by Docker®, Inc.

Docker has had a significant impact on the popularity of containers and made it much simpler on the technological side of things, but on the legal side, there are potentially increased complexities. For example, using containers makes it easier for developers to deploy software, but it also makes it easier to deploy (sometimes inadvertently) the wrong thing. Docker hides many of the implementation details, and developers might end up unknowingly shipping all kinds of software without knowing the license compliance issues that occur as a result.

By making it easy to obtain, build on top of, and deploy containerized software, distribution of software can bypass compliance processes that a company might have in place for more traditionally-distributed products and services. The focus of this article will be on analyzing what the compliance challenges are and how to prepare for and remedy these challenges. The article will touch on how to manage, start, or run a container, to the extent it is relevant for understanding the compliance implications.

This is not meant as the definitive guide about license compliance for containers. Instead, it is meant as a starting point for a more informed discussion about how container compliance should be achieved. There are gaps and likely also errors in this document.

Historical perspective on docker

While containers have become much more prominent in the last few years, they have a long history dating as far back as the IBM® mainframes of the 1960s. The full history of these technologies is out of scope for this article, but it is useful to look at a few of the core concepts in the context of Unix(-like) systems.

Besides containers, there are a few different technologies that are sometimes confused with containers, such as virtualization and hypervisors. These will only be mentioned briefly but not looked at in-depth.

1. **https://deepsec.net/docs/Slides/2015/Chw00t_How_To_Break%20Out_from_Various_Chroot_Solutions_-_Bucsay_Balazs.pdf**

2. FreeBSD handbook, chapter 14 **https://www.freebsd.org/doc/handbook/jails.html**

In this section, the following concepts will be briefly explained:

- Separation and isolation: The processes of one user on a shared physical system should not interfere with and should be invisible to processes run by other users on the system. Containers are the logical extension of this.
- Hardware virtualization: Implementations of CPUs and peripherals in software to allow the creation of virtual machines on a single physical system
- Hypervisor: A program controlling virtual machines (start, stop, pause, snapshots, restore, and so on)

Separation

Beginning in 1979 the so-called `chroot` ("change root") system call was added to various Unix systems. With this mechanism, processes can be partially isolated from each other by only being able to access specific parts of the file system. In essence, a specified directory is set to be the "root directory" for the process, and anything outside of that directory, except subdirectories, cannot be accessed. This mechanism is typically used to isolate processes that the outside world has access to and which are running with special privileges, for example, superuser privileges. A very typical use case for `chroot` is to isolate FTP servers supporting "active FTP," where specific network ports need to be opened that require superuser privileges.

The isolation provided by chroot is partial and only covers the file system, but all other resources in the operating system are still shared. Errors in the operating system could allow an attacker to possibly break out of a chroot environment and access other parts of the system[1].

The FreeBSD project improved upon chroot and created FreeBSD jails[2] which extended this separation to processes as well as the network stack, allowing each jail to have its IP address.

Linux containers took this concept further, providing even more separation. Docker is the dominant technology for Linux-based containers and consists of a set of tools to create, deploy, and manage containers.

Docker also has overlap with software configuration management (SCM) programs and tools (examples: cfengine, Puppet, Ansible) but differs from these because it operates in the context of containers.

Hardware virtualization/emulation

On the other end of the spectrum is hardware virtualization software that emulates a complete system (CPU, peripherals) in software. In these emulators, it is possible to install an operating system as if it were installed on a physical machine. A well-known example is the original VMware® product that emulated the Intel® PC architecture

3. **https://xenproject.org/**

and which would allow people to install other operating systems and work with them, for example, install Linux inside a VMware virtual machine running on Microsoft® Windows, or vice versa.

Other well-known programs that provide hardware virtualization are Bochs (x86), QEMU (various hardware architectures), and Oracle's VirtualBox®.

On Linux, there are also many emulators available, such as for several gaming devices (e.g., Nintendo® Gameboy® and Gamecube®), as well as old home computers (e.g., Apple®] [, Commodore 64®), in which hardware has been partially or completely reimplemented in software.

Hypervisors

Hypervisors are specialized operating systems (either custom or based on a minimalistic version of Linux) that control and manage the launch of virtual machines. A well-known hypervisor is Xen Project®[3].

A difference between Docker and hypervisors is that Docker manages Linux containers (each of which is a program, with its run time dependencies), whereas hypervisors manage entire virtual machines, each with an entire operating system.

Hybrids

The techniques described are not mutually exclusive. It is very well possible to have a hypervisor managing multiple virtual machines running an operating system on which containerized applications have been deployed, so in practice, you can find them used next to or on top of each other.

Docker container technology deep dive

Unfortunately, the term "Docker" is suffering a bit from overload: the technology is called Docker, the company that created it is called "Docker, Inc." and the program that is used to create and deploy containers is called Docker. In this article, it should be clear from the context which is used. If not, it is explicitly mentioned.

4. **https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736**

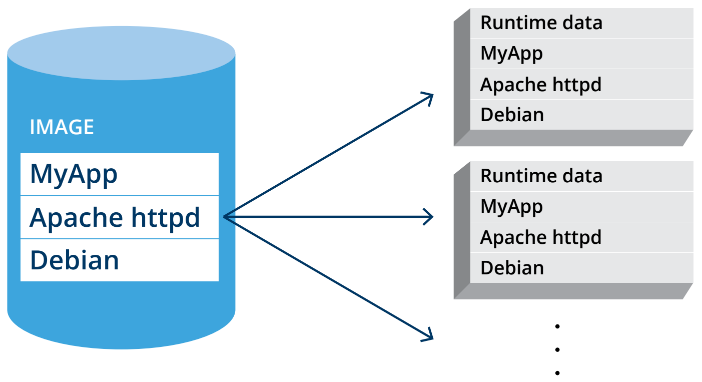
Similar to what Git has done for version control, Docker technology has vastly simplified and democratized creation and deployment of so-called "containerized applications." A Docker "containerized application" is a self-contained environment that contains all of the dependencies and data (shared libraries, programs, configuration files, etc.) that an application would require to run. This methodology allows a strict separation between applications and processes and means that if one application gets compromised the others should not be affected (unless there are errors in the underlying technology[4]).

Environments can use different distributions/versions per container. For example, one container could be based on Fedora, while another one could be based on SUSE, or Debian, or a custom solution. There are benefits to this approach (there can be subtle differences between distributions, and running software in an environment that is known to work reduces headaches) but also potential downsides (possibly running many different environments, complicating maintenance).

Containers and images: what is the difference?

In articles and documentation about Docker, there are frequent references to "containers" and "images." These are not the same, although sometimes used interchangeably in articles or conversations. There is a very fundamental difference: an image is the on-disk collection of software, while a container is a running **instance** of an image, together with run time data and run-time state.

An example image could contain the Apache webserver and all its dependencies, from which a container can be instantiated and run. An image can be instantiated multiple times: these would then all become separate containers.

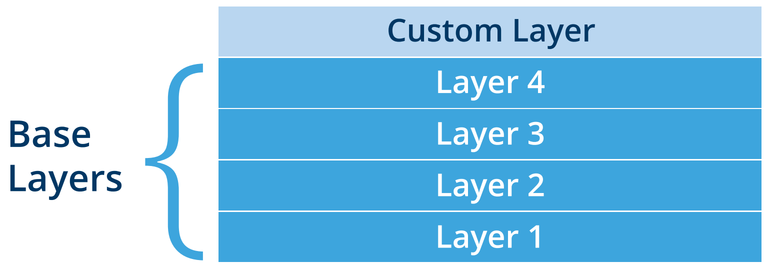


Images can be made available for reuse in public or private repositories, where they can be searched for and downloaded to be reused.

Docker Layers

Each Docker image consists of one or multiple layers that are stacked on top of each other. Some of the layers contain files (programs, files, etcetera); others are meta-layers modifying existing layers. Different images can, and often do, share layers. For example, if two images are both based on a specific Debian layer, then this layer will only be stored on disk once.

If an existing image is reused (for example: downloaded from a repository) and modifications are made, then these modifications are stored as one or more separate layers on top of the existing layers in the image. All the layers of the base image and the new layer with the modifications together form a new image that can be instantiated (to create a container) or exported to be distributed or made available in a repository. A Docker image could be as pictured below: a base image with four base layers (building on top of each other), and a custom layer on top.



5. https://docs.docker.com/engine/reference/builder/

6. Docker is limited to Linux and Linux commands. Although it is possible to run Docker on other operating systems, this is implemented by using a small Linux system in a virtual machine.

As explained before: a container instance is not the same as a Docker image, as the container instance also contains run-time data. The run-time data is stored separately in a data layer.

Process for deploying a docker container

Deploying a Docker container comprises the following steps (installing “docker” and related programs is out of scope for this article, as details are specific to your Linux distribution):

1. Creating or downloading a Docker image
2. Creating a container instance from the image
3. Running the Docker container

Docker images and containers can be searched, created, deployed, and managed (started, stopped, etc.) using the docker tool. There are also complete container orchestration solutions, such as Kubernetes. In this article, these are ignored for the sake of simplicity.

Creating or downloading a docker image

The first step in deploying a Docker container is to create or download a Docker image. This can be done in three different ways:

1. An image can be built from scratch, or
2. A complete image with all the layers can be loaded into the Docker run time, or
3. An image can be assembled “on the fly” by Docker using a combination of recipes and base images (typical use case), i.e., using a Dockerfile

In practice, method 1 is hardly used, except for bootstrapping and creating a base image, and most people use either method 2 or method 3, with method 3 being used the most. These two will be described below.

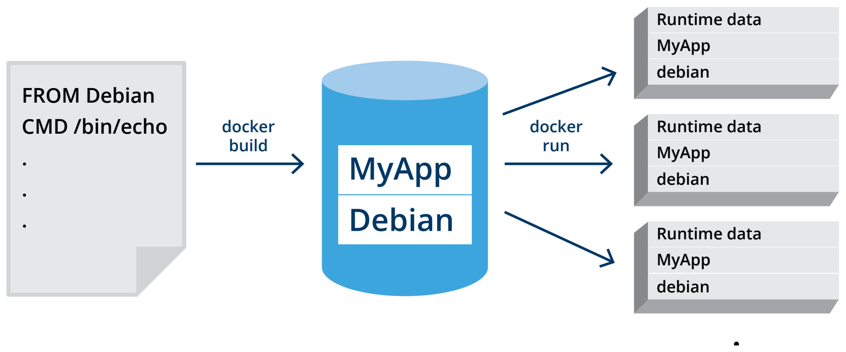
Loading a docker image

A complete image contains all the layers and meta information that is needed by Docker and is loaded into the Docker run time by running the command “docker load.” One possible use case of this method would be to install the image on computers not connected to the Internet.

Creating a docker image using a "dockerfile"

The most common way to create and load images is by using a so-called Dockerfile. A Dockerfile is a file with instructions that describes how to assemble a Docker image. The docker tool reads these instructions and executes them to assemble a Docker image.

A Dockerfile is read to build an image, which in turn can then be used to instantiate a container:



Each of these steps is described in more detail below.

Although Docker defines a limited set of instructions[5] that can be used in a Dockerfile; in practice, there is no limitation to what these instructions can do because any Linux shell command can be run as part of an instruction[6]. Typically a Dockerfile contains instructions about what base image to use, what software to install, which commands to run, and so on.

"Hello World!" Example

A "Hello World!" Dockerfile based on Debian that prints the words "Hello World!" could look like this (line numbers added for clarity):

```
1 FROM debian
2
3 CMD [ "/bin/echo", "Hello World!" ]
```

This file has two instructions: line 1 tells Docker to base the container image on a Debian base image, and line 3 to launch the command `/bin/echo` and print "Hello World!". Each of these commands is stored in a separate layer.

Creating a Docker Image

The image can be created using the docker build command, which gives the following output (line numbers added for clarity):

```
1 $ docker build docker/
2 Sending build context to Docker daemon 2.048 kB
3 Step 1/2 : FROM debian
4 Trying to pull repository docker.io/library/debian ...
5 sha256:d986a531d62903b66e731d475988f5b2ba3a4a90078078cb0f29f9685ee36466: Pulling
6 from docker.io/library/debian
7 dc65f448a2e2: Pull complete
8 Digest: sha256:d986a531d62903b66e731d475988f5b2ba3a4a90078078cb0f29f9685ee36466
9 Status: Downloaded newer image for docker.io/debian:latest
10 ---> a8797652cfd9
11 Step 2/2 : CMD /bin/echo Hello World!
12 ---> Running in 0eb4b6b18619
13 ---> 58a63e12e394
14 Removing intermediate container 0eb4b6b18619
15 Successfully built 58a63e12e394
```

Line 1 is the command that is typed in and executed. On lines 2-9, it can be seen that the Docker daemon (the central program that helps manage containers and images) downloads a Debian base image from a repository (in this case, the central Docker repository) and stores the layer on disk (line 10). This corresponds to the first instruction in the Dockerfile (namely the line "FROM debian").

On lines 11 – 13, the layer for executing the CMD command (line 3 in the Dockerfile) is created and stored separately.

As already discussed, Docker images are modular and can be built upon other images or by reading a stack of Dockerfile files and corresponding images (which might, or might not, be assembled “on the fly” as well).

From a developer standpoint, this is very convenient, as they only need to focus on their application and not on the boilerplate code, which will likely be shared between many containerized applications. Often developers will have a library of Dockerfile files or templates that they will reuse.

Searching and Downloading Docker Images

There are two ways to install containerized applications with Docker:

1. Install from a Docker repository with images (example: official Docker docker.io repository)
2. Download an image and load it separately (already covered in the previous sections)

Docker (the company) has an extensive repository of “official images” called docker.io, which can be searched in a registry using Docker (the tool). As an example, the official Docker image for Nextcloud can be found by using the following command (formatted output edited for clarity):

```
$ docker search nextcloud
INDEX        NAME                                     DESCRIPTION                                 STARS     OFFICIAL   AUTOMATED
docker.io    docker.io/nextcloud                      A safe home for all your data               544       [OK]
docker.io    docker.io/nextcloud                      A safe home for all your data               544       [OK]
docker.io    docker.io/linuxserver/nextcloud          A Nextcloud container, brough ...           72
docker.io    docker.io/linuxserver/nextcloud          A Nextcloud container, brough ...           72
docker.io    docker.io/greyltc/nextcloud              Nextcloud: a safe home for al ...           36                   [OK]
docker.io    docker.io/greyltc/nextcloud              Nextcloud: a safe home for al ...           36                   [OK]
docker.io    docker.io/wonderfall/nextcloud           All-in-one alpine-based Nextc ...           36                   [OK]
docker.io    docker.io/wonderfall/nextcloud           All-in-one alpine-based Nextc ...           36                   [OK]
```

The image can then be installed from the repository using docker (the tool):

```
$ docker pull docker.io/nextcloud
Using default tag: latest
Trying to pull repository docker.io/library/nextcloud ...
sha256:55f042577e61f3ce826476a29aa77435e61707abc88f3665d415c519cca85c13: Pulling from docker.io/library/
nextcloud
be8881be8156: Downloading [==============================>      ] 19.79 MB/22.49 MB
69a25f7e4930: Download complete
65632e89c5f4: Downloading [===============>                   ] 20.43 MB/67.43 MB
```

after which it can be instantiated and run.

Other repositories also exist, and on systems such as Fedora, some will be preconfigured in the Docker configuration file.

Searching for an image is similar to, for example, searching a package using dnf on Fedora, apt-cache on Debian, or other package management systems such as Python's pip, NuGet, and others. By running `docker search`[7] or `docker pull`[8] all configured repositories are automatically searched. If "docker search" cannot find an image, it will report that nothing can be found:

```
$ docker search fbbbrrrtzzzz
INDEX    NAME     DESCRIPTION    STARS    OFFICIAL   AUTOMATED
```

When using `docker pull,` an error will be displayed instead:

```
$ docker pull fbbbrrrtzzzz
Unable to find image 'fbbbrrrtzzzz:latest' locally
Trying to pull repository docker.io/library/fbbbrrrtzzzz ...
Trying to pull repository registry.fedoraproject.org/fbbbrrrtzzzz ...
```

7. https://docs.docker.com/engine/reference/commandline/search/

8. https://docs.docker.com/engine/reference/commandline/pull/

```
Trying to pull repository quay.io/fbbbrrrtzzzz ...
Trying to pull repository registry.access.redhat.com/fbbbrrrtzzzz ...
Trying to pull repository registry.centos.org/fbbbrrrtzzzz ...
Trying to pull repository docker.io/library/fbbbrrrtzzzz ...
repository docker.io/fbbbrrrtzzzz not found: does not exist or no pull access
```

Creating a Docker Container

An instance is created using the command `docker create`. The manpage (short for manual page) of the `docker create` command says the following:

```
Creates a writeable container layer over the specified image and prepares it for running the
specified command.
```

The "docker create" command takes a few parameters as can be seen in the (heavily cut) help text for the command:

```
$ docker create --help

Usage: docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

The important parameter is "IMAGE." To find out which images are available on the local system, the command `docker image list` can be used:

```
$ docker image list
REPOSITORY            TAG     IMAGE ID        CREATED                 SIZE
<none>                <none>  58a63e12e394    About a minute ago      114 MB
docker.io/debian      latest  a8797652cfd9    3 days ago              114 MB
```

The `IMAGE ID` value is what will be used by `docker create`. The "Hello World" container that was built earlier has the image id `58a63e12e394` (lines 13 and 15 of the build log). The Debian image it is built on can also be found in the image list and has an image id corresponding to the build log (line 10).

Creating a docker container can be done as follows:

```
$ docker create 58a63e12e394
34590ba2e6271640a0c26e066afbeb7acf42dabb0ac0a779babedc97f18e1249
```

This command outputs a container identifier.

The result is the identifier of a container, which is an instance of an image. The status of the containers can be shown using the `docker ps -a` command (output edited for clarity):

```
$ docker image list
CONTAINER ID    IMAGE          COMMAND                 CREATED               STATUS
34590ba2e627    58a63e12e394   "/bin/echo 'Hello ..."  28 seconds ago        Created
```

In the output, the container ID is shown (abbreviated), as well as the image identifier.

Running a docker container

The Docker container can be started using the "docker start" command using the Docker container ID:

```
$ docker start 34590ba2e627
34590ba2e627
```

In this case, nothing much happens, but anything that is logged can be found in the container logs:

```
$ docker logs 34590ba2e627Hello World!
```

Of course, this is just a straightforward Docker container and deployed using the simplest tools. Whole orchestration suites have evolved around containers, such as Kubernetes, which allow deployment and management of many containers on many platforms.

Docker repositories and registries

Docker images can be retrieved from repositories. Apart from the already introduced docker.io repository (run by Docker) there are also other repositories, such as quay.io, which is run by Red Hat. Community projects such as Fedora and CentOS also have public repositories, and there are many running their private repositories of Docker images.

Configuring repositories to search

In a default Docker installation, the repositories from Docker will be searched, but the Docker configuration can be changed to use different repositories. As an example, on a current Fedora Linux system the following repositories are defined (in `/etc/containers/registries.conf`):

```
docker.io
registry.fedoraproject.org
quay.io
registry.access.redhat.com
registry.centos.org
```

On other distributions, other repositories might be configured. Companies might also have their own (internal) repositories. Depending on the configuration of the repositories, different images might be found, as different repositories might have overlapping names for images.

Fetching images from a repository

When an image is pulled from a repository (using, for example, `docker pull`), not all data is necessarily downloaded: if some layers are already present on a system, these will not be redownloaded. Metadata (describing the layers) will always be downloaded.

Publishing images in a repository

Apart from downloading images from a repository, they can also be published on a repository using the `docker push` command[9]. Depending on the version of Docker and the Docker API either the full image with all layers will be uploaded, or unique layers (i.e., those not already present in the repository where the "push" is directed) will be uploaded[10].

Docker on-disk representation

Although container images are typically intended to be treated as hermetically-sealed units, they can be analyzed just like any other software stored on disk. This type of analysis can be useful in understanding the compliance implications for using, distributing, and otherwise working with a particular container image or an individual image layer.

When a Docker image is installed, its contents can be inspected. The locations on a Linux machine where these images are installed can differ per installation, but typically the images are in a system directory. On a default install on Fedora Linux, you will, for example, find it in `/var/lib/docker`.

At first glance, it is difficult to find out where the data of a container exactly resides on the disk, as there are multiple directories where data is stored. In this section, the standard Docker "Hello World!" example (note: this is not the same as the earlier "Hello World!" example!) will be dissected. To better understand these concepts, it is recommended to not just read through this section, but to actually perform the steps.

The first step is to pull the image from the online registry:

```
$ docker pull hello-world
Using default tag: latest
Trying to pull repository docker.io/library/hello-world ...
sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788: Pulling from docker.io/library/
hello-world
d1725b59e92d: Pull complete
Digest: sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788
```

9. https://docs.docker.com/engine/reference/commandline/push/

10. https://forums.docker.com/t/why-does-docker-push-base-image-layers-that-are-already-on-docker-hub/2329/7

```
Status: Downloaded newer image for docker.io/hello-world:latest
```

Using the `docker pull` command, the `hello-world` container image is downloaded from the registry, unpacked onto disk, and added to the list of locally available images. The locally available images can be found in the file `image/overlay2/repositories.json`:

```
{
   “Repositories” : {
       “docker.io/hello-world” : {
       “docker.io/hello-world:latest” :
“sha256:4ab4c602aa5eed5528a6620ff18a1dc4faef0e1ab3a5eddeddb410714478c67f”,
       “docker.io/hello-world@sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499
788” : “sha256:4ab4c602aa5eed5528a6620ff18a1dc4faef0e1ab3a5eddeddb410714478c67f”
       }
   }
}
```

In this file, it says that for the repository `docker.io/hello-world` one image is available, which is made available under two names: `latest` and a SHA256 digest. The digest is the same as the value of “Digest” that was printed when the entry was pulled (in this case `0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788`), as only one version of the image was pulled from the repository.

In the file repositories.json file it can be seen that the digest points to another SHA256 hash (`4ab4c602aa5eed5528a6620ff18a1dc4faef0e1ab3a5eddeddb410714478c67f`). In the directory `image/overlay2/imagedb/content/sha256/` a file with this particular hash as its name can be found.

This file, which is another JSON file, describes the image in much more detail, including:

- Creation date
- Architecture
- History of commands used to create the container
- The list of layers
- Etc.

which provides a good starting point for a thorough analysis.

The list of layers that store the content is the most interesting, as these describe what is actually used by the container. The list of all layers used (but not how they are stacked) can be found in the JSON file in the element `rootfs`:

```
"rootfs" : {
       "type" : "layers",
       "diff_ids" : [
       "sha256:428c97da766c4c13b19088a471de6b622b038f3ae8efa10ec5a37d6d31a2df0b"
       ]
```

In this case, there is only one layer, as it is a simple container. In other containers, there could be many more layers. Each layer is referred to by yet another SHA256.

For images installed via a Docker registry, a file with the SHA256 value for each layer is created in `image/overlay2/distribution/v2metadata-by-diffid/sha256` with the same name as the SHA256. This file is a JSON file that contains some more information, including for example the origin of the layer:

```
[
   {
       "SourceRepository" : "docker.io/library/hello-world",
      "HMAC" : "",
       "Digest" : "sha256:d1725b59e92d6462c
        6d688ef028979cc6bb150762db99d18dddc
        7fa54b82b0ce"
   }
]
```

For images that are loaded from a complete image file, these files are possibly not present, as they are not necessarily coming from an online repository, so depending on the setup, these files could be missing.

In the directory `image/overlay2/layerdb/sha256/` there are directories for each of the layers. Inside these directories there are a number of files:

```
cache-id
size
tar-split.json.gz
diff
parent
```

where the file `diff` contains the SHA256 of the layer, and the file parent is not present for the first layer. For the other layers, it contains the hash of the parent layer, and by following the parent hashes for each parent layer, it is possible to find how the layers are stacked. For “hello world,” there is just one layer, so the file parent is missing.

The file `cache-id` contains yet another SHA256 checksum (`e12deaf42bcd370442e87e06ded3f86efaa21c c4014ae57afeb462980e8ffb6d` in this particular install that was tested with, but it will differ per install). A directory with the same name can be found in the directory `overlay2`. The subdirectory `diff` has the actual contents of the layer:

```
$ ls -l ./overlay2/e12deaf42bcd370442e87e06ded3f86efaa21cc4014ae57afeb462980e8ffb6d/diff/

total 4

-rwxrwxr-x 1 root root 1840 Sep  7 21:25 hello
```

The file size contains the size of the layer. This file contains just a single value, and in this case, it is 1840, which is the same as the size of the `hello` executable.

As can be seen, there are lots of indirections to get to the final data that is used in a Docker container, but it is relatively easy to automate this analysis.

Full docker image

A full Docker image can be exported and distributed using `docker save`. The result of this command is a tar archive, one directory per layer, plus associated metadata files that correspond to the metadata of the on-disk representation:

```
manifest.json

repositories

<hash>.json
```

where the third file can have a different name every time, as it contains a cryptographic hash in the file name.

License compliance questions for docker containers

At its core, license compliance for Docker images and containers is no different from compliance for software distributed via other media, such as firmware for routers or phones, or a CD, so your existing analyses on whether the software is “distributed” can continue to apply. There are a few aspects to it that can make it a lot more complicated, such as the fact that assembling Docker images is not reproducible because it depends on the configuration of which docker repositories are searched and the state of those repositories at a particular point in time.

When looking at license compliance for Docker containers, it is essential to take the following into account:

1. What is distributed?
2. Who distributes the software?
3. The license of Dockerfile files versus software inside containers
4. Compliance for all layers, not just the final layer
5. How to collect and publish the required source code?

What software is distributed?

Because there are a few different distribution mechanisms of software with Docker containers, there is no single answer to the question, “how should I comply with license conditions?”. In this section, the following scenarios will be looked at:

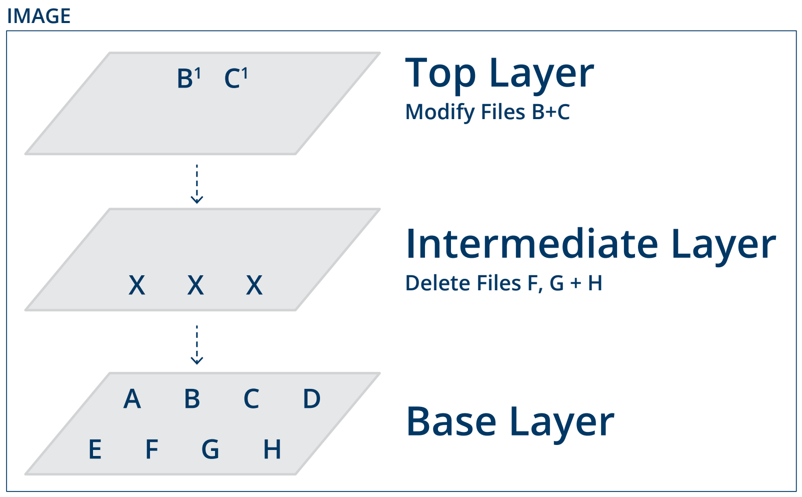
1. Distributing a full image
2. Distributing a Dockerfile file (the “recipe” used to build a container image)

A running container cannot be easily distributed, because it is the actual instantiation of an image that is currently executing. But images from which containers are instantiated, and Dockerfile "recipes" used to build images, are distributed.

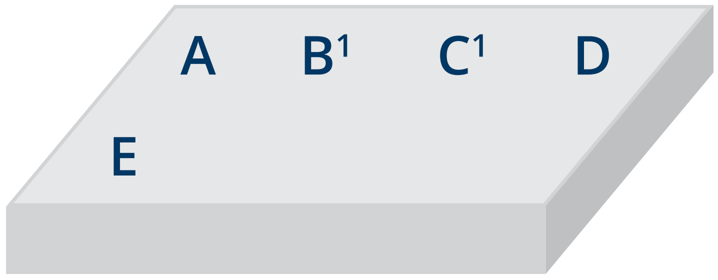
Distributing a full image

Distributing a full image is not any different from distributing a CD with the software on it, or a wireless router with software installed on flash memory. Therefore, it is important to understand what software is actually distributed, which means that the contents of the image need to be inspected.

When inspecting the contents of the image, it is essential to realize that the image that users interact with is simply a view of all layers, and during run-time, only the final view is seen. This view will possibly not show all of the software that is inside all of the layers: each layer can modify the view, but it will not change the content stored in any of the underlying layers.



It could be that one layer installs software, and a new layer overwrites the software with another version (possibly with another version under a different license). In the final view, users see that the older software seems to have been removed, but in the underlying layer, the original software will still be present. For example, with the representation of a container image shown above, a user of the image would only "see" the following:



However, when the image is distributed, all files from all layers will be distributed: each of A through H, and also modified files B’ and C’.

If a complete image is shipped, then the license conditions for software in all layers apply, even if, in the **final view**, software in some of the layers can no longer be seen. This means that for compliance with a full image, every layer that is inside the image should be checked.

An extra complexity could be linking. If a component is linked with other components in a layer and components get overwritten with other versions under a different license, then the license implications might be different depending on which layer is examined.

Distributing a recipe to build a container image

As discussed earlier Docker can also build images in a “just in time” fashion where instead of a full disk image or container image only a recipe to assemble a container image is provided using a Dockerfile.

The software is assembled on the fly from a base image that is downloaded from a repository, or that is available on the local system, with possibly extra software being installed from (other) upstream sources, like installing updates from a Linux distribution.

An example is a recipe that defines that the base image is based on a specific version of Ubuntu Linux, with updates being pulled from the Ubuntu update servers and then having a proprietary program installed from a local server.

The recipes, such as a Dockerfile, can be stacked and depend on other recipe files. These recipe files are typically stored in local files or registries that can be searched.

The recipes used for assembling the images are sometimes released under licenses that are different than the actual software being aggregated (which is perfectly fine), and there is a real chance that people will misinterpret this information and think that the licenses of the Dockerfile files apply to the assembled image, which is incorrect.

Misinterpretation of license provenance within a container image is not an imaginary problem as something similar happened in a non-containerized context in the past with Android, of which large parts have been released under Apache 2. This license confusion led some people to believe that all of Android had been released under Apache 2, even though there were significant portions of Android released under GPL-2.0 (Linux kernel, iptables, etc.) and various versions of the LGPL license.

Who distributes the software?

From a compliance point of view, there is a big difference between distributing an image that has already been fully created (and when all the software is included in the image) and distributing a Dockerfile that only describes how the image should be built. In the former case, the software is distributed in binary form, while in the latter form, possibly only a recipe for constructing an image is distributed.

When a complete image is distributed, only one party is doing distribution of the image -- the party sending the built image out the door. However, when a Dockerfile (the recipe) is distributed, then when the end-user builds the image, it is assembled on the fly with software possibly being pulled from various places. In this scenario, it means distribution is possibly done by several parties, because (e.g.) each layer could come from a different third party. One layer could be distributed by the party operating an image repository, with content in another layer coming from a distribution (example: distribution updates), and content in another layer coming from yet another party (custom download location).

To complicate matters, this also depends on the configuration of the image repository: many Dockerfile files have not explicitly pinned the image that is used to a specific release of the image but simply specifies that the latest image should be used. In theory, each image assembled using a Dockerfile file could be a different image, with different software, possibly under different licenses, pulled from different places, and so on, because the source repositories changed between the builds.

One complication is that for convenience and speed, people provide pre-built images (either of complete disk images or individual packages) covered by free software licenses, without a written offer or the complete and corresponding source code. As an example, many images provide binary builds of BusyBox (licensed under GPL-2.0) for various architectures, without a written offer or source code being made available.

As users can push images or layers to repositories, but the images or individual layers are then indexed, made available and distributed by the operator of the repository, the question arises who should be responsible for license compliance. Should the compliance process be done by the user pushing a full image or a layer to a repository, or the repository owner who then distributes the built image to others?

Hypothetically a company providing “Docker builds as a service” -- where after uploading a Dockerfile, a full image would be created and made available -- could possibly be responsible for license compliance.

The license of dockerfiles versus software inside containers

The Dockerfile files can be licensed under an open source license themselves. It is vital to realize that the scope of this license statement is only the Dockerfile and not the container image.

For example, the Dockerfile itself can be licensed under the MIT license but describing the installation of GPL licensed software. In a typical use case, the license of the Dockerfile and the license of the described software are entirely independent.

Compliance for all layers, not just the final layer

When distributing an image, typically, multiple layers are included and distributed. As these layers are stacked on top of each other, it could be that the contents of one layer obscure the contents of the other layers. From a pure license point of view, the final view does not matter: what matters is what is distributed. It could be that one version of an open source licensed program is distributed in one layer, with another version of the same program distributed in another layer. In this case, the license conditions for both versions need to be met.

Also, to be safe, one should not rely on the fact that specific layers of their image may already exist at the destination or in the repository receiving the push, and that therefore they will not distribute the software for those layers. At some point, that container will land in a place where none of the layers are pre-existing, so all layers of the image must be provided to the recipient, and one will have to comply with distribution obligations for each layer provided.

How to collect and publish the required source code?

An open question is how to collect complete and corresponding source code for containers with software under licenses that require complete and corresponding source code. With the current Docker infrastructure, it is not possible to automatically gather and publish the required source code. This means that extra work needs to be done (either manual or scripted) to gather the right source code, store it, and make it available. There are a few complications:

1. Creating a Docker image is not reproducible but depends on configurations. If different configuration options are chosen every time a Docker image is created, the resulting image could be different. This means that gathering source code at a later time than image creation might not yield the corresponding source code for the image created earlier.
2. Layers can be composed at different times and source code for some layers might have disappeared.
3. Source code might need to be gathered from various places. Focusing on just system packages could lead to missing packages.
4. Gathering source code needs to be done for all layers.

This is currently an unsolved problem.

License compliance checklist for docker

The following section is a compliance checklist that should help companies distributing containers to understand the license obligations better.

Is any software distributed?

The first question to ask is: is any software distributed at all? If the only thing that is distributed is a Dockerfile recipe that needs to be instantiated by the user, and software gets pulled from repositories and the company publishing the Dockerfile does not run the repository and also did not push (base) images that are used in that Dockerfile into that repository, then the company is likely not distributing any software other than the Dockerfile itself. This would require a thorough investigation of the used Dockerfile files and the build process.

What software is distributed?

Knowing which software is shipped and what license this software is under requires analyzing the software in all layers of the container image, not just the final (assembled) layer that is presented to the user. The software might have been hidden from view in the final layer, so a full analysis of all layers is necessary.

How is the software distributed?

Depending on how the software is distributed (as a full image, Dockerfile, etc.), different parties might be responsible for fulfilling license obligations. Question 1 already touched upon this, but a full analysis of the layers and their origin, as well as the install process, is important to understand who is responsible for ensuring compliance.

If a full image is distributed by you, then it is clear that you are responsible for compliance for the full image's entire contents. If an image is assembled on the fly using a Dockerfile and Dockerfiles/images from one or multiple repositories, then it might be either the parties running the repositories, or the parties that created the images that are pulled from the repositories and pushed them into the repositories, who are responsible for compliance for different portions of the container.

Who is distributing the software?

This question comes up often and is directly related to the previous topic. If you are distributing the container as a whole, then you are responsible for license compliance for all of the software it contains. By contrast, if you are distributing just a Dockerfile which tells people how to build a container, and the recipients are then using your Dockerfile to obtain container layers from third-party locations, then you are perhaps not responsible for license compliance for that software. However, even in this case, your downstream recipients may expect that you are attending to container compliance matters, or that you will provide sufficient information to assist them in doing so. Regardless of who is distributing software, all organizations will benefit from adopting a container compliance policy and process.

Existing tools and other research

This document is far from the first attempt at tackling compliance of Docker containers. For example tools from the Tern project[11] can be used to analyze package information inside a Docker container image.

Conclusion

Docker has made the quick deployment of software much simpler, but also introduces a few legal challenges. What the solutions to some of these legal challenges are is currently not clear.

Evaluating compliance challenges requires a basic understanding of the technical specifics of how containers work and how they are built. With this understanding, it becomes evident how the distribution of containers bears some similarities to more historical means of distributing software while making clearer the aspects that can be obscured.

In this document, which will hopefully serve as a starting point for discussions to what the solutions should be, the following challenges were identified:

1. There are different types of distribution and depending on which form of distribution is chosen you might or might not have an obligation to distribute corresponding source code. It is not yet obvious to casual users when or if obligations are present.
2. The Docker tools and ecosystem currently do not make it easy to collect complete and corresponding source code and are focused purely on assembling container images and deploying containers.
3. Because of the layered approach of Docker and only making the final layer visible it is easy to overlook possible distribution of software. A thorough analysis of what is distributed using tools (such as Tern) is necessary in those cases.

Future opportunities to improve the compliance environment for containers should likely focus on further developing tooling and processes that can collect and publish the corresponding source code in a more automated fashion.