PUBLICATION

Docker containers: What are the open source licensing considerations?

DOWNLOAD REPORT

A new Linux Foundation whitepaper seeks to uncover the complexities of open source licensing implications when distributing and deploying Docker containers.

DOWNLOAD REPORT

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 OS distribution with the necessary software and dependencies — these days, it is now much more common to "spin up a Docker 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. Containers allow services to be isolated from each other and require far fewer resources than a virtual machine, so they are becoming extremely popular in extremely dense multi-tenant hosting environments run by hyper-scale cloud providers, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud.

Docker has had a significant impact on the popularity of containers and has 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 containers hide 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.

The Linux Foundation has recently published a whitepaper by Armijn Hemel that seeks to uncover the complexities of open source licensing implications when distributing and deploying Docker containers — it can be downloaded here. This blog post is a summary of the general findings.

Containers and images: what's 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.

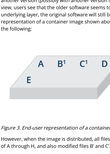

Figure 1. A Docker image instantiated into multiple containers.

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

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 picture book, for example, a base image with four base layers (building on top of each other), and a custom layer on top.

Understanding licensing complexities of layers within containers

When attempting to understand the license implications of software stored within a Docker image, it is essential to realize that the image that users pull is simply a single 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.

Figure 2. A Composite view of a Docker image.

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:

Figure 3. End-user representation of a container image.

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.

Docker repositories, registries and the potential for licensing misinterpretation

Docker images can be retrieved from repositories. Apart from the 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.

In addition to full images being retrieved from repositories, 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 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).

The license of dockerfiles vs. 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.

How do we 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.

A checklist for Docker license compliance

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 result in a significant 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.

Who is distributing the software?

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.

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 our original whitepaper, 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 most 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.