Docker security

Estimated reading time: 11 minutes

There are four major areas to consider when reviewing Docker security:

- the intrinsic security of the kernel and its support for namespaces and cgroups;
- the attack surface of the Docker daemon itself;
- loopholes in the container configuration profile, either by default, or when customized by users.
- the “hardening” security features of the kernel and how they interact with containers.

Kernel namespaces

Docker containers are very similar to LXC containers, and they have similar security features. When you start a container with `docker run`, behind the scenes Docker creates a set of namespaces and control groups for the container.

Namespaces provide the first and most straightforward form of isolation: processes running within a container cannot see, and even less affect, processes running in another container, or in the host system.

Each container also gets its own network stack, meaning that a container doesn’t get privileged access to the sockets or interfaces of another container. Of course, if the host system is setup accordingly, containers can interact with each other through their respective network interfaces — just like they can interact with external hosts. When you specify public ports for your containers or use *links* (https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks/) then IP traffic is allowed between containers. They can ping each other, send/receive UDP packets, and establish TCP connections, but that can be restricted if necessary. From a network architecture point of view, all containers on a given Docker host are sitting on bridge interfaces. This means that they are just like physical machines connected through a common Ethernet switch; no more, no less.

How mature is the code providing kernel namespaces and private networking? Kernel namespaces were introduced between kernel version 2.6.15 and 2.6.26 (http://man7.org/linux/man-pages/man7/namespaces.7.html). This means that since July 2008 (date of the 2.6.26 release), namespace code has been exercised and scrutinized on a large number of production systems. And there is more: the design and inspiration for the namespaces code are even older. Namespaces are actually an effort to reimplement the features of OpenVZ (http://en.wikipedia.org/wiki/OpenVZ) in such a way that they could be merged within the mainstream kernel. And OpenVZ was initially released in 2005, so both the design and the implementation are pretty mature.

Control groups

Control Groups are another key component of Linux Containers. They implement resource accounting and limiting. They provide many useful metrics, but they also help ensure that each container gets its fair share of memory, CPU, disk I/O; and, more importantly, that a single container cannot bring the system down by exhausting one of those resources.

So while they do not play a role in preventing one container from accessing or affecting the data and processes of another container, they are essential to fend off some denial-of-service attacks. They are particularly important on multi-tenant platforms, like public and private PaaS, to guarantee a consistent uptime (and performance) even when some applications start to misbehave.

Control Groups have been around for a while as well: the code was started in 2006, and initially merged in kernel 2.6.24.

Docker daemon attack surface

Running containers (and applications) with Docker implies running the Docker daemon. This daemon currently requires `root` privileges, and you should therefore be aware of some important details.

First of all, only trusted users should be allowed to control your Docker daemon. This is a direct consequence of some powerful Docker features. Specifically, Docker allows you to share a directory between the Docker host and a guest container; and it allows you to do so without limiting the access rights of the container. This means that you can start a container where the `/host` directory is the `/` directory on your host; and the container can alter your host filesystem without any restriction. This is similar to how virtualization systems allow filesystem resource sharing. Nothing prevents you from sharing your root filesystem (or even your root block device) with a virtual machine.

This has a strong security implication: for example, if you instrument Docker from a web server to provision containers through an API, you should be even more careful than usual with parameter checking, to make sure that a malicious user cannot pass crafted parameters causing Docker to create arbitrary containers.

For this reason, the REST API endpoint (used by the Docker CLI to communicate with the Docker daemon) changed in Docker 0.5.2, and now uses a UNIX socket instead of a TCP socket bound on 127.0.0.1 (the latter being prone to cross-site request forgery attacks if you happen to run Docker directly on your local machine, outside of a VM). You can then use traditional UNIX permission checks to limit access to the control socket.

You can also expose the REST API over HTTP if you explicitly decide to do so. However, if you do that, be aware of the above mentioned security implications. Ensure that it is reachable only from a trusted network or VPN or protected with a mechanism such as `stunnel` and client SSL certificates. You can also secure API endpoints with HTTPS and certificates (https://docs.docker.com/engine/security/https/).

The daemon is also potentially vulnerable to other inputs, such as image loading from either disk with `docker load`, or from the network with `docker pull`. As of Docker 1.3.2, images are now extracted in a chrooted subprocess on Linux/Unix platforms, being the first-step in a wider effort toward privilege separation. As of Docker 1.10.0, all images are stored and accessed by the cryptographic checksums of their contents, limiting the possibility of an attacker causing a collision with an existing image.

Finally, if you run Docker on a server, it is recommended to run exclusively Docker on the server, and move all other services within containers controlled by Docker. Of course, it is fine to keep your favorite admin tools (probably at least an SSH server), as well as existing monitoring/supervision processes, such as NRPE and collectd.

Linux kernel capabilities

By default, Docker starts containers with a restricted set of capabilities. What does that mean?

Capabilities turn the binary "root/non-root" dichotomy into a fine-grained access control system. Processes (like web servers) that just need to bind on a port below 1024 do not need to run as root: they can just be granted the `net_bind_service` capability instead. And there are many other capabilities, for almost all the specific areas where root privileges are usually needed.

This means a lot for container security; let's see why!

Typical servers run several processes as `root`, including the SSH daemon, `cron` daemon, logging daemons, kernel modules, network configuration tools, and more. A container is different, because almost all of those tasks are handled by the infrastructure around the container:

- SSH access are typically managed by a single server running on the Docker host;
- `cron`, when necessary, should run as a user process, dedicated and tailored for the app that needs its scheduling service, rather than as a platform-wide facility;
- log management is also typically handed to Docker, or to third-party services like Loggly or Splunk;
- hardware management is irrelevant, meaning that you never need to run `udevd` or equivalent daemons within containers;
- network management happens outside of the containers, enforcing separation of concerns as much as possible, meaning that a container should never need to perform `ifconfig`, `route`, or ip commands (except when a container is specifically engineered to behave like a router or firewall, of course).

This means that in most cases, containers do not need "real" root privileges *at all*. And therefore, containers can run with a reduced capability set; meaning that "root" within a container has much less privileges than the real "root". For instance, it is possible to:

- deny all "mount" operations;
- deny access to raw sockets (to prevent packet spoofing);
- deny access to some filesystem operations, like creating new device nodes, changing the owner of files, or altering attributes (including the immutable flag);
- deny module loading;
- and many others.

This means that even if an intruder manages to escalate to root within a container, it is much harder to do serious damage, or to escalate to the host.

This doesn't affect regular web apps, but reduces the vectors of attack by malicious users considerably. By default Docker drops all capabilities except those needed (https://github.com/moby/moby/blob/master/oci/defaults.go#L14-L30), a whitelist instead of a blacklist approach. You can see a full list of available capabilities in Linux manpages (http://man7.org/linux/man-pages/man7/capabilities.7.html).

One primary risk with running Docker containers is that the default set of capabilities and mounts given to a container may provide incomplete isolation, either independently, or when used in combination with kernel vulnerabilities.

Docker supports the addition and removal of capabilities, allowing use of a non-default profile. This may make Docker more secure through capability removal, or less secure through the addition of capabilities. The best practice for users would be to remove all capabilities except those explicitly required for their processes.

Docker Content Trust Signature Verification

The Docker Engine can be configured to only run signed images. The Docker Content Trust signature verification feature is built directly into the `dockerd` binary.
This is configured in the Dockerd configuration file.

To enable this feature, trustpinning can be configured in `daemon.json`, whereby only repositories signed with a user-specified root key can be pulled and run.

This feature provides more insight to administrators than previously available with the CLI for enforcing and performing image signature verification.

For more information on configuring Docker Content Trust Signature Verificiation, go to Content trust in Docker (https://docs.docker.com/engine/security/trust/content_trust).

Other kernel security features

Capabilities are just one of the many security features provided by modern Linux kernels. It is also possible to leverage existing, well-known systems like TOMOYO, AppArmor, SELinux, GRSEC, etc. with Docker.

While Docker currently only enables capabilities, it doesn't interfere with the other systems. This means that there are many different ways to harden a Docker host. Here are a few examples.

- You can run a kernel with GRSEC and PAX. This adds many safety checks, both at compile-time and run-time; it also defeats many exploits, thanks to techniques like address randomization. It doesn't require Docker-specific configuration, since those security features apply system-wide, independent of containers.
- If your distribution comes with security model templates for Docker containers, you can use them out of the box. For instance, we ship a template that works with AppArmor and Red Hat

comes with SELinux policies for Docker. These templates provide an extra safety net (even though it overlaps greatly with capabilities).
- You can define your own policies using your favorite access control mechanism.

Just as you can use third-party tools to augment Docker containers, including special network topologies or shared filesystems, tools exist to harden Docker containers without the need to modify Docker itself.

As of Docker 1.10 User Namespaces are supported directly by the docker daemon. This feature allows for the root user in a container to be mapped to a non uid-0 user outside the container, which can help to mitigate the risks of container breakout. This facility is available but not enabled by default.

Refer to the daemon command (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-user-namespace-options) in the command line reference for more information on this feature. Additional information on the implementation of User Namespaces in Docker can be found in this blog post (https://integratedcode.us/2015/10/13/user-namespaces-have-arrived-in-docker/).

Conclusions

Docker containers are, by default, quite secure; especially if you run your processes as non-privileged users inside the container.

You can add an extra layer of safety by enabling AppArmor, SELinux, GRSEC, or another appropriate hardening system.

If you think of ways to make docker more secure, we welcome feature requests, pull requests, or comments on the Docker community forums.

Related information

- Use trusted images (https://docs.docker.com/engine/security/trust/)
- Seccomp security profiles for Docker (https://docs.docker.com/engine/security/seccomp/)
- AppArmor security profiles for Docker (https://docs.docker.com/engine/security/apparmor/)
- On the Security of Containers (2014) (https://medium.com/@ewindisch/on-the-security-of-containers-2c60ffe25a9e)
- Docker swarm mode overlay network security model (https://docs.docker.com/engine/userguide/networking/overlay-security-model/)

Docker (https://docs.docker.com/glossary/?term=Docker), Docker documentation (https://docs.docker.com/glossary/?term=Docker%20documentation), security (https://docs.docker.com/glossary/?term=security)