



# Contents

<b>Contents</b>	<b>2</b>
<b>1 Docker Security</b>	<b>3</b>
1.1 Introduction . . . . .	3
1.2 Mutual TLS and Security by Default . . . .	5
1.3 Content Trust . . . . .	8
1.4 Overlay Network . . . . .	21
1.5 Docker Bench Security . . . . .	21
1.6 Process Restriction and Capabilities . . . .	24
1.7 Open Source . . . . .	27
1.8 Linux Kernel Security . . . . .	27
1.9 Cilium . . . . .	32
1.10 About your images . . . . .	40
1.11 Docker Security Scanner . . . . .	45
1.12 Secret Manager . . . . .	46
1.13 Immutability . . . . .	49
<b>2 Biography</b>	<b>51</b>

# Chapter 1

## Docker Security

**Reviewers:** Gareth Pelly, Riyaz Faizullahoy

### 1.1 Introduction

When you have to manage a production environment security is one of the first points that you have to address. In this chapter I will share best practices and tips around this topic.

Shipping your application out of your laptop creates more problems and it doesn't matter if it's inside or out of a container:

- You need to be sure that what are you sending out of your laptop is compliant and doesn't contain compromised content.

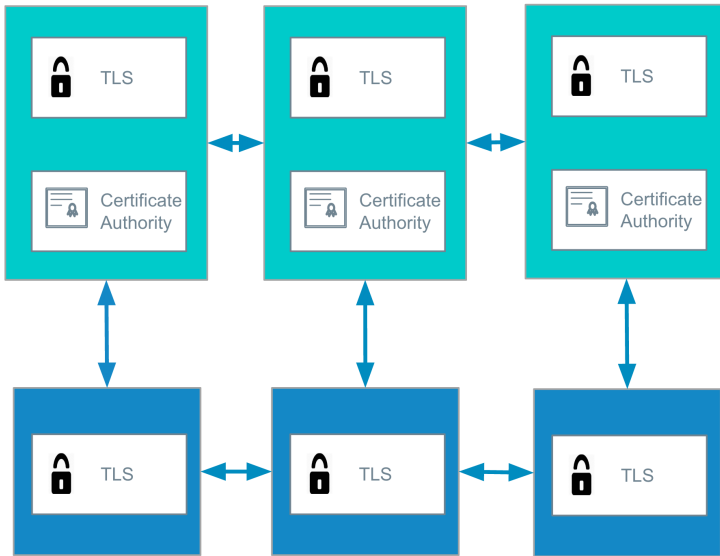
- You need to check if what you are downloading, for example the binary of your application is what you need and not something different.

But from a security point of view a container is a new layer of isolation between different applications, not only between different instances of the same application but also between the app and the operation system. Docker offers more flexibility: you can update your application quickly, scale containers on different hosts and apply security scan to discover vulnerabilities. Containers add a new layer of abstraction which adds some complexity but also a set of utilities to make your production environment solid and safe.

Usually, in order to have a safe environment, you have to delete what is not really important for you, everything that is unused could be a cause of problems.

There are also choices that you can make to secure your environment like using a firewall, encryption and so on. We will cover all these elements throughout this chapter.

## 1.2 Mutual TLS and Security by Default



In SwarmKit, Docker implemented a “Security by default” layer that is enabled by default in every cluster. It is embedded in Swarm, and unlike security flags implemented in other tools, it does not have a flag to disable them in a development environment. In practice, this security layer implements the Mutual TLS (mTLS) concept to manage server-to-server authentication, the Swarm managers are also have a Certification Authority

(CA) built-in. It signs and knows the identity of every worker and it's replicated across multiple masters to be highly available. Every worker has a proper identity in order to allow the master to understand if the worker is part of the cluster and if it's operating as expected and also to avoid possible fake and malicious workers.

All these certificates and signatures are used during the gossip protocol because all traffic into the cluster is encrypted. There are a lot of stories from different companies about attacks and exploits and usually they need 6 months to detect them. For this very reason is extremely important to rotate these certificates in order to reduce the range or exposure. By default, Swarm rotates keys every 3 months and it works with a whitelist of validate certificates which is unusual as other applications generally work with a blacklist of invalid certificates which can become really long and hard to manage. Thankfully, this process is automatic and totally transparent for your cluster.

It also supports the use of an external CA, if you have already one that you trust then you can continue to use it.

Follow the link in the note to see a great presentation that was given during DockerCon 2016, it is great to have an idea about how this flow works in practice.

When you start a swarm cluster you also receive two tokens, one to identify and add new managers and the other to add the workers. You will need to provide these

tokens when you join a node. You can rotate these tokens too, this means that the old one will become invalid and you need to use the new one. You can decide to rotate just one of them and the reasons for this are usually:

- someone copy them on slack, email or other 3rd party service
- you put your key in git

These tokens have a specific format:

```
SWMTKN-1-3mqolc2i75ygkj51df3339mkhdtel6ynjexqfvb6vhc2viywx-3yo
```

SWMTKN is standard and you can use it to scan your environments and understand if someone committed that token in git for example or it was sent in an email. The second (3mqolc2i75ygkj51df3339mkhdtel6ynjexqfvb6vhc2viywx) part is the encrypted hash of the CA root certificate, it's the section used during the bootstrap to identity your cluster. The last part (3yonjuwq8wclj1r7g3ke3ha0p) is a generated secret.

There is no automatic policy to rotate your key but you can always use the proper command in order to invalidate the old ones and generate new tokens:

```
docker swarm join-token --rotate worker
docker swarm join-token --rotate master
```

These tokens are the keys to identity the role of the new node in the cluster but also if the node can join it.

If the token is wrong or old then the node can not be added but it also means that someone is trying to add an unsafe node.

By design the communication between node are mono directional. The masters can take the decision and communicate what the workers need to do, this decision guarantees that a compromised worker can not take decision itself or manipulate the cluster.

## 1.3 Content Trust

Securing software updates is a challenging problem - whether we're updating our own application or downloading a new package from the Internet. In both cases, dependencies represent a risk for our current environment. New packages come from a source that needs to be trusted in some way. We need to understand what we are downloading, does it contain the right content? Is the version that we required the same one that are we getting?

Dependencies represent a huge part of our application but also of our whole stack, for example how many system packages did you download or update on your server? This is an important question for many different sources: package managers and registries like Apt, Yum but also rubygem, composer and many others from application point of view. With Docker you have the same problem, how can you trust the source of the image and the content



from Docker Hub? Docker comes with a solution called Docker Content Trust but I've renamed it "The world behind a pull".

You might think this problem is already solved by signing images with GPG and sending them over TLS, but this approach falls flat because it does not provide context around the signature.

For example: The identity of the registry: where does my image come from? Or even better, is it coming from the place that I hope?

Between the packaging and the reception, what happened? You need to verify the integrity of your content. Are the signatures for this image too out of date? Are they still valid? What was the intent of the publisher when pushing this image? And should they have been able to push this image in the first place? Docker works on all these mechanisms with an implementation of the "The Update Framework" (TUF). It's an open source framework and specification designed to make the update lifecycle safe. The studies behind this framework are backed by peer-reviewed academic papers published in top-tier conferences because they identified common and security vulnerabilities across package managers and designed TUF to be robust against the attack vectors they discovered. The framework is inspired by Thandy the Tor's secure updating system, the authors of the framework abstracted Thandy's concepts to make them reusable in different environment.

Notary is an opinionated implementation of the TUF spec by Docker Inc. It's also integrated in Docker itself and in the Hub. TUF not only signs the content of the package but it also signs contextual information about your tarball, files, or images since it can be used to secure any kind of content. Another powerful decision around TUF is the F. It is a framework. It means that it's not a platform that you need to take as you find but it exposes a set of primitives, utilities and concepts that you can connect to support your flow.

It is based on a couple of principles:

- Responsibility separation to decrease the scope of a specific role and by consequently increase the security of your system. Someone can sign all your releases and another one can sign and declare the latest version of the software for example.
- Survivable key compromise and scoped keys. The idea is to make key revocation easy and also define roles with set privileges for each key. In some cases, you can also manage different sources for your keys based on security requirements. For example, you can restrict delegation keys to only have privilege to sign for specific package paths.
- Multi-signature thresholding to only allow signatures to be valid for a package if a quorum of designated collaborators have signed explicit and im-

licit revocation. Explicit should be easy within the framework, if your keys are compromised. Implicit is when you key expires with an expiration date.

A TUF repository contains all metadata to trust your resources and they are totally separated from the package itself which could be in your HDD, in a CDN, S3 or anywhere. The various metadata files are signed from different keys with different rules. All metadata has an expiration date and it is different depending on the type.

There are five metadata roles:

- Root
- Targets
- Snapshot
- Timestamp
- Delegation

All these roles have one or more keys and requires a threshold of signatures to trust a metadata. Threshold is not required, by default in Docker/Notary is 1 but it's good to know if you need to build a complex solution that requires more than one signatory.

Root

Root is the root of trust, it defines all keys to trust also itself. The client uses the root key to store information about the registries that it already trusted. The root key needs to be stored in a safe place offline. It means that it's not easy to manage and for this reason, it has the longest expiration date. The root signs all keys but it has no information about the final resources itself.

#### Targets

Information about files and artifacts is signed by the Targets. It contains the mapping between the human name of the resource and the hash used to address internal communication between client and repository. The Targets key doesn't sign the file itself but it signs related metadata such as content hash, size and so on. The Targets key defines which individual packages are trusted.

#### Delegation

The Targets key also specifies collaborators to define who can sign what and in the meantime these collaborators are delegators and they sign the Targets like a Delegation.

In practice for example you can declare that a new release, to be trusted content needs to be signed from two collaborators. These collaborators can be for example the maintainer of the project.

#### Snapshot

The Snapshot gives you a valid picture of the TUF repository, it lists the hash and the size of all metadata. It allows the client to verify if the metadata presented in

the TUF repository at a particular time is not presented to clients by an attacker, probably with different content.

The Snapshot role is very important because one of the possible attacks is based on old releases, or mixing and matching previous releases

#### Timestamp

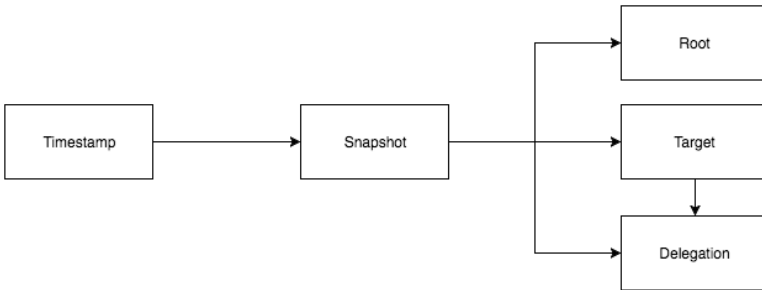
Timestamp gives information about the latest version of the resource. It's the perfect way to understand if you are downloading a fresh version of the release. It contains the hash of the Snapshot and has a very short expiration date in order to oblige clients to ask for fresh metadata frequently.

Timestamp and Snapshot together contrast the most common replay attacks such as roll-back freeze. Roll-back attack is when an attacker forces the user system to serve an old version of the package, probably with a public security vulnerability that it can use later to broke into the system (ex: openssl with Heartbleed).

Freeze attack is when the attacker manipulates the repository in order to always serve a specific previous version after the team has released a new valid one.

Now we know why GPG is not enough, all the roles and keys of TUF reassure clients about what they are downloading is authentic and fresh. One of the reasons that you have to release a new version of your library is because it has an horrible security vulnerability that other people already discovered. In the example of using GPG signatures, since that old version of your library

was valid before the vulnerability was discovered, it was probably signed as a valid piece of software – however, since GPG does not provide any expiration or revocation, that signature is still valid now even though the software is vulnerable. With only the Timestamp role you can be sure that you can trust what you are downloading only if you get the latest version but if for some reason, the most common is because it's not 100% compatible and you have not time to update, you are download an old version only the Snapshot role can tell you that that version is the right version.



This graph represent how they are connected and the information that they contain. Docker implements all this flow in Notary, a standalone library that you can use to create your update lifecycle. It's also fully integrated into the Docker Hub and into the CLI. This feature in Docker CLI can be enabled by setting this environment variable

```
export DOCKER_CONTENT_TRUST=1
```

Docker Content Trust leverages Notary by signing mappings from image tags (which are human readable) to digests (SHA256 hashes). It does so by translating the following commands that operate over tags into commands that use hashes, after it verifies the information from a specified Notary server. If the TUF validation fails from the Notary server, the operation is rejected outright, before any image data is pulled or run.

Docker Content Trust commands:

- push
- build
- create
- pull
- run

You can always use the option `-disable-content-trust` to disable this check runtime.

A repository can contain both signed and unsigned tags but if you force your client to work with `DOCKER_CONTENT_TRUST=1` the client will see only signed images, the others will effectively behave as if they do not exist since the Notary TUF metadata will not validate.

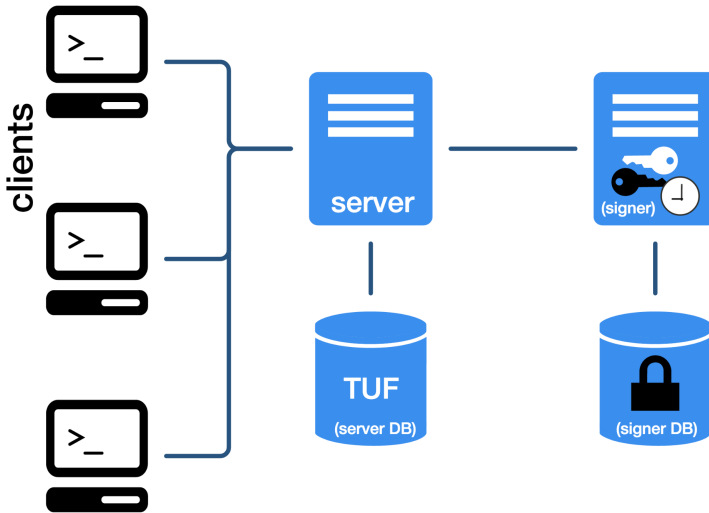
Notary provides a CLI and a server component that you can use to sign and verify your data as a simple, tarball, application artifact. It's a good example to help understand that it works not just for containers but for other kinds of data. We can add this to our pipeline to verify that every deployment in each of our environments contain what we expect, it also allows verification if make the data publicly available that the content is still correct.

The first step is to clone the repository and download Notary CLI. Docker provides some binaries that you can download directly from the GitHub repository (in this example I am working on Linux):

```
go get github.com/docker/notary
cd \${GOPATH}/src/github.com/docker/notary
make binaries
./bin/notary version
notary
Version: 0.4.2
Git commit: c8aa8c
```

The last command is only to check that everything is working as expected.





Notary is made of different parts, notary-server, notary-signer, database and a command line tool to interact with the server, we clone the repository because it contains a few utilities to start a server with docker-compose:

```
docker-compose build
docker-compose up -d
mkdir -p ~/.notary && cp cmd/notary/config.json
    cmd/notary/root-ca.crt ~/.notary
```

And we also need to add a new entry in our `/etc/hosts`

```
127.0.0.1 notary-server
```

Or the ip of the docker-machine if you are using docker-machine.

Now that we have everything up and running we can init a notary TUF repository

```
bin/notary init scaledocker-demo
```

It will create for us few keys one for each role and we need to remember all the passphrases. We can store them in some a safety password manager as KeepassX.

At this point we can publish our scaledocker-demo repository.

```
./bin/notary publish scaledocker-demo
./bin/notary list scaledocker-demo
No targets present in this repository.
```

Because our repository is empty, we can create a very simple file:

```
echo "echo hello! Enjoy your book dockers!" >
  ./hello.sh
./hello.sh
```

We can sign it with notary

```
./bin/notary add scaledocker-book hello-script
  ./hello.sh
./bin/notary publish scaledocker-book
```

The last command asks us to type targets and snapshots passphrase because we need to sign a new hash as

a target for `hello.sh`, and a new version of the `targets` file that now includes `hello.sh` into our snapshot. The `add` command supports `-p` option to publish the content directly without the explicit `push` command.

```
./bin/notary verify -i ./hello.sh
  scaledocker-demo hello-script
```

Imagine that `./hello.sh` is a file downloaded from a repository or from the web, `notary` checks it and in this case comes back with a success because the hash of the file matches the hash signed into `notary` - which has a valid signature from our trusted `targets` key and no metadata is out of date

```
./bin/notary verify -i /evil-script.sh
  scaledocker-demo hello-script
```

\* fatal: data not present in the trusted collection, sha256 checksum for `hello-script` did not match: expected `06c8971ac4183b56f3e75f84702541dacd51dc6fd6c3b298e8578a27c740135`

If you try to check another file such as `/etc/hosts` `notary` blocks you because the file is different. You can try to update the `hello.sh` file in your site and with `curl`, download and verify it. I can not setup this test for you because it's not nice ask you to download a malicious file from a random server. As you can see its very easy to use!

```
./bin/notary key list
```

You can check the current target key id, remember it and now we can rotate:

```
./notary key rotate scaledocker-demo targets
```

Where targets is the type of key that you need to update and scaledocker-demo is your repository. Now you can list another time your keys and here we are, the key id for the targets role is different. We have a new key. Try to verify hello.sh again to check that all is working like expected. Rotation key is easy and transparent for your files, all the packages signed with the old keys are still valid.

These are the default expiration dates in Notary. Your content shouldn't expire for the next three years. But you need to have a mechanism to re-sign your content after that time because if the Targets key expires the content is not valid for the client, it means that the download will fail. Same for the Root key, after its expiration all the TUF Repository will be invalid for the client. In this scenario, it's important to note that Notary and TUF make the subtle differentiation between old-but-good and out-of-date software because old versions that are still good must be explicitly re-signed.

Root - 10 years Targets and Delegations - 3 years  
Snapshot - 3 years Timestamp - 14 days

I know, you cannot wait to use it in a real environment!

## 1.4 Overlay Network

The overlay network by default is not encrypted, by default swarm encrypts all communication between nodes but the communication between containers is not. When you create a network you can enable encryption:

```
docker network --opt encrypted --driver overlay
  tick-net
```

This option is not enabled by default primarily because swarm doesn't know how your application works and in some cases it could create a problem. Before allowing it by default we need to know the real degradation created by the introduction this new layer.

But remember that having this security layer which is easy to enable is great.

## 1.5 Docker Bench Security

Frequently, best practices help you to have a safe environment, docker-bench-security is an open source project that runs in a container and scans your environment to report a set of common mistakes like:

- Your docker is not up to date
- Your kernel is too old

- Some Docker daemon configurations are not good enough to run a production environment
- Your container runs 2 processes

It's a great idea to run it at some stage in each host to have an idea about the status of your environment. To do that you can just use this command when running a container.

```
docker run -it --net host --pid host --cap-add
  audit_control \
  -v /var/lib:/var/lib \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v /usr/lib/systemd:/usr/lib/systemd \
  -v /etc:/etc --label docker_bench_security \
  docker/docker-bench-security
```

You can try it in your local environment. Run the command and check what you can do to make your local environment safe.

This tool is open source on GitHub, it's a great example of collaboration and how a community can share experiences and help other members to improve an environment.

This is a partial output:

```
Initializing Thu Nov 24 21:35:24 GMT 2016
```

```
[INFO] 1 - Host Configuration
[WARN] 1.1 - Create a separate partition for
          containers
[PASS] 1.2 - Use an updated Linux Kernel
[PASS] 1.4 - Remove all non-essential services
          from the host - Network
[PASS] 1.5 - Keep Docker up to date
[INFO]      * Using 1.13.01 which is current as
          of 2016-10-26
[INFO]      * Check with your operating system
          vendor for support and security maintenance
          for docker
[INFO] 1.6 - Only allow trusted users to control
          Docker daemon
[INFO]      * docker:x:999:gianarb
[WARN] 1.7 - Failed to inspect: auditctl command
          not found.
[WARN] 1.8 - Failed to inspect: auditctl command
          not found.
[WARN] 1.9 - Failed to inspect: auditctl command
          not found.
[INFO] 1.10 - Audit Docker files and directories
          - docker.service
[INFO]      * File not found
[INFO] 1.11 - Audit Docker files and directories
          - docker.socket
[INFO]      * File not found
```

## 1.6 Process Restriction and Capabilities

We already know that behind the word containers are chroot, namespace and other bunch of kernel features. They offer a great granularity for system capabilities, it means that we can restrict our application for a subset of resources like memory and cpu but also for networking and filesystem. If our application is under attack we can protect our host and other containers by optimizing resources.

If we know our application requires 500M of RAM, we can set a limit on our container:

```
docker run -m=500M gianarb/micro
```

You can also limit other resources via some other options: `-kernel-memory`, `-memory-swap`, `-cpu-period`, `-cpu-quota`, `-cpu-shares`, `-device-read-iops`, `-device-read-bps`, `-device-write-iops`, `-device-write-bps`.

You can monitor your container with the command:

```
docker stats <container-id>
```

And discover how you configured it with the inspect command:

```
docker inspect -f '{{ .HostConfig }}'
```

If we are under attack this doesn't resolve all problems because an out of memory container will be killed but we



can mitigate bad behavior and save our cluster. Usually a root user has access to all Linux capabilities like creating files, manipulating processes, mount, reboot while a non-root user has access to less privileges but can ask to become root. If your application can not access a particular capability that it needs then it doesn't work, for this reason, by default docker balances the access between security and simplicity. Right now a default container has access to:

- chown
- dac\_override
- fowner
- kill
- setgid
- setuid
- setpcap
- net\_bind\_service
- net\_raw
- sys\_chroot
- mknod

- setfcap
- audit\_write

It's pretty much impossible to find a standard configuration that everyone can use and for this reason docker allows you to add or drop capabilities into the run command:

```
docker run --cap-drop setuid --cap-drop setgid
-ti debian /bin/sh
docker run --cap-add all --cap-drop setgid -ti
debian /bin/sh
```

Leave out any access and OS tools that your application doesn't need, it's a useless risk. For this reason, having a studied capability configuration is a good security layer for your host.

NET\_RAW allows you to use a raw socket and packet socket, in practice and at a high level it allows you to receive a package from the web. However, if we drop this package and you run an alpine container that drops that capability, then try and ping google in the container:

```
docker run -it --cap-drop NET_RAW alpine
/bin/bash
/ # ping google.com
ping: permission denied (are you root?)
```

Because our container does not have that capability.

## 1.7 Open Source

It may seem crazy or obvious but docker is an open but Docker is an open source project, one of the biggest open source projects This means that there is a community around the project that finds and fixes bugs and security issues 24 hours a day. The code is fully open and anyone can contribute and make it safe. There are other philosophies about this topic like how can public code could be secure? But you know, this subject covers far too much to be discussed here.

## 1.8 Linux Kernel Security

It is not just chroot and namespace but also there are other linux kernel security tools like SELinux, Apparmor, Seccomp that work with Docker to enable you to configure profiles and to allow your container to have visibility only for what it really needs in order to isolate it from the host and from other containers.

### SELinux

SELinux is a security tool created in 2003, it is based on the label concept. Everything inside a system has labels: files, network, hosts, mount, directory. You can write a role based on these labels to allow particular action to a specific resource. The owner of a file does not have a particular permission on what he created, every-

thing is managed by labels and by default everything is deny. The iteration between this resource and what a process can do is called a policy.

Let me explain with an example: in a football team there are two kinds of players, a goalkeeper and the normal players.

A player can have one of two labels, normal and goalkeeper. Ball and hand are also labels and we need to describe a policy because the goalkeeper has permission to touch the ball with their hands:

```
allow + goalkeeper + ball:hand + touch
```

What you can do is to describe the policy and attach them in the container, however you need to allow your host to work with SELinux. It's complex and hard to maintain but it's also very powerful and flexible, you have a high granularity and many possible configurations.

### AppArmor

AppArmor is a Mandatory Access Control for Linux. It's included in the Linux Kernel from version 2.6.35. It's quite young and it's built as an SELinux alternative but judged too complex.

You can write a policy to describe what your application can or cannot do and in the case of Docker Container you can attach that policy in a running container.

To run this example, we need to have AppArmor installed, we can not do that in our Mac or with boot2docker,

for this reason I use a droplet on digitalocean and docker-machine.

First of all, we need to create our server:

```
docker-machine create -d digitalocean
  --digitalocean-access-token $DO app-do
docker-machine ssh app-do
```

Where \$DO is my access key and app-do is the name of the problem. Now that we are inside we can have a look of `/etc/apparmor.d/docker` with the command:

```
cat /etc/apparmor.d/docker
```

This file contains the default configuration loaded by docker in every container, but it's time to add our custom policy:

Copy this:

```
#include <tunables/global>

profile sample-one
  flags=(attach\_disconnected,mediate\_deleted)
  {
    #include <abstractions/base>
    network,
    capability,
    file,
    umount,

    deny /etc/** w,
```

```
}

```

In `/etc/apparmor.d/sample-one` and reload the apparmor service

```
service apparmor restart
```

And start a new container with the profile `sample-one` previously declared:

```
docker run --security-opt="apparmor:sample-one" --rm -it alpine /bin/sh
```

Before trying to do some tests in our container, let me explain to you what that policy means. We created a new profile called `sample-one` that allows our container to work with network, capability, file and unmount but we also made `/etc/**` read-only with the role:

```
deny /etc/** w
```

Now what we can do is to write two files in our container

```
touch /tmp/hello.txt
touch /etc/bad.txt
```

The first command runs successfully, the second one returns:

```
touch: /etc/ciao: Permission denied
```

We can create a more complex policy to make our container and our host secure. I like AppArmor more than other similar tools because it's not too complex and

offers good flexibility for normal use cases. It's always easy to have a long and hard file to read, I'm happy to quote Jess Frazelle:

AppArmor profile pull requests is the bane of my existence

---

*Jess Frazelle*

She created a tool called `bane` to create and maintain AppArmor policy from a YAML specification file. You can think of `seccomp` like a `syscall` firewall. `Syscall` is the mechanism that a process uses to communicate with the kernel of the operating system like: access to the hard drive, start a process, communicate with the scheduler and, in practice, everything the kernel can do. Docker has a default profile for every container that disables about 40 system calls but you can override them with a specific profile for your container. The real problem with a `seccomp` profile is that it is very hard to write and maintain, for this reason it's very hard to work with it.

Every one of these tools has the same entrypoint in Docker, you can specify a custom role for your container during the `run` command with `-security-opt` options. It is the common entry point, here is an example:

```
docker run --rm -it \
```

```
--security-opt
    seccomp=/path/seccomp/profile.json \
    gianarb/micro:1.0.0
```

## 1.9 Cilium

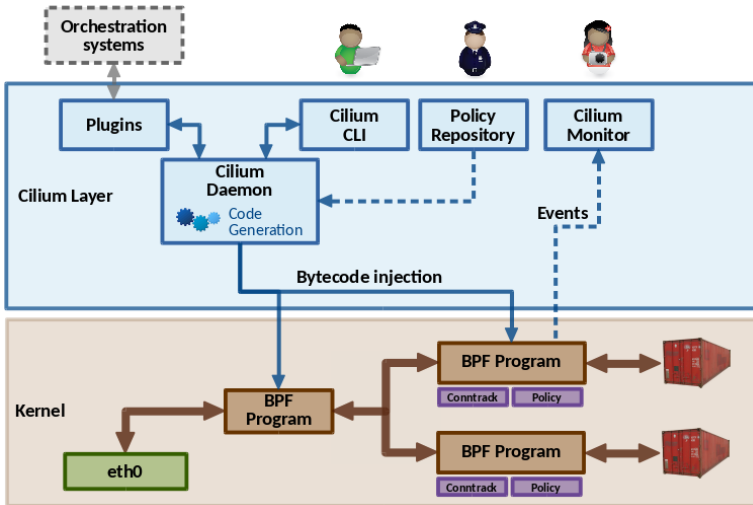
Each architecture has iptables that manage connections with the outside but also that block or allow communication between servers.

In AWS, we have security group that manages this for us and it's always good practice to have a strong configuration that disables all kinds of traffic and only allows what our application requires. We can do the same with our containers.

iptables is not designed to support the new dynamic environment that contains produce and also it's not very scalable. iptables with thousands of rules will start to be slow.

Cilium is an open source project supported by Cisco to manage networking and security policies between containers. It requires Linux Kernel  $\geq$  4.8 because it uses eBPF to provide a fast and in-kernel implementation. This means that it works also with Kubernetes and other container platforms.





This image is taken from the original documentation and explains the architecture on top of Cilium. BPF is a bytecode interpreter introduced to filter network packets and Cilium provides a CLI to define network policies.

It also exposes a monitor and a set of plugins, one of them allows you to manage integration with Docker.

Cilium daemon receives policies described by you that contain information about which containers can speak with another, it compiles a BPF and injects it into the system. Docker plugins use labels to translate these policies, this means that you need to attach a specific `-label` when you run your containers. We can continue with an example.

---

```
git clone git@github.com:cilium/cilium.git
  ~/cilium-test
cd ~/cilium-test
NUM\_NODES=1 ./contrib/vagrant/start.sh
```

Cilium has a Vagrantfile that helps you to start a demo, we are using that solution. It means that you need to have virtualbox and vagrant up and running in your machine. In this case we started a 2 node swarm cluster.

Go into the master and install docker-compose

```
vagrant ssh
```

We need to create a cilium network to allow docker to use the plugin:

```
docker network create --ipv6 --subnet ::1/112
  --ipam-driver cilium --driver cilium cilium
cd ~/
```

If you run `docker ps` you can see that there are 2 containers un and running. This box is built to run few example. Cilium uses a key value storage, in our case Consul to map the network and the container available. You can also get some information about the cilium daemon and also check that it's up and running.

```
cilium daemon status
KVStore:      OK - 172.17.0.3:8300
Docker:       OK
```

```

Kubernetes: Disabled
Cilium:      OK
V4 addresses reserved:
 10.1.0.1
 10.1.28.238
 10.1.116.202
 10.1.138.214
V6 addresses reserved:
 f00d::c0a8:210b:0:f236
 f00d::c0a8:210b:0:f7e8

```

At this point all is good and we can start our test. We are creating a client and a server. In our case what we are going to set up is a one way communication from the client to the server and not vice versa.

```

docker run -d --name server --net cilium --label
  io.cilium.service.server alpine sleep 30000
docker run -d --name client --net cilium --label
  io.cilium.service.client alpine sleep 30000

```

Now let's use cilium to understand if it registered our two containers:

```

cilium endpoint list
ENDPOINT ID LABEL ID LABELS
  (source:key[=value])   IPv6
  IPv4                   STATUS
29898                258
  cilium:io.cilium.service.client
  f00d::c0a8:210b:0:74ca 10.11.247.232 OK

```

```
35542          257
  cilium:io.cilium.service.server
  f00d::c0a8:210b:0:8ad6 10.11.28.238 OK
```

cilium endpoint list shows the two endpoints based on the labels of our containers.

```
docker exec -it client ping server
```

The normal behavior of the ping is that it will work. The two containers are in the same network and usually they are able to ping each other. But as you can see it's not what it's happening right now. At this point what you are going to do is start to think about a Docker's bug and start to debug with tcpdump and so on. Cilium has a command called cilium monitor. It's a very good utility that filter package that come from and to a container and also figure out what is happening in our particular scenario.

```
sudo cilium monitor
CPU 01: MARK 0x1eb5e162 FROM 63464 DEBUG: CT
  verdict: New
CPU 01: MARK 0x1eb5e162 FROM 63464 DEBUG: Policy
  denied from 258 to 257
CPU 01: MARK 0x1eb5e162 FROM 63464 Packet dropped
  133 (Policy denied) 98 bytes ifindex=15
  258->257 to lxc 63464
00000000 92 d4 f6 b5 85 3f d2 e1 17 af 39 af 08
  00 45 00 |.....?....9...E.|
```

```

00000010 00 54 5a 4b 40 00 40 01 cc bb 0a 01 74
      ca 0a 01 |.TZK@.@.....t...|
00000020 8a d6 08 00 40 16 05 00 00 05 2b d4 87
      10 00 00 |....@.....+.....|
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 |.....|

```

The monitor is telling you “Policy denied from 258 to 257”. These numbers are the endpoints id and the monitor is telling us that the client is unauthorized to connect with the server.

```

cilium policy allowed -s
  cilium:io.cilium.service.client -d
  cilium:io.cilium.service.server
Resolving policy for context &{Trace:1
  Logging:0xc42177b590
  From:[cilium:io.cilium.service.client]
  To:[cilium:io.cilium.service.server]}
Root rules decision: undecided
No matching children in io.cilium
Root children decision: undecided
Final tree decision: deny

```

With this command we are just asking the cilium daemon to tell us about the current relation between two endpoints a source (-s) and a destination (-d). In our case the final decision is deny. It means that our containers are not able to speak. The problem is that we didn't load a policy.

A policy is a hierarchical tree that explain connections between endpoints, let's add one to allow client to speak with sever. You can copy that policy in ./cs.policy and load it with the command cilium policy import ./cs.policy

```
{
  "name": "io.cilium",
  "children": {
    "service": {
      "name": "service",
      "children": {
        "client": {
          "name": "client"
        },
        "server": {
          "name": "server",
          "rules": [{
            "allow": [{
              "action": "accept",
              "label": {
                "key": "host",
                "source": "reserved"
              }
            }
          ]
        },
        {
          "action": "accept",
          "label": {
            "key":
              "../client",
            "source":

```

```
    "cilium"  
  }  
}]  
}]  
}  
}  
}  
}
```

Now that we loaded that policy we are able to ping the server from the client but not vice versa.

Cilium is a very powerful tool and well design. CISCO is doing a very good work to close the gap between the traditional application firewall and the high scalable and “containerized” ecosystem.

It’s also well optimised to reduce the amount of code generated to implement your policies. This is important when you are working on this level, into an interface you can manage a big traffic and every unnecessary line of code makes the difference.

At the moment the project has not a stable release and also the kernel’s features required are not easy to accomplish but it’s still a project that you need to know and you need to follow. The network feature provided by Docker has not a powerful granularity and define how your network works make the difference between a safe and unsafe environment.

## 1.10 About your images

Everything unnecessary in your system could be a very stupid vulnerability. We already spoke about this idea in the capability chapter and the same rule exists when we build an image. Having tiny images with only what our application needs to run is not just a goal in terms of distribution but also in terms of cost of maintenance and security.

If you have some small experience with docker already you probably know the alpine image. It is build from the Alpine distribution and it's only 5MB size, if your application can run inside it then this is a very good optimization that you can do.

What about your binaries? Can your application run standalone? If the answer is yes you can think about a very very minimal image. scratch is usually used as a base for other images like debian and ubuntu but you can also use it to run your golang binary and let me show you something with our micro application.


In the release page, there are a list of binaries already compiled and ready to be used. In this case we can download the linux\_386 binary.



**Latest release**

1.0.0  
7fc5af  
Verified

## 1.0.0

 gianarb released this on Jul 17 · 4 commits to master since this release

```
docker run gianarb/micro:1.0.0
```

I create this application to have a good example around few topic:






- HealthCheck
- HTTP Microservices
- Docker
- 12factor application

GET /

```
<ip>
```

GET /health  
Response: it return 200 like status code.

### Downloads

 <a href="#">micro_1.0.0_darwin_386</a>	4.82 MB
 <a href="#">micro_1.0.0_linux_386</a>	4.78 MB
 <a href="#">micro_1.0.0_linux_arm</a>	4.86 MB
 <a href="#">Source code (zip)</a>	
 <a href="#">Source code (tar.gz)</a>	

```
curl -SsL https://goo.gl/o6tle5 > micro
```

And we know we can include this binary in the scratch image with this Dockerfile

```
FROM scratch

ADD ./micro /micro
EXPOSE 8000

CMD ["/micro"]
```

```
docker build -t micro-scratch .  
docker run -p 8000:8000 micro-scratch
```

The expectation is an http application on port 8000 but the main difference is the size of the image, the old one from alpine is 12M the new one is 5M.

The scratch image is impossible to use with all applications but if you have a binary you can remove a lot of unused overhead.

Another way to understand the status of your image is to scan it to detect security vulnerabilities or exposures. Docker Hub and Docker Cloud can do it for private images.

This is a great feature to have in your pipeline to scan an image after a build.

CoreOS provides an open source project called clair to do the same in your environment.

It is an application in Golang that exposes a set of HTTP API to pull, push and analyse images. It downloads vulnerabilities from different sources like Debian Security Tracker or RedHat Security Data. Each vulnerability is stored in Postgres. Clair works like static analyzer, this means that it doesn't need to run our container to scan it but it persists different checks directly into the filesystem of the image.

```
docker run -it -p 5000:5000 registry
```

With this command we are running a private registry to use as a source for the image to scan

```
docker pull gianarb/micro:1.0.0
docker tag gianarb/micro:1.0.0
  localhost:5000/gianarb/micro:1.0.0
docker push localhost:5000/gianarb/micro:1.0.0
```

Now that we pushed in our private repo the micro image we can setup clair.

```
mkdir $HOME/clair-test/clair\_config
cd $HOME/clair-test
curl -L https://goo.gl/2fcpra -o
  clair\_config/config.yml
curl -L https://goo.gl/MzTrNL -o
  docker-compose.yml
```

Modify HOME/clair.config/config.yml and add the proper source postgresql://postgres:password@postgres:5432?sslmode=now you can run the following command to start postgres and clair:

```
docker-compose up
```

To make our test easier, we will use another CLI called hyperclair that is just a client to work with this application. If you are using Mac OS, you can follow the above commands, if you are in another OS you can find the correct url in the release page

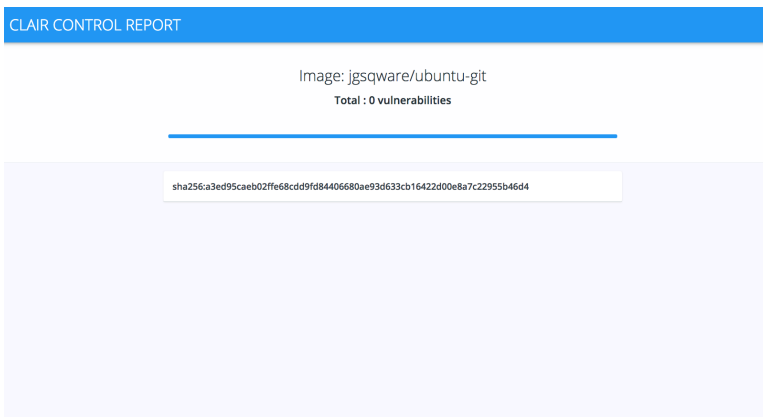
```
curl -SSl https://goo.gl/8WlkpS > ~/hyperclair
```

```
chmod 755 ~/hyperclair
```

Now we have an executable in `~/hyperclair`

```
~/hyperclair pull
  localhost:5000/gianarb/micro:1.0.0
~/hyperclair push
  localhost:5000/gianarb/micro:1.0.0
~/hyperclair analyze
  localhost:5000/gianarb/micro:1.0.0
~/hyperclair report
  localhost:5000/gianarb/micro:1.0.0
```

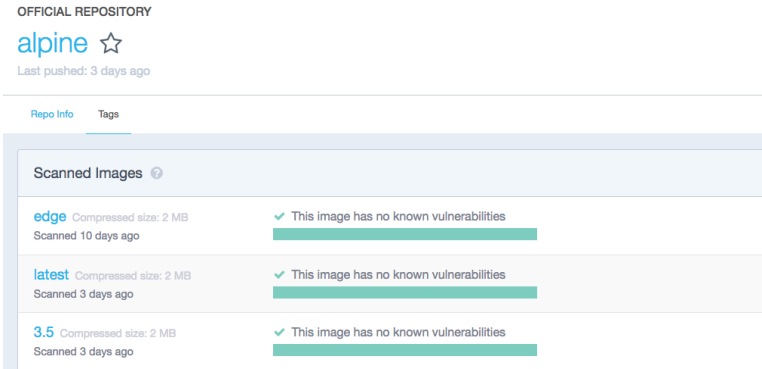
The generated report looks like this:



Hyperclair is just one of the implementations of clair, you can decide to use it or build your own implementation in your pipeline.

## 1.11 Docker Security Scanner

Docker Cloud and Docker Datacenter offer a similar feature called Docker Security Scanning. It is also available in Docker Hub to evaluate from a security point of view the status of the official images.



Docker Security Scanning is not only there to scan and check signature and SHA but also to recognize compiled binaries that contains common vulnerabilities and exposures (CVEs) for example if you image contains OpenSSL with the heathbleed vulnerability Security Scanning will notify you about this problem.

This feature is enabled in Docker Hub has beta, it scans private images and report back to you all issues for every tag.

To have a safe image is a good starting point to have a secure environment. Sometime is not a problem of your

application but if you are starting from a base image that it's not well designed you inherit a set of vulnerabilities. Be aware of them is important but Docker Security Scanning notify you with a report similar to CVE-2014-9912 that contains information about the single vulnerability. It means that you can try to update the package to a new version if a fix is available.

CVE is an open vulnerabilities directory when a new vulnerability is added into the system Docker Security Scanning re-scan all the images involved to check if they are safe or not.

We read about Clair previously, this service is not free and it's not opensource but it's well integrated into the powerful Docker workflow. If you have already Docker Datacenter or if you are using Docker Hub with private images it's something that you can start to use right now.

## 1.12 Secret Manager

Modern applications use a lot of third party services to ship particular features, these APIs require tokens and credentials. All your services like MySQL, Elasticsearch have credentials that you need to store in some safe place and you also need to put them into the container.

It's also important that a specific container has only credentials and secrets that it needs to run a specific application.

Probably your frontend application doesn't care about MySQL credentials like your backend application does but it requires other tokens and vice versa. Splitting the secrets is important because in the case of a vulnerability, you won't compromise all your tokens but only a subset of them.

Every configuration manager like Chef, Puppet and Ansible has their own security storage, Docker released their own embedded solution in Docker 1.13.

This means that from Docker 1.13 in SwarmMode, we can use a built-in security database to store and ship inside a service encrypted files.

At the moment it supports only files and not environment variables because it's not good practice and they prefer to release only file support.

We have a new command 'docker secret', it is the entrypoint of the feature where we can create, inspect and remove secrets.

First of all, we need to start a docker swarm cluster. When we have it up and running we can create a secret file and encrypt it in the Swarm:

```
echo '{"username":"root", "password": "root"}' >
~/secret-test.json
docker secret create myapp -f ~/secret-test.json
```

At this point, we can list all the secrets:

```
\begin{lstlisting}[language=Bash]
```

```
docker secret ls
```

ID	Digest	Size	Name	Created
---	-----	----	----	-----
njn6256a42476epuhh9awmk27	sha256:77cf	942	myapp	5 seconds ago

Or inspect one of them

```
docker secret inspect myapp
```

ID	: njn6256a42476epuhh9awmk27
Name	: myapp
Digest	: sha256:77cf
Size	: 942
Created	: 2016-11-01T16:16:53.105065598Z

We can create a service with the option `--secret secret_name:TARGET` in our case the secret `myapp` will be available in the container at `/opt/credential`

```
docker service create \
  --name backend \
  --secret myapp:/opt/credential \
  --image gianarb/micro:1.2.0
```

We can add and remove a secret at runtime with the command `docker service update`:

```
docker service update --secret-add foo
  --secret-rm myapp
```



There are also other solutions like HashiCorp's Vault but at the moment there is no native support for external storage. There are a few side projects to mount encrypted volumes with Vault and it is a great solution if you are interested in managing secrets not only for your swarm cluster but for infrastructure not managed by Docker Swarm. One of them is `docker-volume-libsecret` buildt by Evan Hazlett.

## 1.13 Immutability

Docker containers are in fact immutable. This means that a running container never changes because in case you need to update it, the best practice is to create a new container with the updated version of your application and delete the old one.

This aspect is important from a security point of view also because you will have a fresh container after each update and in the case of a vulnerability or injection they will be cleaned during the update.

You have also an instrument to analyse the attacked container with the command

```
docker diff <container_id>
```

This command shows the differences in the filesystem. It supports 3 events:

A - Add D - Delete C - Change

In case of attack, you can commit the attacked container to analyse it later and replace it with the original image.

This flow is interesting but if you know that your application doesn't need to modify the filesystem you can use `--read-only` parameters to make the fs read only or you can share a volume with the `ro` suffix `-v PWD:/data:ro`.

Docker can't fix the security issues for you, if your application can be attacked by a code injection then you need to fix your app but Docker offers a few utilities to make life hard for an hacker and to allow you to have more control over your environment.

During this chapter we covered some practices and tools that you can follow or use to build a safe environment. In general, you need to close your application in an environment that provides only what you need and what you know. If your distribution or your container has something that you don't have under your control or is unused then it is a good idea remove these dark points.

# Chapter 2

## Biography

I am Gianluca Arbezano (@gianarb) and I work as Software Engineer in different languages: PHP, Golang, JS and so on.

I am passionate about automation and all the DevOps philosophy during my work experience I used different cloud providers like Amazon Web Service, OpenStack, Digitalocean.

I have a good experience about different web layers I worked on AngularJs to build mobile and web application but also developed backend and scalable infrastructure



Figure 2.1: Thanks Ivan Frantar for this photo.

but I am focused right now on the last one.

I started my experience as developer with open source product and framework like Linux, PHP, Zend Framework, Vagrant and for this reason I am actively involved on different open source community Zend Framework, Doctrine, PHP, Golang and Docker.

I am a Docker Captain, it's a group of Docker experts and leaders in their communities who demonstrate a commitment to sharing their Docker knowledge with others (by Docker Inc).<sup>1</sup> This book is a way to share my experience with the big and great community around Docker, containers and distributed systems.

You can find me on Twitter<sup>2</sup> and GitHub<sup>3</sup>, they are my mainly social network, I am also a blogger<sup>4</sup> and an active and passionate speakers<sup>5</sup>.

Out of the web I am a volunteer involved on different ONG and ONLUS, I like the idea to leave the world better than you found it. I love ski, play football with friend, have a chat in some nice pub with a pint, eat and cook and son on.

---

<sup>1</sup><https://www.docker.com/community/docker-captains>

<sup>2</sup><https://twitter.com/gianarb>

<sup>3</sup><https://github.com/gianarb>

<sup>4</sup><http://gianarb.it/blog>

<sup>5</sup><http://gianarb.it/conferences>