Securing Docker with TLS certificates (http://tech.paulcz.net/2016/01/secure-docker-with-tls/)

January 3, 2016

By default Docker (and by extension Docker Swarm) has no authentication or authorization on its API, relying instead on the filesystem security of its unix socket `/var/run/docker.sock` which by default is only accessible by the root user.

This is fine for the basic use case of the default behavior of only accessing the Docker API on the local machine via the socket as the root user. However if you wish to use the Docker API over TCP then you'll want to secure it so that you don't give out root access to anyone that happens to poke you on the TCP port.

Docker supports using TLS certificates (both on the server and the client) to provide proof of identity. When set up correctly it will only allow clients/servers with a certificate signed by a specific CA to talk to eachother. While not providing fine grained access permissions it does at least allow us to listen on a TCP socket and restrict access with a bonus of also providing encryption.

Here I will detail what is required to secure Docker (and in turn Docker Swarm) running on a CoreOS (http://coreos.com/) server. I will assume you already have a CoreOS (http://coreos.com/) server running as described in my Docker Swarm post (http://tech.paulcz.net/2016/01/running-ha-docker-swarm/).

If you are only interested in securing Docker itself and not Docker Swarm then this should apply to any server with Docker installed that uses systemd. Even on systems without systemd it should provide enough details to secure Docker.

Creating Certificates

I will offer two methods to create the certificates, the first by using `openssl` to create a CA and then sign a key/cert pair, the second by using the paulczar/omgwtfssl (https://hub.docker.com/r/paulczar/omgwtfssl/) Docker Image which automates the certificate creation process.

Either way you'll want to start off by creating directories for both the server and client certificate sets:

```
$ sudo mkdir -p /etc/docker/ssl
$ mkdir -p ~/.docker
```

For this example we're creating the keys and certificates on the server itself, ideally you would do this on your laptop or via configuration management and never store the CA key on a public server.

OpenSSL

First run `openssl` to create and sign a CA key and certificate and copy the CA certificate into `/etc/docker/ssl`:

```
$ openssl genrsa -out ~/.docker/ca-key.pem 2048
.+++
...................................................................................
+++
e is 65537 (0x10001)

$ openssl req -x509 -new -nodes -key ~/.docker/ca-key.pem \
    -days 10000 -out ~/.docker/ca.pem -subj '/CN=docker-CA'

$ ls ~/.docker/
ca-key.pem  ca.pem

$ sudo cp ~/.docker/ca.pem /etc/docker/ssl
```

Next we'll need an openssl configuration file for the Docker client `~/.docker/openssl.cnf`:

```
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name
[req_distinguished_name]
[ v3_req ]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth, clientAuth
```

Followed by a configuration file for the Docker server `/etc/docker/ssl/openssl.cnf`. Add any DNS or IPs that you might use to access the Docker Server with, this is critical as the Golang SSL libraries are very strict:

```
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name
[req_distinguished_name]
[ v3_req ]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth, clientAuth
subjectAltName = @alt_names

[alt_names]
DNS.1 = docker.local
IP.1 = 172.17.8.101
IP.2 = 127.0.0.1
```

Next create and sign a certificate for the client:

```
$ openssl genrsa -out ~/.docker/key.pem 2048
....................................+++
.............+++
e is 65537 (0x10001)

$ openssl req -new -key ~/.docker/key.pem -out ~/.docker/cert.csr \
    -subj '/CN=docker-client' -config ~/.docker/openssl.cnf

$ openssl x509 -req -in ~/.docker/cert.csr -CA ~/.docker/ca.pem \
    -CAkey ~/.docker/ca-key.pem -CAcreateserial \
    -out ~/.docker/cert.pem -days 365 -extensions v3_req \
    -extfile ~/.docker/openssl.cnf
Signature ok
subject=/CN=docker-client
Getting CA Private Key
```

Then do the same for the server:

```
$ sudo openssl genrsa -out /etc/docker/ssl/key.pem 2048
................................................................................+
++
....................................+++
e is 65537 (0x10001)

$ sudo openssl req -new -key /etc/docker/ssl/key.pem \
    -out /etc/docker/ssl/cert.csr \
    -subj '/CN=docker-server' -config /etc/docker/ssl/openssl.cnf

$ sudo openssl x509 -req -in /etc/docker/ssl/cert.csr -CA ~/.docker/ca.pem \
    -CAkey ~/.docker/ca-key.pem -CAcreateserial \
    -out /etc/docker/ssl/cert.pem -days 365 -extensions v3_req \
    -extfile /etc/docker/ssl/openssl.cnf
Signature ok
subject=/CN=docker-client
Getting CA Private Key
```

OMGWTFSSL

If you want to skip manually creating the certificates you can use the paulczar/omgwtfssl (https://hub.docker.com/r/paulczar/omgwtfssl/) image which is a small (< 10mb) Docker image built specifically for creating certificates for situations like this.

First we'll create our client certs and use a docker volume binding to put the CA and certs into `~/.docker` :

```
$ docker run --rm -v $(pwd)/.docker:/certs \
    paulczar/omgwtfssl
----------------------------
| OMGWTFSSL Cert Generator |
----------------------------

--> Certificate Authority
====> Using existing CA Key ca-key.pem
====> Using existing CA Certificate ca.pem
====> Generating new config file openssl.cnf
====> Generating new SSL KEY key.pem
Generating RSA private key, 2048 bit long modulus
.............+++
..........+++
e is 65537 (0x10001)
====> Generating new SSL CSR key.csr
====> Generating new SSL CERT cert.pem
Signature ok
subject=/CN=example.com
Getting CA Private Key
```

Next we'll take ownership of them back from root (because of the docker volume binding) and then create the server certificates using the same CA using a second volume binding to `/etc/docker/ssl`:

Since this is a server certificate we need to pass the IP and DNS that the server may respond to via the -e command line arguments.

```
$ sudo cp ~/.docker/ca.pem /etc/docker/ssl/ca.pem
$ chown -R $USER ~/.docker
$ docker run --rm -v /etc/docker/ssl:/server \
    -v $(pwd)/.docker:/certs \
    -e SSL_IP=127.0.0.1,172.17.8.101 \
    -e SSL_DNS=docker.local -e SSL_KEY=/server/key.pem \
    -e SSL_CERT=/server/cert.pem paulczar/omgwtfssl
----------------------------
| OMGWTFSSL Cert Generator |
----------------------------

--> Certificate Authority
====> Using existing CA Key ca-key.pem
====> Using existing CA Certificate ca.pem
====> Generating new config file openssl.cnf
====> Generating new SSL KEY /server/key.pem
Generating RSA private key, 2048 bit long modulus
.................................+++
..................+++
e is 65537 (0x10001)
====> Generating new SSL CSR key.csr
====> Generating new SSL CERT /server/cert.pem
Signature ok
subject=/CN=example.com
Getting CA Private Key
```

Using the TLS certificates with Docker

Now we have our TLS certificates created and in the correct locations you need to tell Docker to use the TLS certificate and also verify the client. You do this by creating a drop in systemd unit to modify the existing Docker systemd unit.

Create the file `custom.conf` in `/etc/systemd/system/docker.service.d/` :

If you want to restrict local users from using the docker unix socket remove the second -H command line option, if you already have a custom drop in unit you can add the -H and –tls arguments to it.*

```
[Service]
Environment="DOCKER_OPTS=-H=0.0.0.0:2376 -H unix:///var/run/docker.sock --tlsveri
fy --tlscacert=/etc/docker/ssl/ca.pem --tlscert=/etc/docker/ssl/cert.pem --tlskey
=/etc/docker/ssl/key.pem"
```

Reload systemd and the Docker service:

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
```

Now when you try to access Docker via the TCP port you should get a TLS error:

```
$ docker -H tcp://127.0.0.1:2376 info
Get http://127.0.0.1:2376/v1.21/containers/json: malformed HTTP response "\x15\x0
3\x01\x00\x02\x02".
* Are you trying to connect to a TLS-enabled daemon without TLS?
```

This is because the Docker client does not know to use TLS to communicate with the server. We can set some environment variables to enable TLS for the client and use the client key we created:

```
$ export DOCKER_HOST=tcp://127.0.0.1:2376
$ export DOCKER_TLS_VERIFY=1
$ export DOCKER_CERT_PATH=~/.docker
$ docker info
docker info
Containers: 0
Images: 0
Server Version: 1.9.1
Storage Driver: overlay
 Backing Filesystem: extfs
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 4.3.3-coreos
Operating System: CoreOS 899.1.0
CPUs: 1
Total Memory: 997.4 MiB
Name: core-01
ID: RGVQ:VDUC:Z5LU:IE7I:J6UJ:TFBJ:SSCO:EWG2:QKAW:5FY6:EIAV:MROK
```

Using the TLS certificates with Docker Swarm

To secure Docker Swarm using these TLS certificates you will need to create TLS certificate/key pairs for each server using the same CA.

to add some arguments to the `docker run` command that you start Swarm Manager with the following:

```
$ docker run -d --name swarm-manager \
    -v /etc/docker/ssl:/etc/docker/ssl \
    --net=host swarm:latest manage \
    --tlsverify \
    --tlscacert=/etc/docker/ssl/ca.pem \
    --tlscert=/etc/docker/ssl/cert.pem \
    --tlskey=/etc/docker/ssl/key.pem \
    etcd://127.0.0.1:2379
```

Which you can then access using the docker client:

```
$ export DOCKER_HOST=tcp://127.0.0.1:2375
$ export DOCKER_TLS_VERIFY=1
$ export DOCKER_CERT_PATH=~/.docker

$ docker info
Containers: 6
Images: 5
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 3
 core-01: 172.17.8.101:2376
  └ Status: Healthy
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.023 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.3.3-coreos, operatingsyst
em=CoreOS 899.1.0, storagedriver=overlay
 core-02: 172.17.8.102:2376
  └ Status: Healthy
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.023 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.3.3-coreos, operatingsyst
em=CoreOS 899.1.0, storagedriver=overlay
 core-03: 172.17.8.103:2376
  └ Status: Healthy
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.023 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.3.3-coreos, operatingsyst
em=CoreOS 899.1.0, storagedriver=overlay
CPUs: 3
Total Memory: 3.068 GiB
Name: core-01
```

Related Posts

0 Comments **tech.paulcz.net** 1 **Login**

Recommend **Share** **Sort by Best**

Start the discussion…

Be the first to comment.

ALSO ON **TECH.PAULCZ.NET**

Moving from a self hosted Wordpress blog to Blogger

2 comments • 4 years ago•

The Dame Intl — Oh my gosh, I'm gonna have to hire someone to do this for me...

Creating immutable servers with chef and docker.io

6 comments • 3 years ago•

Bret Mogilefsky — It seems like one of the nice things about using chef this way is that the Dockerfiles you make are essentially …

Creating a Github Pages Blog With Octopress

2 comments • 4 years ago•

nickboucart — Super helpful, thanks!

Managing docker services with this one easy trick

1 comment • 3 years ago•

Roger Qiu — Does runit support processes that fork child processes?

Subscribe **Add Disqus to your site Add Disqus Add** **Privacy**