CHRIS WARD
WRITER, SPEAKER, AND DEVELOPER

Docker Secrets Management

About the Author.

Chris Ward is a technical writer, speaker, and developer. His mission is helping others understand technical subjects through technical writing, blogging, networking and educating people through presentations and workshops.

Codeship is a fully customizable hosted Continuous Integration and Delivery platform that helps you build, test, and deploy web applications fast and with confidence.

Learn more about Codeship here.

Docker Secrets Management

I'm sure we've all been there. That moment when you realize that important and sensitive access details have leaked online into a public space and potentially rendered your services to unrequited access.

With the ever-growing amount of services we depend on for our development stack, the number of sensitive details to remember and track has also increased. To cope with this problem, tools have emerged in the field of "secrets management." In this post, I am going to look at Docker Secrets, the new secrets management feature available in Docker 1.13 and higher.

The feature doesn't require much work on your part from a Docker perspective, but you may need to refactor your applications to take advantage of it. I will present ideas on how to do this, but not exhaustively.

ABOUT THIS EBOOK

To cope with the ever-growing amount of services we depend on for our development stack, let's look at Docker Secrets, the newly added secrets management feature in Docker 1.13.

Swarm Only

Docker Secrets only work across Docker swarms, mostly because that's the area where secrets management makes the most sense. After all, Swarm is aimed at production usage where multiple Docker instances need to share access details between themselves. If you want to use secrets management in a standalone container, you need to run a container with a `scale` value set to '1'. Docker for Mac and Windows doesn't support multinode swarm mode, but you can use them to create a multinode swarm with Docker Machine.

Create two machines and then a swarm of two nodes, and run the rest of the commands in this article from a manager in that swarm.

Getting Secrets

As you create secrets from the command line, you have the whole gamut of tools available to you to for creating random passwords and piping output. For example, to create a random password for a database user:

CODE

```
1  openssl rand -base64 20 | docker secret create mariadb_password -
```

This will return an ID for the secret.

You need to issue this command a second time to generate a password for the MariaDB root user. You will need this to get started, but you won't need this for every service.

```
CODE
1  openssl rand -base64 20 | docker secret create mariadb_root_password -
```

If you're already forgetting the secrets you've created, then the time-honored `ls` command also works here:

```
CODE
1  docker secret ls
```

Exchanging Secrets

To keep the secrets secret, communication between the services happens in an overlay network that you define. They're only available in that overlay network by calling their ID.

```
CODE
1  docker network create -d overlay mariadb_private
```

This will also return an ID for that network. Again you can use `docker network ls` to remind yourself of what's available.

Create Service(s)

This example will have a Docker node running MariaDB, and a node running Python. In a final application, the Python application would read and write to the database.

First, add a MariaDB service. This service uses the network you created to communicate, and the secrets created earlier are saved into two files: one for the root password and one for a default user password. Then pass all the variables you need to the service as environment variables.

CODE

```
docker service create \
  --name mariadb \
  --replicas 1 \
  --network mariadb_private \
  --mount type=volume,source=mydata,destination=/var/lib/mariadb \
  --secret source=mariadb_root_password,target=mariadb_root_password \
  --secret source=mariadb_password,target=mariadb_password \
  -e MARIADB_ROOT_PASSWORD_FILE="/run/secrets/mariadb_root_password" \
  -e MARIADB_PASSWORD_FILE="/run/secrets/mariadb_password" \
  -e MARIADB_USER="python" \
  -e MARIADB_DATABASE="python" \
  mariadb:latest
```

The Python instance again uses the private network you created and copies the secrets accessible in the network. A better (production-ready) option would be to create the databases your application needs in an admin program and never give the application access to root passwords, but this is purely for an example.

CODE

```
docker service create \
  --name cspython \
  --replicas 1 \
  --network mariadb_private \
  --publish 50000:5000 \
  --mount type=volume,source=pydata,destination=/var/www/html \
  --secret source=mariadb_root_password,target=python_root_password,mode=0400 \
  --secret source=mariadb_password,target=python_password,mode=0400 \
  -e PYTHON_DB_USER="python" \
  -e PYTHON_DB_ROOT_PASSWORD_FILE="/run/secrets/python_root_password" \
  -e PYTHON_DB_PASSWORD_FILE="/run/secrets/python_password" \
  -e PYTHON_DB_HOST="mariadb:3306" \
  -e PYTHON_DB_NAME="python" \
  chrischinchilla/cspython:latest
```

The example above uses a simple Docker image I created that sets up packages for creating a web application using Flask for serving web pages and PyMySQL for database access. The code doesn't do much but shows how you could access the environment variables from the Docker container.

For example, to connect to the database server with no database specified:

CODE

```
import os
import MySQLdb

db = MySQLdb.connect(host=os.environ['PYTHON_DB_HOST'],
user=os.environ['PYTHON_DB_ROOT_USER'],
passwd=os.environ['PYTHON_DB_PASSWORD_FILE'])

cur = db.cursor()

print(db)

db.close()
```

Rotating Secrets

It's good practice to frequently change sensitive information. However, as you likely know, updating those details in applications is a dull process most would rather avoid. Via Services, Docker Secrets management allows you to change the values without having to change your code.

Create a new secret:

CODE

```
1  openssl rand -base64 20 | docker secret create mariadb_password_march -
```

Remove the access to the current secret from the MariaDB service:

CODE

```
1  docker service update \
2    --secret-rm mariadb_password \
3    mariadb
```

And give it access to the new secret, pointing the target to the new value:

```
CODE
1  docker service update \
2    --secret-add source=mariadb_password_march,target=mysql_password \
3    mariadb
```

Update the Python service:

```
CODE
1  docker service update \
2    --secret-rm mariadb_password \
3    --secret-add source=mariadb_password_march,target=python_password,mode=0400 \
4    cspython
```

And remove the old secret:

```
CODE
1  docker secret rm mariadb_password
```

Docker Secrets Management and Codeship Pro

Now that we know how **Docker** handles things, the next question is: what do you do with your secrets if you're using Docker in a CI/CD service like Codeship Pro? This can bring its own set of challenges, so let's take a look specifically at Docker-related secrets managements issues and solutions using Codeship Pro.

Build Args, Env Vars, Trade-offs

The first, and most common, ways of including secrets in your build is through build arguments and environment variables. The distinction between the two is one of build time vs run time. Inside your Dockerfiles, you use Build arguments, which are essentially the same as environment variables but are consumed when the container builds, meaning they are not available to your application, in the environment, once the container is running. This makes Build arguments perfect for things like access keys to private dependencies or context-awareness in the build environment. Build arguments are also strictly required, meaning any declared build argument must be used for the container to build successfully.

On Codeship Pro, we allow you to encrypt your build arguments in your `codeship-services.yml` file. To learn more about your Services file, and how Codeship Pro works in general, you can watch this short demo video. This encryption helps on a service like Codeship because otherwise they would need to be visible in your repo for the CI/CD process to work, but other than this Codeship uses build arguments identical to how Docker and Docker Compose use them.

Using encrypted build arguments on Codeship is pretty simple. You start with adding an encrypted args file to your Services file (encrypted with our Jet CLI tool):

CODE

```
app:
  build:
    dockerfile: Dockerfile
    encrypted_args_file: build_args.encrypted
```

Then you declare and use them in your Dockerfile:

CODE

```
FROM ubuntu:latest

ARG build_env
# build_env=test would make test the default value

RUN script-requiring-build-env.sh "$build_env"
```

Using environment variables is similar, as Codeship Pro also offers encryption of your environment variables. Unlike build arguments, environment variables declared via your Compose or Services file are not used in your Dockerfile and are instead loaded into the container environment once it has built, at run time. Declaring your environment variables looks like this:

CODE

```
aws:
  build:
     image: codeship/aws-deployment:latest
  encrypted_env_file: deployment.env.encrypted
```

So, environment variables and build arguments are two easy ways to include secrets in your builds, why not just use them every time?

For build arguments, the downside is that your secrets live in your Docker image history. Anybody who gets access to the image will be able to look at the history of the image to see the secrets. Not the biggest problem, but it can be a cause of concern for some teams.

For environment variables, the biggest issue is that they need to either be visible in your Compose file, or encrypted but stored in your repo for your Codeship Pro `codeship-services` file. While the AES encryption is pretty secure, and it's easy enough to keep your repo private, some teams may conditionally decide that they do not like this approach.

Externally Hosting Secrets

One more option would be to store your secrets externally, on a CDN or accessible remote endpoint. This has a somewhat ironic problem of needing to include keys that can be used to access the remote location where your keys are, but there can still be some benefits.

For starters, if used in the Dockerfile context, you can fetch, use and delete the keys on the same layer - meaning they are never saved in the image history.

If used in the run-time context, you don't need to add your important keys to the repo at any point.

While it may seem awkward to include a key just to access another, more important key, some teams have found this triages the risk a bit. You can install more security on your external location where you store the higher-risk asset than you can in your source control or image registry, and there's more auditing capability for security purposes as well. If there is a breach, it's also easier to remove and swap out a key stored externally than it is one embedded in the application itself.

Spread the Word

Docker Secrets is a new feature (March 2017), but Docker encourages image maintainers to add support for it as soon as possible for better security among Docker users. This entails allowing for a similar process to the example above, where a container can read every parameter it needs from a file(s) created by generating a secret instead of hard-coded into an application. This enforces the idea of containerized applications as containers can come and go, but always have access to the important information you need for your application to run.

For more details on the `secret` command read the reference guide here.

Share this Codeship Guide

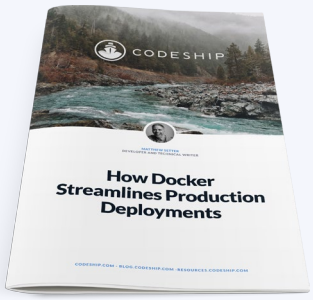
More Codeship Resources.

EBOOKS

How Docker Streamlines Production Deployments.

In this eBook, find out some of the core reasons how Docker helps to predictably streamline the deployments of your application.

Download this eBook



EBOOKS

Best Practices for Building Minimal Docker Images.

In this eBook we will look at some ways to streamline your Docker image as small as possible..

Download this eBook

EBOOKS

Container Image Immutability and the Power of Metadata.

In this eBook you will learn how to use LABEL to add meaningful metadata to your container images..

Download this eBook

CODESHIP

Share this Codeship Guide

About Codeship.

Codeship is a hosted Continuous Integration service that fits all your needs. Codeship Basic provides pre-installed dependencies and a simple setup UI that let you incorporate CI and CD in only minutes. Codeship Pro has native Docker support and gives you full control of your CI and CD setup while providing the convenience of a hosted solution.

Codeship Basic

A simple out-of-the-box Continuous Integration service that just works.

Starting at $0/month.


Works out of the box


Preinstalled CI dependencies


Optimized hosted infrastructure


Quick & simple setup

LEARN MORE

Codeship Pro

A fully customizable hosted Continuous Integration service.

Starting at $0/month.


Customizability & Full Autonomy


Local CLI tool


Dedicated single-tenant instances


Deploy anywhere

LEARN MORE