Manage sensitive data with Docker secrets

About secrets

In terms of Docker Swarm services, a secret is a blob of data, such as a password, SSH private key, SSL certificate, or another piece of data that should not be transmitted over a network or stored unencrypted in a Dockerfile or in your application's source code. In Docker 1.13 and higher, you can use Docker secrets to centrally manage this data and securely transmit it to only those containers that need access to it. Secrets are encrypted during transit and at rest in a Docker swarm. A given secret is only accessible to those services which have been granted explicit access to it, and only while those service tasks are running.

You can use secrets to manage any sensitive data which a container needs at runtime but you don't want to store in the image or in source control, such as:

- Usernames and passwords
- TLS certificates and keys
- SSH keys
- Other important data such as the name of a database or internal server
- Generic strings or binary content (up to 500 kb in size)
- Another use case for using secrets is to provide a layer of abstraction between the container and a set of credentials. Consider a scenario where you have separate development, test, and production environments for your application. Each of these environments can have different credentials, stored in the development, test, and production swarms with the same secret name. Your containers only need to know the name of the secret to function in all three environments.

***Note**: Docker secrets are only available to swarm services, not to standalone containers. To use this feature, consider adapting your container to run as a service*

How Docker manages secrets

When you add a secret to the swarm, Docker sends the secret to the swarm manager over a mutual TLS connection. The secret is stored in the Raft log, which is encrypted. The entire Raft log is replicated across the other managers, ensuring the same high availability guarantees for secrets as for the rest of the swarm management data.

When you grant a newly-created or running service access to a secret, the decrypted secret is mounted into the container in an in-memory filesystem. The location of the mount point within the container defaults to **/run/secrets/<secret_name>** in **Linux containers**

Useful Commands:

- **docker secret create**
- **docker secret inspect**
- **docker secret ls**
- **docker secret rm**
- **--secret flag for docker service create**
- **--secret-add and --secret-rm flags for docker service update**

Lab:

1. Add a secret to Docker. The docker secret create command reads standard input because the last argument, which represents the file to read the secret from, is set to -.

```
printf "This is a secret" | docker secret create my_secret_data -
```

2. Create a redis service and grant it access to the secret. By default, the container can access the secret at /run/secrets/<secret_name>, but you can customize the file name on the container using the target option.

```
docker service  create --name redis --secret my_secret_data redis:alpine
```

3. Verify that the task is running without issues using docker service ps. If everything is working, the output looks similar to this:

```
docker service ps redis
```

4. If there were an error, and the task were failing and repeatedly restarting, you would see something like this:

```
docker service ps redis
```

5. Get the ID of the redis service task container using docker ps

```
docker ps --filter name=redis -q
<Container ID>

docker container exec $(docker ps --filter name=redis -q) ls -l /run/secrets

docker container exec $(docker ps --filter name=redis -q) cat
/run/secrets/my_secret_data

<Your Secret will be listed here>
```

6. Verify that the secret is **not** available if you commit the container.

```
docker commit $(docker ps --filter name=redis -q) committed_redis
docker run --rm -it committed_redis cat /run/secrets/my_secret_data
cat: can't open '/run/secrets/my_secret_data': No such file or directory
```

7. Try removing the secret. The removal fails because the redis service is running and has access to the secret.

```
docker secret ls
docker secret rm my_secret_data
```

<Error response from daemon: rpc error: code = 3 desc = secret

'my_secret_data' is in use by the following service: redis >

8. Remove access to the secret from the running redis service by updating the service.

```
docker service update --secret-rm my_secret_data redis
```

9. Stop and remove the service, and remove the secret from Docker.

```
docker service rm redis

docker secret rm my_secret_data
```

Lab 2:

Secrets are not loaded into the container's environment, they are mounted to /run/secrets/

Here is a example:

1) Project Structure:

```
|
|---   docker-compose.yml
|---   super_secret.txt
```

2) **docker-compose.yml** contents:

```
version: "3.6"

services:

  my_service:
    image: ubuntu
    entrypoint: "cat /run/secrets/my_secret"
    secrets:
      - my_secret

secrets:
  my_secret:
    file: ./super_secret.txt
```

3) **super_secret.txt** contents:

```
my secret prakash
```

4) Run this command from the project's root to see that the container does have access to your secret, (Docker must be running and docker-compose installed):

```
docker-compose up --build my_service
```