Background

- Most containerized apps have been implemented in a way to inject both configs and secrets as environment variables.
- Secrets shouldn't be stored in plain-text as a best practice.
- Kubernetes provides integrations with secure vaults so that the secrets can be stored in encrypted forms in the vault and injected to the container during runtime.
- These secrets are mounted into the containers filesystem(tmpfs) as individual files instead of exposing them as environemnt variables.
- However, K8s also provides constructs to easily expose the content of these secret files as environment variables.

Problem

- During development, usually Kubernetes is not used in developer machines. The popular choices are Docker and Compose.
- Developers now need to find a way to inject the secrets **safely** to the containers running in their developer machines.
- They could pass them as plain-text env variables to the containers. However, this is not safe as secrets can be leaked easily in this form.
- Docker comes with Swarm mode which lets us to create secrets in encrypted form.
- Similar to K8s, Compose can be used to mount these Swarm secrets to the filesystem(tmpfs) of the containers.
- However, Swarm/Compose **does not include** constructs to expose these mounted secrets as environment variables. Why? Explained here.
- How can we pass these secrets to the containers safely without modifying the application code or the Dockerfile?

Solution

1. Initialize the swarm mode in Docker.

```
docker swarm init
```

2. Create the secrets in Swarm.

```
printf "myapipassword" | docker secret create api_password -
printf "mydbpassword" | docker secret create db_password -
```

3. Modify your compose file(`compose.yml`) with secret references and modify the entrypoint to use a script.

```
services:
  myapp:
    image: sample-app:v1
    entrypoint: ["sh", "-c", "/app/entrypoint.sh"]
    secrets:
      - db_password
      - api_password
secrets:
  db_password:
    external: true
  api_password:
    external: true
```

4. Create a new file(`entrypoint.sh`) in the source code directory with the following content. Note that the final image has to include this file along with the source code.

```
#!/bin/sh

db_password=$(cat /run/secrets/db_password)
api_password=$(cat /run/secrets/api_password)

db_password=$db_password api_password=$api_password node /app/app.js
```

The above script does the following.

- Sets two env variables and initializes it with the content of the secret files mounted to the container.
- Pass these env variables to the original command (`node /app/app.js`)

5. Make sure to build the docker image again, so that the entrypoint script gets copied to the image.

```
docker build -t sample-app:v1
```

6. Deploy the app into swarm

```
docker stack deploy --compose-file=compose.yml myapp
```

Your application should now have access to the secrets as env variables. We successfully injected the secrets in a safe manner to the containers without modifying the application code.

Optional

Note that an `exec` into the containers terminal will not show these env variables as it opens up a new terminal session. However, since we passed these values to the main process via the entrypoint script, the process has access to the env variables. Adding a console log to the app code or debugging the process will help you to verify this.

```
const db_password = process.env.db_password;
const api_password = process.env.api_password;

console.log(`db_password is ${db_password}`);
console.log(`api_password is ${api_password}`);
```

Citations and helpful links

- Why you shouldn't use ENV variables for secret data
- Manage sensitive data with Docker secrets
- How to use secrets in Docker Compose