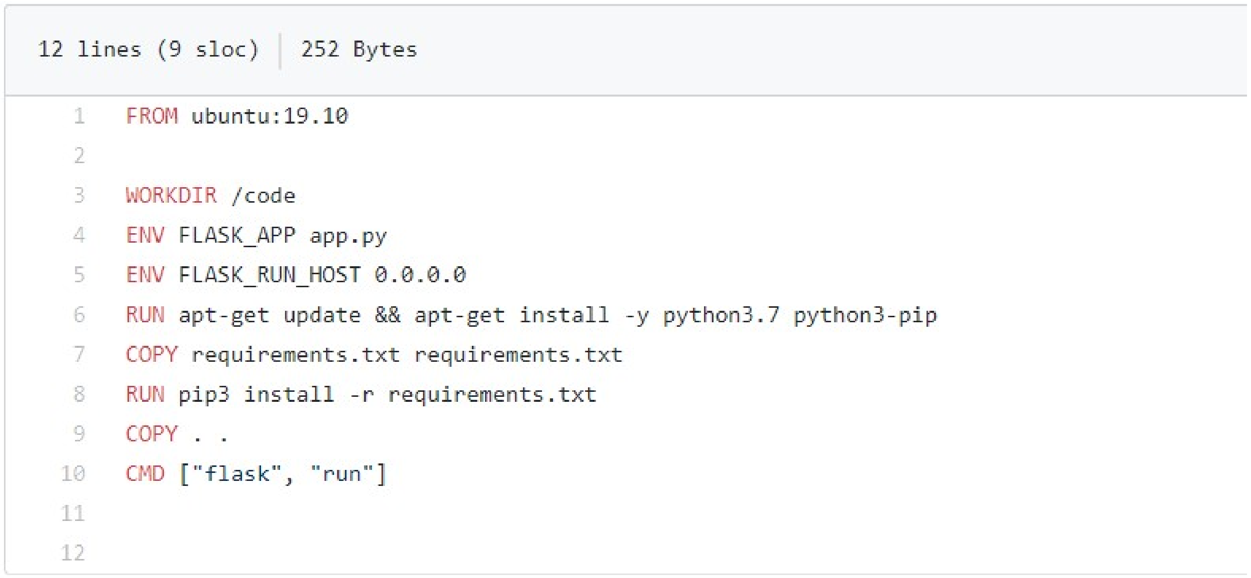
Matthew Hibshman (mrh583@nyu.edu)
Student: N17252302
Application Security Fall 2019, Assignment #4

GitHub Repo: https://github.com/crazyzete/AppSecAssignment4

Docker

This assignment took the Flask Web Service created in previous assignments and packed it in a docker container. Subsequent sections talk about utilizing Secrets and Docker Swarm, but for this section I am focusing on what I did in order to containerize and build as a docker image.

First, I created a Dockerfile. This file contains the information necessary to build the containerized image. The Docker file is shown below in Figure 1.



12 lines (9 sloc) | 252 Bytes

```
1   FROM ubuntu:19.10
2
3   WORKDIR /code
4   ENV FLASK_APP app.py
5   ENV FLASK_RUN_HOST 0.0.0.0
6   RUN apt-get update && apt-get install -y python3.7 python3-pip
7   COPY requirements.txt requirements.txt
8   RUN pip3 install -r requirements.txt
9   COPY . .
10  CMD ["flask", "run"]
11
12
```

Figure 1: Dockerfile

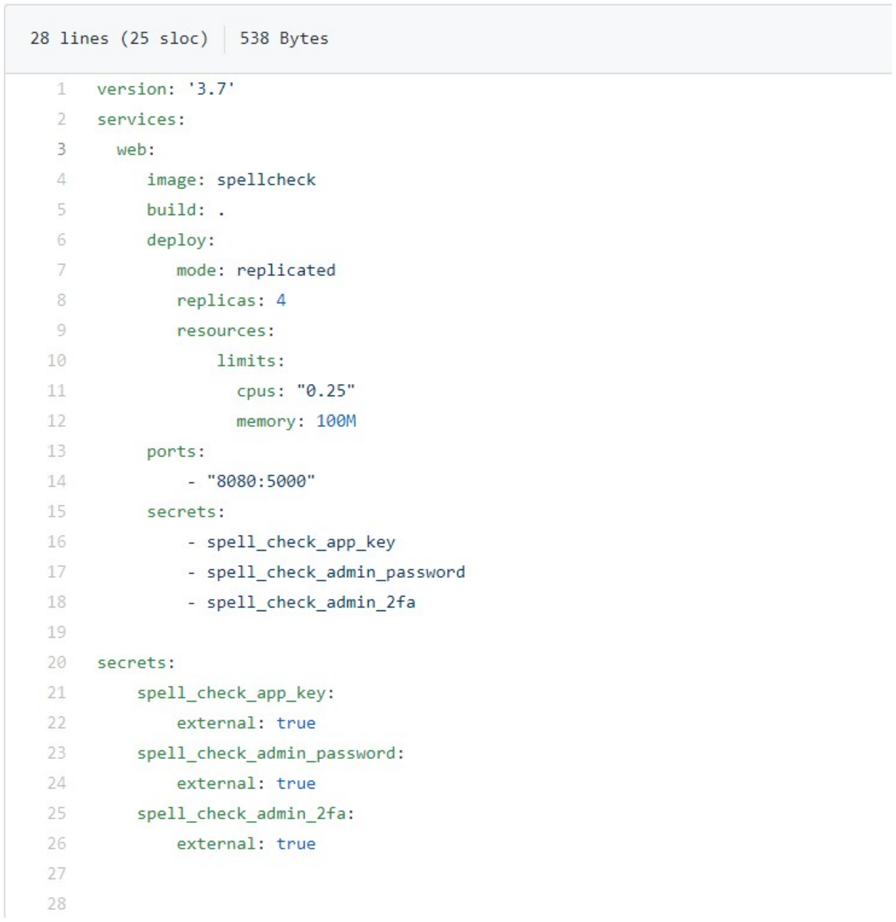
In the above Dockerfile, there are a few things to point out. On line #1 I used ubuntu:19.10 as a base image. This is the latest release "Eoan Ermine" and the version I am running in my virtual machines. Alternatively, I could have used ubuntu:latest, which would pull the latest LTS release, but that is Ubuntu 18.04, which is older than what I am running. However, if LTS release is enough, using latest does offer the benefit of building with the latest images as they become available.

From there on line #3 I established a working directory for the code. Lines #4 and #5 set the Flask app name, and the 0.0.0.0 indication for host to indicate any host IP address. Line #6 updates apt-get and the installs python 3.7 and pip3 so that Line #7 can copy the requirements.txt into the container so Line #8 can install the requirements (including the flask environment). That appears to be all that I need. I looked at the image because I was a little unclear how it worked without installing sqlite3, and I found the sqlite3 shared libraries already on the default image. Line 9 then copies the code files into the container and Line 10 is the default command that invokes ‘flask run’. There it is, the docker container is setup and able to execute.

Docker Compose & Deploy with Swarm

Once the base Dockerfile is in place, it is time to consider a deployment strategy. Docker-compose can be used to bring up a single container instance, but it can also be used to deploy to a container cluster using Docker Swarm. To be able to deploy to a swarm, I first had to issue the command “docker swarm init”. This initialized a swarm manager on my virtual machine that persists after reboots. This service starts when the VM instance starts. It also provides a unique key to use to request additional VM nodes to joint the swarm as worker nodes. At this point after issuing the command, the Docker Swarm infrastructure is running, but no container services are running.

To facilitate building and deploying, I created a docker-compose.yml file shown in Figure 2 below.



```
28 lines (25 sloc)  |  538 Bytes
```

```
version: '3.7'
services:
  web:
     image: spellcheck
     build: .
     deploy:
        mode: replicated
        replicas: 4
        resources:
            limits:
              cpus: "0.25"
              memory: 100M
     ports:
         - "8080:5000"
     secrets:
         - spell_check_app_key
         - spell_check_admin_password
         - spell_check_admin_2fa

secrets:
    spell_check_app_key:
        external: true
    spell_check_admin_password:
        external: true
    spell_check_admin_2fa:
        external: true


```

Figure 2: docker-compose.yml File

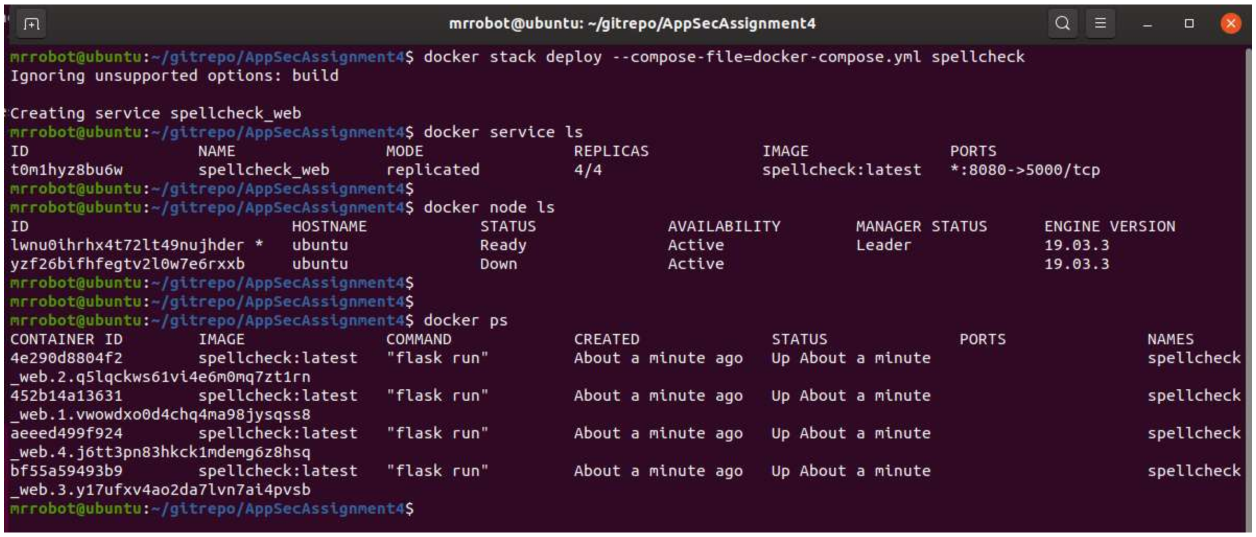
Key elements to note in this file are:

- Line 3 defines a web service using the spellcheck image on Line 4, with build command referencing the current directory (where it looks for the Dockerfile).
- The deploy section starting at Line 6 allows me to set the deployment to replicated (Line 7), with 4 replicas (Line 8), limiting CPU utilization to 25% (Line 11) and memory to 100M per container (Line 12).
- Line 14 handles the port mapping of the service, exposing the service on port 8080 externally and mapping to port 5000 inside the container (where Flask sets up the listener).

That is really all that is to the deployment. I left out talking about the secrets section as I will discuss that in the next section of this report. Once the above file is created, to deploy it, all that is needed is:

```
docker stack deploy –compose-file docker-compose.yml spellcheck
```

This tells docker to deploy the container to the swarm and use the deployment information in the docker-compose.yml file to deploy the 'spellcheck' container. In this case, 4 replicas with the limits established in the above file are deployed to the cluster. Even if there is only 1 VM in the cluster, 4 replicas will be deployed (but the docker mesh will automatically load balance access).



```
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$ docker stack deploy --compose-file=docker-compose.yml spellcheck
Ignoring unsupported options: build

Creating service spellcheck_web
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
t0m1hyz8bu6w        spellcheck_web      replicated          4/4                 spellcheck:latest   *:8080->5000/tcp
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
lwnu0ihrhx4t72lt49nujhder *   ubuntu              Ready               Active              Leader              19.03.3
yzf26bifhfegtv2l0w7e6rxxb     ubuntu              Down                Active                                  19.03.3
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS              PORTS               NAMES
4e290d8804f2        spellcheck:latest   "flask run"         About a minute ago   Up About a minute                       spellcheck
_web.2.q5lqckws61vi4e6m0mq7zt1rn
452b14a13631        spellcheck:latest   "flask run"         About a minute ago   Up About a minute                       spellcheck
_web.1.vwowdxo0d4chq4ma98jysqss8
aeeed499f924        spellcheck:latest   "flask run"         About a minute ago   Up About a minute                       spellcheck
_web.4.j6tt3pn83hkck1mdemg6z8hsq
bf55a59493b9        spellcheck:latest   "flask run"         About a minute ago   Up About a minute                       spellcheck
_web.3.y17ufxv4ao2da7lvn7ai4pvsb
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$
```

Figure 3: Docker Swarm Deployment

In Figure 3 above, you can see execution of the docker stack deployment. A subsequent execution of 'docker service ls' shows the spellcheck_web service running, in 'replicated' mode with 4/4 replicas running, and additionally shows the port 8080->5000/tcp mapping. The 'docker ps' command shows 4 separate container IDs running with the 'flask run' command. At this point, we have a swarm cluster running (all on a single VM in this case), serving out the spellcheck web service.

Note that as the assignment says, an external load balancer can be configured and used. This assignment did not require separation of the database from the flask web service docker image. The side effect here is that when multiple containers are run on the same machine, the docker swarm mesh automatically load balances each request. Because each container maintains a local copy of its database, this becomes a problem when one client issues a POST request to register a new user to the database, and the next login POST request attempts to log in the client to another container. Because the database is not on a shared storage location or replicated across container, this creates a consistency issue. Therefore, this setup is not a completely realistic deployment scenario but demonstrates how container deployment and orchestration function.

Docker Secrets

In the Docker Swarm section above, I alluded to the use of docker secrets in the docker-compose.yml file. Once a docker swarm is in place, secrets can be added manually by typing a docker create secret command. As you can see in Figure 4 below, a docker secret was created for the app_key (that is used to generate the CSRF token), the admin password, and the admin 2FA token. A 'docker secret ls' shows the secret names that are applied but does not actually reveal the secret. Note that the app_key can be really set to anything, but as demonstrated below, I utilized 'opens rand -hex 48' to generate the key.

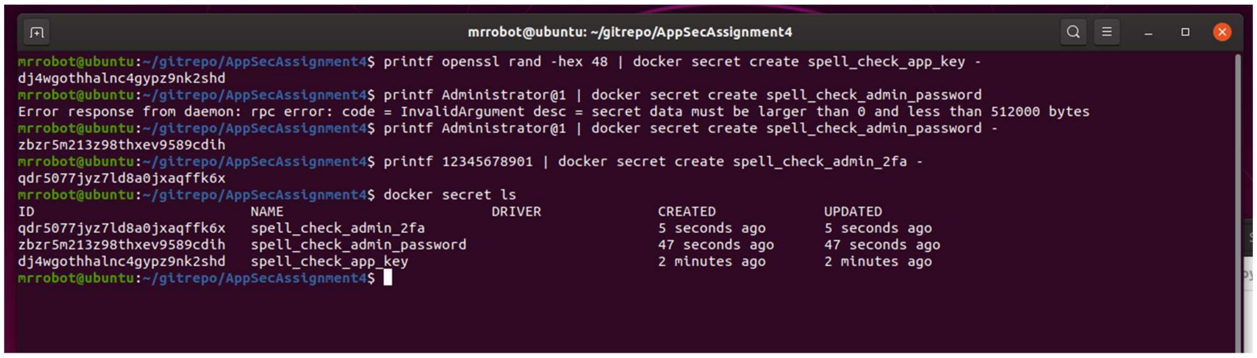
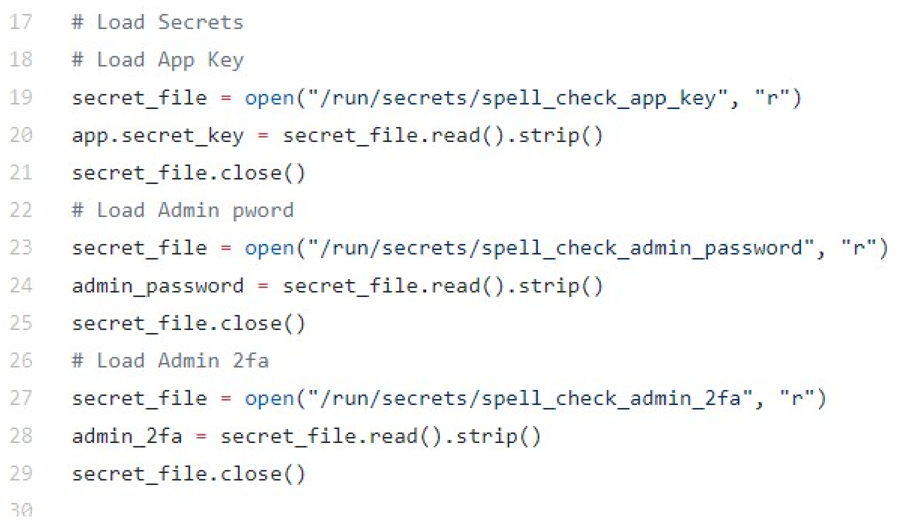

```
mrrobot@ubuntu: ~/gitrepo/AppSecAssignment4
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$ printf openssl rand -hex 48 | docker secret create spell_check_app_key -
dj4wgothhalnc4gypz9nk2shd
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$ printf Administrator@1 | docker secret create spell_check_admin_password
Error response from daemon: rpc error: code = InvalidArgument desc = secret data must be larger than 0 and less than 512000 bytes
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$ printf Administrator@1 | docker secret create spell_check_admin_password -
zbzr5m213z98thxev9589cdih
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$ printf 12345678901 | docker secret create spell_check_admin_2fa -
qdr5077jyz7ld8a0jxaqffk6x
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$ docker secret ls
ID                          NAME                          DRIVER              CREATED             UPDATED
qdr5077jyz7ld8a0jxaqffk6x   spell_check_admin_2fa                             5 seconds ago       5 seconds ago
zbzr5m213z98thxev9589cdih   spell_check_admin_password                        47 seconds ago      47 seconds ago
dj4wgothhalnc4gypz9nk2shd   spell_check_app_key                               2 minutes ago       2 minutes ago
mrrobot@ubuntu:~/gitrepo/AppSecAssignment4$ 
```

Figure 4: Docker Secrets

I investigated how the secrets are stored and learned they are stored encrypted in the Raft log with the docker engine. This protects the data at rest, and they are loaded with a decryption key into memory when the engine starts. Alternatively, I learned the docker engine can be set to not auto restart and an unlock key generated which an administrator can use to unlock the docker engine on startup. Docker recommends this key be stored in a password manager. In either case, once added to the docker engine by an administrator, the secrets are protected and are securely provide to containers permitted to access them.

The docker container permissions to access the secrets are provided in the docker-compose.yml file. Referencing the previous screenshot of the docker-compose.yml file in Figure 2 above, lines 16-18 list the docker secrets that should be made available to my spellcheck web service. This just lists the secret names as entered into the docker engine and shown in Figure 4 above. Additionally, lines 21-26 indicate each of those secrets are external. This means the deployment should expect to obtain the secrets from the docker engine. They can alternatively be provided in another file and referenced from the docker-compose.yml file, but this means that the secrets are potentially stored unencrypted on disk and pulled into the deployment.

Once the docker container is deployed, the 3 secrets it has been granted access to are placed in the container unencrypted in text files in /run/secrets/secret_name, where secret_name is a file named for the secret. All that needs to change in the app.py file is instead of the secret values being hardcoded or generated, the python app needs to open and read the secret out of the text file, as seen in the code snippet in Figure 5 below. This is potentially easily testable outside a deployment environment by manually copying text files into a /run/secrets directory and verifying the functionality before deploying to a swarm.



```
# Load Secrets
# Load App Key
secret_file = open("/run/secrets/spell_check_app_key", "r")
app.secret_key = secret_file.read().strip()
secret_file.close()
# Load Admin pword
secret_file = open("/run/secrets/spell_check_admin_password", "r")
admin_password = secret_file.read().strip()
secret_file.close()
# Load Admin 2fa
secret_file = open("/run/secrets/spell_check_admin_2fa", "r")
admin_2fa = secret_file.read().strip()
secret_file.close()

```

Figure 5: app.py Secrets Updates

At this point, a docker container has been built, deployed to a swarm using docker-compose with multiple replicas and CPU/memory limits established, and secrets have been pulled out of the source code and stored securely in the docker engine allowing access to those container instances authorized to access them. From the application viewpoint, the unencrypted secrets are securely delivered into the container and access is as easy as a regular python file read. A little configuration and minor code changes have converted the Assignment 4 flask web service into a replicated containerized deployment. Note that I only executed my deployments on my Ubuntu Linux VM. I would have liked to try to run this under Windows, but the Docker Engine does not install under Windows Home (only Professional or Enterprise), so this was not a viable option for my testing. But I think the flexibility of being able to deploy and run a container on a Windows environment containing an Ubuntu image truly brings us closer to the 'write once, run anywhere' approach that Java touted at its inception.

Docker Content Trust

The final portion of this report is a discussion on Docker Content Trust. Per the assignment directions, I was not required to implement this. Just discuss its purpose as it pertains to the containerized web service I just built.

The Docker Content Trust framework is built off the Docker Notary framework, which employs "The Update Framework" (TUF) as a security model (built by our own Dr. Justin Cappos). This provides a way for content publishers to sign images such that a user of that image (consumer) can trust that the image they are acquiring was legitimately produced by the publisher and has not been modified.

Like we learned about the TUF framework in Information Security and Privacy class, the DCT framework utilizes multiple keys to secure the image publication and delivery process. An offline key is used create a tagging key. The tagging key is then what is maintained by authorized individuals to sign a tag. The image itself is not signed, rather, the tag for an image is signed. Additional keys such as a timestamp key provide additional protection to the integrity of the images signed in the repository.

The main advantage to using DCT is to provide integrity of the images published and confidence to the end user that the image is provided unaltered from the stated publisher. This is all very similar to other code signing mechanisms that have been utilized in the Java language and web-based plug-ins such as ActiveX. With Docker images being often publicly available for download or inclusion in other images, it is important that the end user has trust in what they are accessing. An unsigned Linux image, for example, may contain malicious kernel (or other code) and so assurance that the image came from an authoritative source is important. Of course, you still need to trust the publisher that is providing the content. The docker engine provide an ability to indicate that only trusted content should be utilized and automatically checks signing information to mitigate risk to endpoint clients for running unsigned images. A publisher does not need to sign all (or any image) for that matter, but it is good practice to sign that help enable trust in the image content.

Conclusion

Overall this was an interesting exercise. I had not previously had any containerization experience and learned how easy it was to create and deploy a container with replicas to a swarm and protect the secrets so they can be extracted from the code. Coming from a more traditional software development background, I was impressed at the ease at which an entire environment can be setup and deployed. From a security perspective, I see the value in being able to do processing work within an environment and then completely destroy and re-instantiate the environment back to a known good state from an image, as well as the ability to quickly deploy images to nodes that have a docker engine and do not need a set of dependencies pre-installed.