**Michael Tse**

**11/24/19**

**Application Security – Assignment 4 – Fall 2019**

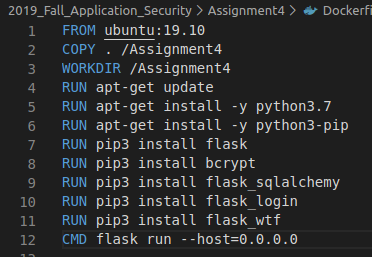
**Github Repository -** <https://github.com/mt1836/Assignment4>

**INTRODUCTION**

In this assignment we containerized our web application from assignment 3 by using Docker. Aside from containerization, we also utilized features in Docker that provides automated builds, replica creation and secrets management through the use of docker-compose, docker stack, docker swarm and docker secrets.

**IMPLEMENTATION**

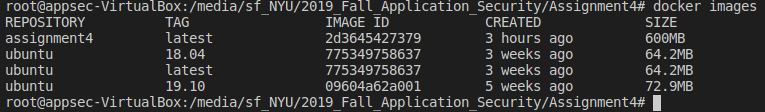
The first step in containerizing our web app involved creating a dockerfile which is essentially a text file that tells docker how to build your applications environment with all of the pre-requisites to run your service/application. Below is an image of the dockerfile we used:



Once the dockerfile has been created we can build the image by running the following command. In the example below we built an image that gets stored in a local repository called assignment4.

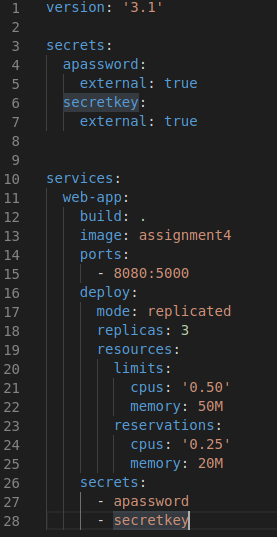


We can see the image in the following screenshot along with a few other images we have for the base Ubuntu OS.



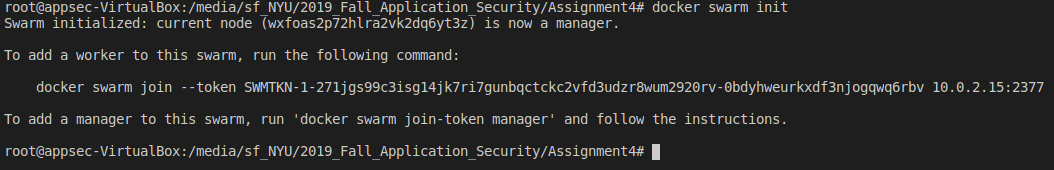
From here we can perform a proof of concept to ensure that our application works under one container by creating the container using docker create and then starting it using docker start, or you could use docker run which combines the create/start into one command. Having a containerized application auto built from a dockerfile is a very robust way to deploy code as others can download the dockerfile and be able to build the environment that will run your application without issue very quick and easily.

Being able to do this with one container is great but in more complex deployments where you may have a web server and a db server (we have a combined web and db server in this assignment) you don’t want to manually run docker commands to spin up each individual container. This is where docker-compose.yml files come into play. Docker-compose.yml files take the images built by the dockerfile and spin up instances of the images in the form of containers. It can spin up a web server a database server and set port mapping settings as shown in the screenshot below:

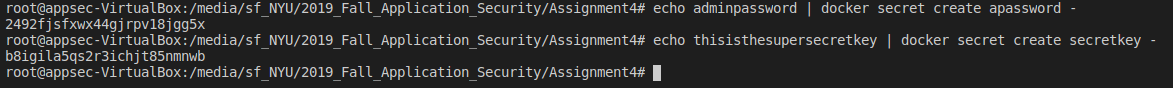


Docker-compose up would be the command you use to build based off the dockerfile and docker-compose.yml file with some limitations. Docker-compose does not recognize the deploy and secrets section of the docker-compose.yml file. For this reason we need to utilize docker stack deploy which will execute the docker-compose.yml file in its entirety.

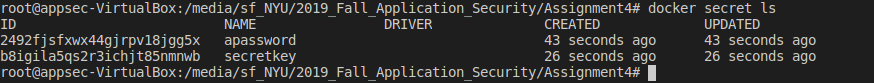
Before running this command however we need to handle secrets. Secrets are a way to securely store sensitive data such as passwords and tokens needed for automatic deployments/builds. By using secrets these sensitive data points can be excluded from plain text view in dockerfiles and our code which is uploaded to a public repository that anyone would be able to see. To setup secrets we need to initialize docker swarm using the following command:

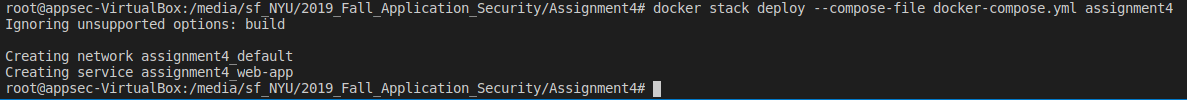


Once this is done we can create the secret by having the owner of the web app input it via the command line using the command below:

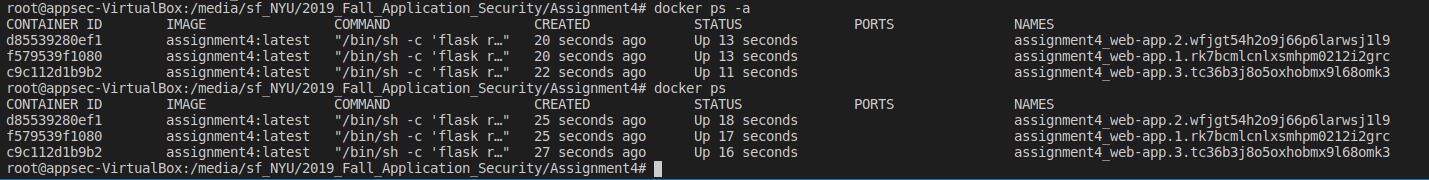


We can see the secrets created by typing docker secret ls. In our assignment we created two secrets. One called apassword which stores the admin password “adminpassword” and another that stores our secretkey “thisisthesupersecretkey”.

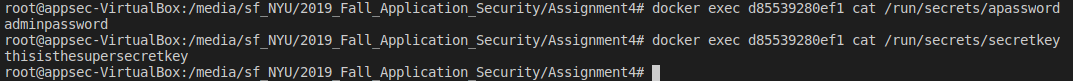


Now that we have secrets setup we can create our containers/replicas with resources limited for CPU and memory by running the following docker stack deploy command:

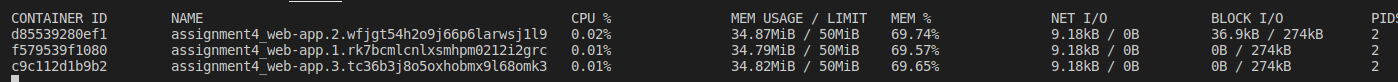
Once the command has been run we can docker ps to see our running containers and we see 3 replicas which match our docker-compose.yml file.

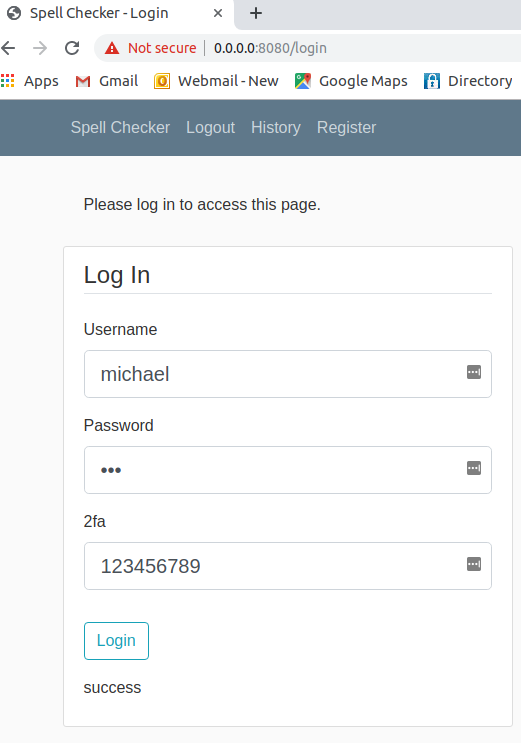
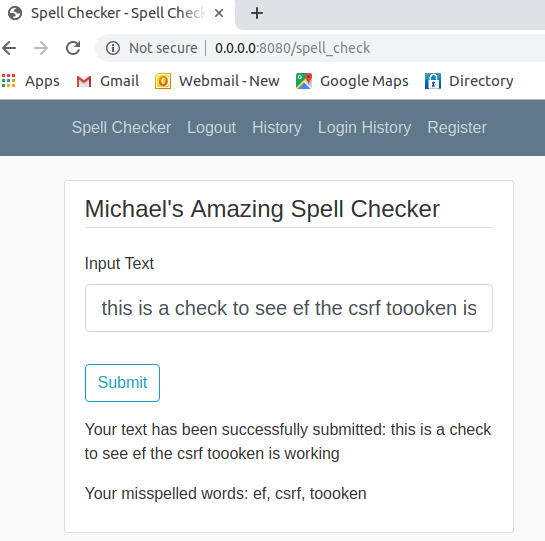


We can manually check that a particular container has the apassword and secretkey secrets by running the cat command below:



Finally we run docker stats to check that our CPU and memory is indeed limited based on the spec in our docker-compose.yml file.



One last check with the app on the browser itself shows that logins and spellcheck posts work to ensure that our CSRF token is being used properly with a secretkey in docker secrets: 

**DOCKER CONTENT TRUST**

One of the benefits of docker is the simplicity of building a containerized solution by simply creating a dockerfile. The dockerfile provides instructions for the build but the actual images for the pre-requisites (i.e. Ubuntu) are pulled from public or private registries. In the case of Ubuntu, we pulled this image from a public registry into our local registry and our web app (app.py) we pulled locally. If we wanted others to use app.py we would have to push to a public registry. How can one be certain that they are getting the image that I created and not some malicious version? This is issue that docker content trust addresses and does so by signing the images with trust keys (key sets).