EE379K Enterprise Network Security Lab 2b Report

Student: Brian Cheung bc32427
Professor: Mohit Tiwari
TA: Antonio Espinoza
Department of Electrical & Computer Engineering
The University of Texas at Austin

October 5, 2019

Part 3 - Orchestration

3a - Orchestration with Kubernetes

Docker applications

Questions:

1. What IP address and port does the web-service use to connect to the SQL DB? Explain what you see on the homepage `http://localhost:8000`.

   The port that the web-service uses to connect to the SQL DB is port 3306. As shown in Figure 1, the web-server is serving at `http://localhost:8000`.

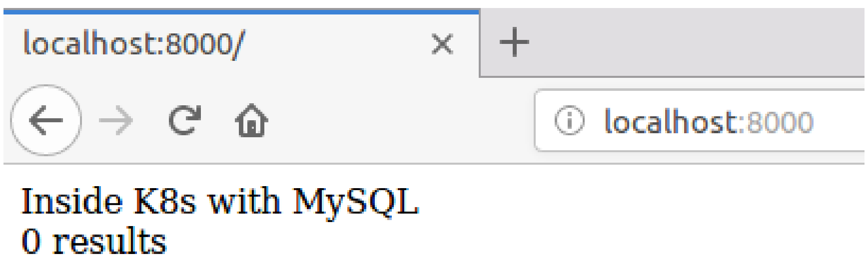


Figure 1: Screenshot of the web-server serving at http://localhost:8000.

2. Do necessary changes so that the web-server now serves at localhost:9000. Explain the change and give screenshots.

   In order to change the port that the web-server is serving at, the host port specified in the `docker-compose.yml` must be changed from 8000 to 9000 as shown in the code snippets below.

   The original specifications:

```
...
  website:
  container_name: php72
  build:
    context: ./
  ports:
    - 8000:80
```

were changed to:

```
 ...
  website:
  container_name: php72
  build:
    context: ./
  ports:
    - 9000:80
```

As shown in Figure 2, the web-server now serves at `http://localhost:9000`.

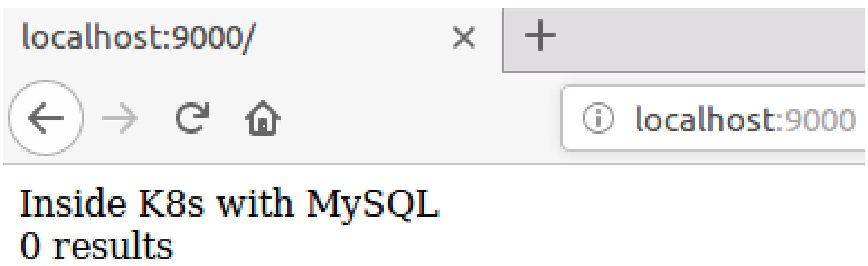


Figure 2: Screenshot of the web-server serving at http://localhost:9000.

Kubernetes

The first step was to tag and push the web-service image with the following commands:

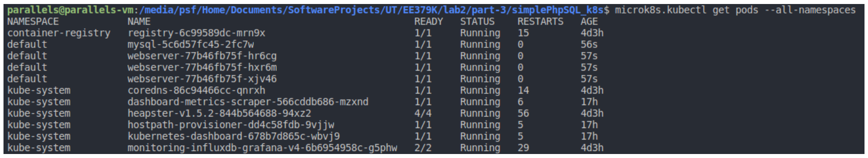
```
docker tag simplephpsqlk8s_website localhost:32000/simplephpsql_k8s_website:k8s
docker push localhost:32000/simplephpsql_k8s_website
```

To run the web-application in kubernetes:

```
microk8s.kubectl apply -f webserver.yaml
microk8s.kubectl apply -f webserver-svc.yaml
microk8s.kubectl apply -f mysql.yaml
microk8s.kubectl apply -f mysql-svc.yaml
```

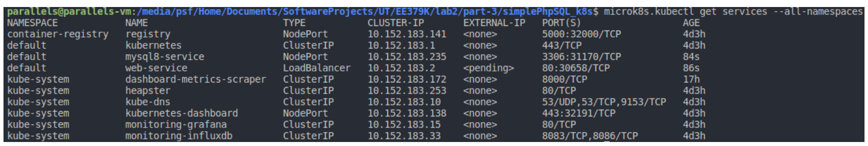
The following commands display information about the pods and services:

```
$ microk8s.kubectl get pods --all-namespaces
$ microk8s.kubectl get services --all-namespaces
```



```
parallels@parallels-vm:/media/psf/Home/Documents/SoftwareProjects/UT/EE379K/lab2/part-3/simplePhpSQL_k8s$ microk8s.kubectl get pods --all-namespaces
NAMESPACE            NAME                                             READY   STATUS    RESTARTS   AGE
container-registry   registry-6c99589dc-mrn9x                         1/1     Running   15         4d3h
default              mysql-5c6d57fc45-2fc7w                           1/1     Running   0          56s
default              webserver-77b46fb75f-hr6cg                       1/1     Running   0          57s
default              webserver-77b46fb75f-hxr6m                       1/1     Running   0          57s
default              webserver-77b46fb75f-xjv46                       1/1     Running   0          57s
kube-system          coredns-86c94466cc-qnrxh                         1/1     Running   14         4d3h
kube-system          dashboard-metrics-scraper-566cddb686-mzxnd       1/1     Running   6          17h
kube-system          heapster-v1.5.2-844b564688-94xz2                 4/4     Running   56         4d3h
kube-system          hostpath-provisioner-dd4c58fdb-9vjjw             1/1     Running   5          17h
kube-system          kubernetes-dashboard-678b7d865c-wbvj9            1/1     Running   5          17h
kube-system          monitoring-influxdb-grafana-v4-6b6954958c-g5phw  2/2     Running   29         4d3h
```

Figure 3: Screenshot of the output from command microk8s.kubectl get pods –all-namespaces



```
parallels@parallels-vm:/media/psf/Home/Documents/SoftwareProjects/UT/EE379K/lab2/part-3/simplePhpSQL_k8s$ microk8s.kubectl get services --all-namespaces
NAMESPACE            NAME                        TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)                  AGE
container-registry   registry                    NodePort       10.152.183.141   <none>        5000:32000/TCP           4d3h
default              kubernetes                  ClusterIP      10.152.183.1     <none>        443/TCP                  4d3h
default              mysql8-service              NodePort       10.152.183.235   <none>        3306:31170/TCP           84s
default              web-service                 LoadBalancer   10.152.183.2     <pending>     80:30658/TCP             86s
kube-system          dashboard-metrics-scraper   ClusterIP      10.152.183.172   <none>        8000/TCP                 17h
kube-system          heapster                    ClusterIP      10.152.183.253   <none>        80/TCP                   4d3h
kube-system          kube-dns                    ClusterIP      10.152.183.10    <none>        53/UDP,53/TCP,9153/TCP   4d3h
kube-system          kubernetes-dashboard        NodePort       10.152.183.138   <none>        443:32191/TCP            4d3h
kube-system          monitoring-grafana          ClusterIP      10.152.183.15    <none>        80/TCP                   4d3h
kube-system          monitoring-influxdb         ClusterIP      10.152.183.33    <none>        8083/TCP,8086/TCP        4d3h
```

Figure 4: Screenshot of the output from command microk8s.kubectl get services –all-namespaces

The different namespaces are specified in the NAMESPACE column in the Figures above. `kube-system` refers to the namespace created by the `Kubernetes` system and includes pods/services like the dashboard. `default` is the default namespace for objects with no other namespace. [1]

The number of instances of each application to be deployed is specified in the `yaml` file in the `replicas` field under `spec`. The original specifications of the `webserver.yaml` file:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver
  labels:
    app: apache
spec:
  replicas: 3
...
```

were changed to:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver
  labels:
```

```
    app: apache
spec:
  replicas: 2
...
```

in order to deploy 2 instances of the webserver application.
The following command:

```
$ microk8s.kubectl -n kube-system edit service kubernetes-dashboard
```

opens the dashboard service script to edit the type to NodePort and find out the exposed port number of the dashboard. The exposed port number is 32191 as specified on line 5:

```
1  spec:
2  clusterIP: 10.152.183.138
3  externalTrafficPolicy: Cluster
4  ports:
5  - nodePort: 32191
6    port: 443
7    protocol: TCP
8    targetPort: 8443
9  selector:
10   k8s-app: kubernetes-dashboard
11 sessionAffinity: None
12 type: NodePort
```

In order to open the dashboard, the secret token must be inputted to login. Once logged in, the dashboard shows all of the pods in the `default` namespace as shown in Figure 5. When viewing other namespaces, no pods show up as the `user-sa` service account does not have access to other namespaces.

In comparison to the output of running `microk8s.kubectl get pods --all-namespaces` as shown in Figure 3, not all of the pods are shown on the dashboard because the namespace of the service account is `default` as specified on line 9 of the `sa-role-bind.yaml` file:

```
1 kind: RoleBinding
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: sa-rolebinding
5   namespace: default
```

```yaml
subjects:
- kind: ServiceAccount
  name: user-sa
  namespace: default
roleRef:
  kind: Role
  name: user-role
  apiGroup: rbac.authorization.k8s.io
```

Only the pods with the default namespace are shown on the dashboard, since the service account only has access to the default namespace. [2]
First, create a service account named `kube-system-sa` for the namespace `kube-system`:

```
$ microk8s.kubectl create serviceaccount kube-system-sa -n kube-system
```

The next step is to create a role with permissions to get, list, create, update and delete. These specificationsare defined in `kube-system-role.yaml`. Then create the role with the following command:

```
$ microk8s.kubectl apply -f kube-system-role.yaml
```

Then to bind the service account and role, a role bind yaml file must specify the service account and role. The `kube-system-sa-role-bind.yaml` file specifies to bind the `kube-system-sa` service account to the `kube-system-role` role. Simply execute the following command to bind:

```
$ microk8s.kubectl apply -f kube-system-sa-role-bind.yaml
```

Next, copy the secret token to login to the dashboard. To check if the service account was succesfully created:

```
$ microk8s.kubectl get serviceaccounts -n kube-system
```

To get the token name for the service account:

```
$ microk8s.kubectl get secret -n kube-system
```

Then get the token by specifying the token name:

```
$ microk8s.kubectl describe secret kube-system-sa-token-rtv8t -n kube-system
```

After logging in with the secret from the service account `kube-system-sa`, the dashboard displays the pods from the `kube-system` namespace as shown in Figure 6 below. When viewing the `default` namespace, no pods show up as the `kube-system-sa` service account does not have access to the `default` namespace.

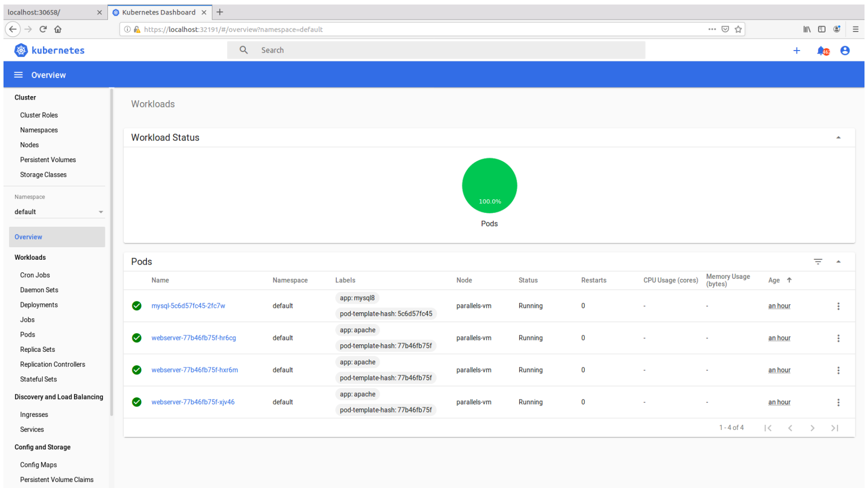


Figure 5: Screenshot of the dashboard viewing the pods in the default namespace with the user-sa service account

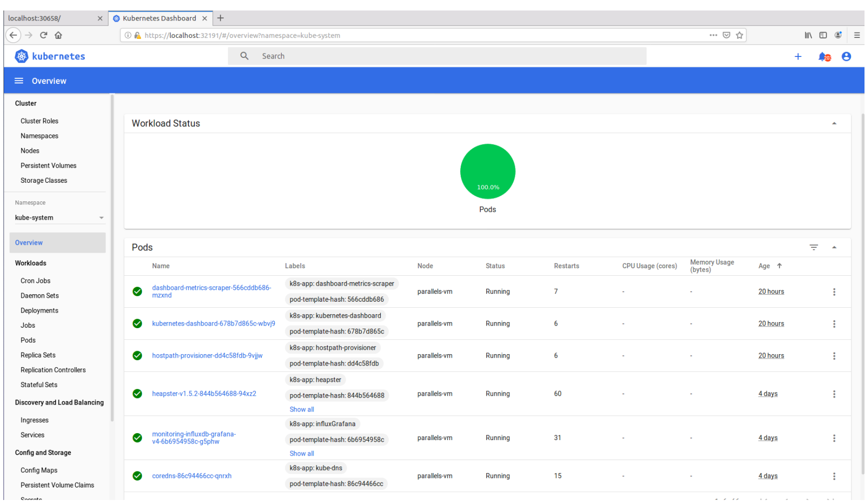


Figure 6: Screenshot of the dashboard viewing the pods in the kube-system namespace with the kube-system-sa service account

3b - Creating a kubernetes cluster for DVWA

The objective of this section was to split up the DVWA to run the MySQL DB and Webserver each in their own containers. The source code for this section is in the `docker-vulnerable-dvwa` directory (part-3/part-b/docker-vulnerable-dvwa) and `dvwa-k8s` directory (part-3/part-b/dvwa-k8s).

Splitting DVWA with Dockerfiles and Docker Compose

This section uses the `docker-vulnerable-dvwa` directory. The first step was to split up the Dockerfile in the DVWA repo into two separate Dockerfiles (one for the MySQL DB and the other for the Apache Webserver). Next, the `docker-compose.yml` (part-3/part-b/docker-vulnerable-dvwa/docker-compose.yml) file creates and runs the containers using the images created by the Dockerfiles. The `docker-compose.yml` also specifies the ports to expose along with the environment variables needed for the MySQL DB.
The following commands create the images and run the containers:

```
$ docker-compose build  # creates the images
$ docker-compose up     # runs the containers
```

Figure 7 shows the DVWA successfully connecting to the MySQL DB.

Split and Deploy DVWA into a Kubernetes cluster

This section uses the `dvwa-k8s` directory. The next step was to deploy the DVWA into a Kubernetes cluster. This is done by creating .yaml files (part-3/part-b/dvwa-k8s/*.yaml)that defines each pod and service. For the DVWA, a pod and a service was created for each deployment of the MySQL DB and Apache Webserver. Follow the `README.md` (part-3/part-b/dvwa-k8s/README.md) for instructions on how to deploy the Kubernetes cluster. The Kubernetes cluster successfully deploys, but the DVWA fails to connect to the MySQL DB.

Fork Bomb Attack in a Kubernetes Pod

The objective of the following experiment is to crash a Kubernetes pod with a fork bomb attack in order to demonstrate the power of splitting up deployments into a Kubernetes cluster.

A fork bomb attack is a form os a denial of service (DOS) attack that depletes system resources in order to slow down and ultimately crash the victim's system. [3] It can be implemented with a simple script that runs

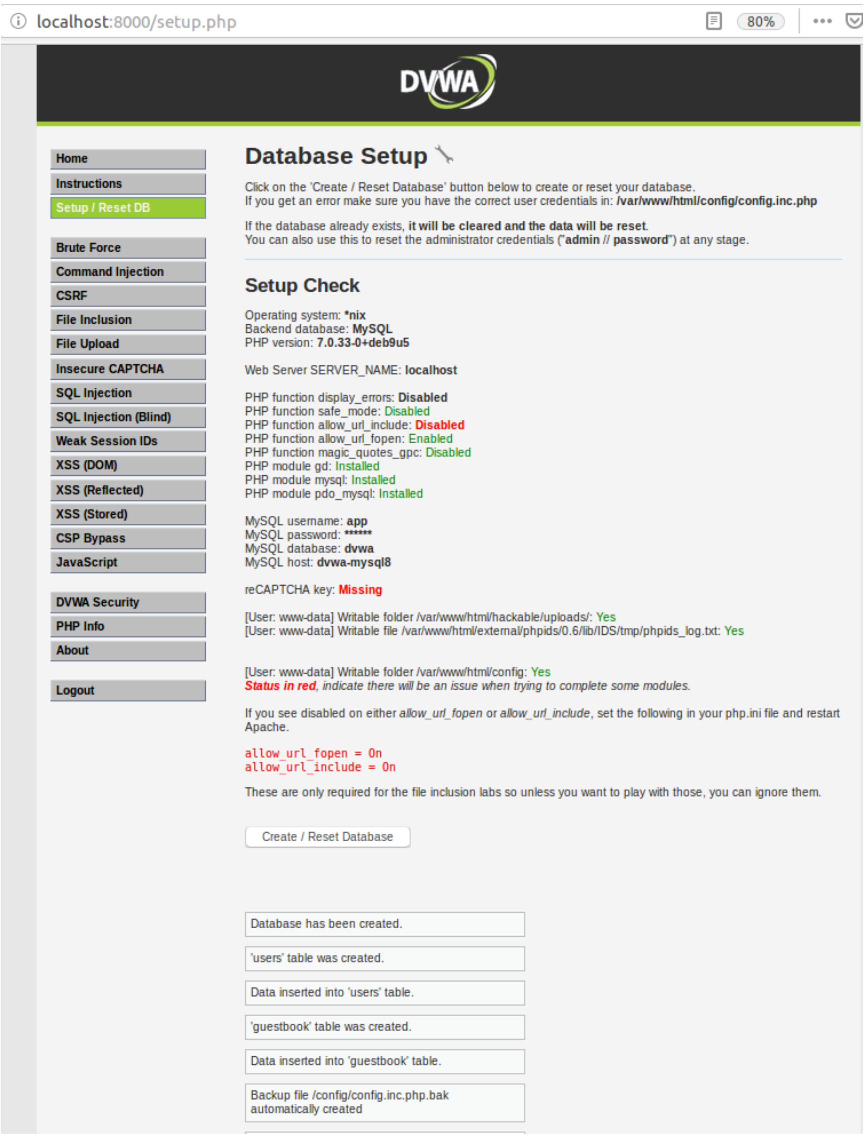


Figure 7: S Screenshot of the DVWA successfully connecting to and creating a MySQL DB

in the background and calls itself twice over and over without any way of stopping the script. The following line is a fork bomb:

```
bomb(){ bomb|bomb& };bomb
```

Since the DVWA could not connect to the MySQL DB in the Kubernetes cluster, there was no way to test the fork bomb to observe what happens in the Kubernetes pod. Instead, the fork bomb was tested with the deployment of DVWA using the docker-compose.yml file. This crashed the entire virtual machine that the docker container was running on because the resource limits weren't set in the docker-compose.yml.
Hypothesis:

If the fork bomb was executed in a single DVWA Kubernetes pod, the pod would have crashed, but Kubernetes would automatically restart the crashed pod. (if configured to automatically restart) Then once the pod is ready, clients would be able to reconnect to the DVWA. However, if there were multiple DVWA Kubernetes pods, clients would be able to connect to the DVWA as long as there is at least one running pod that can handle the traffic. The load balancer would just distribute the traffic to the rest of the pods that are up and running while the crashed pod restarts.

This experiment demonstrated how DOS attacks like the fork bomb attack can be mitigated or even prevented. This is extremely important in production where services need to be running 24/7. Any crash or down time could be very costly to a company.

Conclusion

I really enjoyed learning about the concepts of this lab. It was my favorite lab so far because all the concepts are very applicable to software engineering in the industry. I also never knew how load balancing and distributing traffic works, so it was cool to finally get the chance to explore how software systems handle high volumes traffic. However, part 3b was very vague and did not really give much background. It would have been helpful to know what dependencies are need in each of the containers because I spent a lot of time trying to figure out how DVWA works rather than learning about how Docker, Docker Compose files, and Kubernetes works, which is the main objective of the lab. This lab also took longer than expected. I would say I spent over 30 hours just trying to figure things out and read about the topics.

References

[1] "Namespaces."

[2] "Using rbac authorization."

[3] "Fork bomb."

[4] "Docker documentation."

[5] "Kubernetes documentation."