Assignment 3: Raj Shah

Part 1: Securing Secrets:

To enhance security, the first step was to review the settings.py file and identify the presence of hardcoded secrets. To address this, the hardcoded secret key was removed and stored in an environment variable.

Further examination of the provided files revealed that the django-deploy.yaml and dp-deployment.yaml files also contained hardcoded secrets, which posed a security risk. To address this issue Kubernetes secrets needed to be used. So a secretkey.yaml file was created using the same format as the django-admin-pass-secret.yaml file. This file would store all sensitive information and secrets in base64 format and replace all the sensitive information with the secrets. The command “echo <key> | base64” was used to achieve this.

Once the secretkey.yaml file was created, it was applied by running the command “kubectl apply -f secretkey.yaml”. The application of the file was confirmed by running the command “kubectl get secrets".

- Storing sensitive information, such as passwords, API keys, and tokens, in a public Git repository increases the risk of compromise. Compliance standards may also require that sensitive data is stored securely and not accessible to unauthorized users. Additionally, access control is difficult to manage in Git repositories, and accidentally committing sensitive information can be tracked and stored in the repository's history.

To implement the changes in the django-deploy.yaml and dp-deployment.yaml files, all hardcoded secrets were removed and replaced with secrets from the secretkey.yaml file using (secretKeyRef).

To ensure that the changes were successfully applied, the command "kubectl delete pods <pod name>" was ran. Finally, the command “kubectl exec -it <pod name> /bin/sh” was used to confirm that the plaintext secrets had been replaced with the encrypted secrets from the secretkey.yaml file.

Part 2: Applying Migrations:

To run the migrations, I started off by creating a migrationjob.yaml file that contains all of the necessary variables that need to be migrated into the database. Within the yaml file I added the commands “python3, manage.py, migrate” to ensure that the Django migration job is triggered by the manage.py file. Once the file was created, I ran the command “kubectl apply -f migrationjob.yaml” which triggered the migration flow.. Then in order to confirm the migration has been applied I ran the commands “kubectl get job” and “kubectl get pods”, which indicated that the container was running and properly applied.

To populate records from one table to another, I created a seedingjob.yaml file that calls the database using the stored credentials within the yaml file. To implement this file and confirm its successful completion, I made necessary modifications to the Docker file, including commenting out the data folder and setup.sql command since they had already been executed and we wanted to minimize any possible duplicates.

I then ran the following commands to ensure that the yaml file had been properly applied and was running: "kubectl apply -f seedingjob.yaml", "kubectl get job", and "kubectl get pods". These commands allowed me to confirm that the yaml file had been implemented correctly and was functioning smoothly.

Part 3: Monitoring with Prometheus:

Part 3.1 – Removing unwanted monitoring:

After reviewing an article, I analyzed the view.py code to ensure that there were no sensitive information exposures. Upon inspection, The use of graphs[pword].inc() in the view.py code incremented a Prometheus metric with a password value. However, this password value was also displayed in the logs during a POST request, which could potentially expose sensitive information to unauthorized individuals. To remediate this issue, I commented out a portion of the code (line 51 - 57). Additionally, I commented out the tracker for the number of login requests (line 17 - 30) to prevent revealing a user's interactions.

Part 3.2 – Removing Reasonable Monitoring:

For this part, I analyzed the view.py file and noticed that the '404 not found' message was being used in multiple parts of the code. I decided to track the number of times this message is returned in the code, so I added the line graphs['error_return_counter'].inc() beneath each line of code where '404 not found' was returned. I identified four specific locations in the code where this message was being returned: lines 107-109, lines 114-116, lines 161-163, and lines 168-170

By adding the graphs['error_return_counter'].inc() line in each of these locations, I can effectively track the number of times the '404 not found' message is returned in the code. Tracking 404 not found errors can be useful for security reasons because it can help identify potential security vulnerabilities in the code. When a user requests a resource that does not exist on the server, the server returns a 404 not found response. However, if an attacker is probing the server for sensitive information or trying to exploit a vulnerability, they may intentionally request resources that do not exist in order to elicit a 404 not found response. By tracking the frequency of 404 not found errors, developers can identify whether there is an unusual level of such requests, which may indicate an attack or attempted exploit.

Part 3.3 – Adding Prometheus:

1. Adding Prometheus was pretty simple, I started out by downloading helm by running the commands:

```
curl -fsSL -o get_helm.sh h
https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```

2. I then installed Prometheus by using previously installed helm by running the commands:

```
helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
helm install prometheus-community/prometheus --generate-name
```

3. To ensure Prometheus was running properly I ran the following commands:

```
kubectl get services
kubectl get pods
```

4. Then in order to see all the running services and to ensure Prometheus service was up I ran the command:

```
minikube service --all
```

5. Then in order to find the proper Prometheus server to point to its port server I ran the command:

```
kubectl expose service prometheus-19862113000-server --type=NodePort --
target-port=9090 --name=prometheus-server-np
```

6. Doing this opened up a webpage from the running prometheus server. Now I had to play around with the configmap so that the server would be able to read/use our data from our website. In order to do this I ran the following commands:

```
kubectl get configmap
kubectl edit configmap prometheus-19862113000-server
```

7. I then added the following code for the server to monitor the giftcardsite below (line 25 - 29) within the prometheus-server.yaml.

   After, copied the configfile which I made edits to a yaml file called prometheus-server.yaml by running the commands:

```
kubectl get configmap prometheus-19862113000-server -o yaml > prometheus-
server.yaml
```

8. After I checked to make sure that the server was pointing to a port (80) and was running properly by running the command:

```
minikube service list prometheus
kubectl delete <pod>
kubectl get pods
```

GitHub Link:

https://github.com/Raj-NYU/AppSec3/blob/main/Part%201/secrets.md