CENG 489 - Computer Security
Spring 2024
HONEYPOT PROJECT REPORT

Gürhan İlhan Adıgüzel
`e244802@metu.edu.tr`

Anıl İçen
`e244848@metu.edu.tr`

Murat Kaş
`e238058@metu.edu.tr`

May 10, 2024

Group A
GitHub Repository Link
https://github.com/gurhanadiguzel/Honeypot-Project

1 Introduction

This report documents the Honeypot Project that is dedicated to simulating a dynamic network environment in an attack scenario.

2 Environment Setup

2.1 Flask

2.1.1 What is Flask?

Flask is a lightweight and flexible web framework for Python. It provides the necessary tools and features to develop web applications and APIs efficiently. Known for its simplicity and modularity, Flask is widely used for building various types of web services.

2.1.2 Why do we use Flask in this project?

In the Honeypot Project, Flask serves as the backbone of the backend infrastructure. Here's why Flask is chosen as the framework of choice:

- **Ease of Development:** Flask offers a straightforward and intuitive development experience, allowing developers to focus on implementing the core functionalities of the project without unnecessary complexity.
- **Modularity:** Flask promotes a modular design, enabling the separation of concerns within the project. Each Flask file represents a distinct component or service, facilitating scalability and maintainability.
- **Containerization:** Flask applications can be containerized using Docker, providing isolation and reproducibility across different deployments. Docker ensures consistency and simplifies management within the Honeypot environment.
- **Dynamic Behavior:** Flask applications implement dynamic behavior to respond to events within the Honeypot environment. This includes handling incoming requests, monitoring database container status, and triggering port switches in response to attacks.

2.1.3 Implementing Dynamic Port Switching in Flask

Each Flask application running on different ports facilitates the implementation of dynamic port switching. When an attack is detected, Flask applications can dynamically switch network configuration to ensure security. This is achieved through custom logic within each Flask application, monitoring the status of the database container, and updating configurations accordingly.

2.1.4 Conclusion

The use of Flask in the Honeypot Project's backend infrastructure provides the necessary flexibility, modularity, and scalability to simulate a complex network environment effectively. With 14 Flask applications running on different ports, the project can accurately mimic real-world scenarios while implementing custom logic, such as dynamic port switching, to enhance security measures.

2.2 Docker

2.2.1 Introduction to Docker

Docker is a popular containerization platform that enables developers to package applications into containers—standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment. Containers are lightweight, portable, and provide a consistent runtime environment, making them ideal for simplifying the development, testing, and deployment processes.

2.2.2 Project Overview

In our project, we utilized Docker to containerize a multi-component application composed of several Flask microservices. Our architecture comprised 14 containers, including:

- 6 primary containers: These served as the main runtime environments for our Flask applications.
- 6 replica containers: Each served as a switchable container to a corresponding primary container, enhancing reliability and security.
- 1 gateway container: This acted as the entry point for all incoming requests, routing them to the appropriate services.
- 1 database container: This container managed all data storage and retrieval operations, effectively acting as a port changing station within our architecture.

2.2.3 Communication Flow

The communication between containers was orchestrated using Flask, starting from the gateway container. The database container played a crucial role in facilitating dynamic port management and service discovery within the containerized environment.

2.2.4 Encountered Challenges and Solutions

Networking Challenges Post-Containerization **Problem:** Initially, when the Flask applications were containerized, they were configured to run on localhost. However, this setup failed in Docker's isolated network environment where containers communicate via internal IPs.
Solution: We resolved this by using host.docker.internal, which is a special DNS name used to refer to the host machine's network. This allowed the containers to communicate with services running on the host or in other containers reliably.

Deployment Issues with Kubernetes **Problem:** Locally deploying our Docker containers on Kubernetes posed significant challenges. Kubernetes, a powerful orchestration tool for managing containerized applications, requires configurations and networking adjustments that were not immediately compatible with our local setup.
Solution: To overcome this, we uploaded our Docker containers to Docker Hub, a public container registry. From there, Kubernetes could efficiently pull the containers and manage their deployment. This not only simplified the deployment process but also ensured that our containers were stored and managed centrally, enhancing our deployment workflow.

2.2.5 Conclusions and Recommendations

The transition to a fully containerized environment using Docker presented initial networking and deployment challenges, primarily related to port management and Kubernetes integration. By addressing these issues through strategic DNS usage and leveraging Docker Hub for container management, we successfully streamlined our deployment process.

2.3 Kubernetes

2.3.1 Introduction to Kubernetes

In our project, we utilized Kubernetes for container orchestration, optimizing the deployment and management of our application's components.

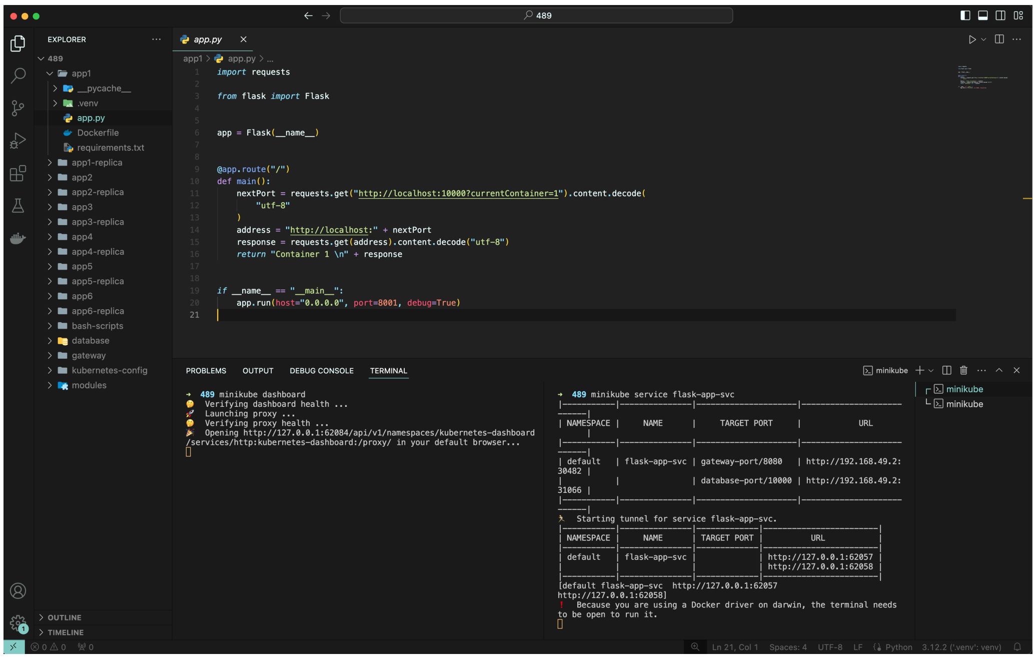
2.3.2 Project Overview

Our strategy includes breaking down the configuration tasks into two YAML files: one dedicated to deployment specifics and the other to service definitions. In the deployment YAML file, we carefully decided where to put each container and set up things like where to find them (in docker.io) and which ports they should use. Similarly, we created the service YAML file to outline the specifics of our backend service, such as the port it will operate on and the type of service it will be (LoadBalancer, NodePort, etc.).

2.3.3 Deployment Methodology

By utilizing 'kubectl create' commands, we integrated these configurations into our Kubernetes environment. This methodology not only simplified the deployment process but also improved the reliability and scalability of our application infrastructure.

3 Screenshots

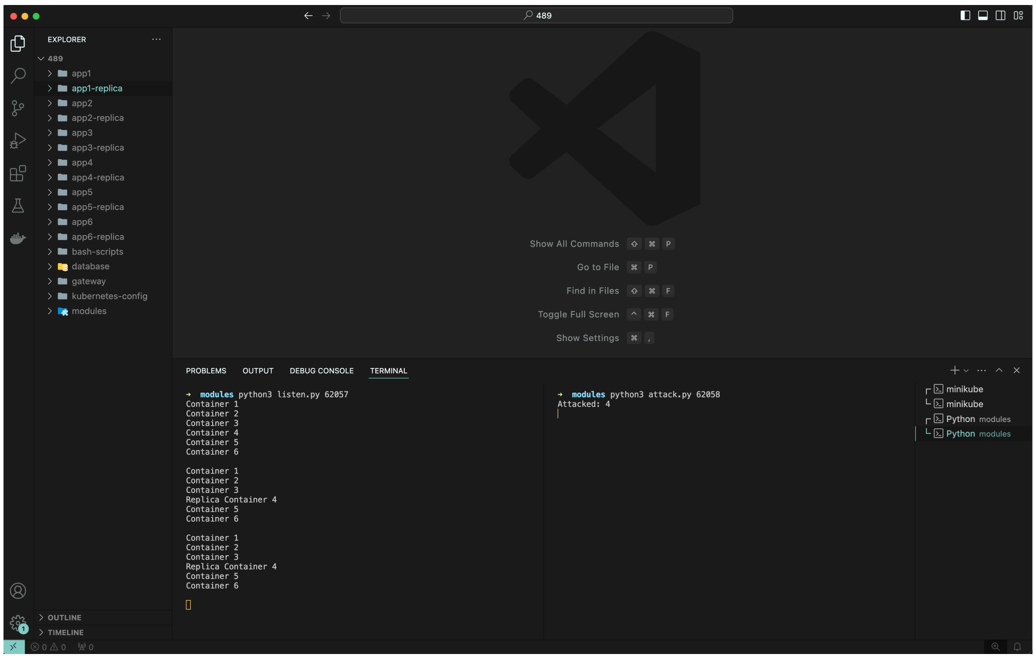


```python
import requests

from flask import Flask


app = Flask(__name__)


@app.route("/")
def main():
    nextPort = requests.get("http://localhost:10000?currentContainer=1").content.decode(
        "utf-8"
    )
    address = "http://localhost:" + nextPort
    response = requests.get(address).content.decode("utf-8")
    return "Container 1 \n" + response


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8001, debug=True)
```

```
➜  489 minikube dashboard
🤔  Verifying dashboard health ...
🚀  Launching proxy ...
🤔  Verifying proxy health ...
🎉  Opening http://127.0.0.1:62084/api/v1/namespaces/kubernetes-dashboard
/services/http:kubernetes-dashboard:/proxy/ in your default browser...
```

```
➜  489 minikube service flask-app-svc
|-----------|---------------|---------------------|---------------------
-------|
| NAMESPACE |     NAME      |     TARGET PORT     |             URL
       |
|-----------|---------------|---------------------|---------------------
-------|
| default   | flask-app-svc | gateway-port/8080   | http://192.168.49.2:
30482 |
|           |               | database-port/10000 | http://192.168.49.2:
31066 |
|-----------|---------------|---------------------|---------------------
-------|
🏃  Starting tunnel for service flask-app-svc.
|-----------|---------------|-------------|------------------------|
| NAMESPACE |     NAME      | TARGET PORT |          URL           |
|-----------|---------------|-------------|------------------------|
| default   | flask-app-svc |             | http://127.0.0.1:62057 |
|           |               |             | http://127.0.0.1:62058 |
|-----------|---------------|-------------|------------------------|
[default flask-app-svc  http://127.0.0.1:62057
http://127.0.0.1:62058]
❗  Because you are using a Docker driver on darwin, the terminal needs
to be open to run it.
```

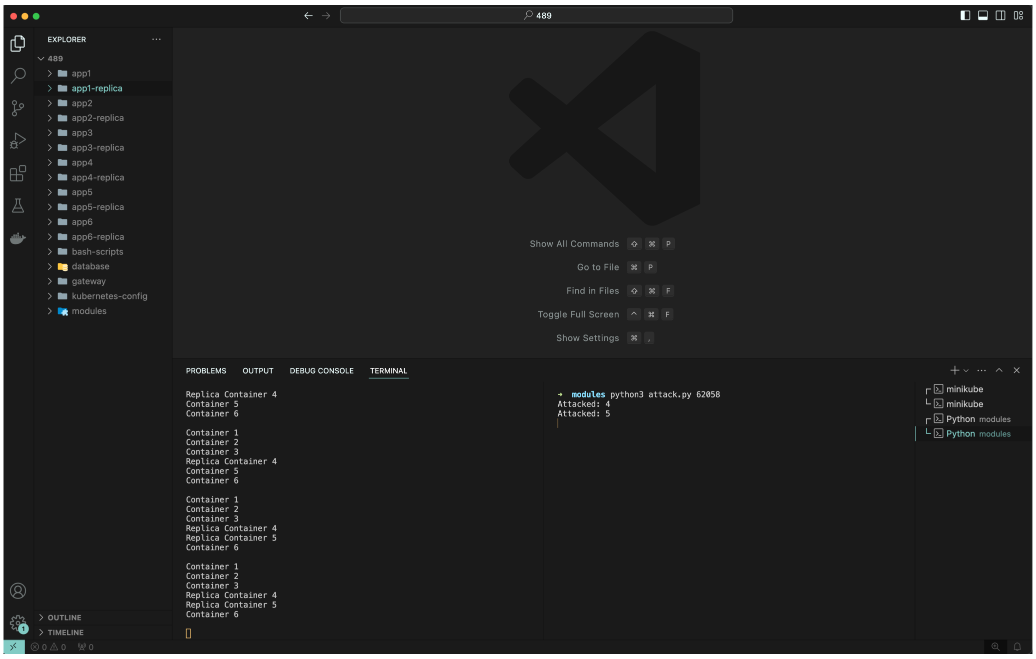
Figure 1: Minikube Dashboard Start



```
➜  modules python3 listen.py 62057
Container 1
Container 2
Container 3
Container 4
Container 5
Container 6

Container 1
Container 2
Container 3
Replica Container 4
Container 5
Container 6

Container 1
Container 2
Container 3
Replica Container 4
Container 5
Container 6
```

```
➜  modules python3 attack.py 62058
Attacked: 4
```

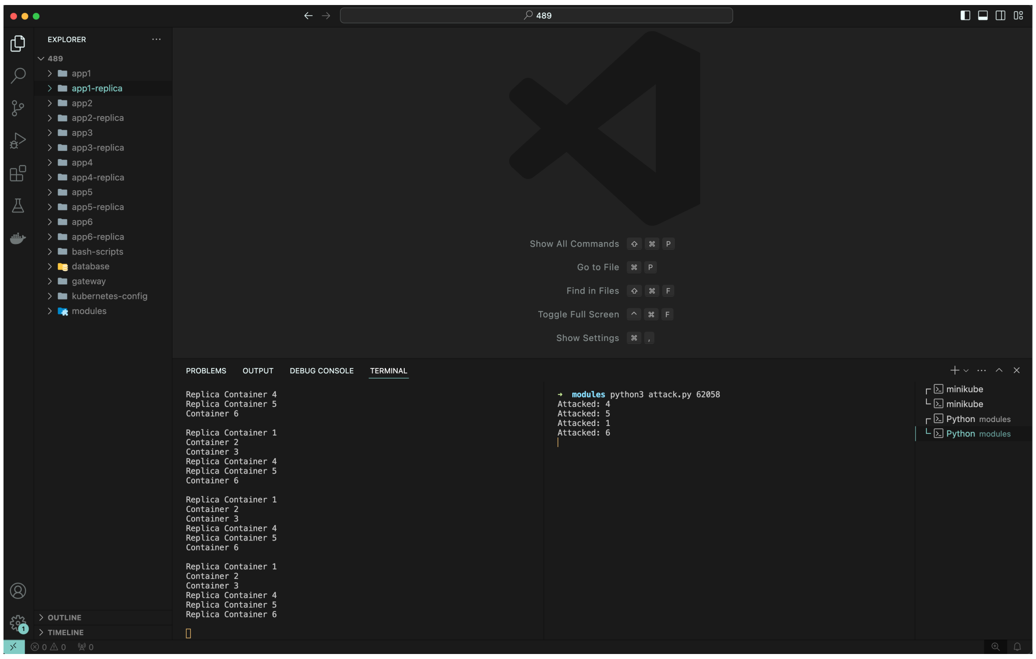
Figure 2: Attack Scenario 1



```
Replica Container 4
Container 5
Container 6

Container 1
Container 2
Container 3
Replica Container 4
Container 5
Container 6

Container 1
Container 2
Container 3
Replica Container 4
Replica Container 5
Container 6

Container 1
Container 2
Container 3
Replica Container 4
Replica Container 5
Container 6
```

```
➜  modules python3 attack.py 62058
Attacked: 4
Attacked: 5
```

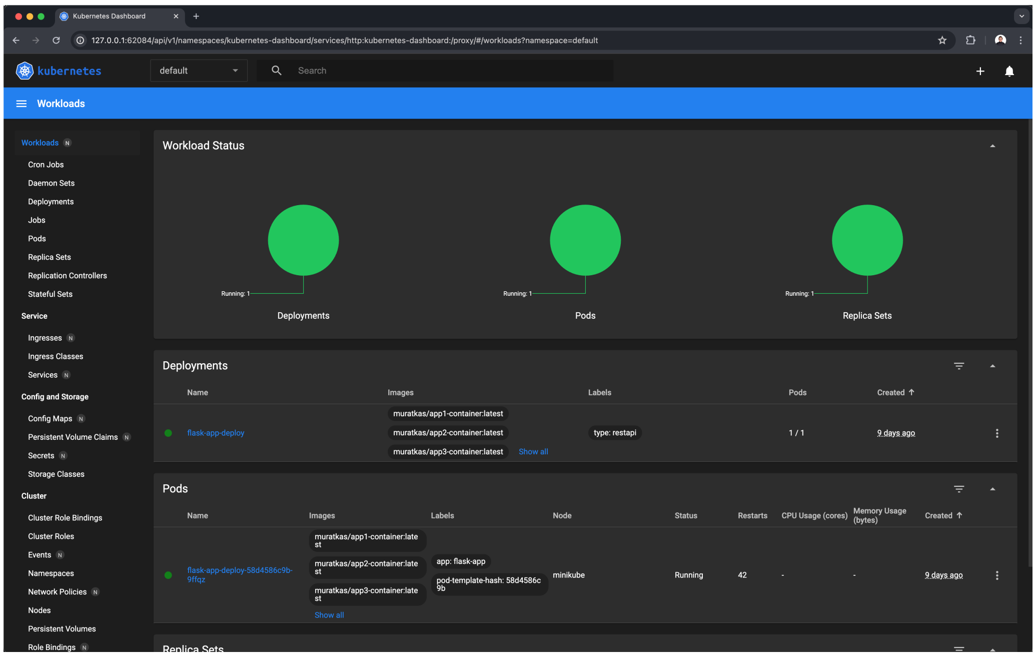
Figure 3: Attack Scenario 2



```
Replica Container 4
Replica Container 5
Container 6

Replica Container 1
Container 2
Container 3
Replica Container 4
Replica Container 5
Container 6

Replica Container 1
Container 2
Container 3
Replica Container 4
Replica Container 5
Container 6

Replica Container 1
Container 2
Container 3
Replica Container 4
Replica Container 5
Replica Container 6
```

```
➜  modules python3 attack.py 62058
Attacked: 4
Attacked: 5
Attacked: 1
Attacked: 6
```

Figure 4: Attack Scenario 3



127.0.0.1:62084/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/#/workloads?namespace=default

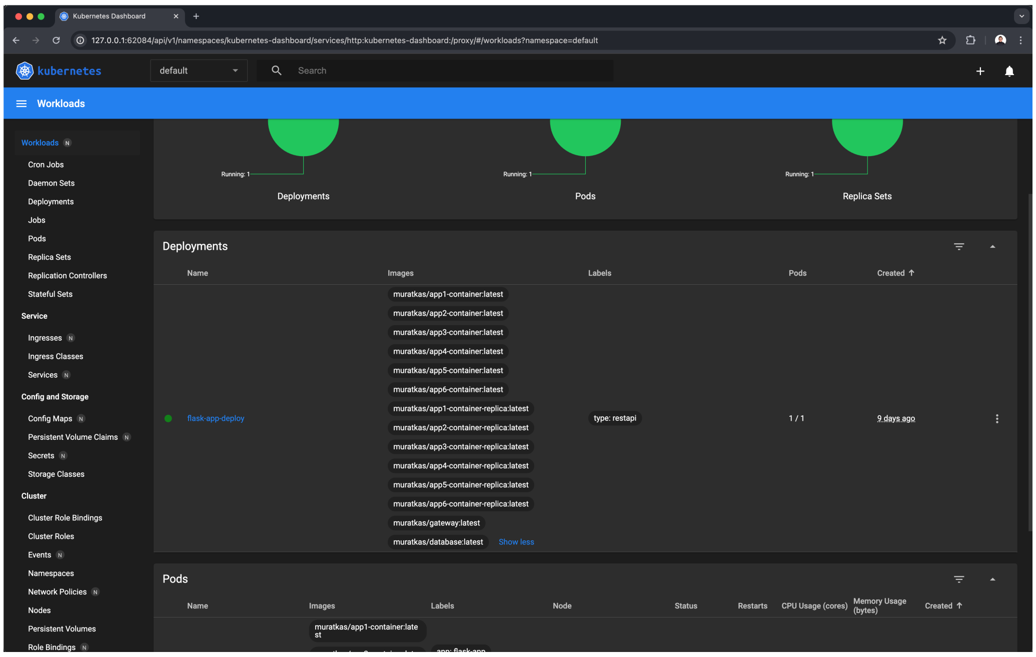
Deployments

	Name	Images	Labels	Pods	Created ↑
●	flask-app-deploy	muratkas/app1-container:latest muratkas/app2-container:latest muratkas/app3-container:latest Show all	type: restapi	1 / 1	9 days ago

Pods

	Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
●	flask-app-deploy-58d4586c9b-9ffqz	muratkas/app1-container:latest muratkas/app2-container:latest muratkas/app3-container:latest Show all	app: flask-app pod-template-hash: 58d4586c9b	minikube	Running	42	-	-	9 days ago

Figure 5: Kubernetes Dashboard



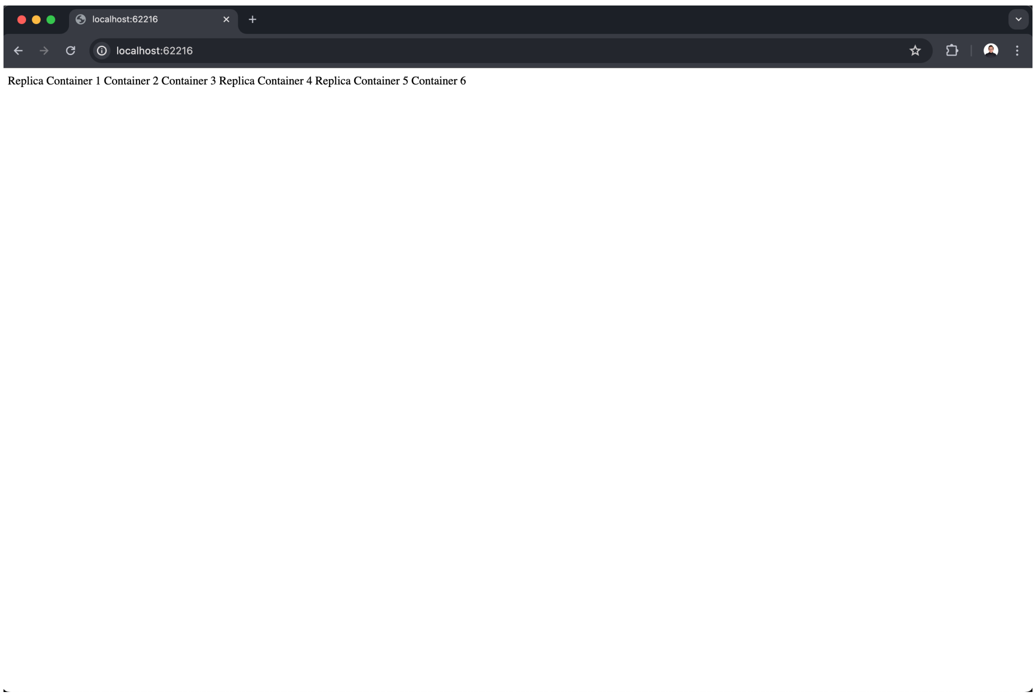
Deployments

Name	Images	Labels	Pods	Created ↑
● flask-app-deploy	muratkas/app1-container:latest muratkas/app2-container:latest muratkas/app3-container:latest muratkas/app4-container:latest muratkas/app5-container:latest muratkas/app6-container:latest muratkas/app1-container-replica:latest muratkas/app2-container-replica:latest muratkas/app3-container-replica:latest muratkas/app4-container-replica:latest muratkas/app5-container-replica:latest muratkas/app6-container-replica:latest muratkas/gateway:latest muratkas/database:latest Show less	type: restapi	1 / 1	9 days ago

Pods

Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
	muratkas/app1-container:latest	app: flask-app						

Figure 6: Pod and Containers Configuration



Replica Container 1 Container 2 Container 3 Replica Container 4 Replica Container 5 Container 6

Figure 7: Website Screen

4 Conclusion

The Honeypot Project aims to create a resilient and adaptive network environment. Our utilization of Flask for backend development provided the necessary flexibility and modularity to implement sophisticated features, while Kubernetes served us while we are creating our container orchestration strategy, ensuring seamless deployment and management of our application's components.

Furthermore, our decision to utilize Minikube helped us to use and test Kubernetes features locally. We implemented a mini SDN (Software-Defined Network) to dynamically change network configurations. By doing that we simulated a dynamic network configuration in an attack scenario.

Docker played a crucial role in ensuring platform independence and seamless deployment across different hosting services. By containerizing our application components, we achieved consistency and reproducibility, simplifying the deployment process and improving overall system reliability.

In conclusion, the integration of Flask, Kubernetes, Minikube, and Docker enabled us to create a robust and adaptive network environment capable of responding to dynamic security challenges.