Security Product and Kubernetes Technical Tasks

This document contains detailed answers and deliverables for three problem statements:
1. Product Requirement and Low-Fidelity Wireframes
2. Kubernetes Security Scan
3. Go, Docker, and Kubernetes Technical Implementation

Problem Statement 1: Product Requirement and Low-Fidelity Wireframes

Background/Task:

A security product requires scanning container images and showing users the findings. Container images contain applications with their dependencies, which may have known vulnerabilities. Users need visibility into vulnerable images, severity, and guidance for remediation.

Product Requirements Document:

Functional Requirements:

- Dashboard overview of total images scanned and vulnerabilities by severity (Critical, High, Medium, Low).
- Search, filter, and sort container images by severity, repository, and last scanned date.
- Image-level vulnerability details: CVE ID, severity, affected package, version, and fix available.
- Highlight critical/high vulnerabilities for prioritization.
- Export findings in JSON/CSV for integration with other systems.
- Scheduled scans and history of past results.
- Role-based access to restrict sensitive vulnerability data.

Non-Functional Requirements:

- Scalability: Support scanning thousands of images.
- Performance: Scan results should be processed and displayed quickly.
- Security: Ensure vulnerability data is stored securely.
- Usability: Simple UI for technical and non-technical users.

Low-Fidelity Wireframes:

- Dashboard Page: KPIs (Total Images, Critical Vulnerabilities, High Vulnerabilities) + Graphs.
- Images List: Table view with Image Name, Repo, Critical, High, Medium, Low, Last Scanned.
- Image Detail View: Vulnerability details with CVE ID, Severity, Package, Fixed Version, Reference Links.

Development Action Items:

- Integrate image scanning tool (e.g., Trivy, Clair, Anchore).
- Design backend services to manage scan jobs and vulnerability database.
- Build APIs to fetch results for frontend consumption.
- Implement frontend UI with dashboard, tables, and detail views.
- Add authentication/authorization for secure access.
- Enable scheduling and notifications (e.g., Slack, email alerts).

Problem Statement 2: Kubernetes Security Scan

Background/Task:

Install a local Kubernetes cluster (Minikube, Kind, K3s, etc.) and use a tool such as Kubescape, Kube-bench, or Kube-hunter to scan the environment. Findings should be exported as JSON.

Example Findings (JSON Format):

```
[ { "id": "1", "resource": "Pod/frontend", "namespace": "default",
"severity": "High", "description": "Container running as root user." }, {
"id": "2", "resource": "Deployment/backend", "namespace": "production",
"severity": "Critical", "description": "No CPU/memory resource limits
defined." }, { "id": "3", "resource": "Service/database", "namespace":
"dev", "severity": "Medium", "description": "Service exposed publicly
without authentication." } ]
```

Problem Statement 3: Go, Docker, and Kubernetes Deployment

Step 1: Go Program

Write a simple GoLang program to display current date & time. Push the code to GitHub, then build and push the Docker image to DockerHub.

```
package main import ( "fmt" "net/http" "time" ) func handler(w
http.ResponseWriter, r *http.Request) { fmt.Fprintf(w, "Current Date &
Time: %s", time.Now().Format(time.RFC1123)) } func main() {
http.HandleFunc("/", handler) http.ListenAndServe(":8080", nil) }
```

Step 2: Kubernetes Deployment

```
apiVersion: apps/v1 kind: Deployment metadata: name: datetime-app spec:
replicas: 2 selector: matchLabels: app: datetime template: metadata:
labels: app: datetime spec: containers: - name: datetime image:
/datetime-app:latest ports: - containerPort: 8080 --- apiVersion: v1
kind: Service metadata: name: datetime-service spec: type: LoadBalancer
selector: app: datetime ports: - port: 80 targetPort: 8080
```

Step 3: Expose the App

Use a LoadBalancer service (as shown above) or Ingress to expose the app publicly. Once deployed on a cloud provider, an external IP will be assigned.