Project Title: Container Image Vulnerability Assessment

Short Project Description:

The system scans images for vulnerabilities, misconfigurations, and embedded secrets, generating actionable reports to secure containerized environments.

Component	Role
Container Image Fetcher	Downloads container images from registries
Vulnerability Scanner	Analyzes packages, libraries inside images
Configuration Checker	Checks Dockerfile, Kubernetes manifests for issues
Secret Detector	Searches for embedded secrets in images
Report Generator	Prepares detailed vulnerability and misconfiguration report

---

Component Details:

1. **Container Image Fetcher**:
    - Pulls images from:
        - Docker Hub
        - Private registries
        - Etc
2. **Vulnerability Scanner**:
    - Scans images using:
        - CVE databases (e.g., Trivy, Clair, etc)
        - OS package vulnerabilities
        - Etc
3. **Configuration Checker**:
    - Looks for:
        - Insecure default settings
        - Unnecessary privileges (e.g., root user)
        - Etc
4. **Secret Detector**:
    - Identifies hardcoded credentials, API keys inside containers.
5. **Report Generator**:
    - Creates comprehensive security reports.

Overall System Flow:

- Input: Container image/tag
- Output: Full vulnerability analysis report
- Focus on **static scanning** without running containers

Internal Functioning of Each Module:

1. Container Image Fetcher

- **How it works:**
  - Connects to registries:
    - DockerHub
    - AWS ECR
    - GCP Container Registry
    - Etc
  - Pulls and saves container images locally using **Docker pull** API commands.

---

2. Vulnerability Scanner

- **How it works:**
  - Unpacks container layers.
  - Scans binaries, libraries, and packages.
  - Matches installed packages to known CVEs using tools like:
    - **Trivy**, **Anchore**, **Clair**, etc.
  - Checks OS base image (e.g., Ubuntu, Alpine, etc) vulnerabilities too.

---

3. Configuration Checker

- **How it works:**
  - Parses Dockerfile from the image (if possible).
  - Checks for insecure patterns:
    - USER root (bad)
    - Unrestricted ports
    - No security profiles (AppArmor/SELinux)
    - Etc
  - Also checks for missing best practices:
    - Missing HEALTHCHECK instructions
    - Exposing unnecessary volumes
    - Etc

---

4. Secret Detector

- **How it works:**
  - Greps through files for high-entropy strings.
  - Detects:
    - Hardcoded AWS credentials
    - SSH private keys

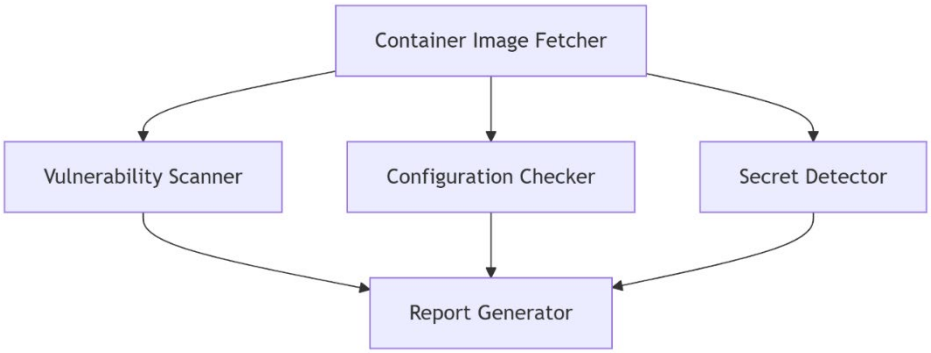
        - OAuth tokens
        - Database passwords
        - Etc
- **Techniques:**
    - Regex patterns
    - Entropy analysis (detect random-looking secrets)
    - Etc

---

5. Report Generator

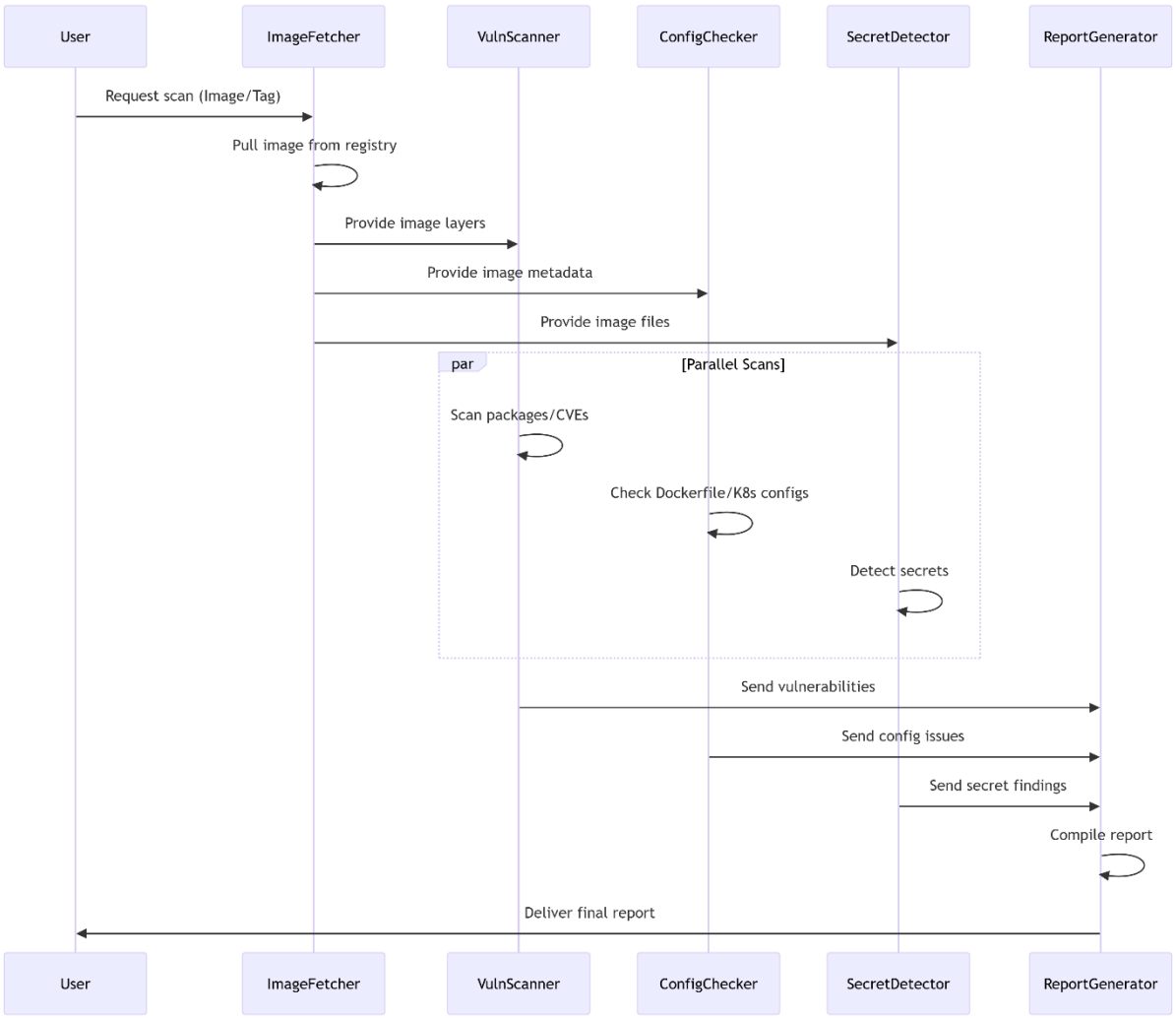
- **How it works:**
    - Summarizes vulnerabilities by:
        - CVSS score
        - CVE ID
        - Package name and version
        - Recommendation (update, remove, rebuild)
    - Adds secret findings separately.

Component Diagram



- The **Container Image Fetcher** retrieves images and distributes data to three parallel components: **Vulnerability Scanner**, **Configuration Checker**, and **Secret Detector**.
- All three analysis modules send their findings to the **Report Generator**, which consolidates the results.

Sequence Diagram



- The **User** initiates a scan by specifying an image/tag.
- The **Image Fetcher** pulls the image and distributes its layers, metadata, and files to the respective scanners.
- Scans for vulnerabilities, configuration issues, and secrets occur **in parallel**.
- Results are aggregated by the **Report Generator**, which produces a final report for the **User**.

Detailed Project Description: Container Image Vulnerability Assessment

A system for static security analysis of container images. The system scans images for vulnerabilities, misconfigurations, and embedded secrets, generating actionable reports to secure containerized environments.

---

1. System Overview

The tool analyzes container images without executing them, focusing on:

- **Vulnerability Detection**: CVEs in OS packages and dependencies.
- **Configuration Audits**: Insecure Dockerfile/Kubernetes settings.
- **Secret Detection**: Hardcoded credentials and API keys.
- **Reporting**: Consolidated findings in developer-friendly formats.

---

2. Component Design & Implementation

2.1 Container Image Fetcher

Functionality:

- Pulls container images from public/private registries.

Implementation Steps (e.g.):

1. **Registry Integration**:
    - Use Docker SDK (Python) or Skopeo CLI to pull images.
    - Support registries (e.g.):
        - **Docker Hub**: Use `docker pull` or API with authentication tokens.
        - **AWS ECR**: Authenticate via AWS CLI (`aws ecr get-login-password`).
        - **GCP Artifact Registry**: Use `gcloud auth configure-docker`.
2. **Authentication**:

    - Handle credentials securely using environment variables or vaults (e.g., HashiCorp Vault, etc).
3. **Image Extraction**:
    - Save images to a local directory (e.g., `/var/lib/images`).
    - Extract image layers using `docker save` or `umoci` for OCI-compliant images.

Output:

- Extracted image layers and metadata (e.g., `manifest.json`).

2.2 Vulnerability Scanner

Functionality:

- Identifies CVEs in OS packages and application dependencies.

Implementation Steps (e.g.):

1. **Tool Integration**:
    - Use **Trivy** (lightweight), **Clair** (enterprise-grade), etc, for scanning.
    - Example Trivy command:
      ```
      trivy image --format json --output vulns.json <image:tag>
      ```
2. **Dependency Analysis**:
    - Scan extracted layers for:
        - OS packages (e.g., `apt`, `apk`, `yum` databases).
        - Language-specific dependencies (`node_modules`, `requirements.txt`).
3. **CVE Database Updates**:
    - Schedule daily updates using `trivy --download-db-only`.

Output:

- JSON report listing CVEs, affected packages, and CVSS scores.

2.3 Configuration Checker

Functionality:

- Audits Dockerfiles and Kubernetes manifests for security best practices.

Implementation Steps (e.g.):

1. **Dockerfile Extraction**:
    - Reconstruct Dockerfile from image history:
      ```
      docker history --no-trunc <image:tag> > dockerfile
      ```
    - Use **Dive** to analyze layer efficiency and commands.
2. **Rule Engine**:
    - Define rules in YAML:
    - Example
      ```yaml
      rules:
        - id: "ROOT_USER"
          description: "Container runs as root user"
          condition: "USER root"
          severity: "High"
      ```
    - Check for:
        - `USER root`, exposed ports, missing `HEALTHCHECK`, privileged mode.
3. **Kubernetes Manifest Checks**:
    - Use **kube-score** or **Checkov** to validate YAML files for security policies.

Output:

- List of configuration issues with remediation advice.

2.4 Secret Detector

Functionality:

- Finds hardcoded secrets (API keys, passwords) in image files.

Implementation Steps (e.g.):

1. **Regex Patterns**:
    - Define patterns for common secrets (AWS keys, JWT tokens):
      `AWS_KEY_PATTERN = r"(?i)AKIA[0-9A-Z]{16}"`
2. **Entropy Analysis**:
    - Detect high-entropy strings (e.g., random tokens, etc) using Shannon entropy.
3. **Directory Exclusions**:
    - Ignore binary files (e.g., `.png`, `.so`) and common paths (`/usr/bin`).

Tools (e.g.):

- **TruffleHog**: Open-source secret scanner.
- **Gitleaks**: For regex-based detection.
- **Etc**.

Output:

- List of detected secrets with file paths and matched patterns.

2.5 Report Generator

Functionality:

- Consolidates findings into structured reports.

Implementation Steps (e.g.):

1. **Data Aggregation**:
    - Merge JSON outputs from Vulnerability Scanner, Configuration Checker, and Secret Detector.
2. **Template Design**:
    - **HTML**: Use Jinja2 templates for interactive tables and filters.
    - **SARIF**: Export for CI/CD integration (e.g., GitHub Code Scanning).
    - **PDF**: Generate via `WeasyPrint` or `pandoc`.
    - **Etc**.

3. **Severity Prioritization**:
    - Sort findings by CVSS score (Critical > High > Medium > Low).

Output:

- Final report with:
    - Vulnerability details (CVE ID, package, fix version).
    - Configuration misconfigurations.
    - Secrets and their locations.
    - Remediation steps (e.g., "Upgrade `openssl` to v3.0.7").

---

3. Technology Stack (e.g.)

- **Image Handling**: Docker SDK, Skopeo, Umoci, etc.
- **Vulnerability Scanning**: Trivy, Clair, etc.
- **Secret Detection**: TruffleHog, custom regex/entropy checks, etc.
- **Configuration Audits**: Dive, kube-score, Checkov, etc.
- **Reporting**: Jinja2, SARIF SDK, WeasyPrint, etc.

---

4. Evaluation & Validation

1. **Accuracy Testing**:
    - Use intentionally vulnerable images (e.g., `vulhub/nginx:unsafe`).
    - Verify detection of known CVEs, misconfigurations, and test secrets.
2. **Performance**:
    - Measure scan time for large images (e.g., 5GB+).
3. **False Positives**:
    - Review secret detection results to refine regex patterns and entropy thresholds.

---

5. Development Roadmap

1. **Phase 1**: Build Image Fetcher and integrate Trivy.
2. **Phase 2**: Implement Configuration Checker and Secret Detector.
3. **Phase 3**: Develop reporting engine with SARIF/HTML support.
4. **Phase 4 (optional)**: Optimize performance and validate on real-world images.

---

6. Challenges & Mitigations (optional)

- **Large Images**: Streamline layer extraction and parallelize scans.
- **Private Registries**: Implement OAuth2/Token-based authentication.
- **Complex Dependencies**: Use language-specific scanners (e.g., `npm audit`, `safety check, etc`).

---

7. Glossary

- **CVE**: Common Vulnerabilities and Exposures
- **CVSS**: Common Vulnerability Scoring System
- **OCI**: Open Container Initiative
- **SARIF**: Static Analysis Results Interchange Format
