FAKULTÄT FÜR INFORMATIK

TECHNISCHE HOCHSCHULE ROSENHEIM

Projektarbeit im Modul Software-Qualitätssicherung

Statische Analyse von Docker-Containern mit Clair

Author: Johann Neuhauser
Datum der Abgabe: 08.07.2019

Zusammenfassung

Die Projektarbeit „Statische Analyse von Docker-Containern mit Clair“ befasst sich, im Rahmen des Master-Moduls „Software-Qualitätssicherung (SQS)“, mit der statischen Analyse von Anwendungscontainern auf bekannte Sicherheitslücken. Sie bewertet dazu das quelloffene Projekt „Clair“ von CoreOS® auf Basis von bestehenden und selbst erstellten Docker-Containern. Die Evaluierung von Clair findet dabei zunächst auf einem lokalen Entwicklungs-Rechner statt. Im Anschluss wird die statische Analyse mit „Clair“, mittels einer selbst aufgesetzten Testumgebung von GitLab, in eine GitLab CI/CD Pipeline integriert, um Container-Images nach dem automatisierten Bau und vor der Veröffentlichung in einer Docker Registry auf Sicherheitslücken zu untersuchen. Schlägt die Untersuchung aufgrund von Sicherheitslücken, welche nicht auf einer Whitelist stehen oder über einer definierten Gefährdungsstufe liegen, fehl, so soll der Container nicht für die weitere Verwendung in der Docker Registry veröffentlicht werden. Abschließend findet eine kleine Gegenüberstellung mit einem anderen Open Source Tool zur statische Container-Analyse statt.

Inhaltsverzeichnis

Zusammenfassung **ii**

Listings **iv**

Abkürzungsverzeichnis **v**

1 Einleitung **1**
1.1 Ausgangssituation 1
1.2 Zielsetzung 2

2 Scannen von Container-Images mit Clair **3**
2.1 Installation von Clair mit PostgreSQL und Clair Scanner 3
2.2 Scannen von lokalen Container-Images 4

3 Integration von Clair in GitLab CI/CD **7**
3.1 GitLab EE Omnibus Installation 7
3.2 GitLab Runner Installation 9
3.3 Installation von Clair mit PostgreSQL und Clair Scanner 11
3.4 Analyse eines Docker-Containers in GitLab CI/CD 11

4 Fazit **14**

Listings

2.1 Installation von Clair, Postgres-Datenbank und Clair Scanner 4
2.2 Scannen von lokalen Container-Image 5
2.3 Beispiel einer Whitelist für Clair Scanner 5
2.4 Scannen von lokalen Container-Images mit Whitelist 6

3.1 Benötigte Pakete in Ubuntu 18.04 installieren 7
3.2 Installation von GitLab EE auf Ubuntu 18.04 8
3.3 Aktivieren von GitLab Container Registry in GitLab EE 8
3.4 Installation von Docker auf Ubuntu 18.04 9
3.5 Installation von GitLab Runner auf Ubuntu 18.04 10
3.6 Registrieren von GitLab Runner in GitLab EE 10
3.7 Installation von Clair, Postgres-Datenbank und Clair Scanner 11
3.8 GitLab CI/CD Container Job Definition mit Clair Analyse 12

Abkürzungsverzeichnis

CVE	Common Vulnerabilities and Exposures
SQS	Software-Qualitätssicherung
YAML	YAML Ain't Markup Language

1 Einleitung

Der seit den letzten Jahren stetig wachsende Trend zur Containerisierung von Anwendung hat vor allem für Entwickler und Administratoren vieles einfacher und unkomplizierter gemacht. Mit ein paar wenigen Befehlen kann, z. B. mittels Docker, in wenigen Minuten ein Container mitsamt Anwendung aus einer scheinbar unbegrenzten Auswahl hochgefahren werden. Dies kommt nicht nur Entwicklern mit mehr Zeit für die Entwicklung von ihren eigenen Anwendungen entgegen, sondern z. B. auch der Flexibilität beim Bereitstellen von Anwendung als Microservices oder dem Reduzieren der Komplexität beim Bereitstellen von Services durch einen Administrator.

Sobald nun aber containerisierte Anwendung nicht nur für Entwicklungszwecke verwendet werden, stellt sich auch die Frage, wie es um die Sicherheit von fertigen Anwendungscontainern steht. Dazu gibt es natürlich nicht nur eine Betrachtungsweise, denn Sicherheit hängt immer von vielen Faktor ab. Ein Ansatz zur Verbesserung der Sicherheit von Anwendungscontainern bietet das Open Source Tool „Clair“ von CoreOS®. Dieses Tool zur statischen Analyse von Anwendungscontainern, bzw. Container im Allgemeinen, soll in dieser Projektarbeit im Rahmen des Master-Moduls „Software-Qualitätssicherung“ genauer betrachtet werden.

Dazu erläutert diese Dokumentation die Ausgangslage und die Zielsetztung der Projektarbeit „Statische Analyse von Docker-Containern mit Clair“ und stellt mit dem Kapitel 2 und Kapitel 3 die erarbeitete Umsetzung der lokalen bzw. automatisierten Analyse von Anwendungscontainern vor. Abschließend rundet ein kleiner Vergleich und ein Fazit vom Author die Dokumentation ab.

1.1 Ausgangssituation

Wenn in der Softwareentwicklung mit Docker-Containern gearbeitet wird, dann ist nicht nur die Anwendung, sondern auch ein Teil des Betriebssystems darin enthalten. Dabei ist es wichtig zu wissen, welche Art von Bibliotheken in dem Container für bekannte Schwachstellen bzw. Sicherheitslücken anfällig sein können.

Eine Möglichkeit diese Informationen zu finden, besteht darin, die Sicherheitsüberprüfung von bekannten Docker-Registries wie „Docker Hub“ oder „Quay.io“ zu verwenden. Dies bedeutet allerdings, dass sich dann das potenziell gefährdete bzw. noch nicht geprüfte Image bereits in der Docker-Registry befindet und an die Benutzer verteilt

wird. Dies liegt daran, dass diese Sicherheitsüberprüfungen asynchron im Hintergrund stattfinden und es kein Regelwerk gibt, welche ein Veröffentlichen unter bestimmten Bedingungen verhindert.

1.2 Zielsetzung

Das Ziel dieser Projektarbeit soll mit Hilfe von Clair eine Möglichkeit für Entwickler und Benutzer von Anwendungscontainern bieten, um Container vor ihrer Veröffentlichung bzw. Verwendung mit der statischen Analyse auf Sicherheitslücken zu überprüfen. Die Vorgehensweise soll dabei in etwa wie folgt sein.

Ablauf für Entwickler:

- Bauen und Testen der Anwendung
- Bauen vom Container mit Anwendung
- Untersuchen vom Container auf bekannte Sicherheitslücken
- Überprüfe ob nur erlaubte bzw. geringe Sicherheitslücken enthalten
- Veröffentliche Container auf Docker Hub o. Ä.

Ablauf für Benutzer:

- Beziehen des Anwendungscontainers von einer Container-Registry
- Untersuchen vom Container auf bekannte Sicherheitslücken
- Überprüfe auf gefährliche Sicherheitslücken
- Verwende Container

Zur Erreichung dieser zwei leicht unterschiedlichen Ziele ist es von Vorteil, dass die Analyse für Entwickler möglichst automatisiert und für Benutzer möglichst einfach abläuft. Für die Integration in eine automatisierte Umgebung, soll zunächst das lokale und manuelle Analysieren von Containern erarbeitet werden und im Anschluss in einer GitLab CI/CD Pipeline Anwendung finden.

2 Scannen von Container-Images mit Clair

Die Evaluierung und das Verwenden des Open Source Projekt „Clair“, zum Untersuchen von Container-Images auf bekannte Sicherheitslücken, kann von jedem Benutzer lokal auf seinem Rechner durchgeführt werden. Dieser Abschnitt dient einerseits als Dokumentation und andererseits als Hilfestellung für jeden der Container auf bekannte Schwachstellen untersuchen will.

Voraussetzung für die Installation:

- Windows, Linux oder MacOS
- x86 oder x86-64 CPU
- Docker Engine (18.06.0+)
- Docker Compose (1.22.0+)

Alle nachfolgenden Befehle wurden nur auf Linux und MacOS getestet. Die Befehle für Windows können sich teilweise unterscheiden und werden hier nicht weiter behandelt, da dem Author kein solches System zur Verfügung steht.

2.1 Installation von Clair mit PostgreSQL und Clair Scanner

Die Open Source Software „Clair“ besteht aus insgesamt zwei essentiellen Komponenten. Das ist zum einen eine Postgres-Datenbank zum Speichern der Vulnerability-Daten und zum anderen der Dienst Clair selbst. Clair kommt selbst als Docker-Container und stellt seine Analyse-Funktionen nur über eine Netzwerk-API bereit. Für die einfache Verwendung von Clair verwenden wir hier das Tool „Clair Scanner[1]“. Das Verwenden eines externen Tools zum Nutzen von Clair ist erst seit Version 2 nötigt, da mit dem Versionssprung die Funktionalität für das lokale Untersuchen von Container-Images aus Version 1 vollständig entfallen ist.

[1] `https://github.com/arminc/clair-scanner`

Für eine benutzerfreundlichere Installation von Clair, ist das Setup mittels Docker Compose und einem BASH-Skript vereinfacht worden. Das Listing 2.1 zeigt die Installationsschritte mittels den vorbereiteten Konfigurationsdateien für Docker Compose und Clair und dem BASH-Skript für den Clair Scanner. Der Docker Compose Service für Clair ist so konfiguriert, dass ein Zugriff auf die API nur über „*http://127.0.0.1:6060/*“ möglich ist. Das BASH-Skript benötigt für die Installation Root-Rechte, da es die ausführbare Datei vom Clair Scanner unter „*/usr/local/bin*“ für die globale Verwendung installiert.

```
johann@ubuntu-desktop:~# git clone https://github.com/jneuhauser/clair-local-scan.git
johann@ubuntu-desktop:~# cd clair-local-scan
johann@ubuntu-desktop:~# docker-compose up -d
johann@ubuntu-desktop:~# sudo ./install_clair_scanner.sh
```

Listing 2.1: Installation von Clair, Postgres-Datenbank und Clair Scanner

Ist die Software einmal auf dem Computer eingerichtet und gestartet, so kann jederzeit eine Analyse mit dem Clair Scanner beauftragt werden. Die Datenbank mit den Vulnerability-Daten aktualisiert sich nach jedem Start bzw. alle zwei Stunden automatisch.

Clair und die zugehörige Datenbank startet in der vorgegebenen Konfiguration nach jedem Neustart automatisch sofern diese mit „**docker-compose up -d**“ einmal gestartet wurden. Zum dauerhaften Stoppen von Clair und der Datenbank reicht ein Wechseln in den Ordner mit der Docker Compose Datei und ein „**docker-compose down**“.

2.2 Scannen von lokalen Container-Images

Das Analysieren von lokalen Container-Images gestaltet sich mit dem Client-Tool Clair Scanner sehr einfach. Zunächst muss das zu analysierende Container-Image lokal auf den Rechner vorhanden sein. Dies erfolgt im einfachsten Fall mit Befehl „*docker pull*“ und dem anschließenden Aufruf von Clair Scanner wie es exemplarisch für das Container-Image „*quay.io/coreos/clair:v2.0.8*“ in Listing 2.2 gezeigt ist. Der Befehl „*clair-scanner*“ verwendet per Default als Adresse für Clair „*http://127.0.0.1:6060/*“ und benötigt deswegen bis auf den Namen vom Container-Image keinen weiteren Parameter. Das exemplarisch verwendete Container-Image ist das selbe, welches in unserer Docker Compose Datei für die lokale Analyse verwendet wird.

```
johann@ubuntu-desktop:~# docker pull quay.io/coreos/clair:v2.0.8
...
Status: Image is up to date for quay.io/coreos/clair:v2.0.8
johann@ubuntu-desktop:~# clair-scanner quay.io/coreos/clair:v2.0.8
[INFO] -> Start clair-scanner
...
[ERRO] -> Image [quay.io/coreos/clair:v2.0.8] contains 1 unapproved vulnerabilities
+------------+---------------------+--------------+-----------------+-------------+
| STATUS     | CVE SEVERITY        | PACKAGE NAME | PACKAGE VERSION | DESCRIPTION |
+------------+---------------------+--------------+-----------------+-------------+
| Unapproved | High CVE-2018-20843 | expat        | 2.2.6-r0        | Link ...    |
+------------+---------------------+--------------+-----------------+-------------+
```

Listing 2.2: Scannen von lokalen Container-Image

Wie in den letzten Zeilen in Listing 2.2 zu sehen, enthält dieses Container-Image eine als Hoch eingestufte Sicherheitslücke in der Bibliothek *„expat“* laut der Ausgabe vom Clair Scanner. Folgt man dem Link in der Beschreibung, so lässt sich schnell nachlesen, dass die Sicherheitslücke trotz der Einstufung als High eine sogenannte *„denial-of-service attack“* unter bestimmten Umständen ermöglicht. Ein Durchsuchen vom Quellcode von Clair deutet nur auf die Verwendung von *„expat“* als XML-Parser in einem Test hin und somit sollte diese Sicherheitslücke keine Bedenken beim Einsatz von diesem Container-Image hervorrufen.

Mit Hilfe einer Whitelist können durch den Benutzer für den Einsatzzweck unkritische Sicherheitslücken als akzeptabel definiert werden. Dazu bietet der Clair Scanner mit dem Parameter *„-w“* die Möglichkeit eine im YAML-Format definierte Whitelist wie in Listing 2.3 zu übergeben. Die Whitelist kennt dabei zwei Möglichkeiten die Ausnahmen für Sicherheitslücken zu definieren. Zu einem können CVE-Nummern in der Sektion *„generalwhitelist“* allgemein für alle oder in der Sektion *„images:image-name“* nur für bestimmte Images ausgenommen werden. Die Angabe einer Notiz hinter der CVE-Nummer ist optional, sollte aber für einen besseren Überblick angegeben werden.

```
generalwhitelist:
  CVE-2018-20843: libexpat DoS
images:
  quay.io/coreos/clair:
    CVE-2018-20843: libexpat DoS (only used in test)
```

Listing 2.3: Beispiel einer Whitelist für Clair Scanner

Die Angabe der Schwachstelle von „*expat*“ in der Sektion „*generalwhitelist*“ ist eher gefährlich, denn wenn ein anderes Container-Image den XML-Parser für Eingabedaten und nicht nur für einen Testfall verwendet, so ist die Anwendung im Container dadurch verwundbar und kann dementsprechend negative Folgen nach sich ziehen. Im Listing 2.3 ist dies nur zu Demonstrationszwecken geschehen.

Das erneute Analysieren vom Container-Image unter der Angabe der Whitelist wie in Listing 2.4 produziert eine sehr ähnliche Ausgabe wie in Listing 2.2. Der Unterschied in den beiden Ausgaben liegt darin, dass dem Vorhandensein der Sicherheitslücke mit der Whitelist zugestimmt wurde und der Aufruf vom Clair Scanner den Retrun-Wert 0 zurückgibt.

```
johann@ubuntu-desktop:~# clair-scanner -w clair-whitelist.yml quay.io/coreos/clair:v2.0.8
[INFO] -> Start clair-scanner
...
[WARN] -> Image [quay.io/coreos/clair:v2.0.8] contains 1 total vulnerabilities
[INFO] -> Image [quay.io/coreos/clair:v2.0.8] contains NO unapproved vulnerabilities
+----------+---------------------+--------------+-----------------+-------------+
| STATUS   | CVE SEVERITY        | PACKAGE NAME | PACKAGE VERSION | DESCRIPTION |
+----------+---------------------+--------------+-----------------+-------------+
| Approved | High CVE-2018-20843 | expat        | 2.2.6-r0        | Link ...    |
+----------+---------------------+--------------+-----------------+-------------+
```

Listing 2.4: Scannen von lokalen Container-Images mit Whitelist

Somit ist die Sicherheitslücke protokolliert, ihr ausmaß für den Betrieb vom Container-Image untersucht und als unbedeutend für den Einsatzzweck eingestuft worden. Damit kann das Container-Image für den abgesehen Einsatzzweck mit gutem Gewissen eingesetzt werden.

3 Integration von Clair in GitLab CI/CD

Mit der in Kapitel 2 geleisteten Vorarbeit ist die Integration in eine Instanz von GitLab bzw. in den ausführenden Part der GitLab CI/CD Pipeline, dem GitLab Runner, ein Kinderspiel.

Für die Versuche im Rahmen der Projektarbeit ist die benötigte Systemumgebung in einer virtuellen Maschine mit zwei virtuellen CPUs und 4 GB RAM auf einem privatem Server aufgesetzt worden.

Verwendete Softwareversionen für Systemumgebung:

- Ubuntu Server 18.04
- Docker CE 18.09.7
- Docker Compose 1.24.1
- GitLab EE 12.0.3
- GitLab Runner 12.0.1

Als Vorbereitung zur Installation der weiteren Softwarepakete muss das Downloaden von Installationsskripten mittels HTTPS funktionieren und deswegen die in Listing 3.1 gezeigten Schritte zur Installation der benötigten Ubuntu-Pakete durchgeführt werden.

```
johann@ubuntu-gitlab:~# sudo apt-get update
johann@ubuntu-gitlab:~# sudo apt-get install -y curl wget ca-certificates
```

Listing 3.1: Benötigte Pakete in Ubuntu 18.04 installieren

3.1 GitLab EE Omnibus Installation

Die Installation von GitLab EE auf einem eigenen Server funktioniert dank dem Omnibus-Paket sehr einfach. Die benötigten Installationsschritte sind der Anleitung[1]

[1] `https://about.gitlab.com/install/`

auf GitLab.com entnommen und wurden für die eigene Umgebung angepasst.

Das Listing 3.2 zeigt die einfache Installation für die verwendete und auf die Server-IP zeigende Domain *„gitlab.yesterdaymorning.de“*. Dazu wird, mittels dem per Curl bezogenen Installationsskript, das Paket-Repository von GitLab EE konfiguriert und aktualisiert und im Anschluss mit dem Paketmanager Apt das GitLab Omnibus-Paket *„gitlab-ee“* installiert. Das Omnibus-Paket erledigt alle relevanten Schritte von der Installation der Datenbank, über die Konfiguration von Nginx als Reverse-Proxy, bis hin zum beziehen vom Zertifikat für die angegebene externe HTTPS-Adresse vollkommen selbstständig. Nach der erfolgreichen Installation sollte sofort die angegebene externe Web-Adresse geöffnet und das Passwort für den Administrator-Account festgelegt werden.

```
johann@ubuntu-gitlab:~# curl -sSL \
    https://packages.gitlab.com/install/repositories/gitlab/gitlab-ee/script.deb.sh \
    | sudo bash
...
The repository is setup! You can now install packages.
johann@ubuntu-gitlab:~# sudo EXTERNAL_URL="https://gitlab.yesterdaymorning.de" \
    apt-get -y install gitlab-ee
...
Thank you for installing GitLab!
GitLab should be available at https://gitlab.yesterdaymorning.de
```

Listing 3.2: Installation von GitLab EE auf Ubuntu 18.04

Abschließend aktivieren wir noch die GitLab interne Container Registry zur späteren Verwendung in eigenen Projekten. Dies erfolgt, wie in Listing 3.3 gezeigt, mit der Definition der externen Adresse mittels *„registry_external_url“* in der Konfigurationsdatei von GitLab und dem darauf folgenden Aufruf von *„gitlab-ctl reconfigure“* zur Neukonfiguration. Selbstverständlich kann als externe Adresse für die Registry auch die selbe Adresse wie für GitLab selbst verwendet werden.

```
johann@ubuntu-gitlab:~# echo \
    'registry_external_url "https://registry.yesterdaymorning.de"' \
    >> /etc/gitlab/gitlab.rb
johann@ubuntu-gitlab:~# gitlab-ctl reconfigure
...
gitlab Reconfigured!
```

Listing 3.3: Aktivieren von GitLab Container Registry in GitLab EE

3.2 GitLab Runner Installation

Die Installation vom GitLab Runner muss nicht, wie in dem folgenden Beispiel, auf der selben Maschine wie GitLab selbst installiert werden. Damit wir später in der GitLab CI/CD Pipeline überhaupt Docker Container bauen können und auch die Docker Compose Umgebung von Clair aus Abschnitt 2.1 aufsetzten können, benötigen wir auf dem Rechner, wo später der GitLab Runner läuft, die Docker Engine und das Tool Docker Compose. Der Vollständigkeit halber ist die Installation der Docker Engine und vom Tool Docker Compose in Listing 3.4 für Ubuntu Server dargestellt.

```
johann@ubuntu-gitlab:~# curl -sSL https://get.docker.com | sh
...
Server: Docker Engine - Community
 Engine:
  Version:          18.09.7
  API version:      1.39 (minimum version 1.12)
  Go version:       go1.10.8
  Git commit:       2d0083d
  Built:            Thu Jun 27 17:23:02 2019
  OS/Arch:          linux/amd64
  Experimental:     false
...
johann@ubuntu-gitlab:~# FALVOR="docker-compose-$(uname -s)-$(uname -m)"
johann@ubuntu-gitlab:~# sudo curl \
    -L "https://github.com/docker/compose/releases/download/1.24.0/$FALVOR" \
    -o "/usr/local/bin/docker-compose"
johann@ubuntu-gitlab:~# sudo chmod +x /usr/local/bin/docker-compose
```

Listing 3.4: Installation von Docker auf Ubuntu 18.04

Die Installation vom GitLab Runner ist ebenso einfach gestaltet wie von GilLab EE mittels dem Omnibus-Paket und ist für die meisten Distributionen in der Anleitung[2] auf GitLab.com genauer erläutert. Nach dem Setup der Paketquellen mittels des Installationsskripts folgt die Installation vom GitLab Runner als Ubuntu-Paket wie in Listing 3.5 gezeigt. Der wichtige Schritt nach der Installation ist das Hinzufügen des Benutzers vom GitLab Runner zur Gruppe *„docker“*. Für Debian und Ubuntu ist das der Benutzer *„gitlab-runner“*. Dies ist notwendig damit der GitLab Runner den Docker Daemon mittels der Socket-Datei *„/var/run/docker.sock“* steuern darf.

[2] `https://docs.gitlab.com/runner/install/linux-repository.html`

```
johann@ubuntu-gitlab:~# curl -sSL \
    https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh \
    | sudo bash
...
The repository is setup! You can now install packages.
johann@ubuntu-gitlab:~# sudo apt-get -y install gitlab-runner
johann@ubuntu-gitlab:~# sudo usermod -aG docker gitlab-runner
```

Listing 3.5: Installation von GitLab Runner auf Ubuntu 18.04

Zum Registrieren des GitLab Runners benötigen wir einen Registrierungs-Token von der GitLab Instanz. Für einen „shared“ Runner für alle Repositories der Instanz kann dieser im Adminbereich unter Runners bezogen werden. Für einen Runner, welcher nur für ein Projekt Aufträge annimmt, ist der Token im Repository unter Settings -> CI/CD -> Runners zu finden. Wir verwenden hier einen „shared“ Runner, welcher für alle Projekte verwendet werden kann. Als „Executer“ wählen wir „shell“, da wir mit diesem Runner lediglich das lokal verfügbare Docker und Clair verwenden wollen. Das Bauen und Analysieren von Docker Containern wäre zwar auch aus einem Docker Container möglich, ist jedoch mit einer „Docker in Docker“ Lösung weniger performant und nur unnötig komplizierter. Das Listing 3.6 zeigt den Registrationsprozess unseres GitLab Runners bei unserer GitLab EE Instanz mit einem einzelnen Befehl. Zu beachten sei hier der Parameter „*–tag-list shell*“, welcher unseren registrierten Runner später in der CI/CD-Konfiguration explizit auswählen lässt.

```
johann@ubuntu-gitlab:~# EXTERNAL_URL="https://gitlab.yesterdaymorning.de"
johann@ubuntu-gitlab:~# PROJECT_REGISTRATION_TOKEN="Z6fCm9RV_8SdUsPswoxm"
johann@ubuntu-gitlab:~# sudo gitlab-runner register \
    --non-interactive \
    --url "$EXTERNAL_URL" \
    --registration-token "$PROJECT_REGISTRATION_TOKEN" \
    --executor shell \
    --description "shell-executor" \
    --tag-list "shell" \
    --run-untagged="true" \
    --locked="false"
...
Registering runner... succeeded
...
```

Listing 3.6: Registrieren von GitLab Runner in GitLab EE

3.3 Installation von Clair mit PostgreSQL und Clair Scanner

Der Installationsvorgang von Clair erfolgt auf unserem Ubuntu-Server ebenso mit dem Git-Repository `https://github.com/jneuhauser/clair-local-scan` und beschleunigt das Setup somit stark. Die Clair-API ist somit auch nur für den lokalen Zugriff über *„http://127.0.0.1:6060/“* verwendbar und sollte somit keine größeren Sicherheitsbedenken erzeugen. Bei Bedarf kann der Service von Clair und PostgreSQL natürlich auf einen Nachbarserver ausgelagert werden. Das Tool Clair Scanner muss allerdings lokal auf dem Computer oder im verwendeten Container vom GitLab Runner vorhanden sein.

```
johann@ubuntu-gitlab:~# sudo su
root@ubuntu-gitlab:~# cd /opt
root@ubuntu-gitlab:~# git clone https://github.com/jneuhauser/clair-local-scan.git
root@ubuntu-gitlab:~# cd clair-local-scan
root@ubuntu-gitlab:~# docker-compose up -d
root@ubuntu-gitlab:~# ./install_clair_scanner.sh
```

Listing 3.7: Installation von Clair, Postgres-Datenbank und Clair Scanner

3.4 Analyse eines Docker-Containers in GitLab CI/CD

Nachdem erfolgreichen Installieren und Konfigurieren von GitLab EE, GitLab Runner und Clair mit PostgreSQL kann das Bauen und Analysieren von Container-Images in einer GitLab CI/CD Pipeline durchgeführt werden. Das Listing 3.8 zeigt eine vollständige *„.gitlab-ci.yml“* zum Bauen eines Container-Images aus einem Dockerfile im Hauptverzeichnis des Repositories, dem anschließenden Analysieren mittels Clair Scanner, Clair und der Vulnerablility-Daten aus der PostgreSQL-Datenbank und dem anschließenden Pushen des Images in die eigene Docker Registry des Repositories.

Die Stage *„scan_container“* ist dabei so konfiguriert, dass eine im Hauptverzeichnis vorhandene *„clair-whitelist.yml“* als eine wie in Listing 2.3 gezeigte Whitelist für Clair dient. Zudem kann mittels der CI/CD-Variable *„CLAIR_THRESHOLD“* ein Schwellwert für ungewollte Sicherheitslücken definiert werden. Der Standardwert für den Schwellwert liegt bei *„Unknown“*, welcher der niedrigste Schwellwert ist und somit bei allen Sicherheitslücken auslöst. Die möglichen Schwellwerte sind „Unknown“, „Negligible“, „Low“, „Medium“, „High“, „Critical“ und „Defcon1“.

Die Sektion *„artifacts:reports:container_scanning“* definiert das Hochladen des Reports über die gefundenen Sicherheitslücken als JSON-Textdokument zur GitLab Instanz. GitLab in der Enterprise Edition unterstützt das Auswerten dieser Datei und stellt die

gesammelten Informationen im „Security Dashboard“, dem Reiter „Security“ in der Job-Übersicht der CI/CD-Pipeline und bei „Merge Requests“ an.

```
stages:
  - build
  - scan
  - push

build_container:
  stage: build
  tags:
    - shell
  script:
    - docker build -t $CI_REGISTRY_IMAGE/$CI_COMMIT_REF_SLUG:$CI_COMMIT_SHA .

scan_container:
  stage: scan
  tags:
    - shell
  variables:
    GIT_STRATEGY: none
  allow_failure: false
  script:
    - touch clair-whitelist.yml
    - if [ -z "$CLAIR_THRESHOLD" ]; then CLAIR_THRESHOLD="Unknown"; fi
    - clair-scanner
        -w clair-whitelist.yml
        -t $CLAIR_THRESHOLD
        -r gl-container-scanning-report.json
        -l clair.log
        $CI_REGISTRY_IMAGE/$CI_COMMIT_REF_SLUG:$CI_COMMIT_SHA
  artifacts:
    reports:
      container_scanning: gl-container-scanning-report.json

push_container:
  stage: push
  tags:
    - shell
  script:
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"
    - docker push $CI_REGISTRY_IMAGE/$CI_COMMIT_REF_SLUG:$CI_COMMIT_SHA
```

Listing 3.8: GitLab CI/CD Container Job Definition mit Clair Analyse

Liefert das Tool „Clair Scanner“ einen Rückgabewert ungleich 0, so schlägt die Stage in der Pipeline fehl und die weiteren Stages kommen nicht mehr zur Ausführung. Dieses Verhalten kann mit einem „true“ in der Sektion „allow_failure“ deaktiviert werden. Der Bericht mit den Sicherheitslücken wird dann trotzdem zur GitLab Instanz hochgeladen.

4 Fazit

Das Open Source Projekt „Clair“ von CoreOS® kann beim Entwickeln und beim Verwenden von Anwendungscontainern einen erheblichen Beitrag zum sicheren Umgang mit diesen leisten.

Mit Ancore[1] existiert ein weiteres Tool für die statische Analyse von Container-Images, welches in der Open Source Version ein fast identisches Feature-Set[2] bietet. Die Vulnerability-Listen sind bei Anchore bis auf die zusätzliche Liste für Pakete vom Node Package Manager „npm“ identisch mit denen von „Clair“.

Aufgrund der Unterstützung der Reports von der Analyse mit „Clair Scanner“ durch GitLab lässt sich das hier vorgestellte Konzept besser als Anchore in die GitLab CI/CD Pipeline integrieren. Zumindest wenn die Enterprise Edition von GitLab verwendet wird.

Zusammengefasst lässt sich sagen, dass die bestehenden Tools zur statischen Analyse von Container-Images auf jeden Fall zum Einsatz kommen sollten wenn Anwendungen für den Produktiven Einsatz in Container verpackt werden.

[1] `https://anchore.com/`
[2] `https://anchore.com/enterprise/`