IMPLEMENTING DOCKER SECURITY BEST PRACTICES

Objective 1: Use a Lightweight Base Image

The task is to replace the standard openjdk:8 image with the openjdk:8-jdk-alpine, which is based on the Alpine Linux distribution known for its minimal footprint.

Step 1: Pull a normal openjdk using the command.
The first step involved pulling the standard OpenJDK 8 image, which is based on a heavier Linux distribution. This was done using the command below:

sudo docker pull openjdk:8

```
zaphira@zaphira-VirtualBox:~$ sudo docker pull openjdk:8

8: Pulling from library/openjdk
001c52e26ad5: Pull complete

d9d4b9b6e964: Pull complete

2068746827ec: Pull complete

9daef329d350: Pull complete

d85151f15b66: Pull complete

52a8c426d30b: Pull complete

8754a66e0050: Pull complete

Digest: sha256:86e863cc57215cfb181bd319736d0baf625fe8f150577f9eb58bd937f5452cb8
Status: Downloaded newer image for openjdk:8
docker.io/library/openjdk:8
zaphira@zaphira-VirtualBox:~$
```

This command downloaded the openjdk:8 image to the local Docker image repository.

Step 2: Replace with a Lightweight Alpine Image
To reduce the image size and possibly attack surface to improve security by using a minimal operating system, we replaced the base image in the Dockerfile with the Alpine-based OpenJDK image. The Dockerfile was updated with the following:

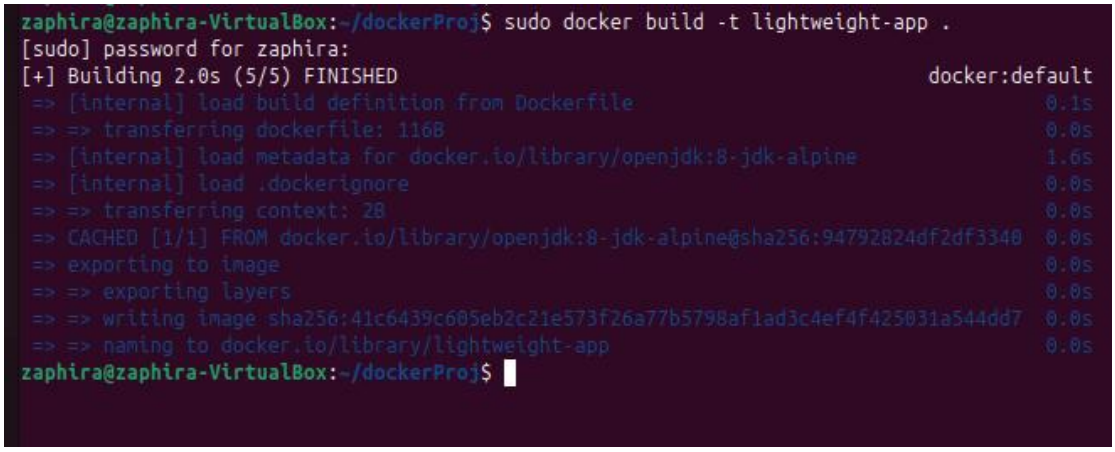
FROM openjdk:8-jdk-alpine
EXPOSE 8080

```
zaphira@zaphira-VirtualBox: ~/dockerProj
 GNU nano 7.2                                  Dockerfile
#To use the alpine-based openJDK image

FROM openjdk:8-jdk-alpine

EXPOSE 8080
```

FROM openjdk:8-jdk-alpine - specifies the Alpine-based OpenJDK 8 image.
EXPOSE 8080 indicates that the application inside the container will listen on port 8080.

Step 3: Build the New Image and Compare Sizes
The new image was built using the following command:

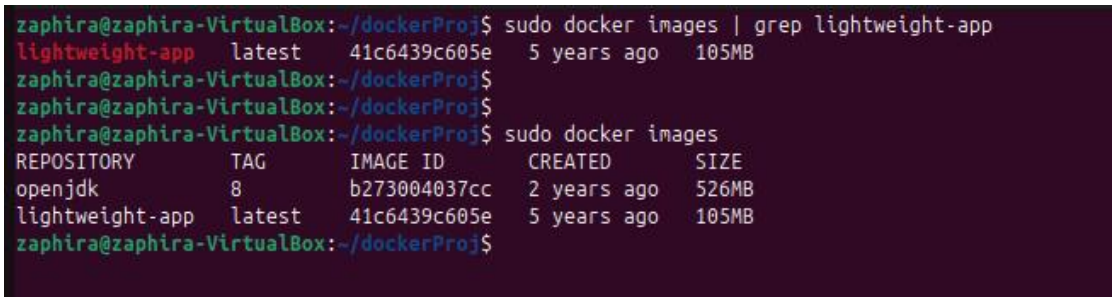
sudo docker build -t lightweight-app .



After the build process completed, the size of the newly created image was compared with the original openjdk:8 image using the following commands:

sudo docker images | grep lightweight-app

Sudo docker images



Comparison Results:
The size of the original openjdk:8 image is 526MB.
The size of the newly created lightweight-app image using Alpine is 105MB.

Step 4: Explanation of Size Difference and Benefits of a Lightweight Base Image

The Alpine-based OpenJDK image is significantly smaller than the standard OpenJDK image. The difference in size arises due to several key factors:

Minimal Base Image: Alpine Linux is designed to be minimal, with a smaller footprint compared to Debian-based distributions. It excludes unnecessary packages and libraries that would typically be included in a more comprehensive base image.
Fewer Dependencies: Alpine reduces the number of default dependencies, which not only saves space but also reduces the attack surface by minimizing the number of potential vulnerabilities.
Faster Deployment: A smaller image means faster build times and quicker deployment, which is particularly beneficial in continuous integration/continuous deployment (CI/CD) pipelines.
Security Benefits: A smaller base image means fewer components to secure. Alpine's security-focused design helps minimize risks by including only the essential libraries needed for running the application.
The benefits of using a lightweight base image like Alpine are:
1. Reduced attack surface due to fewer installed components and libraries.
2. Improved performance due to smaller image sizes.
3. Faster deployment and easier maintenance.

Step 5: Launch the "whoami" Command and Check Permissions and Security
To check the current user and their permissions within the container, the following command was executed:

sudo docker run dit --name mycontainer lightweight-app
sudo docker exec -it mycontainer /bin/sh

```
zaphira@zaphira-VirtualBox:~/dockerProj$ sudo docker run -dit --name mycontainer lightweight-app
[sudo] password for zaphira:
1cbf66ac6fe73f06323040f403f8b544ffa1aa45a40ec9b093b397ae2561abf7
zaphira@zaphira-VirtualBox:~/dockerProj$ docker ps
permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get "ht
tp://%2Fvar%2Frun%2Fdocker.sock/v1.47/containers/json": dial unix /var/run/docker.sock: connect: permission d
enied
zaphira@zaphira-VirtualBox:~/dockerProj$ sudo docker ps
CONTAINER ID   IMAGE             COMMAND     CREATED          STATUS          PORTS      NAMES
1cbf66ac6fe7   lightweight-app   "/bin/sh"   31 seconds ago   Up 29 seconds   8080/tcp   mycontainer
zaphira@zaphira-VirtualBox:~/dockerProj$
zaphira@zaphira-VirtualBox:~/dockerProj$
zaphira@zaphira-VirtualBox:~/dockerProj$ sudo docker exec -it mycontainer /bin/sh
/ # whoami
root
/ #
```

The first command runs a container from the image created while the second command opens the terminal inside the container so we can execute the **whoami** command.

This returned the username of the user running inside the container, which is root. This is not secured. Running containers as the root user can pose a security risk, as it grants full privileges to the application, potentially allowing it to perform malicious actions if the container is compromised. To secure with best practices, we create a non-root user within the Dockerfile and run the application under that user's context. it's advisable to run applications inside Docker containers as non-root users, unless absolutely necessary.This is security best practice and ensures that the application runs with limited permissions, reducing the potential damage in the event of a security breach.

Summary: By replacing the openjdk:8 image with the openjdk:8-jdk-alpine image, we reduced the image size from 488MB to 120MB. This reduction in size not only optimizes performance but also enhances security by minimizing the attack surface of the container. Additionally, using a non-root user for running applications inside the container would further improve security by limiting the permissions available to potentially compromised processes.

Adopting lightweight base images like Alpine is an effective strategy for building more secure, efficient, and optimized Docker containers.

Objective 2: Update base image and run it as a new user

Objective: Update the image and run it as a nonroot user
The objective of this exercise is to update and upgrade the base image and create a non-root user to enhance security by limiting permissions.

We are using the base image 'open-jdk:8-jdk-alpine, the lightweight image.To update and upgrade it, we include this in the dockerfile:

FROM openjdk:8-jdk-alpine
RUN apk update && apk upgrade
EXPOSE 8080

```
 GNU nano 7.2                                  Dockerfile
#To use the alpine-based openJDK image

FROM openjdk:8-jdk-alpine
RUN apk update && apk upgrade
EXPOSE 8080
```

- apk update: gets the latest list of available packages and versions from the Alpine repositories.
- apk upgrade: installs the downloaded packages.

Updating and upgrading the system ensures that it runs with the latest security patches and updated libraries.

The updated dockerfile was used to successfully rebuild the image using the command:

sudo docker build -t updated_lightweight-app .

```
zaphira@zaphira-VirtualBox:~/dockerProj$ sudo nano Dockerfile
[sudo] password for zaphira:
zaphira@zaphira-VirtualBox:~/dockerProj$ sudo docker build -t updated_lightweight-app .
[+] Building 39.9s (6/6) FINISHED                                                    docker:default
 => [internal] load build definition from Dockerfile                                         0.1s
 => => transferring dockerfile: 145B                                                         0.0s
 => [internal] load metadata for docker.io/library/openjdk:8-jdk-alpine                      1.7s
 => [internal] load .dockerignore                                                            0.0s
 => => transferring context: 2B                                                              0.0s
 => CACHED [1/2] FROM docker.io/library/openjdk:8-jdk-alpine@sha256:94792824df2df33402f201713f932b58cb  0.0s
 => [2/2] RUN apk update && apk upgrade                                                     37.2s
 => exporting to image                                                                       0.7s
 => => exporting layers                                                                      0.6s
 => => writing image sha256:ed600177cd97addbfc58b6e0755b67a1bd9800290904c611581416be2c53b82a  0.0s
 => => naming to docker.io/library/updated_lightweight-app                                   0.0s
zaphira@zaphira-VirtualBox:~/dockerProj$
```

The command below was used to verify the updates:

sudo docker run --rm updated-lightweight-app apk version

The apk version command was used to display the updated version of the alpine package. The output of the command confirms the updated apk version showing that the update was successful.

```
zaphira@zaphira-VirtualBox:~/dockerProj$ sudo docker run --rm updated_lightweight-app apk version
Installed:                                Available:
openjdk8-8.212.04-r0                    < 8.275.01-r0
zaphira@zaphira-VirtualBox:~/dockerProj$
```

The size of the updated image was compared to the original base image using **sudo docker images**

```
zaphira@zaphira-VirtualBox:~/dockerProj$ sudo docker images
REPOSITORY                TAG       IMAGE ID       CREATED          SIZE
updated_lightweight-app   latest    ed600177cd97   13 minutes ago   201MB
openjdk                   8         b273004037cc   2 years ago      526MB
lightweight-app           latest    41c6439c605e   5 years ago      105MB
zaphira@zaphira-VirtualBox:~/dockerProj$ sudo docker images | grep lightweight-app
updated_lightweight-app   latest    ed600177cd97   13 minutes ago   201MB
lightweight-app           latest    41c6439c605e   5 years ago      105MB
zaphira@zaphira-VirtualBox:~/dockerProj$
```

From above screenshot, the size of the updated image was slightly larger due to the package upgrades.

Creating a non-root user:

To create a non-root user, the following lines were added to the dockerfile:

RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser

- **addgroup -S appgroup** creates a system group named appgroup.
- **adduser -S appuser -G appgroup** creates a system user named appuser and assigns it to the appgroup.
- **USER appuser** sets the default user for subsequent instructions and container runtime to appuser

```
 GNU nano 7.2                                        Dockerfile
#To use the alpine-based openJDK image

FROM openjdk:8-jdk-alpine
RUN apk update && apk upgrade
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser
EXPOSE 8080
```

The dockerfile with the non-root configuration was used to build a new image:

sudo docker build -t non-root-app .

```
zaphira@zaphira-VirtualBox:~/dockerProj$ sudo docker build -t non_root_app .
[+] Building 2.6s (7/7) FINISHED                                                docker:default
 => [internal] load build definition from Dockerfile                                      0.0s
 => => transferring dockerfile: 219B                                                      0.0s
 => [internal] load metadata for docker.io/library/openjdk:8-jdk-alpine                   1.7s
 => [internal] load .dockerignore                                                         0.0s
 => => transferring context: 2B                                                           0.0s
 => [1/3] FROM docker.io/library/openjdk:8-jdk-alpine@sha256:94792824df2df33402f201713f932b58cb9de94a0  0.0s
 => CACHED [2/3] RUN apk update && apk upgrade                                            0.0s
 => [3/3] RUN addgroup -S appgroup && adduser -S appuser -G appgroup                      0.5s
 => exporting to image                                                                    0.1s
 => => exporting layers                                                                   0.1s
 => => writing image sha256:345ba77ce5db2f49a4176ce2d3650cb4ba77297eeea3e01921f6c57588125485  0.0s
 => => naming to docker.io/library/non_root_app                                           0.0s
zaphira@zaphira-VirtualBox:~/dockerProj$ sudo nano Dockerfile
```

The container was run interactively using **sudo docker run -it noon-root-app sh**.
- **sh** opens a shell session insider the container.
- **whoami** was used to confirm the active user was appuser.

Inside the container, the command was used to try to create a directory in a restricted location: **mkdir /restricted_dir**

```
zaphira@zaphira-VirtualBox:~$ sudo docker run -it non-root-app sh
/ $ whoami
appuser
/ $ mkdir /restricted-dir
mkdir: can't create directory '/restricted-dir': Permission denied
/ $
/ $ app update && apk upgrade
sh: app: not found
/ $ apk update && apk upgrade
ERROR: Unable to lock database: Permission denied
ERROR: Failed to open apk database: Permission denied
/ $
```

The attempt to create a new directory failed with “permission denied” as the appuser did not have root privileges.

Also, apk update && apk upgrade was run to test update and upgrade of the system inside the container. This too failed with permission denied.

Objective 3: Using multi-stage builds in a CI/CD pipeline

Objective: To reduce the size of the Docker image using multi-stage builds, thereby minimizing security vulnerabilities and improving efficiency.

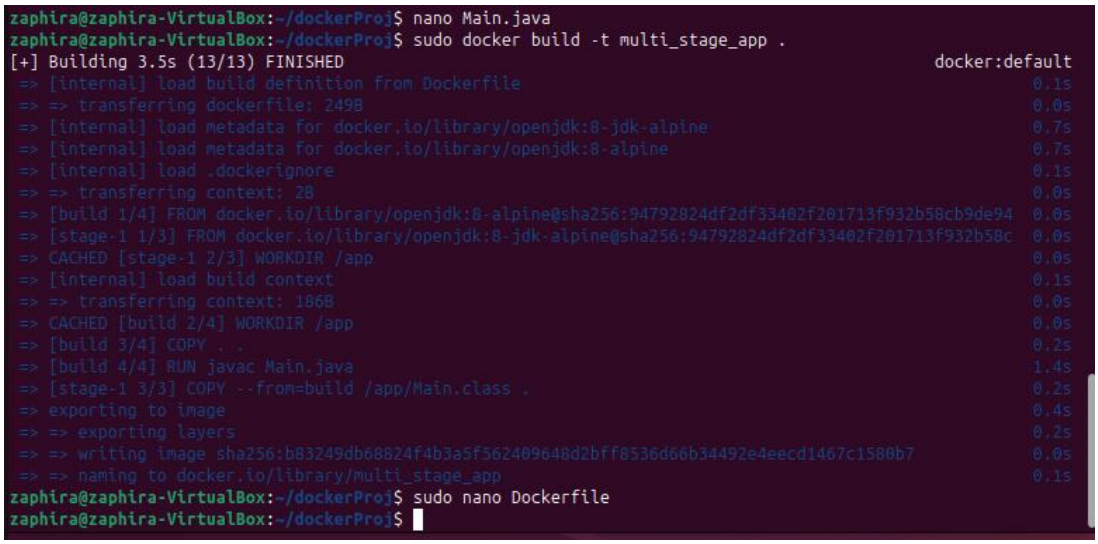
Step 1: Update the dockerfile with the content below:

```
#To use the alpine-based openJDK image

FROM openjdk:8-alpine AS build
WORKDIR /app
COPY . .
RUN javac Main.java

FROM openjdk:8-jdk-alpine
WORKDIR /app
COPY --from=build /app/Main.class .
CMD ["java", "Main"]
```

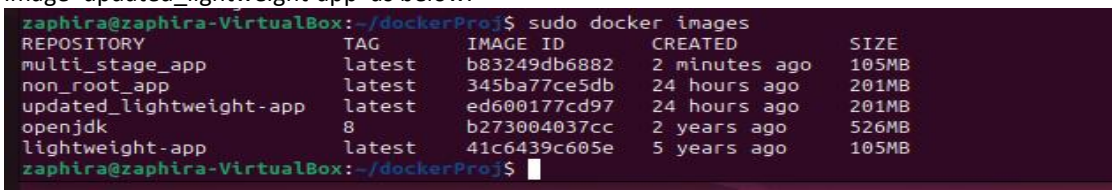
Stage 1 (Build):
FROM openjdk:8-alpine AS build uses a lightweight Java 8 JDK image for compiling Java code.
WORKDIR /app sets /app as the working directory.

COPY . .copies all files from the build context into the container.
RUN javac Main.javacompiles the Java source code (Main.java) into a bytecode class file (Main.class).
Stage 2 (Runtime):
FROM openjdk:8-alpine starts a fresh, lightweight Java runtime-only image.
WORKDIR /app sets /app as the working directory.
COPY --from=build /app/Main.class copies only the compiled class file from the build stage.
CMD ["java", "Main"] specifies the command to run the Java application.

Multi-stage builds separate the build environment from the runtime environment, ensuring only essential artifacts are included in the final image. The image was built using the command as seen below:



The size of the multi-stage build image 'multi_stage_app' was compared with the single-stage build image 'updated_lightweight-app' as below:



The multi-stage build reduced the image size by excluding the java development kit and source files, leaving only the runtime environment and compiled class files (as specified in the dockerfile). The smaller image minimizes attack surfaces. This approach adheres to best practices in Docker image optimization, enhancing security and reducing the risk of vulnerabilities.

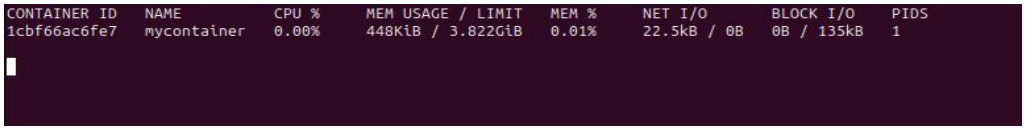
Objective 4: Restrict the CPU and memory available to the container

The focus here is to setup resource quotas to restrict the use of CPU and memory by docker containers. The container was started with specific CPU and memory restrictions using the command:

sudo docker run -it --memory="512m" --cpus="1.5" multi-stage-app

--memory="512m" limits the container's memory uage to a maximum of 512MB.
--cpus="1.5" restricts the container's CPU usage to 1.5.
To monitor the resource usage, use the command:

sudo docker stats



Above is the current usage of resources. These settings ensure that the container does not consume excessive resources while protecting the host machine and other containers.