- Overview
- Creating the Application
- Multi-Stage Build Using Chainguard Images
- Advantages of Chainguard Images

Overview

Containerization technologies such as Docker have revolutionized the development to production pipeline, making it easier than ever to reproduce environments and set up and maintain infrastructure. Unfortunately, containers come with their own risks and overhead. While standard images can be convenient for development, putting large images into production increases the attack surface and can demand additional and unnecessary resources in deployment.

In this tutorial, we'll build on a Chainguard Image to containerize a Python application in a multi-stage build process. By using a Chainguard Image, our containerized application benefits from a minimalist design that reduces attack surface and from a set of features focused on security and ease of development. In addition, the use of a multistage build process will allow us to develop our application with access to tools such as package managers while removing these potential vulnerabilities later in the build process.

Creating the Application

The Python application we will containerize in this tutorial, Chainguard Timeteller, will provide the user with the local time in ten randomly chosen international timezones. It also displays the time in UTC and in the user's local timezone. The application depends on `pytz`, allowing us to draw from the Olson tz database in selecting our random timezones.

Let's start by creating our main application script. First, open your terminal and run the below line to create a new folder called `timeteller` in your home directory:

```
mkdir ~/timeteller && cd ~/timeteller
```

Open a new file called `main.py` in your preferred text editor. We'll use the widely available Nano editor in this tutorial.

```
nano main.py
```

Copy the following Python code into `main.py`:

```
from datetime import datetime
from time import tzname
from random import sample
from pytz import timezone, common_timezones


def tz_aware_now(tz=timezone('utc')):
    """Create a timezone-aware datetime for the current time and a provided timezone. Default
    now = datetime.now(tz)

    return now


def pretty_print_time(t):
    """Given a datetime object, return a human-readable and nicely-formatted time string."""

    # Generate an ordinal suffix, i.e. "th" or "st" based on the day
    day = t.day
    day_ordinal_suffix = str(day) + (
        "th"
        if 4 <= day % 100 <= 20
        else {1: "st", 2: "day", 3: "rd"}.get(day % 10, "th")
    )

    date_pretty = " ".join([t.strftime("%-I:%M %p on %B"), day_ordinal_suffix])

    return date_pretty


def generate_timezone_message():
    """Create the message to display to a user. The message includes a greeting, the current

    local_time = tz_aware_now(tz_aware_now().astimezone().tzinfo)

    random_timezones = sample(common_timezones, 10)

    timezones_map = {
        zone: pretty_print_time(tz_aware_now(timezone(zone)))
        for zone in random_timezones
    }

    printable_timezones = [
        " ".join(["    ", "▸", zone, timezones_map[zone]]) for zone in timezones_map
    ]

    message = "\n".join(
        [
            "\nWelcome to Chainguard Timeteller! \n",
            f"🌍 The current time in UTC is {pretty_print_time(tz_aware_now())}.",
            f"❓ Your current timezone is {tzname[0]}.",
            f"🕛 Your local time is {pretty_print_time(local_time)}.\n",
            "Local times from ten randomly chosen timezones around the world:\n",
            *printable_timezones,
        ]
    )

    return message


if __name__ == "__main__":
    print(generate_timezone_message())
```

If you're using Nano, you can save the file by pressing `Control-x`, `y`, and `Enter` in sequence.

In this script, we define functions to return the current time in a specific timezone, generate nicely-formatted lines for each region, and pull together a message to the user. When run directly, the script prints the generated message, including the time in ten randomly selected timezones, to the console. The code depends on a library, pytz, not in the standard library, and we'll have to install it during our build process.

Because our code depends on `pytz`, a package not in Python's standard library, we'll also need to specify our dependencies. Open a `requirements.txt` file using your text editor:

```
nano requirements.txt
```

Copy the below into the file.

```
pytz==2024.1
```

Here, we specify the version of `pytz` we want to use. Once you're done, save the file.

Before we build our container, let's test that our script works. First, install our dependency using the pip package manager:

```
pip install -r requirements.txt
```

Depending on your system, you may need to use the `pip3` command instead of the `pip` command.

Once `pytz` has installed, run the script with the below command:

```
python main.py
```

Depending on your system, you may need to use the `python3` command instead of the `python` command. You should receive output similar to the following:

```

Welcome to Chainguard Timeteller!

🌍 The current time in UTC is 6:07 PM on February 18th.
❓ Your current timezone is EST.
🕛 Your local time is 1:07 PM on February 18th.

Local times from ten randomly chosen timezones around the world:

     ▸ Europe/Dublin 6:07 PM on February 18th
     ▸ Africa/Lagos 7:07 PM on February 18th
     ▸ America/Fortola 2:07 PM on February 18th
     [...]
```

When you see the above output with local times in ten randomly-selected timezones, you'll know the application is working. You're ready to containerize Timeteller using a base image from Chainguard Images.

Multi-Stage Build Using Chainguard Images

Now that we have our application in place, we're ready to containerize it using a multi-stage build process. This build process works as follows:

1. We begin the build by pulling a development version of the `python-latest` Chainguard Image as our base image.
2. We copy the `requirements.txt` file to the image, activate our virtual environment, and install our dependency using `pip`.
3. We now pull the minimal runtime version of the `python-latest` image. This image does not contain pip or an interactive shell.
4. We copy our virtual environment (now with access to our dependency) from the dev image to the minimal runtime image.
5. We activate our virtual environment on the minimal runtime image and run the application.

Create a file named `Dockerfile` in your `timeteller` folder:

```
nano Dockerfile
```

Copy the following build instructions to the Dockerfile:

```
FROM cgr.dev/chainguard/python:latest-dev as builder

ENV LANG=C.UTF-8
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1
ENV PATH="/timeteller/venv/bin:$PATH"

WORKDIR /timeteller

RUN python -m venv /timeteller/venv
COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

FROM cgr.dev/chainguard/python:latest

ENV TZ="America/Chicago"

WORKDIR /timeteller

ENV PYTHONUNBUFFERED=1
ENV PATH="/venv/bin:$PATH"


COPY main.py ./
COPY --from=builder /timeteller/venv /venv

ENTRYPOINT [ "python", "/timeteller/main.py" ]
```

Change the timezone in the line `ENV TZ="America/Chicago"` to your own local timezone. (Without providing this information, the detected timezone in the container would be UTC.)

Save the file. We should now be ready to perform the build. Run the following:

```
docker build . -t timeteller
```

This will build the image from the instructions in our Dockerfile and tag it with the name `timeteller`.

If the image build is successful, you're ready to run it with the following:

```
docker run --rm timeteller
```

You should see the output from our application as above, including the ten randomly selected local times. Congratulations! You've successfully containerized and run an application using Chainguard Images in a multi-stage build process.

The final image based on python:latest does not contain either pip or interactive shells such as `sh` or `bash`. Since package managers and shells are common vectors for attackers, having neither as part of our runtime image decreases the attack surface of our application in production. The multi-stage build process allows us to use tools such as shells and package managers during development while allowing us to keep our production image securely minimal.

Advantages of Chainguard Images

Chainguard Images provide a happy medium between superminimal images such as scratch and more complex distribution-based images such as Alpine or Debian. Chainguard Images aim specifically to reduce complexity, intentionally including only those software components necessary for runtime. Further, Chainguard Images are based on a distroless philosophy, meaning that they strip out additional software components traditionally associated with a distribution. Typically, a Chainguard Image contains only an application runtime, root certificates, a minimal file structure, and a small number of core system libraries.

Each Chainguard Image comes with a comprehensive SBOM (Software Bill of Materials). This allows users of Chainguard Images to check against known vulnerabilities, adhere to the legal terms of software licenses, and ensure software integrity.

Finally, the focus on minimal builds results in significantly fewer CVEs on your runtime images. Before we end this tutorial, let's scan for CVEs in our Timeteller image using an industry-standard tool, Docker Scout.

To use Docker Scout, you'll first have to have a Docker Hub account. Follow the installation instructions for Docker Scout on GitHub. Once Docker Scout is installed, you can sign in to Docker Hub on the command line with the `docker login` command.

Once we have Docker Scout installed, we can use it to scan for vulnerabilities with the following command:

```
docker scout cves timeteller
```

Running Docker Scout on our Timeteller image (built from a Chainguard Image) on February 18th, 2024 produced the following report:

```
    ✓ Image stored for indexing
    ✓ Indexed 32 packages
    ✓ No vulnerable package detected


## Overview

                    |                   Analyzed Image
--------------------+-----------------------------------------------------
  Target            |  timeteller:latest
    digest          |  0cca410be7e4
    platform        |  linux/amd64
    vulnerabilities |    0C     0H     0M     0L
    size            |  28 MB
    packages        |  32


## Packages and Vulnerabilities

  No vulnerable packages detected
```

As you can see, no CVEs were detected in the Timeteller image, a relatively rare outcome in the fast-paced world of container vulnerabilities and exposures. While your results with Chainguard Images won't always be this free of vulnerabilities, using Chainguard Images as your base will reduce your CVE incidence rate by 80% compared to comparable industry alternatives.

In this tutorial, you containerized a Python application with Chainguard Images in a multi-stage build process. This resulted in a runtime image with only the software components required to run our application. This focus on reducing software complexity resulted in a runtime image with a demonstrably low number of CVEs—zero in this case. Now that you understand the advantages of building your production infrastructure on Chainguard Images, you're ready to go forth and secure your own production environment.