RedHawk

Creating and Running Docker Containers on RedHawk Linux Systems

Info@concurrent-rt.com
www.concurrent-rt.com
(800) 666.4544 or (954) 974-1700

Creating and Running Docker Containers on RedHawk Linux Systems

Overview

This document provides instructions for creating and running a simple “Hello, World!” application container on systems running RedHawk™ Linux® 8.x. Both RedHawk Linux systems utilizing a RHEL base distribution and an Ubuntu® base distribution are covered.

Introduction

A container is a unit of software that houses your code and all its dependencies so your application can run reliably in different computing environments.

Docker is a container solution that uses a client-server architecture where the client talks to the Docker daemon. The Docker daemon handles building, running and distributing Docker containers.

Podman is a daemonless container solution which allows regular users to run containers without interacting with a root-owned daemon. This allows for rootless containers where users can create, run and manage containers without requiring processes with admin privileges.

Privileged containers are containers that have root access to host resources. This solution may be desired for containers that require direct hardware access or access to host filesystems. The container’s root user is mapped to the host’s root user. This is considered unsafe and, if improperly set up, can allow the container to be abused and fall victim to “container escape.” Malicious actors can leverage the container’s vulnerabilities to breach isolation and access host resources with privilege escalation.

Docker and Podman containers are unprivileged by default. They make use of namespaces to isolate containers from the host. The container’s root user is mapped to an unprivileged user outside of the container and only has rights to the resources that the container is assigned. The container will have its own process namespace and users can use cgroups to manage resource limits of the container. Both Docker and Podman provide a `--privileged` option to allow for the creation of privileged containers. By default, Podman containers will not be given any additional access beyond the processes launched by the user. The use of privileged containers is not recommended.

Docker Example

The instructions in this section explain how to create and run containers using Docker on RedHawk systems that are utilizing an Ubuntu base distribution.

Step 1, install the Docker software.

```
$ sudo apt install docker.io
$ sudo snap install docker
```

Step 2, setup the container build directory. This directory is where we will store all the files needed to create our container image.

```
$ mkdir build
```

Step 3, create the hello program source. This simple program will become what is executed when launching our container.

```
$ cd /path/to/build
$ cat <<EOF > hello.c
#include <stdio.h>

const char message[] =
      "\n"
      "Hello World!\n"
      "\n"
      "Running a custom container on RedHawk!\n"
      "\n";

int main() {
      printf("%s", message);
      return 0;
}
EOF
```

Step 4, compile the hello program to create the hello binary that will be placed into the container image.

```
$ sudo apt install gcc
$ cd /path/to/build
$ gcc -o hello -static hello.c
```

Step 5, create a docker ignore file. This file lists all files and directories to exclude from the container image build.

```
$ cat <<EOF > .dockerignore
# Files to exclude from image
hello.c
EOF
```

Step 6, create a Dockerfile. The Dockerfile contains each instruction needed to setup the container image.

```
$ cat <<EOF > Dockerfile
# syntax=docker/dockerfile:1
FROM scratch
ADD hello /
CMD ["/hello"]
EOF
```

The significance of each line in the Dockerfile is explained below:

```
FROM scratch
```

This line tells Docker that this container will begin as a minimal image and the next command will be the first filesystem layer in the image.

```
ADD hello /
```

This line tells Docker to copy over the hello binary to the root directory of the image.

```
CMD ["/hello"]
```

This line tells the image builder what the default program to execute will be when launching the container. In this case we will launch our hello application that was copied into the container.

Step 7, build a Docker image using the Dockerfile. Now that all the pieces are in place, we tell Docker to create our container image based on the instructions in our Dockerfile. Docker will automatically look for a Dockerfile in the directory supplied. We name the image hello-redhawk with a tag indicating this is the latest build.

```
$ sudo docker build -t hello-redhawk:latest /path/to/build
```

You can view all created images with the following command:

```
$ sudo docker images
```

Sample output is shown as follows:

```
REPOSITORY      TAG       IMAGE ID       CREATED          SIZE
hello-redhawk   latest    d26523e725da   34 seconds ago   872kB
```

Step 8, create a container from the hello-redhawk image. We create a container called mycontainer and supply the name:tag of the image we created earlier.

```
$ sudo docker container create -i -t --name mycontainer hello-redhawk:latest
```

You can view all created Docker containers with the following command:

```
$ sudo docker ps -a
```

Sample output is shown as follows:

```
CONTAINER ID  IMAGE                 COMMAND   CREATED         STATUS    PORTS  NAMES
a76a20310f25  hello-redhawk:latest  "/hello"  6 seconds ago   Created          mycontainer
```

Step 9, start the container.

```
$ sudo docker container start --attach -i mycontainer
```

You should see the following output:

```
Hello World!

Running a custom container on RedHawk!
```

Podman Example

The instructions in this section explain how to create and run containers using Podman on RedHawk systems that are utilizing a RHEL base distribution.

Because RHEL does not support static linking, this example uses an existing RHEL container image as a base for our application container. We will copy hello.c into the image and compile it there. The program will run in an isolated RHEL environment while still making use of the RedHawk host kernel.

This example will require gcc to be installed in the container. Depending on the RHEL container, you may be required to attach a valid Red Hat subscription to the host.

Step 1, install the Podman software.

```
$ sudo dnf install podman
```

Step 2, pull in the RHEL 8 base container image. The Red Hat Universal Base Image 8 is designed to be the base layer for any RHEL 8 container images.

```
$ podman pull ubi8
```

Note that Red Hat provides several other containers to serve more specific purposes. Depending on the registry (e.g., redhat.io), Red Hat may require registry authentication to pull the container image. If a RHEL clone is sufficient, users can search and pull images from other registries that don't require authentication. For example, the following commands will pull in a Rocky 8 container:

```
$ podman search rockylinux
$ podman pull docker.io/library/rockylinux:8
```

You can view all downloaded Podman containers with the following command:

```
$ podman images
```

Sample output is shown as follows:

```
REPOSITORY                        TAG     IMAGE ID      CREATED     SIZE
registry.access.redhat.com/ubi8   latest  1b7b40f4f1ee  5 days ago  227 MB
docker.io/library/rockylinux      8       8cf70153e062  5 days ago  202 MB
```

Step 3, setup the container build directory. This directory is where we will store all the files needed to create our container image.

```
$ mkdir build
```

Step 4, create the hello program source. This simple program will become what is executed when launching our container.

```
$ cd /path/to/build
$ cat <<EOF > hello.c
#include <stdio.h>

const char message[] =
      "\n"
      "Hello World!\n"
      "\n"
      "Running a custom container on RedHawk!\n"
      "\n";

int main() {
      printf("%s", message);
      return 0;
}
EOF
```

Step 5, create a container ignore file. This file lists all files and directories to exclude from the container image build. Like the Docker example above, we create an ignore file, however Podman requires a .containerignore file. Also, because we're copying hello.c into the container, we don't have anything to add to our ignore file in this example.

```
$ cd /path/to/build
$ touch .containerignore
```

Step 6, create a Containerfile. The Containerfile contains each instruction needed to setup the container image.

```
$ cat <<EOF > Containerfile
FROM ubi8:latest
RUN dnf -y install gcc
ADD hello.c /
RUN gcc -o /hello /hello.c
RUN rm -f /hello.c
CMD ["/hello"]
EOF
```

The significance of each line in the Containerfile is explained below:

`FROM ubi8:latest`

This line tells Podman to use the ubi8:latest image we pulled earlier as the base for our container image build. If the image does not exist on our system, Podman will attempt to pull the best fit from the registry.

`RUN dnf -y install gcc`

This line installs the gcc packages needed to compile our program.

`ADD hello.c /`

This line copies hello.c into the root directory of our image.

`RUN gcc -o /hello /hello.c`

This line compiles the hello program.

`RUN rm -f /hello.c`

This line performs some cleanup. Now that the hello binary exists, we can remove hello.c.

`CMD ["/hello"]`

This line tells the image builder what the default program to execute will be when launching the container. In this case we will launch our hello application that was compiled inside the container.

Step 7, build a container image using the Containerfile. Now that all the pieces are in place, we tell Podman to create our container image based on the instructions in our Containerfile. Podman will automatically look for a Containerfile in the directory supplied. We name the image hello-redhawk with a tag indicating this is the latest build.

```
$ podman build -t hello-redhawk:latest /path/to/build
```

You can view all created images with the following command:

```
$ podman images
```

Sample output is shown below. Notice that both the base Red Hat ubi8 image and our hello-redhawk image are shown.

```
REPOSITORY                      TAG     IMAGE ID      CREATED         SIZE
localhost/hello-redhawk         latest  9990b933279b  39 seconds ago  366 MB
registry.access.redhat.com/ubi8 latest  1b7b40f4f1ee  5 days ago      227 MB
docker.io/library/rockylinux    8       8cf70153e062  3 months ago    202 MB
```

Step 8, Create a container from the hello-redhawk image. We create a container called mycontainer and supply the name:tag of the image we created earlier.

```
$ podman container create -i -t --name mycontainer hello-redhawk:latest
```

You can view all created Podman containers with the following command:

```
$ podman ps -a
```

Sample output is shown as follows:

```
CONTAINER ID  IMAGE                          COMMAND  CREATED        STATUS   NAMES
bf3fbe99dadf  localhost/hello-redhawk:latest  /hello   4 seconds ago  Created  mycontainer
```

Step 9, start the container.

```
$ podman container start --attach -i mycontainer
```

You should see the following output:

```
Hello World!

Running a custom container on RedHawk!
```

Docker Compatibility

Podman users can optionally install a Docker compatibility package. For users coming from a Docker environment, the package emulates the Docker CLI. This allows users to type Docker commands that will be executed using Podman. Issue the following command to install the compatibility package:

```
$ sudo dnf install podman-docker
```

Container Image Layers

In the examples above, each step of our Dockerfile/Containerfile adds a layer to the image.

Knowing this information, you can optimize the container build process and use certain layers of the image build to create different images.

Recall the output of the image build for the Podman example above:

```
$ podman build -t hello-redhawk:latest /path/to/build
STEP 1/6: FROM ubi8:latest
STEP 2/6: RUN dnf -y install gcc
...
Installing:
 gcc
...
Complete!
--> dce84915b9f
STEP 3/6: ADD hello.c /
--> f08b9da848d
STEP 4/6: RUN gcc -o /hello /hello.c
--> 1238d0dc976
STEP 5/6: RUN rm -f /hello.c
--> cefc2da146e
STEP 6/6: CMD ["/hello"]
COMMIT hello-redhawk:latest
--> 9990b933279
Successfully tagged localhost/hello-redhawk:latest
9990b933279ba46a1edc1c182cc8b066a205bb5a7700a76cfcd78301cc139ca6
```

Notice how each step of the image build generates an image ID. Compare these image IDs to the output of the following:

```
$ podman images --all
REPOSITORY                          TAG         IMAGE ID      CREATED         SIZE
localhost/hello-redhawk             latest      9990b933279b  5 minutes ago   366 MB
<none>                              <none>      cefc2da146eb  5 minutes ago   366 MB
<none>                              <none>      1238d0dc9761  5 minutes ago   366 MB
<none>                              <none>      f08b9da848d9  5 minutes ago   366 MB
<none>                              <none>      dce84915b9fc  5 minutes ago   366 MB
registry.access.redhat.com/ubi8     latest      1b7b40f4f1ee  5 days ago      227 MB
docker.io/library/rockylinux        8           8cf70153e062  3 months ago    202 MB
```

The image ID listed can be specified for future image builds or to create containers based on that layer. You can also assign a name and tag to the image using the Podman tag command.

For example, image ID dce84915b9fc can be used to create a Red Hat container with gcc already installed and renamed to rhel8-gcc:latest. It can also be used in future Containerfiles as a base image with the `FROM` instruction.

Further Documentation

The following websites contain additional container documentation.

- https://hub.docker.com/
- https://docs.docker.com/
- https://catalog.redhat.com/
- https://docs.podman.io/

About Concurrent Real-Time

Concurrent Real-Time is a leading provider of high-performance real-time computer hardware and software for commercial and government markets worldwide. The Company's RedHawk Linux solutions deliver hard real-time performance for the most sophisticated X-In-the-Loop simulation, high-speed data acquisition, process control and low-latency transaction processing applications. With over 55 years of real-time expertise, Concurrent Real-Time is headquartered in Pompano Beach, FL, and provides sales and support from offices throughout North America, Europe and Asia. Concurrent Real-Time is part of HBK's Virtual Test Division.

More Information

www.concurrent-rt.com

concurrent REAL-TIME