Containers

Building better container images

by Amit Khanal and Sajjan AVS | on 24 JUL 2023 | in Amazon Elastic Container Registry, Amazon Elastic Container Service, Amazon Elastic Kubernetes Service, Compute, Containers, Technical How-to | Permalink | Share

Introduction

Many applications built today or modernized from monoliths are done so using microservice architectures. The microservice architecture makes applications easier to scale and faster to develop, which enables innovation and accelerating time-to-market for new features. In addition, microservices also provide lifecycle autonomy enabling applications to have independent build and deploy processes, which provides technological freedom such that they can be implemented in different programming languages and provide scaling flexibility to scale up or scale down independently based on workload utilization.

While microservices provide a lot of flexibility, the process of building and deploying them, ensuring the right application versions, and required dependencies are used is a tedious process. This is where containers come in.

Microservices can be packaged into a single lightweight and standalone executable artifact called container image that includes everything to run an application. This process of packaging a microservice into container image is called containerization.

Containerization offers a lot of benefits, such as portability, which allow containers to be deployed to different infrastructures. It also offers fault isolation, which ensures that different containers are running as isolated process within their own user space in the host OS so that one container's crash or failure wouldn't impact the other and provide ease-of-management for deployment and version management.

With the benefits that are offered via microservices and subsequent containerization, creation of container images have increased at a rapid scale. As the use of containerized applications continue to grow, it is important to ensure that your container images are optimized, secure, and reliable. Taking these tenets into account in this post, we discuss best practices for building better container images for use on Amazon Elastic Container Service (Amazon ECS), Amazon Elastic Kubernetes Service (Amazon EKS), and other services. For the container image repositories, we focus on Amazon Elastic Container Registry (Amazon ECR).

Building better images

We'll look at the following approaches to build better images. While this isn't an exhaustive list, these topics provide a good base for your image builds, and you can adopt them as you wish.

Walkthrough

Use trusted images and source

Using a base image from a trusted source can improve the security and reliability of your container. This is because you can be confident that the base image has been thoroughly tested and vetted. It can also reduce the burden of establishing provenance, because you only need to consider the packages and libraries that you include in your image, rather than the entire base image. Here is an example creating a Dockerfile using the official Python base image from the Amazon ECR repository. In fact, all of the Docker official images are available on Amazon ECR public gallery.

Note: The code samples in the blog are for Docker. If you don't have Docker, then please refer to Docker installation guide for information about how to install Docker on your particular operating system. Alternatively, you can also consider using Finch an open source client for container development.

```
FROM public.ecr.aws/docker/library/python:slim

# Install necessary packages
RUN pip install flask

COPY app.py /app/

ENTRYPOINT ["python", "/app/app.py"]
```

Here's another example of using latest version (as of the data of this post) of Amazon Linux image, which is a secure and lightweight Linux distribution provided by AWS.

```
FROM public.ecr.aws/amazonlinux/amazonlinux:2023
```

It is important to keep images up-to-date by regularly updating to latest secure versions of the software and libraries included in the image. As new versions of the images get created, they should be explicitly tagged with versions such as v1, v2, rc_03122023, etc instead of tagging as latest. Using explicit tags instead of latest tag could prevent situations where the image with latest tag isn't actually updated and instead gives a false appearance that the image contains the latest version of the application. If you're confident in your automation, then vanity tags such as latest or prod might be acceptable to use, but avoiding them also reduces ambiguity. This can avoid confusion about which application version may actually being used.

Once images are created, they can be pushed into the Amazon ECR repository for secure storage and highly available distribution. Amazon ECR encrypts data at rest and offers lifecycle management, replication, and caching features. Amazon ECR can also scan the images to help in identifying software vulnerabilities through Basic and Enhanced Scanning. Stored images from Amazon ECR can then be pulled and run by services such Amazon ECS, Amazon EKS, or other services and tools.

Here is an example of using AWS Command Line Interface (AWS CLI) and Docker CLI to pull a versioned image from Amazon ECR.

Step 1 – Authenticate your Docker client to the Amazon Linux Public registry. Authentication tokens are valid for 12 hours. For more information, see Private registry authentication. Alternatively, you can also use Amazon ECR Docker Credential Helper, which is a credential helper for the Docker daemon that makes it easier to use Amazon Elastic Container Registry. Amazon ECR Docker credential helper automatically gets credentials for Amazon ECR on docker push and docker pull. Note that this would only be required for Amazon ECR, but ECR Public doesn't need authentication.

```
$ aws ecr-public get-login-password --region us-east-1 | docker login --username AWS --passw
```

Step 2 – Pull the Amazon Linux container image using the docker pull command. To view the Amazon Linux container image on the Amazon ECR Public Gallery, see Amazon ECR Public Gallery – amazonlinux.

```
docker pull public.ecr.aws/amazonlinux/amazonlinux:2023
```

Step 3 – Run the container locally.

```
docker run -it public.ecr.aws/amazonlinux/amazonlinux /bin/bash
```

Bonus

While Amazon Linux is a secure, trusted, and lightweight container image, AWS also offers Bottlerocket which is a Linux-based open-source operating system that is purpose-built for running containers. Bottlerocket includes only the essential software required to run containers, and ensures that the underlying software is always secure. Additionally, Bottlerocket is available at no cost as an Amazon Machine Image (AMI) for Amazon Elastic Compute Cloud (Amazon EC2) and can be used on Amazon EKS and Amazon ECS setups.

Sign container images

Container image signing can help verify that the trusted images you have selected and vetted are in use throughout your build pipelines and deployments. This process involves trusted parties cryptographically signing images so that they can be verified when used. This can be used to also sign and verify images throughout your organization.

Container image signing is fairly lightweight. Because container images and runtimes have built-in integrity checks and all image content is immutable, signing solutions can simply sign image manifests. Signatures are stored alongside images in the registry, and at any point in time a consumer of the image can retrieve its signature and verify against a trusted publisher's identity or public key.

It is a good practice to sign and verify container images as part of overall security practices, and verifying public content establishes trust in content authenticity. With signed images, you can implement a solution that blocks images from running in your container environment unless they can be verified as trusted. This not only guarantees the authenticity of container images but also reduces the need for additional work to validate the container images in other ways prior to their use.

For a full walkthrough of such a solution, see the recent launch of Container Image Signing with AWS Signer and Amazon EKS.

Limit the number of layers

It is a good practice to limit the number of layers in your container images. Having a large number of layers can increase the size and complexity of the image, which can make it more difficult to manage and maintain.

For example, consider the following Dockerfile:

```
FROM public.ecr.aws/docker/library/alpine:3.17.2

# Install all necessary dependencies in a single layer
RUN apk add --no-cache \
  curl \
  nginx \
  && rm -rf /var/cache/apk/*

# Set nginx as the entrypoint
ENTRYPOINT ["nginx"]
```

In this example, we install all necessary dependencies in a single layer, and then remove the cache to reduce the number of layers in the final image. This results in a smaller and more efficient image that is easier to manage and maintain.

Make use of multi-stage builds

A multi-stage build is a technique that allows you to separate the build tools and dependencies from the final image content. This can be beneficial for several reasons:

- Reducing image size: By separating the build stage from the runtime stage, you can include only the necessary dependencies in the final image, rather than including all of the build tools and libraries. This can significantly reduce the size of your final image as and reduce the total number of image layers.
- Improved security: By including only the necessary dependencies in the final image, you can reduce the attack surface of the image. This is because there are fewer packages and libraries that could potentially have vulnerabilities.

Here is an example of a multi-stage build in a Dockerfile:

```
# Build stage
FROM public.ecr.aws/bitnami/golang:1.18.10 as builder
WORKDIR /app
COPY . .
RUN go build -o app .

# Runtime stage
FROM public.ecr.aws/amazonlinux/amazonlinux:2023
RUN yum install -y go
WORKDIR /app
```

```
COPY --from=builder/app ./
ENTRYPOINT ["./app"]
```

In this example, we first use the golang:1.18.10 image as the build stage, in which we copy the source code and build the binary. Then, in the runtime stage, we use the amazonlinux:2023 container image, which is a minimal base image, and copy the built binary from build stage. This results in a smaller and more secure final image that only includes the necessary dependencies.

Secure your secrets

Secrets management is a technique for securely storing and managing sensitive information such as passwords and encryption keys and also externalizing environment specific application configuration. Secrets should not be stored in an image but instead stored and managed externally in service such as AWS Secrets Manager. For secrets that are required for your application during runtime, you can retrieve them from AWS Secrets Manager in real-time or use container orchestrator service such as Amazon ECS or Amazon EKS to mount secrets as volumes on the container such that the application can read from the mounted volume. Details on how secrets can be used on Amazon EKS can be found here and how secrets can be used on Amazon ECS can be found here.

If secrets are required during build time, then you can inject ephemeral secrets using Docker's secret mount-type.

For example, the following Dockerfile uses a multi-stage build process where the first stage called builder_image_stage uses mount=type=secret to load the AWS credentials. The second stage then uses the build artifacts from first builder_image_stage. The resulting final image from second stage won't contain any build tools or secrets from the builder_image_stage thereby not maintaining secrets on it.

```
FROM public.ecr.aws/amazonlinux/amazonlinux:2023 AS builder_image_stage

WORKDIR /tmp

RUN yum install -y unzip && \
    curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip" && \
    unzip /tmp/awscliv2.zip && \
    /tmp/aws/install

RUN --mount=type=secret,id=aws,target=/root/.aws/credentials \
    aws s3 cp s3://... /app/...

# cd /app
# make

FROM public.ecr.aws/amazonlinux/amazonlinux:2023

COPY --from=builder_image_stage /app/files_from_builder_image /app
```

Reduce the attack surface

Images, by default, are not secure and can potentially be vulnerable for attacks. Following approaches can help mitigate the attack surface.

- Build from scratch

Most images are based on an existing base image such as Amazon Linux, ubuntu, alpine, etc. However, if you need to create a completely new base image with your own distribution, or if your application or service can be deployed as a single statically linked binary, you can use the special scratch image as a starting point.

The scratch base image is empty and doesn't contain any files. All of the official base images are actually built on top of scratch, and you can place any content on subsequent image layers. Using FROM scratch is basically a signal to the image build process that the layer built by the next command in the Dockerfile should be the first layer of the image.

Scratch gives you the ability to completely control the contents of your image starting from the beginning. It is important to note that using scratch, you'll be responsible for the entire toolchain and all content in the resulting container image. Also, container scanning solutions, such as Amazon ECR Enhanced Scanning, do not support vulnerability scanning of images without an OS packaging system in place. If your container images have many environmental dependencies or use interpreted languages or need vulnerability scanning, then using a minimal base image instead of scratch may be better.

Here are a few examples of using scratch.

The example below adds hello-world executable to the image as a first layer.

```
FROM scratch
ADD hello-world /
CMD ["/hello-wold"]
```

The second example copies the root filesystem to the root of the image as a first layer. The container uses this root filesystem to spin off processes from binaries and libraries in that directory.

```
FROM scratch
ADD root-filesystem.tar.gz /
CMD ["bash"]
```

- Remove unwanted packages

It is a good practice to remove any unwanted or unnecessary packages from your container image to reduce the attack surface and size of the image. For example, if your application does not require certain libraries or utilities, you should remove them from the image.

Here is an example on how to remove package manager that may be unnecessary on a final image.

```
FROM public.ecr.aws/ubuntu/ubuntu:20.04_stable

# Remove package manager
RUN apt-get -y --allow-remove-essential remove apt
```

- Use a Minimal Base Image

Choosing a minimal base image, such as a Distroless image, can help to restrict what is included in your runtime container. Distroless images only include your application and its runtime dependencies that is necessary for the application to run. As such they don't contain package managers or shells. This can improve the security and performance of your container by reducing the attack surface, as there are fewer packages and libraries that could potentially have vulnerabilities or add unnecessary overhead.

Here is an example of using the Distroless base image for a Go application. This base image contains a minimal Linux, glibc-based system and is intended for use directly by mostly-statically compiled languages like Go, Rust or D. For more information, see the GitHub documentation on base distroless image.

```
FROM gcr.io/distroless/base

COPY app /app

ENTRYPOINT ["/app"]
```

Secure image configuration

Images by default may not be secure and can allow privileged access. It is best to ensure that they are setup with least privileged access and remove configurations that are unnecessary for your application. One such approach is to run containers as a non-root user.

Processes within Docker containers have root privileges by default to both the container and the underlying host. This opens up the container and host to security vulnerabilities that can be exploited.

To prevent these vulnerabilities, it is a good practice to minimize the privileges granted to your container images, only giving the necessary permissions to perform required tasks. This helps to reduce the attack surface and potential for privilege escalation. On Dockerfile, you can use USER directive to specify a non-root user to run the container, or use the securityContext field in the Kubernetes pod specification to specify a non-root user and set group ownership and file permissions. Furthermore, on Kubernetes with Amazon EKS, you can also limit the default capabilities assigned to a POD as explained on EKS best practices Guides.

For example, consider the following Dockerfile:

```
FROM public.ecr.aws/amazonlinux/amazonlinux:2023

# 1) install package for adduser
# 2) create new non-root user named myuser
# 3) create app folder
# 4) remove package for adduser
RUN yum install -y shadow-utils && \
    adduser myuser && \
    mkdir /app && \
    yum install -y remove shadow-utils

COPY . /app

# Set group ownership and file permissions
RUN chown -R myuser:myuser /app

# Run as non-root user
USER myuser
```

In this example, we create a non-root user named myuser and set the group ownership and file permissions of the application files to this user. We then set the entry point to run as the myuser user, rather than as root. This helps to reduce the potential for privilege escalation and increase the security of the container.

Tag Images

When building container images, it is recommended to use tags to identify them. If the image is built without any tags, then Docker will assign latest as the default tag. For instance, when building an image, you may use the following command:

```
docker build -t my-pricing-app .
```

With this command, Docker automatically assigns the latest tag to the image, since no specific tag was provided. However, building an updated image using the same command assigns the latest tag to the new image. This can cause ambiguity and potential deployment of outdated or unverified versions. To avoid this, consider tagging the image descriptively as below:

```
docker build -t my-pricing-app:<git-commit-hash>-1.0-prod .
```

This command builds the my-pricing-app image tagged as <git-commit-hash>-1.0-prod. This allows specific version of the image to be deployed unlike latest tag. Additionally, you can also use the tag command to tag existing images.

It is important to note that Docker tags are mutable by design, which allows you to build a new image with an existing tag. If you need to have immutable tags, then you can consider using Amazon Elastic Container Registry (Amazon ECR) to store your images. Amazon ECR supports immutable tags, which is a capability that prevents image tags from being overwritten. This enables users to rely on the descriptive tags of an image as a reliable mechanism to track and uniquely identify images and also trust that they have not been tampered with. More details can be found at Image tag mutability – Amazon ECR.

Conclusion

In this post, we showed you how to build better container images using best practices. As adoption of containers increase, it becomes important to ensure the container images are secure, lightweight, and are built from trusted sources. The options described in this post can act as a starting point to building such images. Additionally, this post also shows how users can use a secure and fully managed Amazon ECR to manage container images.

TAGS: Amazon ECS, Amazon EKS, Amazon Elastic Container Registry (Amazon ECR), container images, docker, microservices