NASA Center for Climate Simulation Containers Best Practices

Jordan A. Caraballo-Vega, Kenneth E. Peck

NASA Center for Climate Simulation, NASA GSFC

April 6, 2021

Contents

1 Overview

This document describes a set of best practices for building and managing user-interactive containers. The main goal is to cover the basics of containers, together with techniques and practices that will help users and system administrators succeed at the time of building and running containers, while keeping our HPC environments secure. Some of these best practices are fundamental and required in order to run containers at the NCCS; others have been marked as optional.

1.1 Containers In A Nutshell

Containers are small versions of operating systems that are meant to speed up the process of shipping new versions of software. As part of Continuous Integration & Continuous Deployment (CI/CD), an application can be compiled and run from a container in any stage of the Software Development Life Cycle (SDLC) for testing or production. As part of DevOps, containers allow greater flexibility for the deployment of applications on immutable runtime systems providing full reproducibility. As part of data science, containers allow a means of sharing code and artifacts for research reproducibility.

1.2 Container Technologies

Most containers are deployed and managed with Docker or Kubernetes. Docker is a simpler way to manage containers and can be used in Swarm mode (multiple hosts with containers load balanced between them) or standalone (one host with multiple containers). Docker also has a feature called docker-compose where multiple containers can be specified for creation in one file called docker-compose.yml. This file is in YAML format and is human-readable.

Kubernetes is similar to Docker but is used for container orchestration. While Docker manages the container creation and management, Kubernetes is used for maintaining the lifecycle of the containers through tasks and orchestration. For example, Kubernetes can be used with Docker. In a Kubernetes environment, Docker would handle running the containers and Kubernetes is responsible for the creation, execution, removal and overall operations and lifecycle of the containers, typically managed in Kubernetes Pods. A similar solution is Docker Swarm.

RedHat also has its own commercial container technology called OpenShift. Based on Kubernetes, the idea of OpenShift is to make orchestration easier to administrators and developers using built-in RedHat tools. Cannonical, the company that distributes Ubuntu, has its own container technology called LXD and ships with all distros of Ubuntu. LXD containers are a bit different than what Docker and Kubernetes use and allow users to push or pull files from their containers, something that is typically not supported.

Singularity is another container platform created to run complex applications on HPC clusters in a simple, portable, and reproducible way. Singularity containers are executed as a single binary file based

container image, easy to move using existing data mobility paradigms and with no root owned daemon processes. Its security paradigm does not allow user contextual changes or root escalation; where the user inside the container is always the same user who started the container.

Only Singularity and Charliecloud containers are allowed for user-interactive applications at the NCCS at this time. Technologies such as Rootless Docker, Podman, Kubernetes and OpenShift are being evaluated for user-interactive and operational scenarios at this time. It is important to note that Docker containers can be converted to Singularity binaries, allowing a wider adoption of Singularity across different HPC Centers. It is always recommended to build Docker containers first, since these containers can be converted into singularity images due to the Open Container Initiative format (OCI). The NCCS currently provides ways of building both Docker and Singularity containers via CI/CD processes in order to test the reliability of the images before deployment. No Docker daemons are allowed in user-interactive systems, thus every Docker image is then converted into Singularity format for execution.

2 Best Practices for Building Containers

This section highlights best practices at the time of building container images. Each container technology offers a mechanism for building containers from configuration files. These best practices will allow users and system administrators to use containers to their fullest potential.

2.1 Only use golden images to start your container build process

One of the great advantages of Docker and Singularity is the sheer number of publicly available images, for all kinds of software. The availability of these images allow users to get started quickly, but can negatively affect both the creation and runtime of the container. Some examples of these are: not being able to control what software the public image includes/removes, multiple dependency layers, not being able to control vulnerabilities in production environments, and end of support from public images.

Recommendation: The NCCS has adopted a hybrid system where base images are taken from the public registry, and are modified through continuous integration to comply with Agency standards. These images are available in NCCS's internal container registry and should be used as base images for the construction of any container that will be run inside the NCCS. All NCCS-approved images are based off the following operating systems:

CentOS 8	Ubuntu 20 Bionic	Debian 10
CentOS 7	Ubuntu 18 Bionic	Debian 9
Alpine 3	SLES 12 SP5	

All containers have the necessary NASA baselines applied and include NASA certificates loaded into the certificate store. This makes it easier for containers to interact with NCCS GitLab or other NASA tools used within the container. In order to allow for more streamlined updates, all images use repositories available to the operating system. This means less compiling from source and quicker image builds. DockerHub provides a vast array of images, however it is required that all images be based on the NASA-approved gold images stored here. NCCS Containers repository (`https://gitlab.nccs.nasa.gov/nccs-ci/nccs-containers`) includes a vast number of images to use, from the base flavors to GoLang to PostgreSQL. These ensure that the user has the necessary NASA-specific security controls applied and provide a better level of security to the container, while maintaining the stability of using images that have gone through extensive regression testing for operational validation.

Base images are available through the format listed below. These images can be pulled directly into the user environment, or be used as base images to start building any application inside a container.

```
gitlab.nccs.nasa.gov:5050/nccs-ci/nccs-containers/base/**operating_system_version**
gitlab.nccs.nasa.gov:5050/nccs-ci/nccs-containers/**application**/nccs-**
    operating_system_version**-**application_version**

Examples:
gitlab.nccs.nasa.gov:5050/nccs-ci/nccs-containers/base/centos7
gitlab.nccs.nasa.gov:5050/nccs-ci/nccs-containers/base/debian10
gitlab.nccs.nasa.gov:5050/nccs-ci/nccs-containers/openjdk/nccs-debian9-openjdk8
```

2.2 Build container from configuration files

Build production containers from a configuration file instead of a sandbox that has been modified manually. This ensures greatest possibility of reproducibility and mitigates the "black box" effect. Additional problems related to filesystem discrepancies and compressing can affect locally modified sandbox environments.

Recommendation: Keep definition files under revision control environments. If needed, keep container definition files in sync with the semantic versioning of the software application in question. Do not modify containers manually, leverage definition files to secure stability and reproducibility.

2.3 Package a single application per container

Because a container is designed to have the same lifecycle as the application it hosts, each container should contain only one application. This allows for faster and easier maintenance of images, while adopting the use of microservices to distribute the load across containers. Adopting a microservices architecture offers additional resilience when running containers in operational environments. Applications might have different lifecycles, or be in different states, thus having them in separate containers allow for greater health monitoring. When a container starts, so should the application, and when the application stops, so should the container. This applies as well to data science and user-interactive containers where a single environment should be installed for a particular application.

Recommendation: The NCCS recommends to design each container with a single application installed. When a complex application requires multiple services to function, these services should be segmented into smaller containers using a microservices architecture. This will offer a greater flexibility at the time of deploying, upgrading, and modifying new releases of the application, while keeping redundancy over services. The following are examples of single application containers.

- Classic Apache/MySQL/PHP stack - best practice is to use two or three different containers: one for Apache, one for MySQL, and potentially one for PHP if you are running PHP-FPM.
- Data science container - best practice is to have a single conda environment per container, and a single directory structure of files and scripts related to the application hosted by the container.
- Application stack managed by Kubernetes - best practices is to use several single application containers for better resilience and monitoring. Kubernetes will perform health checks on individual containers and spawn new containers automatically to replace the afflicted applications.

2.4 Always install packages, programs, data, and files into OS locations

Container images have the ability to mount host directories into the container image. Thus, users must avoid adding data during build time in directories that are commonly binded at locations as /home, /att/nobackup/$user, and /tmp. This will help avoid conflicts when mounting local filesystem directories.

Recommendation: Avoid adding data or files to commonly binded directories such as /home, /att/nobackup/$user, and /tmp during build time. One example would be to store application files under /opt/app. Another example would be to clone the application software in the /opt directory.

```
git clone https://github.com/repository/application_name /opt/application_name
```

2.5 Always include package manager-specific update commands

One of the first lines of any custom image should be to update the image OS. While all images are rebuilt at least on a monthly basis to include all necessary security updates, it is important to also include those lines at the top. This will allow your image to start cleanly and without any known vulnerabilities for software flaws. Always perform extensive testing to identify bugs early in the SDLC.

Recommendation: For each flavor, include the following at the top. Notifications will be sent out by the NCCS Security team when expedite patches are required.

```
dnf -y update  # CentOS8
yum -y update  # CentOS7
apt-get update; apt-get -y upgrade; apt-get -y dist-upgrade # Ubuntu, Debian
apk update; apk upgrade  # Alpine
```

2.6 Build the smallest image possible

The size of your container image will have an effect on your download and installation processes. User-interactive containers that are small allow for easier manipulation and reproducibility. It is always best practice to avoid installing unnecessary packages and to remove any cache data that your application does not need. The base image is the one referenced in the FROM instruction in your Dockerfile, or in the From instruction on your Singularity Definition file. Every other instruction in the configuration file builds on top of this image.

The smaller the base image, the faster the container gets built and deployed. Alpine images are designed for small applications where not all Linux OS components are required. For example, the alpine:3.7 image is 71 MB smaller than the centos:7 image. Building smaller containers also reduces exposure to security vulnerabilities.

Recommendation: One of the easiest steps to keep the image small is to always clean up the package manager cache at the end. One of the last lines of your configuration file should be:

```
rm -rf /var/cache/dnf /var/cache/yum # CentOS 8
rm -rf /var/cache/yum # CentOS 7
rm -rf /var/cache/apt # Ubuntu, Debian
rm -rf /var/cache/apk # Alpine
```

While this step reduces the size of the image by many MB, it is important to select an appropriate base image. For simple running binaries, a base image like Alpine could suffice. For more complex applications such as ElasticSearch, a base image with all CentOS components would be the best option. Another strategy that can be used to lower the size of container images is the Build Pattern Mechanism.

This mechanisms allows compiling the software in one container, and removing all of the unnecessary artifacts from the final image that will allow the compiled software to run. The following is an example of this using Docker. The image goes from 712MB to 12MB of size.

```
FROM golang:alpine AS build-env
WORKDIR /app
ADD . /app
RUN cd /app && go build -o goapp

FROM alpine
RUN apk update && apk add ca-certificates && rm -rf /var/cache/apk/*
WORKDIR /app
COPY --from=build-env /app/goapp /app

EXPOSE 8080
ENTRYPOINT ./goapp
```

The main idea of this best practice is to remove any cache data that might be present within the container and is not necessary for other processes. An example of this is to remove the pip cache from a Python image, and/or any files that were downloaded and are no longer needed.

```
......
wget https://code.to.install.tar.gz
tar -xyz install.tar.gz
compiled code
rm -rf install.tar.gz install  # remove both the file and directory
....
```

2.7 Reduce the amount of clutter in the container image

To reduce the size of the container image, install only what is strictly needed inside it. Some packages are nice to have, like Vim, Nano or other packages. However, containers are meant to be small and headless. Keeping packages that are not necessary reduces the image size and any "bloat" the container may have. Because each instruction of the Dockerfile creates a layer, removing data from the image in a later step does not reduce the size of the overall image (the data is still there, just hidden in a deeper layer).

Recommendation:

```
## Bad Dockerfile Example

FROM ubuntu:20
RUN apt-get update && \
    apt-get install -y \
    [buildpackage]
RUN [build my app]
RUN apt-get autoremove --purge \
    -y [buildpackage] && \
    apt-get -y clean && \
    rm -rf /var/lib/apt/lists/*

## Good Dockerfile Example

FROM ubuntu:20
RUN apt-get update && \
    apt-get install -y \
    [buildpackage] && \
```

```
    [build my app] && \
    apt-get autoremove --purge \
    -y [buildpackage] && \
    apt-get -y clean && \
    rm -rf /var/lib/apt/lists/*
```

In the bad version of the Dockerfile, the [buildpackage] and files in /var/lib/apt/lists/* still exist in the layer corresponding to the first RUN. This layer is part of the image and must be uploaded and downloaded with the rest, even if the data it contains isn't accessible in the resulting image. In the good version of the Dockerfile, everything is done in a single layer that contains only your built app. The [buildpackage] and files in /var/lib/apt/lists/* don't exist anywhere in the resulting image, not even hidden in a deeper layer (Reference: Google Cloud Containers). Another way of lowering clutter is to use the Build Pattern Mechanism explained above.

2.8 Remove unnecessary tools

To protect container applications from attackers, the attack surface of the application should be reduced by removing any unnecessary tools. Most NASA approved images will already have taken care of this step. However, it is important to make sure that security, tracing, profiling, and debugging utilities are not included in your final image. For example, remove utilities like netcat, which you can use to create a reverse shell inside your system. Keep as few things as possible in your image; this best practice is true for any workload, even if it is not containerized. If you can compile your app into a single statically linked binary, adding this binary to the scratch image allows you to get a final image that contains only your app and nothing else. Sudo and SSH packages should not be part of any user-interactive container image.

Recommendation: Remove unnecessary tools. Depending on your base image, these should not be included by default. If necessary, the following commands will allow you to remove selected packages from your container image.

```
dnf remove package-name # CentOS 8
yum remove package-name # CentOS 7
apt purge package-name  # Ubuntu
apk del package-name    # Alpine
```

Another strategy is to use the Build Pattern Mechanism to avoid storing compilation binaries in the final image. These binaries sometimes include unnecessary information that should not be shared or included anywhere other than the build nodes.

2.9 Try to create images with common layers

Common container runtime tools rely on caching container layers to decrease build time. If you must download an image, the container runtime will download exclusively the layers that are not present in the local filesystem. This situation can occur if you previously downloaded another image that has the same base as the image you are currently downloading. Thus, the more your images have in common, the faster they are to download and execute.

Recommendation: Maintain a template of dedicated layers for each base image so they can be reused as required in order to decrease download and build time. Keep your layers similar when possible so existing local filesystem layers can be leveraged. In the example below, the first and second layers are the same, the third layer has the differing package. First and second layers will be executed at a faster

pace and will require less storage for the second container since the first container already executed the same tasks.

```
## Dockerfile for Container #1

FROM ubuntu:20
RUN apt-get update && \
    apt-get autoremove --purge && \
    apt-get -y clean && \
    rm -rf /var/lib/apt/lists/*

RUN apt-get install -y [buildpackage1]

## Dockerfile for Container #2

FROM ubuntu:20
RUN apt-get update && \
    apt-get autoremove --purge && \
    apt-get -y clean && \
    rm -rf /var/lib/apt/lists/*

RUN apt-get install -y [buildpackage2]
```

2.10 Optimize for the build cache

Many of the current container engines use build caches to speed up container building process. This cache mechanism can accelerate building and deploying container images. During a build, the container runtime, when possible, reuses a layer from the previous build done in the local filesystem, and skips a potentially costly step. This step is closely related to the above mentioned best practice.

Recommendation: To fully benefit from the container runtime build cache, you must position the build steps that change often at the bottom of the configuration file. Because a new container is usually built when a new release of software is available, add the source code to the image as late as possible in the configuration file. This will allow the most expensive layers to be built at the end and leverage previous common layers. You must update your repositories in the same RUN command as your package installation to get the latest metadata and packages.

2.11 Properly handle PID 1, signal handling, and zombie processes

Linux signals are the main way to control the lifecycle of processes inside a container. In order to properly handle the lifecycle of the container application, ensure that your application properly handles Linux signals. The most important Linux signal is SIGTERM to terminate the process. Additionally, SIGKILL and SIGINT signals might be appropriate to kill the process non-gracefully and to perform Ctrl+C operations when required. In practice, process identifiers (PIDs) are used as unique identifiers that the Linux kernel gives to each process. These PIDs are namespaced, thus the container PIDs will be mapped to the PIDs on the host.

There are two main concerns when working with PIDs between the container and the host. The first one is the use of the PID 1 by the container, and how Linux kernel handles signals differently for the process that has PID 1. This can result in the application being forced to shutdown non-gracefully causing sometimes interrupted writes, or unwanted alerts in the monitoring system. The second problem is handling orphaned processes inside the container. In a normal Linux host, this is resolved by the

systemd (or equivalent) application which has PID 1. This has to be resolved by the application inside the container or it can cause exhaustive resource consumption.

Recommendation: The best practice to address these two problems will depend entirely on the application hosted inside the container. Some examples are listed below.

- Run as PID 1 and register signal handlers: when using Docker, launch the application process with the CMD and/or ENTRYPOINT instructions in your Dockerfile. In this case, the best practice is to have the container launch a shell script when starting, which will prepare the environment and assign the appropriate PID 1.
- Use a specialized init system: normal init systems such as systemd or SysV can be installed to take care of these two problems. The init system called tini, is recommended for container purposes since it is less complex and smaller compared to the aforementioned solutions.
- Enable process namespace sharing in the container orchestration tool. Kubernetes, for example, uses a single process namespace for all the containers in that Pod. The Kubernetes Pod infrastructure container becomes PID 1 and automatically reaps orphaned processes, taking care of both problems.

```
## Using Docker CMD and ENTRYPOINT

FROM debian:9
RUN apt-get update && \
    apt-get install -y nginx
EXPOSE 80
CMD [ "nginx", "-g", "daemon off;" ]

## Using tini init system

# Add Tini
ENV TINI_VERSION v0.19.0
ADD https://github.com/krallin/tini/releases/download/${TINI_VERSION}/tini /tini
RUN chmod +x /tini
ENTRYPOINT ["/tini", "--"]

# Run your program under Tini
CMD ["/your/program", "-and", "-its", "arguments"]
# or docker run your-image /your/program ...
```

2.12 Keep sensitive data out of the image

The sensitivity of an image will require different mechanisms to secure the data bound to the container. Any application-specific usernames or passwords or images that contain ITAR data shall not be in the public DockerHub repository, or in any public NCCS GitLab repository. These containers shall be stored in a private repository and shall also not be built using the public build servers. If the file contains a generic username or password that is not used in production, it can be a public repository. For example, the MySQL images use "temp" as the MYSQL_ROOT_PASSWORD. This is intended to be changed upon first run using configuration management.

Recommendation: Sensitive or classified images shall be stored in private repositories in NCCS GitLab with access to a limited number of users. Additionally, username and passwords shall be embedded using GitLab's CI artifacts so they are included in the container image without being exposed in a plain

text configuration file. The use of plain text passwords and/or usernames is not recommended and shall be avoided. If the container image requires ITAR data, a new build node shall be requested to support@nccs.nasa.gov to segregate its data to a separate realm. Ensure that sensitive files like /etc/passwd, /etc/group, and /etc/shadow do not contain secrets.

2.13 Properly tag your images

Container images are generally identified by two components: their name and their tag. Container images are a way of packaging pieces of software, thus it is of extreme importance to tag a container to the release policy or version of the software in question. There are two main tagging strategies used when combining container and software: semantic versioning and using git commit hash. In semantic versioning, the software has a three-part version number: X.Y.Z, where:

- X is the major version, incremented only for incompatible API changes.
- Y is the minor version, incremented for new features.
- Z is the patch version, incremented for bug fixes.

Using this policy offers users the flexibility to choose which version of your software they want to use. For more information on semantic versioning, visit the specification website. The git commit hash tagging strategy uses the Git commit SHA-1 hash (or a short version of it) as the version number. By design, the Git commit hash is immutable and references a specific version of your software. This strategy is recommended for operational containers that do continuous delivery often, since it allows a greater way of monitoring the particular changes included in the deployed container image.

Recommendation: Follow a coherent and consistent tagging policy. Document your tagging policy so that image users can easily understand it. For example, Git application called xrasterlib. This application follows the semantic versioning schema, and the latest release is 7.5.1. An example of the identifier of this image would be nccs/xrasterlib:7.5.1, where nccs/xrasterlib is the name and 7.5.1 is the tag. Since this image is the latest, it can be tagged with latest as well, replacing the previous one with the appropriate tag. Another example, a JupyterHub container should be deployed using a Git repository for Infrastructure as Code, and leveraging the git commit hash tagging strategy. The latest commit hash was huhrff7rf933e. The the deployed image was tagged nccs/jupyterhub:huhrff7rf933e, where nccs/jupyterhub is the name and huhrff7rf933e is the tag.

2.14 Files should always be owned by a system account (UID less than 500)

There have been several vulnerabilities found on containers where an attacker leverages file permissions from the container to take over user namespace files. NCCS base images take care of setting secure file permissions inside the container image to avoid misconfiguration. Avoid using host admin permissions inside the container to avoid any root escalation conditions.

Recommendation: All files should be owned by a user (preferably by same user namespace) and without root or any administrative privileges account.

2.15 Define environment variables in container definition files built-in options

If you require any special environment variables to be defined, add them to the %environment and %appenv sections of the build recipe. Using built-in options allows for environment variables to be shared

across containers being built from the same base image. Setting up environment variables using built-in options will also include environment variables in the metadata of the container for reproducibility.

Recommendation: Always use built-in container runtime options to set up environment variables. Docker has the ENV option, Singularity provides the %environment section. An example of these is listed below.

```
## Docker
ENV JAVA_HOME /usr/local/openjdk-11
ENV PATH $JAVA_HOME/bin:$PATH

## Singularity
.....
%environment
    # set JAVA_HOME
    export JAVA_HOME=/usr/local/openjdk-11/bin
```

2.16 Document your container

As any piece of code, all containers should be documented through their built-in options. This lets the user interact with the container and to know from the metadata what to expect. Singularity definition files allow the developer to specify a help section. Docker provides labels and comment options within the Dockerfile. It is a best practice to provide a set of commands that can guide the user on how to use the container image.

Recommendation: If the application runscript does not supply help, write a %help or %apphelp section in Singularity. A good container tells the user how to interact with it and what to expect. It is best practice to document both execution and shell options of the container image in question.

2.17 Use licenses with care

Third-party licenses might impose restrictions on redistribution, which apply when you publish a container image to a public registry. Additionally, proprietary licenses sometimes account for per-seat licenses, which can also affect other systems ability to reach their respective licenses.

Recommendation: Ensure that software licenses in question are allowed to be included inside containers. The following repository has on example of how to distribute a Matlab container using Docker (`https://github.com/mathworks-ref-arch/matlab-dockerfile`).

2.18 Ubuntu Specific - Add DEBIAN FRONTEND noninteractive

Even when "-y" is specified in a command, Ubuntu may try and prompt for an answer. By setting the above variable, this tells Ubuntu to act like there is no user looking at it.

Recommendation: In a Dockerfile, specify it like this:

```
ENV DEBIAN_FRONTEND noninteractive
```

3 Best Practices for Running Containers

This section highlights best practices for running containers. The following best practices are generally available through each container runtime with differing command line options. Several examples have

been included using Docker and Singularity.

3.1 Mount only essential underlying host directories

Some container runtimes mount selected host directories inside the container image by default. This can lead to the inclusion of unnecessary host data inside the container image, which could potentially allow an attacker to read and/or modify host files. Do not mount host directories such as /etc, /var, /tmp, /usr/, and others. Do not mount other users directories.

Recommendation: Only mount essential host directories inside the container image. Do not mount host directories such as /etc, /var, /tmp, /usr/, and others. Best practice is to mount only the subfolders required to execute your container image. For example, on ADAPT, you would mount your /home-/username and/or $NOBACKUP space, but not the entire /home directory. Mounting the entire /home directory would include other users' data. Best practice is to also create a dedicated host directory to store the persistent data of your application, which will allow only mounting a subfolder from the underlying filesystem.

```
## Bad Practice
singularity -B /home:/var exec container-name

## Good Practice
singularity -B /home/user/mysql-data:/var/lib/mysql exec container-name
```

3.2 Create a directory for persistent data to be stored

By nature, when a container is destroyed, it takes its data with it. It is best practice to keep persistent data in a designated location so data is not lost or corrupted. This is not required when container data are ephemeral and non-essential.

Recommendation: To avoid losing persistent data, create a directory on the host for use as the container mount point. For example, with a MySQL image, run the following commands on the host system:

```
## Docker Command
docker run -d --name container-name -v /home/user/mysql-data:/var/lib/mysql

## Singularity Command
singularity -B /home/user/mysql-data:/var/lib/mysql exec container-name
```

3.3 File system security

A container image without tools is not sufficient. Counter measures should be put in place to prevent potential attackers from installing their own tools. The first line of defense between the host and the container is the file system. Some of the mechanisms to prevent an attacker from compromising the filesystem are to remove root privileges from the container during runtime, run the container in read-only mode, and isolate the container from user-facing shared filesystems.

Recommendation: The best practice is to run containers in non-privileged mode. This is the default schema of Singularity since it binds to the current namespace. In Docker, avoid running as root inside the container and disable or uninstall the sudo command from the image. This method offers a first layer of security and could prevent, for example, attackers from modifying root-owned files using a

package manager embedded in your container image. Another best practice is to run the container in read-only mode. This is the default schema of Singularity where containers are immutable by default. In Docker, use the -read-only flag from the docker run command or use the readOnlyRootFilesystem option in Kubernetes. You can enforce this in Kubernetes by using a PodSecurityPolicy.

3.4 Do not update containers during runtime

Containers are supposed to be immutable and ephemeral in nature. If there are critical vulnerabilities found during runtime, containers should be rebuilt from the configuration file. No packages or files should be modified during runtime.

Recommendation: The best practice is to rebuild the image, patches included, and redeploy it. Images should not be modified in place since it removes the reproducibility and immutability of the container lifecycle. Document the lifecycle of your container. Containers should be rebuilt and replaced on a weekly basis to avoid utilizing vulnerable containers. Stable software packages such as MPI and others, should be installed using predefined versions to maintain reproducibility.

4 Best Practices for Operating Containers

This section highlights best practices for operating containers. The following best practices are generally used with container runtime daemons and orchestration tools.

4.1 Use vulnerability scanning in Container Registry

A common way to address software vulnerabilities is to use a centralized inventory system that lists packages installed on servers. There are several container scanning engines used to identify upstream operating systems vulnerabilities to deal with container images. Images are scanned when they are uploaded to the Container Registry and whenever there is an update to the vulnerability database. This allows detecting vulnerabilities before they are used in other containers, providing greater patching flexibility.

Recommendation: Perform automated scans over your Container Registry images to flag vulnerable containers. Enable container scanning capabilities during continuous integration and continuous deployment pipelines to avoid the deployment of vulnerable images. Document an organizational policy on how to deal with vulnerable container images. General recommendations are to:

- Store your images in Container Registry and enable vulnerability scanning.
- Configure a job that regularly checks for new vulnerabilities from the Container Registry and triggers a rebuild of the images if needed.
- When the new images are built, have your continuous deployment system deploy them to a staging environment.
- Manually check the staging environment for problems.
- If no problems are found, manually trigger the deployment to production.

4.2 Allow containers to be executed from a single PATH

One of the main challenges of using container image binaries, such as the ones provided by Singularity (.sif files), is how hard it becomes to track these files across the entire compute system. When a rogue image is detected, it becomes almost impossible to locate these images in large filesystems such as Discover. Another undesired situation is to find images with critically vulnerable image OSs that need to be purged and rebuilt because of known critical findings.

Recommendation: Singularity provides a configuration option to allow containers to be executed from a single path. An example for this would be segregating a directory such as /att/lscratch, where the user would create a directory with the username, and would execute the container from that directory. An example of this is listed below. Conversations such as container image lifecycle need to be discussed at the NCCS to finish the lifecycle policy development.

```
singularity shell /att/lscratch/someone
```

Unfortunately, this option is often not available when working with a limited set of filesystems. Thus an additional way of monitoring container images across big GPFS filesystems is using GPFS Policies to search for ".sif" images or other container extensions. This allows system administrators to have actionable data indicating the location of container images, metadata, and creation date within the filesystem.

4.3 Docker Specific - Ask an SA to install the Docker daemon

Docker daemon is not an approved software package to be run on user facing systems. Docker daemon requires specific configurations to run securely on operational environments. CIS and NIST have released benchmarks to mitigate Docker daemon risks, and these have been developed into an Ansible Playbook available on NCCS GitLab.

Recommendation: If you need the Docker daemon installed on a server or workstation, please submit a ticket to support@nccs.nasa.gov with requirements. We are currently researching on root-less Docker and short-term solutions might involve the use of this solution. This ensures that when running Docker you're complying with the configuration requirements. The SA can then install the daemon using the Ansible playbook created for Linux systems. It configures the Docker daemon to have all container logs write to files on the host. This makes it easier to get a better look at all the container logs instead of having to look at them individually for debugging. Additional requirements to consider from the system administrator process:

- Are you using the latest version of Docker?
- Is the Docker repo installed on the system?
- Is /var/lib/docker on a separate partition?
- Are all containers logging to rsyslogd or Splunk?
- Did you change the default IP subnet for docker0 interface?
- Is the subnet used documented?

4.4 Use the native logging mechanisms of containers

Having container runtime logs available is a great way of monitoring the execution of container images allowing easier debugging capabilities and more security insights. Containers offer an easy and standardized way to handle logs because you can write them to stdout and stderr.

Recommendation: Docker and Singularity offer built-in logging capabilities. Do not disable this feature or lower the level of verbosity provided by both container runtimes. Forward these logs to a centralized log infrastructure such as ELK with Fluentd, or any other log analysis system. Having logs stored in both the local and external system allows for additional forensic capabilities and metric acquisition.

5 Future Work

This is a live document being modified as policies and processes regarding containers change at the NCCS. Many container engines are being evaluated for proper configurations to support additional tools to ease our users development efforts. Tools being evaluated at this time are Podman and Rootless Docker.

References

[1] Singularity at BIOWULF NIH HPC Center, `https://hpc.nih.gov/apps/singularity.html`

[2] Docker Container Risk and Best Practices, `https://www.stackrox.com/post/2019/09/docker-security-101`

[3] Google Cloud Best practices for building containers, `https://cloud.google.com/solutions/best-practices-for-building-containers`

[4] Google Cloud Best practices for operating containers, `https://cloud.google.com/solutions/best-practices-for-operating-containers`