ORNL/TM-2024/3328

Oak Ridge National Laboratory Container Factories in the Oak Ridge Research Cloud

David Heise

March 2024

OAK RIDGE National Laboratory

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2024/3328

National Security Sciences Directorate

Container Factories in the Oak Ridge Research Cloud

David Heise

March 2024

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-Battelle LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

CD Continuous Deployment

CDE Continuous Delivery

CI Continuous Integration

CI/CD CI/CD

CiC container-in-container

DinD Docker-in-Docker

DoD Department of Defense

FQDN fully qualified domain name

OCI Open Container Initiative

ORC ORNL Research Cloud

ORNL Oak Ridge National Laboratory

PinP Podman-in-Podman

PTC pull-through-cache

RBAC Role Based Access Control

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Advanced Materials & Manufacturing Technologies Office (AMMTO) Award Number DE-EE0009046. The views expressed herein do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

This research used birthright cloud resources of the Compute and Data Environment for Science (CADES) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This research used resources from the Knowledge Discovery Infrastructure at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725

ABSTRACT

This technical report describes an implementation of a DevOps *container factory* utilizing ORNL Research Cloud birthright resources. Any researcher or project can implement the strategy described here for their CI/CD needs. This is a technology demonstration of using rootless podman as a direct replacement for docker in a GitLab CI/CD context to enhance security. It is also a stepping stone towards implementation of the *containerized software factory* described in the DoD Enterprise DevSecOps Reference Design.

1. INTRODUCTION

This technical manual describes a repeatable process for researchers or research projects at Oak Ridge National Laboratory (ORNL) to follow for building a containerized GitLab CI runner infrastructure [1]. We utilize resources available to every researcher in the open research environment. CI runner infrastructure utilizing resources available at ORNL. The Department of Defense (DoD) has published a reference design for a *containerized software factory* [5]. The ideal containerized software factory is a secure method to deliver software by utilizing a CI/CD (CI/CD) pipeline to build and test software which is delivered as a container image. For the purposes of this technical report, a distinction will be made between a *software factory* and a *container factory*. A software factory is taken to be the portion of a CI/CD pipeline focused on software code – tasks such as building the software, code linting, unit and functional and other software test cases, and creating and delivering software installers. A container factory is taken to be the CI/CD steps focused on integrating a software package into a container by means of a container build tool and checking the properties of the built container. Accordingly, we will focus on the infrastructure of container building in a CI/CD pipeline. Other topics related to the software life cycle are outside the scope of this manual.

2. MOTIVATION

The motivations for this technology demonstration have changed and grown since its initial implementation. The initial motivation was to use local repository mirrors for ORNL supported Linux distributions in order to speed up Continuous Integration (CI), Continuous Delivery (CDE), and Continuous Deployment (CD)* operations that involved installing large and/or complex packages from the Internet. Building containers within a GitLab pipeline is desirable in order to have a reproducible process and to keep the images up to date with their upstream sources if applicable. However, CI pipelines are themselves generally executed inside the context of a container. Therefore, the problem of building a container inside of a container had to be solved. Several strategies were considered and/or implemented for building container-in-container (CiC): (1) Docker-in-Docker executor, (2) Podman-in-Docker executor, (3) Podman-in-Podman with custom executor, (4) Podman-in-Podman with podman executor†, and (5) VirtualBox executor.

Each of these strategies has trades-offs that will be covered below. However, broadly speaking, DinD execution has negative security properties that revolve around allowing automated arbitrary code execution with rootful privileged access to the host. VirtualBox executors are complex to set up and unwieldy to execute and manage. Podman-in-Podman execution allows for rootless build of OCI compliant containers (e.g., docker, podman). As of this writing, some containerization technologies (e.g., Apptainer/Singularity) require rootful privileged access in order to build their container images, therefore these container technologies require the DinD strategy. However, both Docker and Apptainer have ongoing development of rootless operations.

This technical report will describe how to implement a *container factory* using (1) Docker-in-Docker with docker executor and (4) Podman-in-Podman with a podman executor. Strategies (2) and (3) can be largely considered precursors or workarounds that predated (4), which only became available as a GitLab feature in late 2022. Implementation of strategy (5) is complex and has not been practical to implement.

Strategy (4) is the recommended strategy due to improved security properties, broad capability, and

*for conciseness, all three activities are referred as CI/CD

†GitLab refers to this in configuration files as a "docker" executor due to API interchangeability.

substantial similarity to traditional docker containerized CI operations. Strategy (1) is utilized for the case of building Apptainer/Singularity containers. The end result of following this manual will be gaining the capability for an automatic container build process with the resulting image uploaded to the GitLab Container Registry for use. The specific case study examined here additionally utilizes local ORNL repository mirrors. The images described and built here can be used as base images for other projects utilizing the same container factory pattern to gain this capability.

2.1 CONTINUOUS INTEGRATION

Continuous Integration, Continuous Delivery, and Continuous Deployment are important operations of the DevOps workflow [4]. Continuous Integration, Continuous Delivery, and Continuous Deployment all utilize the ability of a repository manager to trigger automatic workflow executions called *pipelines*. Depending on the repository manager and pipeline execution software, they can define complex relationships between job executions, store artifacts, and take other actions. The core operation, however, is the automatic execution of a sequence of actions.

Figure 1. Job Execution Sequence

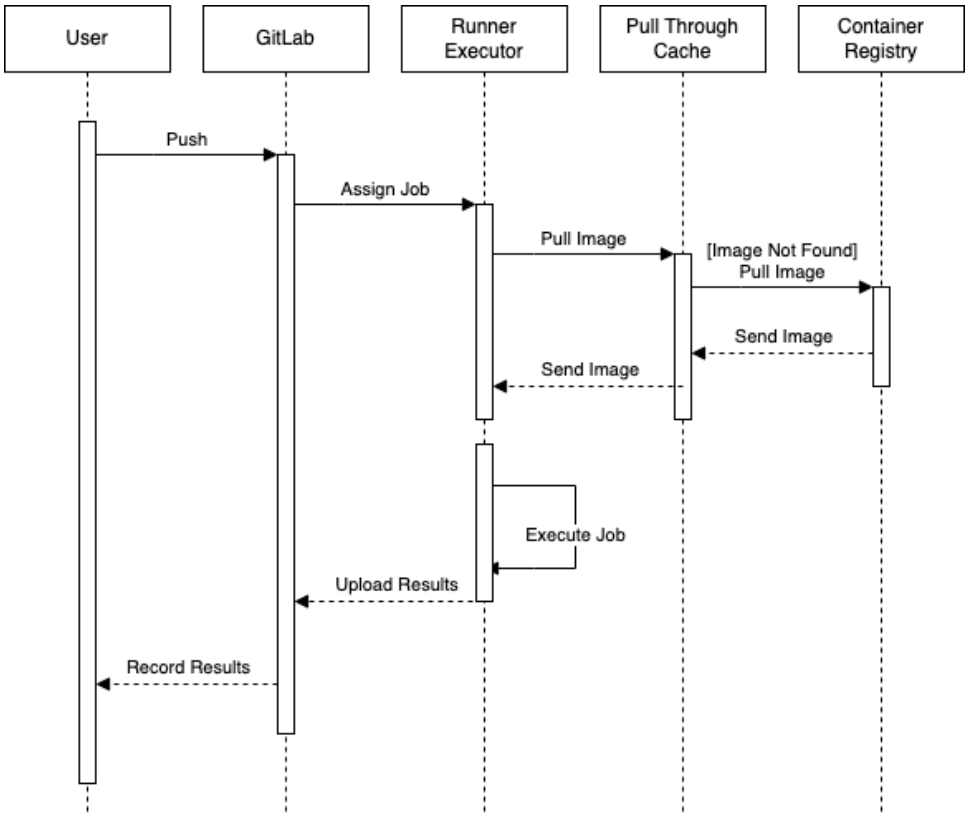


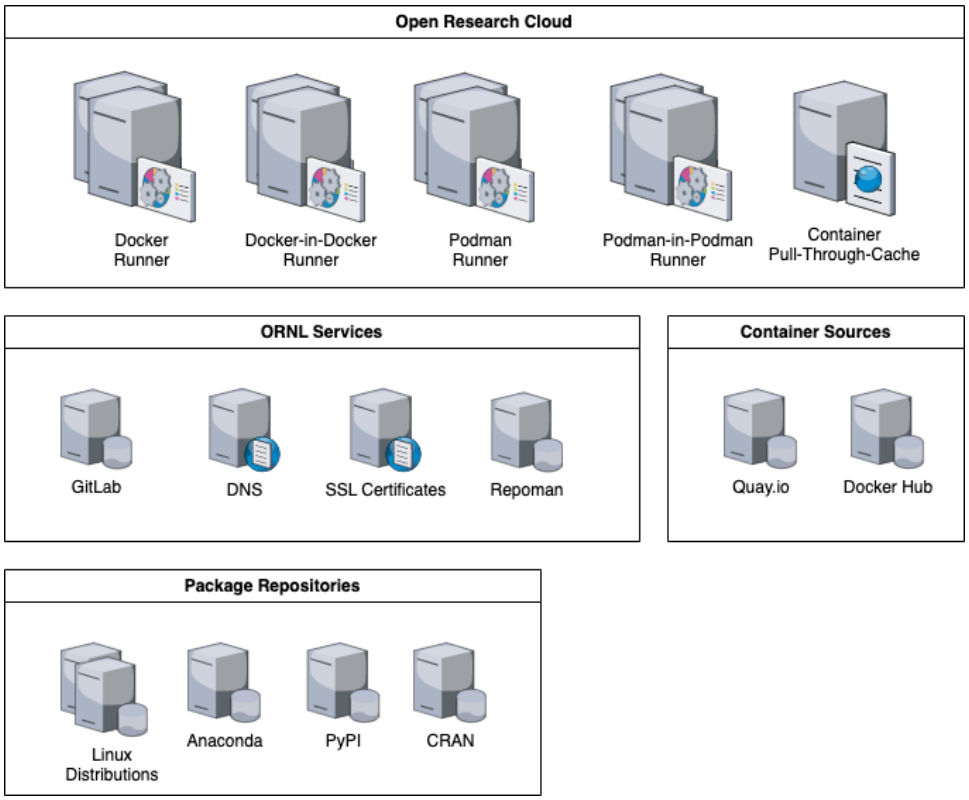
Figure 1 shows the high level sequence of activities that a CI job goes through. A user pushes a code change to a *repository manager* (e.g., GitLab). The repository manager then assigns the job to an appropriate *runner* capable of executing the job. In this case we assume this is a container based runner, so it will request a configured container image that can execute the job. If a pull-through-cache (PTC) is not available, then the image pull would go straight to the container registry. Once the runner has started the

appropriate container it will execute the job script, upload any results (e.g., status code, artifacts, container images) and report those to the user.

A *runner* refers to a distinct configuration for accepting jobs from the repository manager. A runner has a configured type of *executor*, which is the method by which a job script is executed. There is a one-to-one relationship between a runner and an executor. More than one runner can reside on a single host. While technically distinct concepts, *runner* and *executor* will be used interchangeably because of this one-to-one relationship, and for this document an executor will always be an executor that speaks the Docker API.

3. CONTAINER FACTORY

Figure 2. Container Factory System Components

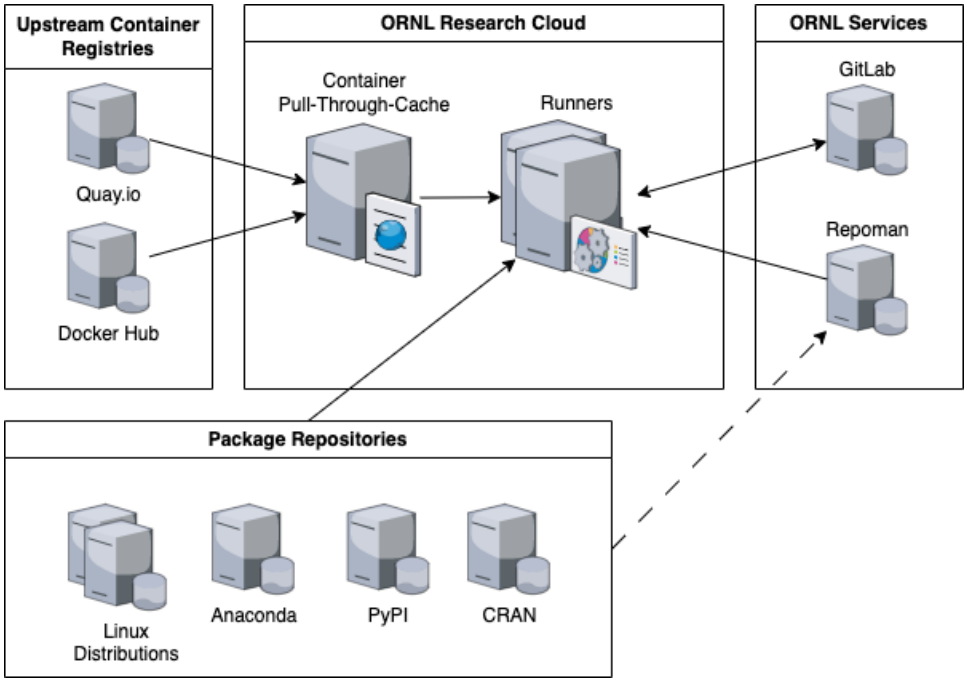


A *containerized software factory* has the goal of "produc[ing] a set of software deployable artifacts with minimal human intervention" [5]. As indicated by the name, the deployable artifact is a container that can execute the target software. The state of the art DevOps practices, this automation is accomplished by triggering CI/CD *pipelines*. Pipelines are a set of automated steps executed in response to code changes sent to a code repository manager, e.g., GitLab, GitHub, etc. Pipelines are often executed in the context of a container. Therefore, a capability to build a container while executing inside the context of another container is desirable. The *container factory* is the architecture that permits a CI/CD system to execute a container build operation inside the context of a container. This can be thought of as a subsystem of the broader software factory.

We will describe a container factory that utilizes the component systems shown ins figure 2. A repository

manager is foundational to the system orchestration; in this case we utilize GitLab which is the repository manager service provided at ORNL. The DNS service is depicted because Docker requires a simple DNS modification to access internal resources; it is not directly related to the CI factory. SSL certificates are an optional security enhancement to have images pulled securely from the PTC. We utilize two upstream container registries: (1) Quay.io, a registry closely associated with Red Hat and Podman, and (2) the Docker Hub, which can optionally have images cached in a PTC. In addition, the GitLab instance functions as a local container registry.

Figure 3. Container Factory Data Flow



This manual will describe how to set up the five services that directly support and execute the container factory inside of the ORNL Research Cloud (ORC): (1) a podman runner for general CI job operations, (2) a Podman-in-Podman runner for container building job operations, (3) a docker runner for general CI job operations, (4) a Docker-in-Docker runner for container building job operations, (5) a pull-through-cache to work around upstream rate limiting and improve job execution latency.

How these services connect is shown in Figure 3. Container images can be sourced from either an upstream container registry, such as Quay.io or the Docker Hub, or from the local GitLab instance. Images from upstream registries can optionally be cached in a local PTC. If the PTC is not present, then images are pulled directly from the upstream registry. Jobs are sent to runners and executed. The results of those executions are then sent to the GitLab instance. Additionally, ORNL mirrors several open source software repositories. If a container image has been configured to do so, then these packages can be pulled from the local mirror.

The fundamental concepts here can generally apply to any repository manager, and any cloud-like computing environment, assuming appropriate network connectivity exists between them. Terminology may differ between particular implementations.

3.1 REPOSITORY MANAGER

A git repository manager is used to store code repositories. Repository managers that support CI/CD are able to trigger automated job executions related to code push events. The repository manager and CI/CD will form the workflow backbone of the container factory.

3.2 CONTAINER REGISTRY

A container registry is a repository of containers that users can push (write) container images to and pull (read) container images from. In this manual we are concerned with three different container registries. There are two upstream registries treated as read-only: the Docker Hub and Quay.io. Both are large sources of base images from which new images are built. The Docker Hub has API rate limitations which motivate the use of a PTC. The third registry is the registry provided by the repository manager. This is the destination for images produced by our container factory. All three registries support access controls to manage push and pull permissions. Pulling from a registry is as simple as giving a URL to the image to the container runtime. If access controls are implemented, the runtime may need to log in to the registry first.

3.2.1 Pipeline Executors

Pipelines are commonly executed in the context of a containerized software environment. The container factory treats the container building and testing process as just another CI/CD job. However, running a container in a container has unique technical requirements.

3.3 CONTAINER-IN-CONTAINER

Several software containerization technologies exist today. The three considered here are Docker, Podman, and Apptainer/Singularity. Docker and Podman are general purpose container technologies, sometimes called "lightweight virtual machines." Docker is the most common container platform for CI. However, Podman has recently added Docker API capabilities to make it a viable drop-in replacement for Docker in the CI context.

For historical-technical reasons, the Docker system runs in a root level daemon[‡]. Users with access to the docker daemon have the ability to execute containers with implicit root-level access to the host through a privileged container execution mode. In the context of a CI executor running code integration tasks, the docker container of the running job typically does not have the privileged mode enabled. However, in order to build or execute a CiC, the job must be given access to the host docker daemon. This requires running the CI executor in privileged mode. Therefore, any CI job running CiC tasks has root access on the host. This requires a high degree of trust between the host and the CI jobs. A rootless solution would be preferable.

Podman is an alternative container runtime that runs containers with the same permissions as the user executing the container. A collaboration between Podman and GitLab has recently brought the Podman socket API to parity with Docker for the purposes of GitLab CI. With this recent change, Podman is a viable, rootless, drop-in replacement for Docker in the GitLab CI context. Similar to Docker, a socket is provided to facilitate the interaction between GitLab CI Runners and Podman. However, in contrast to Docker, the daemon can be configured to run as a regular user on the host. Again, similar to Docker, Podman also has a concept of privileged execution allowing direct access to the host system; however, that

[‡]Rootless Docker is an active area of development

access is still limited to the permissions of the executing user. Therefore, CiC CI jobs do not need to be trusted with root level access to the host. However, all users executing jobs on the same daemon must trust one another.

Apptainer is targeted at HPC workflows and so not used as a general purpose lightweight virtual machine. The typical Apptainer container lifecycle is to be built on a host where the author has privileged access (e.g., a workstation), and then transported to a host where the author does not have privileged access (e.g., an HPC compute cluster). Apptainer container images, by default, are read-only artifacts built for the purpose of packaging up software dependencies to allow a target software to execute successfully on a host where the user does not control the software ecosystem.

3.4 PULL THOUGH REGISTRY CACHE

A PTC is an optional resource for the container factory, but can be a useful capability. Because pipelines are potentially triggered at every code push, the container factory may execute many network operations to obtain prerequisites for the pipeline. Pulling resources from the internet is a much slower operation than pulling local resources if they are available. Additionally, some container image repositories, e.g., Docker Hub, apply rate limits to users which can result in failed pipelines. Runners can also be configured to pull less aggressively.

3.5 LOCAL PACKAGE MIRROR

In addition to the PTC alleviating Internet communication load, ORNL also provides mirrors for several package repositories at `repoman.ornl.gov`. Configuring container package managers to utilize these local mirrors can result in additional performance gains. The case study repository associated with this manual describes how to utilize the local package mirror when building and executing container images.

4. CASE STUDY

The ORNL ORC[§] provides an on-premises cloud-like experience for researchers built on RedHat OpenStack. All staff at ORNL can utilize a "birthright" resource allocation in ORC. The ORC has both an open and moderate instance. This manual is applicable to either environment.

In order to stand up a container factory, we will need a repository manager which will handle version control and pipeline orchestration, a virtual machine, and a pipeline runner service. Optionally, a container image PTC can be included. This is helpful for working around container image providers, such as Docker Hub, that impose a pull rate limit on free tier users.

An ORNL internal demonstration repository is available on `code.ornl.gov` [3]. This case study will examine how each of the parts described above are utilized to create a container factory that can build containers in three ways. The output of this demonstration is a basic (but not minimal) operating system container that utilizes the internal package mirrors hosted on `repoman`. Most ORNL approved Linux operating systems are represented.

[§] https://ornl.servicenowservices.com/kb?id=kb_article_view&sys_kb_id=4253f85e1b756d90f878404fe54bcb34

4.1 ORNL RESEARCH CLOUD VIRTUAL HOSTS

The ORC is a virtual cloud environment built on OpenStack. Access to the ORC is provided to all ORNL researchers.

We utilize three hosts for the container factories:

(1) a host for a Docker-in-Docker runner, (2) a host for Podman and Podman-in-Podman runner, (3) a host for a container registry pull-through-cache.

These three hosts are depicted in Figure 3 sitting in the ORC.

We isolate DinD runners on their own host due to the elevated trust required for the DinD CI jobs. As a mitigation for this level of trust we could, for example, revert to a known-good instance image on these runners on a regular basis to reverse any changes made to the host.

Podman operations only need to be isolated by user, so the same host can be used for both normal CI operations and for PinP operations. This can be accomplished by running the privileged runner as a separate service, although host level isolation may be more straightforward in a cloud-like environment such as ORC. Furthermore, since there is some user-level configuration that the PinP jobs have access to, this host could also be reset to a known-good state regularly.

Finally, we isolate the container registry PTC because it is running, and because it has different virtual hardware and service level needs than the other services. As a cache, this host is permitted to be unreliable. For example, the PTC can be taken down for maintenance or even lose its contents entirely without interrupting normal CI/CD operations. Once the PTC is returned to service it will begin rebuilding its contents based on runner demand.

For the CI hosts, we prioritize CPUs and memory since CI jobs are primarily compute tasks rather than storage tasks. For the container registry pull-through-cache, we prioritize storage performance over CPUs or memory since the retrieval of images is high disk load and low compute load. As a cache, the storage size selection is arbitrary, but larger storage will allow for more images to be cached before needing to delete contents.

In order to have a new enough version of Podman for PinP, we utilize Centos 8 Stream or 9 Stream as our host OS. Any Linux distribution which can install Podman 4.2 or higher will work.

4.1.1 Obtaining a DNS Entry for Virtual Hosts

As of this writing, a support ticket must be submitted to the the ORC team[¶] to get a virtual host entered into Devices where a DNS name can be assigned. Only the pull-through-cache needs a DNS entry if configuring it to use TLS. Our PTC name is `cfcc01.ornl.gov`. In your configuration you will substitute your cache's IP or DNS name.

A method for users to register their own hosts in Devices is expected in the future.

4.2 GITLAB

The git repository manager provided by ORNL is GitLab. GitLab provides all the features needed to set up a container factory. There are two GitLab instances at ORNL: External GitLab (`code.ornl.gov`) and

[¶] https://orc-tickets.ornl.gov/

Internal Gitlab (`code-int.ornl.gov`). The steps in this manual can be used for either GitLab environment.

4.3 GITLAB RUNNER

On each CI host we install and configure the GitLab Runner service. We defer to the GitLab documentation for GitLab Runner software installation[‖].

The GitLab Runner utilizes *executors* to accomplish its tasks. For the purposes of the container factory, only the docker executor matters. Note that GitLab does not distinguish between the docker and podman runtimes when configuring executors; both are "docker" executors according to GitLab with only slight differences in configuration.

Figure 4. Runner Registration Common Options

```
# gitlab-runner register \
  --non-interactive \
  --executor "docker" \
  --url "https://code.ornl.gov/" \
  --registration-token "RUNNER_REGISTRATION_TOKEN" \
  --locked="true" \
  --tag-list "your, tags, here" \
  --description "Free text"
  ...
```

Figure 4 is an excerpt of a registration command. These options are substantially the same for all the executor configurations described below.

The `--non-interactive` option just tells the command to take all parameters from the command line. The `--executor "docker"` option tells the runner to execute jobs utilizing the docker socket API. Both Docker and Podman speak this API[**]. The `--url` is the URL of the GitLab instance – at ORNL this is either https://code.ornl.gov/ or https://code-int.ornl.gov/. The registration token is found under either the Group or Project settings at `Settings -> CI/CD -> Runners - Project (or Group) Runners`. The option `--locked="true"` specifies that this runner cannot be added by other projects/groups by the GitLab Web UI. The `--tag-list` is a comma separated list of arbitrary textual tags. Each tag can be used in CI jobs to restrict jobs to run only when a given set of tags is present on a runner. The `--description` option is an arbitrary text description of the runner and has no functional effect.

4.4 DOCKER EXECUTOR

It is recommended that a DinD runner be isolated on its own virtual host and access to the runner be tightly controlled using GitLab Role Based Access Control (RBAC) due to its privileged access.

[‖] https://docs.gitlab.com/runner/install/

[**] Podman does have its own socket API, but the Docker API is used by GitLab

Figure 5. Docker-in-Docker Runner Registration

```
# gitlab-runner register \
  --non-interactive \
  --executor "docker" \
  --docker-privileged \
  --docker-volumes /certs/client \
  --run-untagged="false" \
  --url "https://code.ornl.gov/" \
  --registration-token "RUNNER_REGISTRATION_TOKEN" \
  --docker-image docker.io/library/docker:latest \
  --docker-allowed-services docker.io/library/dind \
  --locked="true" \
  --tag-list "dind" \
  --description "dind-runner"
```

Figure 5 shows an example command for registering the runner on a CI host. The `--docker-privileged`, and `--docker-volumes /certs/client` are mandatory. The Executor tells the runner how jobs will be run, in this case with Docker. The `--docker-privileged` option runs Docker in privileged mode, which is required for DinD executions. The `--docker-volumes` option creates a volume to store certs, which will be used in jobs to establish secure communication with the Docker daemon. The `--run-untagged="false"` option means this run will only pick up CI jobs explicitly tagged for using DinD, which is included in the `--tag-list`. This is important so that jobs that don't need privileges aren't using them.

The `--docker-image` sets the default image for jobs on this runner, in this case the Docker image. The `--docker-allowed-services` option specifies the services allowed to run on this Runner. In this case, we only permit the DinD service. This option could be left off if other services are needed.

The following descriptions of other runner types will focus on the differences between them and this registration example, since most options are the same or substantially similar.

Figure 6. Docker Runner Registration

```
# gitlab-runner register \
  --non-interactive \
  --url "https://code.ornl.gov/" \
  --registration-token "RUNNER_REGISTRATION_TOKEN" \
  --executor "docker" \
  --docker-volumes /certs/client \
  --description "docker-runner" \
  --tag-list "docker" \
  --run-untagged="false" \
  --locked="true"
```

Figure 6 shows the registration of an "unprivileged" docker runner. Even an unprivileged docker runner is a security hazard since the daemon still runs in a root context, and should therefore be secured in a similar

manner as the DinD runner. Key differences between this and a privileged runner are that `--docker-privileged`, `--docker-image`, and `--docker-allowed-services` are missing. We don't need the elevated privileges of a DinD runner, a user can specify to run their test with any base image, and can specify any service image as well.

4.4.1 Docker Daemon Configuration

To work in the ORNL network environment, it is helpful set a few global docker daemon configuration parameters. An important limitation of the Docker daemon is that a registry mirror can only be configured for the Docker Hub.

Figure 7. /etc/docker/daemon.json Pull-through-cache with TLS

```
{
    "dns": [ "160.91.134.89", "160.91.1.62", "160.91.86.41" ],
    "registry-mirrors": ["https://cfcc01.ornl.gov:4567"]
}
```

Figure 7 shows how to configure the docker daemon to use the ORNL DNS servers so it can locate ORNL internal resources. Be aware that these IP addresses have been observed to change. If you have DNS issues, it is recommended to contact the Solution Center to get an updates list.

The array `registry-mirrors` is used to point to the PTC. If you are not using the PTC, then omit the `registry-mirrors` key.

Figure 8. /etc/docker/daemon.json Pull-through-cache sans TLS

```
{
  "dns": [ "160.91.134.89", "160.91.1.62", "160.91.86.41" ],
  "registry-mirrors": ["http://cfcc01.ornl.gov:4567"],
  "insecure-mirrors": ["http://cfcc01.ornl.gov:4567"]
}
```

If your registry mirror is not configured with TLS, then you must explicitly tell docker to accept the insecure connection with the `insecure-mirrors` setting as in figure 8.

4.5 PODMAN EXECUTOR

By default, the gitlab-runner service is configured to run as a service under an unprivileged user of the same name. This provides the desired level of user privilege for a PinP executor. For ease of configuration, it is recommended that unprivileged and privileged runners are configured on separate hosts. While outside the scope of this document, it is possible to set up more than one gitlab-runner service on a single host and run each service in its own user. It is also possible to register more than one runner on a single host and under a single gitlab-runner service, but this is not recommended.

Figure 9. Podman-in-Podman Runner Registration

```
# gitlab-runner register \
  --non-interactive \
  --url "https://code.ornl.gov/" \
  --registration-token "RUNNER_REGISTRATION_TOKEN" \
  --executor "docker" \
  --docker-privileged \
  --docker-image quay.io/containers/podman:latest \
  --docker-allowed-images quay.io/containers/podman:latest \
  --docker-allowed-services "" \
  --docker-volumes /certs/client \
  --docker-volumes "/etc/containers/registries.conf:/etc/containers/registries.conf:ro" \
  --docker-host "unix:///run/user/$(id -u gitlab-runner)/podman/podman.sock" \
  --env "FF_NETWORK_PER_BUILD=1" \
  --description "pinp-runner" \
  --tag-list "pinp" \
  --run-untagged="false" \
  --locked="true"
```

Figure 9 gives a command to register a PinP runner. We give `--docker-privileged` because there are guardrails that need to be removed for a container build to be possible. However, unlike with Docker, this grants only as much capability as the user the `gitlab-runner` service is running under. By default, this is an unprivileged user. Since this runner is intended only to be used for container builds, we set the official Podman container to be both the default and only allowed image with the `--docker-image` and `--docker-allowed-images` options. We also forbid any additional services from running by giving an empty string to `--docker-allowed-services`. We give two volumes via `--docker-volumes`. We give `/certs/client` so that if the Docker API is configured to use TLS, it will still function. We also pass in the host's `registries.conf` as read-only so that the inner podman will utilize the same registry configuration as the host. In particular, this contains the configuration for the PTC. We point the runner to talk to the podman socket providing the Docker API with `--docker-host`. With `--env` we configure the environment variable `FF_NETWORK_PER_BUILD`, which will create a separate network for every CI job, so each job has a private communication channel. The `--run-untagged="false"` option means this run will only pick up CI jobs explicitly tagged for using PinP, which is included in the `--tag-list`. While not as security critical as with docker, this is still recommended.

Figure 10. Podman Runner Registration

```
# gitlab-runner register \
  --non-interactive \
  --url "https://code.ornl.gov/" \
  --registration-token "RUNNER_REGISTRATION_TOKEN" \
  --executor "docker" \
  --docker-volumes /certs/client \
  --docker-volumes "/etc/containers/registries.conf:/etc/containers/registries.conf:ro" \
  --docker-host "unix:///run/user/$(id -u gitlab-runner)/podman/podman.sock"
  --env "FF_NETWORK_PER_BUILD=1" \
  --description "podman-runner" \
  --tag-list "podman" \
  --run-untagged="true" \
  --locked="true"
```

In figure 10 we see an unprivileged Podman runner for generic CI operations. This is different from the PinP runner mainly by options that are removed: `--docker-privileged`, `--docker-image`, `--docker-allowed-images`, `--docker-allowed-services`. Since this is an unprivileged runner, running as a regular user, the CI job author is given leeway on what to run.

Figure 11. /etc/containers/registries.conf

```
unqualified-search-registries = [
"registry.access.redhat.com",
"registry.redhat.io",
"docker.io",
"quay.io"
]

[[registry]]
prefix = "docker.io"
insecure = false
location = "docker.io"
[[registry.mirror]]
location = "cfcc01.ornl.gov:4567"
# Set insecure = true if your PTC does not use TLS
insecure = false

[[registry]]
prefix = "quay.io"
insecure = false
location = "quay.io"
[[registry.mirror]]
location = "cfcc01.ornl.gov:4568"
# Set insecure = true if your PTC does not use TLS
insecure = false
```

Figure 11 shows the Podman registry configuration. Unlike Docker, Podman is able to be pointed to a mirror for any registry. In this case, we point Podman to a local PTC for both the Docker Hub and Quay.io. The two configurations are substantially similar. In the `[[registry]]` section, we specify the origin location of the registry being mirrored, and specify that the origin uses TLS. This location will be used if the mirror does not contain a given image requested. The `[[registry.mirror]]` section specifies the local location of the mirror, and whether it uses TLS or not. The mirror will be searched first. Since we are configuring a PTC, it is expected that as long as the mirroring service is running, the image should come from the mirror and not the origin registry, saving us pull request limitations.

4.6 PULL-THROUGH-CACHE

Our pull-through-cache is run using the Docker runtime for ease of setting it up to run as a service, and because there will be no user access to the docker daemon; it is strictly a system service. However, in principle, the Podman runtime could be used as a drop-in replacement if appropriate services are configured.

Figure 12. Docker Hub Registry PTC Configuration (/etc/docker/registry/config-dockerhub.yml)

```
version: 0.1
log:
  fields:
    service: registry
storage:
  cache:
    blobdescriptor: inmemory
  filesystem:
    rootdirectory: /var/lib/registry
  delete:
    enabled: true
http:
  addr: :5000
  headers:
    X-Content-Type-Options: [nosniff]
  #tls:
  #  certificate: /etc/docker/registry/tls/yourcert.cert.pem
  #  key: /etc/docker/registry/tls/yourkey.key
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3
proxy:
  remoteurl: https://registry-1.docker.io
#  username: [username]
#  password: [password]
```

Figure 12 is the registry service configuration to mirror the Docker Hub. Most of the items here are boilerplate and beyond the scope of this document. The relevant sections for getting the correct registry mirrored are `remoteurl` and `tls`. Under `tls` we can configure the location to find a TLS certificate for this runner. How to obtain a DNS name and certificate is described below. The `remoteurl` specifies the fully qualified domain name (FQDN) of the registry to be mirrored.

If you are encountering pull rate issues with the Docker Hub, you can give it a username and password to a Docker Hub account in the commented out fields.

If you have obtained a TLS certificate for your PTC, you can also uncomment and configure that section.

Figure 13. Running the Dockerhub Registry Pull-through-cache

```
docker run -d -p 4567:5000 \
    -v /etc/docker/registry/config-dockerhub.yml:/etc/docker/registry/config.yml \
    -v /etc/docker/registry/tls/:/etc/docker/registry/tls/ \
    --restart always --name $REGISTRY_NAME registry:2
```

Figure 13 is the Docker command that will configure the registry mirror to run as a docker service. We pass in the configuration file described in Figure 12 and the TLS certificate directory, if one is available. The port selection is arbitrary, but must match the configuration of the clients given in the executor configurations above.

4.6.1 Quay.io Mirror Configuration

Quay.io does not have pull rate limits, but if you need access to private container images you will need to log in with username and password.

If you have obtained a TLS certificate for your PTC, then you can also uncomment and configure that section.

Figure 14. Quay Pull-through-cache Configuration)

```
version: 0.1
log:
  fields:
    service: registry
storage:
  cache:
    blobdescriptor: inmemory
  filesystem:
    rootdirectory: /var/lib/registry
  delete:
    enabled: true
http:
  addr: :5000
  headers:
    X-Content-Type-Options: [nosniff]
  #tls:
  #  certificate: /etc/docker/registry/tls/yourcert.cert.pem
  #  key: /etc/docker/registry/tls/yourkey.key
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3
proxy:
  remoteurl: https://quay.io
#  username: [username]
#  password: [password]

```

Figure 14 is the registry service configuration to mirror Quay.io. Most of the items here are boilerplate and beyond the scope of this document. The relevant sections for getting the correct registry mirrored are `remoteurl` and `tls`. Under `tls` we can configure the location to find a TLS certificate for this runner.

How to obtain a DNS name and certificate is described below. The `remoteurl` specifies the FQDN of the registry to be mirrored.

If you have obtained a TLS certificate for your PTC, you can also uncomment and configure that section.

Figure 15. Running the Quay Registry Pull-through-cache

```
docker run -d -p 4568:5000 \
       -v /etc/docker/registry/config-quay.yml:/etc/docker/registry/config.yml \
       -v /etc/docker/registry/tls/:/etc/docker/registry/tls/ \
       --restart always --name $REGISTRY_NAME registry:2
```

Figure 14 is the Docker command that will configure the registry mirror to run as a docker service. We pass in the configuration file described in Figure 15 and the TLS certificate directory, if one is available. The port selection is arbitrary, but must match the configuration of the clients given in the executor configurations above.

4.6.2 Securing the Cache

There are several tools available to improve the security posture of the cache. First, a TLS certificate can be installed. Second, a firewall can narrow network access. Finally, user authentication can be added.

Using a TLS certificate provides protection from a man-in-the-middle attack. Using a firewall may be useful for limiting traffic to the cache. Adding authentication may be useful to additionally limit traffic and should be used if you are pulling private images from the upstream registry to avoid leaking containers.

Getting a TLS Certificate

Any staff member at the lab can get a TLS certificate for their devices using the ORNL SSL Certificate Tool[††]. In order to get a DNS name in Devices for an ORC virtual host, you will need to submit an ORC ticket until/unless an automated system is developed.

4.7 CONTAINER BUILDS

For the sake of brevity, simplified excerpts will be used. A repository implementing the techniques described can be found in the associated ORNL Open GitLab repository [3]. The repository tag `1.0-TM` points to the state of the repository as of this writing.

[††]https://certmanager.ornl.gov/index.php

Figure 16. Dockerfile

```
1  FROM quay.io/centos/centos:stream9
2  RUN yum update -y && yum upgrade -y && yum clean all
3
4  # Point conda to repoman
5  COPY linux/condarc /root/.condarc
6  COPY linux/condarc /etc/skel/.condarc
7
8  # Point CRAN to repoman
9  COPY noarch/Rprofile /usr/lib64/R/etc/Rprofile.site
10 COPY noarch/Rprofile /etc/skel/.Rprofile
11
12 # Point pip to repoman
13 COPY linux/pip.conf /etc/pip.conf
14 RUN mkdir -p /etc/skel/.config/pip
15 COPY linux/pip.conf /etc/skel/.config/pip/pip.conf
16 RUN chmod 744 /etc/pip.conf
17
18 # Point CPAN to repoman, and otherwise configure it
19 COPY linux/MyConfig.pm /etc/skel/.cpan/CPAN/MyConfig.pm
20 COPY linux/MyConfig.pm /root/.cpan/CPAN/MyConfig.pm
21
22 RUN dnf autoremove -y && dnf clean all
```

Figure 16 is a sample Dockerfile that defines a container build. It pulls a base Centos Stream 9 image from the public Quay registry (line 1), updates any outdated packages (line 2), and then copies files which point package managers to local package mirrors on the ORNL network (lines 4-29) and finally cleans up the installed packages and cache.

Figure 17. Apptainer Definition

```
1  BootStrap: yum
2  OSVersion: 9-stream
3  MirrorURL: https://repoman.ornl.gov/centos-stream/%{OSVERSION}/BaseOS/$basearch/os
4  Include: yum centos-release centos-stream-repos
5
6  %environment
7  export LC_ALL=C
8  export PATH="${HOME}/perl5/bin${PATH:+:${PATH}}"; export PATH;
9  export PATH="${HOME}/miniconda3/bin:${HOME}/.local/bin:$PATH"
10 export PERL5LIB="${HOME}/perl5/lib/perl5${PERL5LIB:+:${PERL5LIB}}"; \
11 export PERL5LIB;
12 export PERL_LOCAL_LIB_ROOT="${HOME}/perl5${PERL_LOCAL_LIB_ROOT:+:${PERL_LOCAL_LIB_ROOT}}"; \
13 export PERL_LOCAL_LIB_ROOT;
14 export PERL_MB_OPT="--install_base \"${HOME}/perl5\""; export PERL_MB_OPT;
15 export PERL_MM_OPT="INSTALL_BASE=${HOME}/perl5"; export PERL_MM_OPT;
```

```
16
17 %setup
18     mkdir -p ${SINGULARITY_ROOTFS}/usr/lib64/R/etc/ \
19              ${SINGULARITY_ROOTFS}/root/.cpan/CPAN \
20              ${SINGULARITY_ROOTFS}/etc/skel/.cpan/CPAN/ \
21              ${SINGULARITY_ROOTFS}/etc/yum/dnf/ \
22              ${SINGULARITY_ROOTFS}/etc/yum/vars/ \
23              ${SINGULARITY_ROOTFS}/etc/yum/etc/yum.repos.d/ \
24              ${SINGULARITY_ROOTFS}/etc/pki/ca-trust/source/anchors/
25
26 %files
27     etc/pki/ca-trust/source/anchors/Netskope*.pem /etc/pki/ca-trust/source/anchors/
28     centos9/etc/dnf/dnf.conf /etc/dnf/
29     centos9/etc/yum.repos.d/* /etc/yum.repos.d/
30     centos9/etc/yum/vars/* /etc/yum/vars/
31     linux/condarc /etc/skel/.condarc
32     linux/condarc /root/.condarc
33     noarch/Rprofile /etc/skel/.Rprofile
34     noarch/Rprofile  /usr/lib64/R/etc/Rprofile.site
35     linux/pip.conf /etc/pip.conf
36     linux/MyConfig.pm /root/.cpan/CPAN/MyConfig.pm
37     linux/MyConfig.pm /etc/skel/.cpan/CPAN/MyConfig.pm
38     installers/* /root/.local/bin/
39
40 %post
41 update-ca-trust
42 rpmdb --rebuilddb
43 curl -o /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-9
44     http://repoman.ornl.gov/epel/RPM-GPG-KEY-EPEL-9
45 rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-9
46 yum install -y --releasever 9 centos-release centos-stream-repos \
47     epel-release epel-next-release
48 yum -y update && yum -y upgrade
49 yum -y autoremove && yum -y clean all
50 chmod 744 /etc/pip.conf
51 chmod 744 /usr/lib64/R/etc/Rprofile.site
```

Figure 17 is an example Apptainer definition file that performs a substantially similar container build as the Dockerfile in Figure 16. Lines 1-4 define that the image will be bootstrapped from scratch using the ORNL Repoman OS package mirror for the given operating system. There is no base container image pulled from the Internet (although that is also an option with Apptainer). In Figure 17 we bootstrap with `yum`. An example is also available using `debootstrap`. The host and container operating systems do not need to match as long as they have the bootstrap package manager available.

Figure 18. CI Job Variable Definitions

```
variables:
  DOCKER_TLS_CERTDIR: "/certs"
  DOCKER_CERT_PATH: "$DOCKER_TLS_CERTDIR/client"
  DOCKER_TLS_VERIFY: 1
```

Figure 18 is the Docker environment variable boilerplate to set up CiC operations. The variables `DOCKER_*` set up communications over the Docker socket to use TLS. The TLS certificates in this context are self-signed and are just for ensuring jobs can't see one communication with the daemon. These variables are only needed on DinD jobs and can also be set in the runner `config.toml`.

4.7.1 Podman-in-Podman Build

Figure 19. Podman-in-Podman Build CI Job

```
1  podman:
2    image: quay.io/podman/stable
3    tags:
4      - pinp
5    before_script:
6      - podman login --username=$CI_REGISTRY_USER --password=$CI_JOB_TOKEN $CI_REGISTRY
7    script:
8      - podman build --tag $CI_REGISTRY_IMAGE --file Dockerfile .
9      - podman push $CI_REGISTRY_IMAGE
```

Figure 19 shows an example job that will build an Open Container Initiative (OCI) image using PinP. Line 2 specifies that we will use the latest official podman container image (pulled from the PTC if configured). Line 4-5 specifies the runner tags that must be applied to a runner for it to pick up the job. The `pinp` tag means we will be using a the runner we registered using the podman runtime with user-privileged mode enabled.

In order to push images to the GitLab container registry, the CI job must log in (Line 11) to the container registry. GitLab CI makes automatic variables for that purpose available. We then build the container image and push it to the GitLab container registry (Line 9-10). We utilize the GitLab CI automatic variable `CI_REGISTRY_IMAGE` to name the image.

4.7.2 Podman-in-Podman Multi-architecture Manifest Build

Podman has the ability to build and run images that target foreign architectures by utilizing QEMU user level emulation. If your host OS distro has an equivalent of the package `qemu-user-static` on Fedora, then you can install that package. If your distro does not have an equivalent package, then you can utilize the `docker.io/multiarch/qemu-user-static` container to get the needed capability.

Figure 20. QEMU User Static Service (/etc/systemd/system/qemu-user-static.service)

```
[Unit]
Description=QEMU User Virtualization - MultiArch Container Builds
ConditionFileIsExecutable=/bin/podman
ConditionPathExists=!/usr/bin/qemu-x86_64-static
After=syslog.target network.target

[Service]
Type=oneshot
ExecStart=podman run --rm --privileged docker.io/multiarch/qemu-user-static --reset -p yes

[Install]
WantedBy=gitlab-runner.service
```

Figure 20 is a service unit file that can be installed (e.g., at `/etc/systemd/system/qemu-user-static.service`) and enabled (`systemctl enable /etc/systemd/system/qemu-user-static.service`) to start before the GitLab runner. This ensures your QEMU user level emulation will be available before the runner picks up any jobs.

Figure 21. Podman-in-Podman Multi-architecture Manifest Build Job

```
podman:
  image: quay.io/podman/stable
  tags:
    - pinp
  before_script:
    - podman login --username=$CI_REGISTRY_USER --password=$CI_JOB_TOKEN $CI_REGISTRY
  script:
    - podman build --all-platforms --manifest $CI_REGISTRY_IMAGE --file Dockerfile .
    - podman manifest push --format v2s2 $CI_REGISTRY_IMAGE
```

Figure 21 is a sample job that builds a multi-architecture manifest and its images, which requires a few changes from the single architecture image. First, in this example we use `--all-platforms` build flag. This flag will tell podman to attempt to build an image for every architecture listed in the base image's manifest (i.e., taken from the Dockerfile `FROM` directive). Next, the `--manifest` build flag tells podman to produce a *manifest*. A container manifest stores pointers to a container image for each architecture. This allows an OCI runtime to pull down the images needed for the targeted platform.

When the manifest is pushed, the build system will upload the manifest and all associated container images to the container registry. In the case of GitLab, the `--v2s2` argument must be given to upload the manifest in the format it expects. As of this writing, it is a known issue for GitLab that manifests do not display properly in its UI. *

A similar capability for multi-architecture building exists for Docker, but is outside the scope of this manual.

*https://gitlab.com/groups/gitlab-org/-/epics/10434

Figure 22. Docker-in-Docker Build CI Job

```
1  docker:
2    image: docker.io/library/docker:latest
3    tags:
4      - dind
5    services:
6      - name: docker.io/library/docker:dind
7        alias: docker
8        command: ["--registry-mirror", "https://cfcc01.ornl.gov:4567"]
9    before_script:
10     - docker login --username=$CI_REGISTRY_USER --password=$CI_JOB_TOKEN $CI_REGISTRY
11   script:
12     - docker build --tag $CI_REGISTRY_IMAGE --file Dockerfile .
13     - docker push $CI_REGISTRY_IMAGE
```

Figure 22 shows an example job that will build an OCI image using DinD. Line 2 specifies that we will use the latest official docker container image (pulled from the PTC if configured). Line 4-5 specifies the runner tags that must be applied to a runner for it to pick up the job. In this case, the `dind` tag means we will be using the runner we registered using the docker runtime with rootful privileged mode enabled. The `services` section specifies that we will use the official DinD image, available inside the job with the network name `docker`, and that the DinD service should use our pull-through-cache registry mirror for container pulls.

In order to push images to the GitLab container registry, the CI job must log in (Line 11) to the container registry. GitLab CI makes automatic variables for that purpose available. We then build the container image and push it to the GitLab container registry (Line 9-10) using the automatic variable that points to the project registry.

Note that the Podman and Docker strategies as given here are mutually exclusive because they create and push a container image with identical names. You can modify the `--tag` argument to make images distinguishable so you can push multiple images.

Figure 23. Apptainer-in-Docker Build CI Job

```
1  singularity-build:
2    image: quay.io/centos/centos:stream9
3    tags:
4      - dind
5    script:
6      - yum groupinstall -y "Development Tools"
7      - yum install -y apptainer debootstrap
8      - sudo apptainer build -F ApptainerRecipe Apptainer.sif
9      - |
10       pn="Apptainer" # Package Name
11       pv=$(singularity --version | awk '{print $3}' | cut -d'-' -f 1) # Package version
12       curl --header "JOB-TOKEN: $CI_JOB_TOKEN" --upload-file $f \
13       ${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/generic/$pn/$pv/Apptainer.sif
```

In Figure 23 we build and push an Apptainer image. In this example, we pull the official Centos Stream 9 image from quay (Line 2). This image does not come with the tools needed to build Apptainer images, so we install the Centos development tools group, Apptainer itself, and debootstrap to enable bootstrapping of operating systems that use Debian package management(Lines 6-8).

The Apptainer image is distinct from the OCI images in that it is written as a file to the filesystem rather than stored in a container registry. Therefore, we utilize the GitLab Generic Package Registry feature to upload the Apptainer image. To upload, we need to select a package name and a package version. Both are up to the maintainer. In this example, we are using a filler package name, and we match our image version to our Apptainer version. You could also use your git branch name, git hash, or some other value. Be aware that GitLab Generic Packages will store duplicates of uploads with the same name. Therefore, it is important to configure your package retention policy in the project settings. There, you can set GitLab to erase duplicate packages.

4.8 PUTTING IT ALL TOGETHER

Utilizing the strategy described above, a project has been developed that creates base OS images of all ORNL approved operating systems mirrored at `repoman.ornl.gov` [3]. The examples above are simplified for brevity, so you can refer to the code repository for additional details such as building and pushing multiple container images to the same project container registry, utilizing `make` for build management, utilizing the built images in test cases, and demonstrating that Podman and Docker images are cross-compatible thanks to OCI compliance.

These base images can be pulled from the repository for use by any researcher. One particular challenge that has been encountered is the ORNL Netskope security tool blocking access to package repositories from within a containerized environment. These internal images alleviate that issue by (1) installing the Netskope certificates needed to access external sites, and (2) preferring local ORNL mirrors over Internet mirrors, although allowing fallback to public mirrors.

5. ALTERNATIVE STRATEGIES

This technical report describes setting up a GitLab podman based container factory. Several other strategies were either researched, trialed, or even fully developed before podman became the primary method. More information regarding these alternative methods is available in the git repository that is associated with this technical report [3].

5.1 DOCKER-IN-DOCKER EXECUTOR

When this project first started, the only method available to build a container in a container was to utilize DinD[†]. Therefore, this was the first strategy utilized, and is still the strategy utilized for building Apptainer container images, because Apptainer images builds, as of this writing, require more rootful privileges than can be provided container-in-container [6].

The primary drawback to this strategy is that the executor must be run in privileged modes, which removes all guardrails from the docker service. Effectively, the CI pipeline has trivial root access to the host. This strategy should therefore be used to the minimum necessary extent, and access to these executors must be tightly controlled.

5.2 VIRTUALBOX EXECUTOR

To work around the implied root privileges of Docker, a VirtualBox executor was experimented with. The executor was configured to revert the virtual machine to a known state after every job execution. While this method provided ephemeral executions, it is complex to set up and resource intensive. This direction was abandoned as soon as podman capabilities became viable.

5.3 PODMAN-IN-DOCKER EXECUTOR

When nesting the podman runtime in a Docker environment, podman must be configured to use the *vfs* storage driver which causes a substantial performance penalty. Furthermore, because the docker service is involved, a container breakout scenario still results in potential root access to the host. This strategy is not recommended because there are now simpler, more performant, and more secure options available.

5.4 PODMAN-IN-PODMAN CUSTOM EXECUTOR

Prior to the collaboration between GitLab, Red Hat, and Containers[‡] that provided first class podman support, a custom podman executor could be utilized with GitLab [2]. This strategy provided most of the same benefits as first class support would later bring. However, implementation was fairly complex and in the author's experience resulted in clients being hesitant to use it in production. This method is no longer recommended because a simpler, more reliable, and first class method are now available.

5.5 SHELL

Shell executors execute commands directly on the host machine which risks leaving behind data intentionally or unintentionally, and running as a root user in a shell executor would allow users to make

[†] https://docs.gitlab.com/ee/ci/docker/using_docker_build.html
[‡] https://github.com/containers

unhindered changes to the host. The security properties and maintenance labor of a shell executor were therefore determined not to warrant exploration for the purposes of a container factory.

6. CONCLUSION

This report has covered the necessary components to build a container-in-container factory utilizing resources available to all ORNL staff. We have also discussed many different strategies that may be employed and their respective strengths and weaknesses. These principles can be generalized to apply to any CI/CD ecosystem. This capability provides users with powerful automation tools that aid in repeatable and cross platform builds and distributions of their software utilizing containerization tools and common software engineering tools and practices. While this manual covers tools and services provided to researchers at ORNL, analogous tool sets are commonly found in other enterprise and open source ecosystems and the principles described here are easily transferable.

7. FUTURE WORK

There are a number of aspects of DevSecOps that remain to be documented and explored further in the ORNL research environment. Topics include the broader abstraction of a *software factory*, container signing, vulnerability scanning and mitigation, and PTC authorization [5].

7.1 SOFTWARE FACTORY

The *container factory* concepts described here are intended to be a component of a *software factory*. The software factory is an automated end-to-end software lifeycycle concept including software build, test, containerization, delivery, and deployment. This document covers only *containerization* in the context of container-in-container image builds. Future work will cover more of the software life cycle and additional security measures and metrics, e.g., container security scanning.

7.2 CONTAINER VULNERABILITY SCANNING AND MITIGATION

There are several open source tools available that can be incorporated into CI/CD pipelines for container scanning. Among them are dockle, grype, and trivy. These tools can be run in the podman container-in-container environment described here, but may require some configuration to connect to a podman socket rather than the docker socket they expect.

7.3 CONTAINER SIGNING

An emerging topic of interest is trusted container signing. Container signing is analogous to software signing. However, containerization technologies do not currently have strong support for container signing. Provenance of the contents of containers is also a problem nested within this problem. Containers instead have relied on trusted "wateringholes" (e.g., the Docker Hub, Quay.io) and checksum integrity checks rather than the container images themselves being trusted. There are emerging technologies in this area (e.g., skopeo) that warrant investigation.

7.4 CACHE AUTHORIZATION

One weakness of the implementation given here is that the PTC has no user authorization. This presents a few issues. First, if the PTC is logged into the upstream container registry, then new image pulls consume the logged in user's pull quota. Second, the PTC has access to any private repositories of the logged in user and therefore any user with access to the cache can download those private images. This latter issue also applies to quay.io.

The PTC implemented here has been restricted with OpenStack network security groups such that only the runners can access the cache, and it does not log in to any registry watering holes and therefore relies on the anonymous user pull quota and has no access to sensitive data.

8. REFERENCES

[1] GitLab. Gitlab documenttion. https://docs.gitlab.com/, 2023. [Online; accessed 12-April-2023].

[2] J. Goetz. gitlab-executor-podman. https://gitlab.com/johngoetz/gitlab-executor-podman, 2021. [Online; accessed 12-April-2023].

[3] D. Heise. Container factory. https://code.ornl.gov/hd8/container-factory, April 2023.

[4] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[5] D. of Defense. Dod enterprise devsecops reference design. https://dodcio.defense.gov/Portals/0/Documents/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20v1.0_Public%20Release.pdf?ver=2019-09-26-115824-583, August 2019. Accessed: 2023-04-12.

[6] Sylabs. Singularity. https://sylabs.io/, 2023. [Online; accessed 12-April-2023].