SingularityCE Admin Guide

Release 4.3

SingularityCE Project Contributors

Mar 13, 2025

CONTENTS

1 What's New in SingularityCE 4.3 **2**
- 1.1 Configuration . . . 2
- 1.2 Requirements & Packaging . . . 2
- 1.3 Removed Features . . . 2

2 Admin Quick Start **3**
- 2.1 Architecture of SingularityCE . . . 3
- 2.2 SingularityCE Security . . . 4
- 2.3 OCI Compatibility . . . 4
- 2.4 Version Compatibility . . . 5
- 2.5 Installation from Source . . . 6
- 2.6 Installation from RPM/Deb Packages . . . 9
- 2.7 Configuration . . . 10
- 2.8 Test SingularityCE . . . 10

3 Installing SingularityCE **11**
- 3.1 Installation on Linux . . . 11
- 3.2 Installation on Windows or Mac . . . 24

4 SingularityCE Configuration Files **29**
- 4.1 singularity.conf . . . 29
- 4.2 cgroups.toml . . . 38
- 4.3 ecl.toml . . . 40
- 4.4 GPU Library Configuration . . . 42
- 4.5 capability.json . . . 43
- 4.6 seccomp-profiles . . . 45
- 4.7 remote.yaml . . . 46

5 User Namespaces & Fakeroot **48**
- 5.1 User Namespace Requirements . . . 48
- 5.2 Unprivileged Installations . . . 49
- 5.3 –userns option . . . 49
- 5.4 Fakeroot feature . . . 49
- 5.5 Unprivileged Builds Without User Namespaces . . . 53

6 Security in SingularityCE **54**
- 6.1 Security Policy . . . 54
- 6.2 Background . . . 54
- 6.3 Default Native Runtime . . . 55
- 6.4 OCI-Mode . . . 56
- 6.5 Security Implications of Unprivileged User Namespaces . . . 56

6.6 Singularity Image Format (SIF) 57
6.7 Plugins 57
6.8 Configuration & Runtime Options 57

7 Installed Files **59**

8 License **63**

Welcome to the SingularityCE Admin Guide!

This guide aims to cover installation instructions, configuration detail, and other topics important to system administrators working with SingularityCE.

See the user guide for more information about how to use SingularityCE.

CHAPTER

ONE

WHAT'S NEW IN SINGULARITYCE 4.3

This section highlights important changes in SingularityCE 4.3 that are of note to system administrators. See also the "What's New" section in the User Guide for user-facing changes.

1.1 Configuration

- Support for libsubid. Subid mappings will be retrieved from e.g. LDAP according to `nssswitch.conf` if Singularity is built with libsubid support (default). If built without libsubid support, Singularity will retrieve subid directly from `/etc/subid` and `/etc/subgid` regardless of system configuration. Note that `singularity config fakeroot` always modifies the `/etc/subid` and `/etc/subgid` files.

1.2 Requirements & Packaging

- Go 1.23.4 or above is now required to build SingularityCE.
- libsubid headers are now required to build SingularityCE, unless the `--without-libsubid` flag is passed to `mconfig`.
- EL RPM packages are built with libsubid support.
- Ubuntu deb packages are built without libsubid support.
- The RPM spec file no longer includes rules for SLES / openSUSE package builds, which have been untested / unsupported for some time.
- Conmon sources are no longer bundled and built with SingularityCE. Install the `conmon` package from your distribution, or upstream binary, if you need to use the `singularity oci` commands. Note that conmon is not required for `--oci` mode.

1.3 Removed Features

- Plugin `fakerootcallback` functionality for customizing fakeroot subid mappings has been removed. Use the libsubid integration to provide subid mappings from a custom source.

CHAPTER

TWO

ADMIN QUICK START

This quick start gives an overview of installation of SingularityCE from source, a description of the architecture of SingularityCE, and pointers to configuration files. More information, including alternate installation options and detailed configuration options can be found later in this guide.

2.1 Architecture of SingularityCE

SingularityCE is designed to allow containers to be executed as if they were native programs or scripts on a host system. No daemon is required to build or run containers, and the security model is compatible with shared systems.

As a result, integration with clusters and schedulers such as Univa Grid Engine, Torque, SLURM, SGE, and many others is as simple as running any other command. All standard input, output, errors, pipes, IPC, and other communication pathways used by locally running programs are synchronized with the applications running locally within the container.

Beginning with SingularityCE 4 there are two modes in which to run containers:

- The default native mode uses a runtime that is unique to SingularityCE. This is fully compatible with containers built for, and by, SingularityCE versions 2 and 3.
- The optional OCI-mode uses a standard low-level OCI runtime to execute OCI containers natively, for improved compatibility.

SingularityCE favors an 'integration over isolation' approach to containers in native mode. By default only the mount namespace is isolated for containers, so that they have their own filesystem view. Default access to user home directories, `/tmp` space, and installation specific mounts makes it simple for users to benefit from the reproducibility of containerized applications without major changes to their existing workflows.

In OCI-mode, more isolation is used by default so that behavior is similar to Docker and other OCI runtimes. However, networking is not virtualized and SingularityCE's traditional behavior can be enabled with the `--no-compat` option.

In both modes, access to hardware such as GPUs, high speed networks, and shared filesystems is easy and does not require special configuration. Where more complete isolation is important, SingularityCE can use additional Linux namespaces and other security and resource limits for that purpose.

2.2 SingularityCE Security

Note: See also the *security section* (page 54) of this guide, for more detail.

When using the native runtime, a default installation of SingularityCE runs small amounts of privileged container setup code via a `starter-setuid` binary. This is a 'setuid root' binary, used so that SingularityCE can perform mounts, create namespaces, and enter containers even on older systems that lack support for fully unprivileged container execution. The setuid flow is the default mode of operation, but *can be disabled* (page 21) upon build, or in the `singularity.conf` configuration file if required.

If setuid is disabled, or OCI-mode is used, SingularityCE sets up containers within an unprivileged user namespace. This makes use of features of newer kernels, as well as user space filesystem mounts (FUSE).

Note: Running SingularityCE in non-setuid mode requires unprivileged user namespace support in the operating system kernel and does not support all features. This impacts integrity/security guarantees of containers at runtime.

See the *non-setuid installation section* (page 21) for further detail on how to install SingularityCE to run in non-setuid mode.

SingularityCE uses a number of strategies to provide safety and ease-of-use on both single-user and shared systems. Notable security features include:

- When using the default native runtime, with container setup via a setuid helper program, the effective user inside a container is the same as the user who ran the container. This means access to files and devices from the container is easily controlled with standard POSIX permissions.
- When using OCI-mode, or an unprivileged installation, subuid/subgid mappings allows users access to other uids & gids in the container, which map to safe administrator-defined ranges on the host.
- Container filesystems are mounted `nosuid` and container applications run with the prctl `NO_NEW_PRIVS` flag set. This means that applications in a container cannot gain additional privileges. A regular user cannot `sudo` or otherwise gain root privilege on the host via a container.
- The Singularity Image Format (SIF) supports encryption of containers, as well as cryptographic signing and verification of their content.
- SIF containers are immutable and their payload is run directly, without extraction to disk. This means that the container can always be verified, even at runtime, and encrypted content is not exposed on disk.
- Restrictions can be configured to limit the ownership, location, and cryptographic signatures of containers that are permitted to be run.

2.3 OCI Compatibility

SingularityCE allows users to run, and build from, the majority of OCI containers created with tools such as Docker. Beginning with SingularityCE 3.11, there are two modes of operation that support OCI containers in different ways.

SingularityCE's *native runtime*, used by default, supports all features that are exposed via the `singularity` command. It builds and runs containers in SingularityCE's own on-disk formats. When an OCI container is pulled or built into a SingularityCE image, a translation step occurs. While most OCI images are supported as-is, there are some limitations and compatibility options may be required.

SingularityCE 4's OCI-mode, enabled with the `--oci` flag, runs containers using a low-level OCI runtime - either `crun` or `runc`. The container is executed from a native OCI format on-disk. Not all CLI features are currently implemented,

but OCI containers using the `USER` directive or which are otherwise incompatible with SingularityCE's native runtime are better supported. Note that OCI-mode has additional *system requirements* (page 11).

2.4 Version Compatibility

Up to and including version 4, the major version number of SingularityCE was increased only when a significant change or addition was made to the container format:

- v1 packaged applications in a different manner than later versions, and was not widely deployed.
- v2 used extfs or squashfs bare image files for the container root filesystem.
- v3 introduced and switched to the Singularity Image Format (SIF).
- v4 added OCI-SIF images, a variant of SIF encapsulating OCI containers directly. These are used by the new OCI-mode.

Minor versions, e.g. within the 3.x series, frequently introduced changes to existing behavior not related to the basic container format.

Beginning with version 4, SingularityCE aims to follow semantic versioning where breaking changes to the CLI or runtime behavior will also be limited to a new major version. New features that do not modify existing behavior may be introduced in minor version updates.

2.4.1 Backward Compatibility

Execution of container images from 2 prior major versions is supported. SingularityCE 4 can run container images created with versions 2 and 3. Except where documented in the project changelog, differences in behaviour when running v2 or v3 containers using the native runtime in setuid mode are considered bugs.

SingularityCE 4's OCI-mode cannot perfectly emulate the behavior of the native runtime in setuid mode. Although most workflows are supported, complex containers created with SingularityCE 2 or 3 may not run as expected in OCI-mode.

2.4.2 Forward Compatibility

SingularityCE 4 can build SIF container images that can be run with version 3. The scope of this forward compatibility depends on the features used when creating the container, and the 3.x minor version used to run the container:

- The OCI-SIF format (OCI-mode) is not supported before v4.
- The SIF DSSE signature format (key / certificate based signing) was introduced at v3.11.0.
- The SIF PGP signature format was changed at v3.6.0, therefore older versions cannot verify newer signatures.
- Container / host environment handling was modified at v3.6.0.
- LUKS2 encrypted containers are not supported prior to v3.4.0

SingularityCE 4.1 will build container images that can be run with version 4.0, with the exception of multi-layer OCI-SIF images. A multi-layer OCI-SIF created with the 4.1 `--keep-layers` option must be executed using version 4.1 or later.

SingularityCE 4.3 can add a cosign-compatible signature to OCI-SIF images. These cosign signed OCI-SIF images are not compatible with version 4.2 or older.

2.5 Installation from Source

SingularityCE can be installed from source directly, or by building an RPM package from the source. Linux distributions may also package SingularityCE, but their packages may not be up-to-date with the upstream version on GitHub.

To install SingularityCE directly from source, follow the procedure below. Other methods are discussed in the *Installation* (page 11) section.

Note: This quick-start that you will install as `root` using `sudo`, so that SingularityCE uses the default `setuid` workflow, and all features are available. See the *non-setuid installation* (page 21) section of this guide for detail of how to install as a non-root user, and how this affects the functionality of SingularityCE.

2.5.1 Install Dependencies

On Debian-based systems, including Ubuntu:

```
# Ensure repositories are up-to-date
sudo apt-get update
# Install debian packages for dependencies
sudo apt-get install -y \
   autoconf \
   automake \
   cryptsetup \
   fuse2fs \
   git \
   fuse \
   libfuse-dev \
   libseccomp-dev \
   libtool \
   pkg-config \
   runc \
   squashfs-tools \
   squashfs-tools-ng \
   uidmap \
   wget \
   zlib1g-dev
```

On Ubuntu 24.04 and above install additional libsubid headers:

```
sudo apt-get install -y libsubid-dev
```

On RHEL / Alma Linux / Rocky Linux, CentOS Stream, you will need to enable the CRB (9+) or powertools repository (8). This is not necessary on Fedora.

```
sudo dnf install -y dnf-plugins-core
sudo dnf config-manager --enable crb || dnf config-manager --enable powertools
```

You can now install the build dependencies:

```
# Install basic tools for compiling
sudo dnf groupinstall -y 'Development Tools'
```

(continues on next page)

(continued from previous page)

```
# Install RPM packages for dependencies
sudo dnf install -y \
   autoconf \
   automake \
   crun \
   cryptsetup \
   fuse \
   fuse3 \
   fuse3-devel \
   git \
   libseccomp-devel \
   libtool \
   shadow-utils-subid-devel \
   squashfs-tools \
   wget \
   zlib-devel
```

Note: You can build SingularityCE without `cryptsetup` available, but you will not be able to use encrypted containers without it installed on your system.

If you will not use the `singularity oci` commands, or OCI-mode, `crun` / `runc` is not required.

2.5.2 Recommended: Install sqfstar / tar2sqfs for OCI-mode

If you intend to use the `--oci` execution mode of SingularityCE, your system must provide either:

- `squashfs-tools / squashfs` >= 4.5, which provides the `sqfstar` utility. Older versions packaged by many distributions do not include `sqfstar`.
- `squashfs-tools-ng`, which provides the `tar2sqfs` utility. This is not packaged by all distributions.

Debian / Ubuntu

On Debian/Ubuntu `squashfs-tools-ng` is available in the distribution repositories. It has been included in the "Install system dependencies" step above. No further action is necessary.

RHEL / Alma Linux / Rocky Linux

On RHEL and derivatives, the `squashfs-tools-ng` package is now available in the EPEL repositories.

Follow the EPEL Quickstart for you distribution to enable the EPEL repository. Install `squashfs-tools-ng` with `dnf`.

```
sudo dnf install squashfs-tools-ng
```

2.5.3 Optional: Install conmon for singularity oci commands

If you intend to use the *singularity oci* commands, to run Singularity containers in an OCI lifecycle then *conmon* is required.

Debian / Ubuntu

`conmon` is available as a package for Ubuntu 24.10+ and Debian 12+

```
sudo apt get install conmon
```

On older versions, use the latest binary from the conmon GitHub releases page. E.g.

```
sudo curl -L -o /usr/local/bin/conmon https://github.com/containers/conmon/releases/
↪download/v2.1.13/conmon.amd64
sudo chmod +x /usr/local/bin/conmon
```

EL / Fedora

Install the distribution `conmon` package.

2.5.4 Install Go

SingularityCE is written in Go, and you will need Go installed to compile it from source. Versions of Go packaged by your distribution may not be new enough to build SingularityCE.

{SingularityCE} aims to maintain support for the two most recent stable versions of Go. This corresponds to the Go Release Maintenance Policy and Security Policy, ensuring critical bug fixes and security patches are available for all supported language versions.

The method below is one of several ways to install and configure Go.

Note: If you have previously installed Go from a download, rather than an operating system package, you should remove your `go` directory, e.g. `rm -r /usr/local/go` before installing a newer version. Extracting a new version of Go over an existing installation can lead to errors when building Go programs, as it may leave old files, which have been removed or replaced in newer versions.

Visit the Go download page and pick a package archive to download. Copy the link address and download with wget. Then extract the archive to `/usr/local` (or use other instructions on go installation page).

```
$ export VERSION=1.24.1 OS=linux ARCH=amd64 && \
    wget https://dl.google.com/go/go$VERSION.$OS-$ARCH.tar.gz && \
    sudo tar -C /usr/local -xzvf go$VERSION.$OS-$ARCH.tar.gz && \
    rm go$VERSION.$OS-$ARCH.tar.gz
```

Finally, add `/usr/local/go/bin` to the `PATH` environment variable:

```
echo 'export PATH=$PATH:/usr/local/go/bin' >> ~/.bashrc
source ~/.bashrc
```

2.5.5 Download SingularityCE from a GitHub release

You can download SingularityCE from one of the releases. To see a full list, visit the GitHub release page. After deciding on a release to install, you can run the following commands to proceed with the installation.

```
$ export VERSION=4.3.0 && # adjust this as necessary \
    wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/singularity-
↪ce-${VERSION}.tar.gz && \
    tar -xzf singularity-ce-${VERSION}.tar.gz && \
    cd singularity-ce-${VERSION}
```

2.5.6 Compile & Install SingularityCE

SingularityCE uses a custom build system called `makeit`. `mconfig` is called to generate a `Makefile` and then `make` is used to compile and install.

```
$ ./mconfig && \
    make -C ./builddir && \
    sudo make -C ./builddir install
```

By default SingularityCE will be installed in the `/usr/local` directory hierarchy. You can specify a custom directory with the `--prefix` option, to `mconfig`:

```
$ ./mconfig --prefix=/opt/singularity
```

This option can be useful if you want to install multiple versions of Singularity, install a personal version of SingularityCE on a shared system, or if you want to remove SingularityCE easily after installing it.

For a full list of `mconfig` options, run `mconfig --help`. Here are some of the most common options that you may need to use when building SingularityCE from source.

- `--without-libsubid`: Disable libsubid support, on distributions where it is not available. This prevents subuid/subgid mappings being retrieved from a source other than `/etc/subuid` / `/etc/subgid` files.
- `--sysconfdir`: Install read-only config files in sysconfdir. This option is important if you need the `singularity.conf` file or other configuration files in a custom location.
- `--localstatedir`: Set the state directory where containers are mounted. This is a particularly important option for administrators installing SingularityCE on a shared file system. The `--localstatedir` should be set to a directory that is present on each individual node.
- `-b`: Build SingularityCE in a given directory. By default this is `./builddir`.

2.6 Installation from RPM/Deb Packages

Sylabs provides `.rpm` packages of SingularityCE, for mainstream-supported versions of RHEL and derivatives (e.g. Alma Linux / Rocky Linux). We also provide `.deb` packages for current Ubuntu LTS releases.

These packages can be downloaded from the GitHub release page and installed using your distribution's package manager.

The packages are provided as a convenience for users of the open source project, and are built in our public CircleCI workflow. They are not signed, but SHA256 sums are provided on the release page.

2.7 Configuration

SingularityCE is configured using files under `etc/singularity` in your `--prefix`, or `--syconfdir` if you used that option with `mconfig`. In a default installation from source without a `--prefix` set you will find them under `/usr/local/etc/singularity`. In a default installation from RPM or Deb packages you will find them under `/etc/singularity`.

You can edit these files directly, or using the `SingularityCE config global` command as the root user to manage them.

`singularity.conf` contains the majority of options controlling the runtime behavior of SingularityCE. Additional files control security, network, and resource configuration. Head over to the *Configuration files* (page 29) section where the files and configuration options are discussed.

2.8 Test SingularityCE

You can run a quick test of SingularityCE using a container in the Sylabs Container Library:

```
$ singularity exec library://alpine cat /etc/alpine-release
3.9.2
```

See the user guide for more information about how to use SingularityCE.

CHAPTER

THREE

INSTALLING SINGULARITYCE

This section will guide you through the process of installing SingularityCE 4.3.0 via several different methods. (For instructions on installing earlier versions of SingularityCE please see earlier versions of the docs.)

3.1 Installation on Linux

SingularityCE can be installed on any modern Linux distribution, on bare-metal or inside a Virtual Machine. Nested installations inside containers are not recommended, and require the outer container to be run with full privilege.

3.1.1 System Requirements

SingularityCE requires ~180MiB disk space once compiled and installed.

There are no specific CPU or memory requirements at runtime, though 2GB of RAM is recommended when building from source.

Full functionality of SingularityCE requires that the kernel supports:

- **OverlayFS mounts** - (minimum kernel >=3.18) Required for full flexibility in bind mounts to containers, and to support persistent overlays for writable containers.
- **Unprivileged user namespaces** - (minimum kernel >=3.8, >=3.18 recommended) Required to run containers without root or setuid privilege. Required to build containers unprivileged in `--fakeroot` mode. Required to run containers in OCI-mode (`-oci`).
- **FUSE in unprivileged user namespaces** - (minimum kernel >=4.18) Required to run containers in OCI-Mode (`-oci`).
- **Unprivileged overlay** - (minimum kernel >=5.11, >=5.13 recommended) Required to use `--overlay`, to mount a persistent overlay directory onto the container, when running without root or setuid in native mode. OCI-mode will fall-back to the fuse-overlayfs userspace implementation.

Note that the indicated kernel versions correspond to the mainline Linux kernel. Some Linux distributions may backport features to older kernels.

External Binaries

Singularity depends on a number of external binaries for full functionality. The methods that are used to find these binaries have been standardized as below.

Bundled Utilities

In a standard SingularityCE installation, the following are bundled and installed into SingularityCE's `libexec/bin` directory. However, at compilation time `mconfig` options can be used to disable building these tools, in which case they will be searched for on `$PATH` at runtime.

- `squashfuse` or `squashfuse_ll` are used to mount squashfs filesystems from OCI-SIF images in OCI-mode. They may be used to mount squashfs filesystems from SIF images and bare squashfs containers in non-OCI mode.

Configurable Paths

The following binaries are found on `$PATH` during build time when `./mconfig` is run, and their location is added to the `singularity.conf` configuration file. At runtime this configured location is used. To specify an alternate executable, change the relevant path entry in `singularity.conf`.

- `cryptsetup` version 2 with kernel LUKS2 support is required for building or executing encrypted containers.
- `ldconfig` is used to resolve library locations / symlinks when using the `-nv` or `--rocm` GPU support.
- `nvidia-container-cli` is used to configure a container for Nvidia GPU / CUDA support when running with the experimental `--nvccli` option.

For the following additional binaries, if the `singularity.conf` entry is left blank, then `$PATH` will be searched at runtime.

- `go` is required to compile plugins, and must be an identical version as that used to build SingularityCE.
- `mksquashfs` from squashfs-tools 4.3+ is used to create the squashfs container filesystem that is embedded into SIF container images. The `mksquashfs procs` and `mksquashfs mem` directives in `singularity.conf` can be used to control its resource usage.
- `unsquashfs` from squashfs-tools 4.3+ is used to extract the squashfs container filesystem from a SIF file when necessary.

Searching $PATH

The following utilities are always found by searching `$PATH` at runtime:

- `true`
- `mkfs.ext3` is used to create overlay images.
- `conmon` is used for the `singularity oci` commands, but not `--oci` mode.
- `cp`
- `dd`
- `newuidmap` and `newgidmap` are distribution provided setuid binaries used to configure subuid/gid mappings for `--fakeroot` in non-setuid installs, and in OCI-mode.

- `crun` or `runc` are OCI runtimes used for the `singularity oci` commands and OCI-mode for `run / shell / exec`. `crun` is preferred over `runc` if it is available. `runc` is provided by a package in all common Linux distributions. `crun` is packaged in more recent releases of common Linux distributions.
- `proot` is an optional dependency that can be used to permit limited unprivileged builds without user namespace / subuid support. It is packaged in the community repositories for common Linux distributions, and is available as a static binary from proot-me.github.io.
- `sqfstar` or `tar2sqfs` are used in the creation of OCI-SIF images from OCI sources, in OCI-mode (`--oci`).
- `fuse2fs` is used to mount extfs images in unprivileged flows, or when kernel extfs mount is disabled by configuration. Version 1.46.6 or above is required to use OCI-SIF embedded overlays in OCI-mode.
- `fuse-overlayfs` is used to setup overlay filesystems when the kernel does not support unprivileged overlay or the required overlay configuration.
- `fusermount3` or `fusermount` is used to unmount FUSE filesystems safely, in OCI-mode and other flows.

Bootstrap Utilities

The following utilities are required to bootstrap containerized distributions using their native tooling:

- `mount`, `umount`, `pacstrap` for Arch Linux.
- `mount`, `umount`, `mknod`, `debootstrap` for Debian based distributions.
- `dnf`, `rpm`, `curl` for EL derived RPM based distributions.
- `uname`, `zypper`, `SUSEConnect` for SLES derived RPM based distributions.

Installing sqfstar / tar2sqfs for OCI-mode

If you intend to use the `--oci` execution mode of SingularityCE, your system must provide either:

- `squashfs-tools / squashfs` >= 4.5, which provides the `sqfstar` utility. Older versions packaged by many distributions do not include `sqfstar`.
- `squashfs-tools-ng`, which provides the `tar2sqfs` utility. This is not packaged by all distributions.

Debian / Ubuntu

On Debian/Ubuntu `squashfs-tools-ng` is available in the distribution repositories. It has been included in the "Install system dependencies" step above. No further action is necessary.

RHEL / Alma Linux / Rocky Linux

On RHEL and derivatives, the `squashfs-tools-ng` package is now available in the EPEL repositories.

Follow the EPEL Quickstart for you distribution to enable the EPEL repository. Install `squashfs-tools-ng` with `dnf`.

```
sudo dnf install squashfs-tools-ng
```

Installing conmon for singularity oci commands

If you intend to use the *singularity oci* commands, to run Singularity containers in an OCI lifecycle then *conmon* is required.

Debian / Ubuntu

`conmon` is available as a package for Ubuntu 24.10+ and Debian 12+

```
sudo apt get install conmon
```

On older versions, use the latest binary from the conmon GitHub releases page. E.g.

```
sudo curl -L -o /usr/local/bin/conmon https://github.com/containers/conmon/releases/
↪download/v2.1.13/conmon.amd64
sudo chmod +x /usr/local/bin/conmon
```

EL / Fedora

Install the distribution `conmon` package.

Non-standard ldconfig / Nix & Guix Environments

If SingularityCE is installed under a package manager such as Nix or Guix, but on top of a standard Linux distribution (e.g. RHEL or Debian), it may be unable to correctly find the libraries for `--nv` and `--rocm` GPU support. This issue occurs as the package manager supplies an alternative `ldconfig`, which does not identify GPU libraries installed from host packages.

To allow SingularityCE to locate the host (i.e. RHEL / Debian) GPU libraries correctly, set `ldconfig path` in `singularity.conf` to point to the host `ldconfig`. I.E. it should be set to `/sbin/ldconfig` or `/sbin/ldconfig.real` rather than a Nix or Guix related path.

Filesystem support / limitations

SingularityCE supports most filesystems, but there are some limitations when installing SingularityCE on, or running containers from, common parallel / network filesystems. In general:

- We strongly recommend installing SingularityCE on local disk on each compute node.
- If SingularityCE is installed to a network location, a `--localstatedir` should be provided on each node, and Singularity configured to use it.
- The `--localstatedir` filesystem should support overlay mounts.
- `TMPDIR` / `SINGULARITY_TMPDIR` should be on a local filesystem wherever possible.

Note: Set the `--localstatedir` location by by providing `--localstatedir my/dir` as an option when you configure your SingularityCE build with `./mconfig`.

Disk usage at the `--localstatedir` location is negligible (<1MiB). The directory is used as a location to mount the container root filesystem, overlays, bind mounts etc. that construct the runtime view of a container. You will not see these mounts from a host shell, as they are made in a separate mount namespace.

Overlay support

Various features of SingularityCE, such as the `--writable-tmpfs` and `--overlay` options, use overlay mounts to construct a container root filesystem that combines files from different locations. Overlay mounts may use the Linux kernel overlay filesystem driver or the fuse-overlayfs userspace implementation, depending on the workflow and support from the host kernel.

Overlays are mounted with the Linux kernel driver when:

- The native runtime is used in setuid mode.
- The native runtime is used in unprivileged / non-setuid mode, and the kernel supports unprivileged overlay mounts.
- OCI-mode is used without an extfs overlay image, and the kernel supports unprivileged overlay mounts.

Overlays are mounted with the fuse-overlayfs userspace implementation when:

- OCI-mode is used, and the kernel does not support unprivileged overlay mounts.
- OCI-mode is used, with an extfs overlay image.

Not all filesystems can be used with the overlay driver, so when containers are run from these filesystems some SingularityCE features may not be available.

Overlay support has two aspects:

- `lowerdir` support for a filesystem allows a directory on that filesystem to act as the 'base' of a container. A filesystem must support overlay `lowerdir` for you be able to run a Singularity sandbox container on it, while using functionality such as `--writable-tmpfs` / `--overlay`.
- `upperdir` support for a filesystem allows a directory on that filesystem to be merged on top of a `lowerdir` to construct a container. If you use the `--overlay` option to overlay a directory onto a container, then the filesystem holding the overlay directory must support `upperdir`.

Note that any overlay limitations mainly apply to sandbox (directory) containers only. A SIF container is mounted into the `--localstatedir` location, which should generally be on a local filesystem that supports overlay.

Fakeroot & OCI-Mode subuid/gid mapping

When SingularityCE is run using the *fakeroot* (page 49) option, or in OCI-Mode, it creates a user namespace for the container, and UIDs / GIDs in that user namespace are mapped to different host UID / GIDs.

Most local filesystems (ext4/xfs etc.) support this uid/gid mapping in a user namespace.

Most network filesystems (NFS/Lustre/GPFS etc.) *do not* support this uid/gid mapping in a user namespace. Because the fileserver is not aware of the mappings it will deny many operations, with 'permission denied' errors. This is currently a generic problem for rootless container runtimes.

SingularityCE cache / atomic rename

SingularityCE will cache SIF container images generated from remote sources, and any OCI/docker layers used to create them. The cache is created at `$HOME/.singularity/cache` by default. The location of the cache can be changed by setting the `SINGULARITY_CACHEDIR` environment variable.

The directory used for `SINGULARITY_CACHEDIR` should be:

- A unique location for each user. Permissions are set on the cache so that private images cached for one user are not exposed to another. This means that `SINGULARITY_CACHEDIR` cannot be shared.
- Located on a filesystem with sufficient space for the number and size of container images anticipated.
- Located on a filesystem that supports atomic rename, if possible.

In SingularityCE version 3.6 and above the cache is concurrency safe. Parallel runs of SingularityCE that would create overlapping cache entries will not conflict, as long as the filesystem used by `SINGULARITY_CACHEDIR` supports atomic rename operations.

Support for atomic rename operations is expected on local POSIX filesystems, but varies for network / parallel filesystems and may be affected by topology and configuration. For example, Lustre supports atomic rename of files only on a single MDT. Rename on NFS is only atomic to a single client, not across systems accessing the same NFS share.

If you are not certain that your `$HOME` or `SINGULARITY_CACHEDIR` filesystems support atomic rename, do not run `singularity` in parallel using remote container URLs. Instead use `singularity pull` to create a local SIF image, and then run this SIF image in a parallel step. An alternative is to use the `--disable-cache` option, but this will result in each SingularityCE instance independently fetching the container from the remote source, into a temporary location.

NFS

NFS filesystems support overlay mounts as a `lowerdir` only, and do not support user-namespace (sub)uid/gid mapping.

- Containers run from SIF files located on an NFS filesystem do not have restrictions.
- You cannot use `--overlay mynfsdir/` to overlay a directory onto a container when the overlay (upperdir) directory is on an NFS filesystem.
- When using `--fakeroot` to build or run a container, your `TMPDIR` / `SINGULARITY_TMPDIR` should not be set to an NFS location.
- You should not run a sandbox container with `--fakeroot` from an NFS location.

Lustre / GPFS / PanFS

Lustre, GPFS, and PanFS do not have sufficient `upperdir` or `lowerdir` overlay support for certain SingularityCE features, and do not support user-namespace (sub)uid/gid mapping.

- You cannot use `--overlay` or `--writable-tmpfs` with a sandbox container that is located on a Lustre, GPFS, or PanFS filesystem. SIF containers on Lustre, GPFS, and PanFS will work correctly with these options.
- You cannot use `--overlay` to overlay a directory onto a container, when the overlay (upperdir) directory is on a Lustre, GPFS, or PanFS filesystem.
- When using `--fakeroot` to build or run a container, your `TMPDIR/SINGULARITY_TMPDIR` should not be a Lustre, GPFS, or PanFS location.
- You should not run a sandbox container with `--fakeroot` from a Lustre, GPFS, or PanFS location.

OCI-mode Limitations

Because SingularityCE 4's OCI-mode is unprivileged, and never uses a setuid starter executable for container configuration, it has requirements that may not be satisified by older Linux distributions.

OCI-mode, including Dockerfile builds to OCI-SIF, will generally operate correctly on Linux distributions that use kernel 4.18 or later and v2 cgroups. Some distributions that use earlier kernels may have backported functionality that allows OCI-Mode to be used, but certain features may be limited as below. Distributions using v1 cgroups also have limitations, discussed below.

RHEL / Alma Linux / Rocky Linux

On RHEL 9, all features of OCI-mode are supported. `crun` is the recommended low-level runtime, and is listed as a requirement by SingularityCE RPM packages.

On RHEL 8, container resource limits cannot be applied as v1 cgroups are used by default. `crun` is the recommended low-level runtime, and is listed as a requirement by SingularityCE RPM packages.

Ubuntu

On Ubuntu 22.04 LTS, `runc` is the recommended low-level runtime, and is listed as a requirement by SingularityCE Deb packages. The `--no-setgroups` option, to preserve host supplementary group membership, is not supported by `runc`.

On Ubuntu 20.04 LTS, container resource limits cannot be applied as v1 cgroups are used by default. `runc` is the recommended low-level runtime, and is listed as a requirement by SingularityCE Deb packages. The `--no-setgroups` option, to preserve host supplementary group membership, is not supported by `runc`.

3.1.2 Install from Provided RPM / Deb Packages

Sylabs provides `.rpm` packages of SingularityCE, for mainstream-supported versions of RHEL and derivatives (e.g. Alma Linux / Rocky Linux). We also provide `.deb` packages for current Ubuntu LTS releases.

These packages can be downloaded from the GitHub release page and installed using your distribution's package manager.

The packages are provided as a convenience for users of the open source project, and are built in our public CircleCI workflow. They are not signed, but SHA256 sums are provided on the release page.

3.1.3 Install from Source

To use the latest version of SingularityCE from GitHub you will need to build and install it from source. This may sound daunting, but the process is straightforward, and detailed below.

If you have an earlier version of SingularityCE installed, you should *remove it* (page 23) before executing the installation commands. You will also need to install some dependencies and install Go.

Install Dependencies

On Debian-based systems, including Ubuntu:

```
# Ensure repositories are up-to-date
sudo apt-get update
# Install debian packages for dependencies
sudo apt-get install -y \
   autoconf \
   automake \
   cryptsetup \
   fuse2fs \
   git \
   fuse \
   libfuse-dev \
   libseccomp-dev \
   libtool \
   pkg-config \
   runc \
   squashfs-tools \
   squashfs-tools-ng \
   uidmap \
   wget \
   zlib1g-dev
```

On Ubuntu 24.04 and above install additional libsubid headers:

```
sudo apt-get install -y libsubid-dev
```

On RHEL / Alma Linux / Rocky Linux, CentOS Stream, you will need to enable the CRB (9+) or powertools repository (8). This is not necessary on Fedora.

```
sudo dnf install -y dnf-plugins-core
sudo dnf config-manager --enable crb || dnf config-manager --enable powertools
```

You can now install the build dependencies:

```
# Install basic tools for compiling
sudo dnf groupinstall -y 'Development Tools'
# Install RPM packages for dependencies
sudo dnf install -y \
   autoconf \
   automake \
   crun \
   cryptsetup \
   fuse \
   fuse3 \
   fuse3-devel \
   git \
   libseccomp-devel \
   libtool \
   shadow-utils-subid-devel \
   squashfs-tools \
   wget \
```

(continues on next page)

(continued from previous page)

```
   zlib-devel
```

Note: You can build SingularityCE without `cryptsetup` available, but you will not be able to use encrypted containers without it installed on your system.

If you will not use the `singularity oci` commands, or OCI-mode, `crun` / `runc` is not required.

Install Go

SingularityCE is written in Go, and aims to maintain support for the two most recent stable versions of Go. This corresponds to the Go Release Maintenance Policy and Security Policy, ensuring critical bug fixes and security patches are available for all supported language versions.

Building SingularityCE may require a newer version of Go than is available in the repositories of your distribution. We recommend installing the latest version of Go from the [official binaries](https://golang.org/dl/).

This is one of several ways to install and configure Go.

Note: If you have previously installed Go from a download, rather than an operating system package, you should remove your `go` directory, e.g. `rm -r /usr/local/go` before installing a newer version. Extracting a new version of Go over an existing installation can lead to errors when building Go programs, as it may leave old files, which have been removed or replaced in newer versions.

Visit the Go download page and pick a package archive to download. Copy the link address and download with wget. Then extract the archive to `/usr/local` (or use other instructions on go installation page).

```
$ export VERSION=1.24.1 OS=linux ARCH=amd64 && \
    wget https://dl.google.com/go/go$VERSION.$OS-$ARCH.tar.gz && \
    sudo tar -C /usr/local -xzvf go$VERSION.$OS-$ARCH.tar.gz && \
    rm go$VERSION.$OS-$ARCH.tar.gz
```

Then, set up your environment for Go.

```
$ echo 'export GOPATH=${HOME}/go' >> ~/.bashrc && \
    echo 'export PATH=/usr/local/go/bin:${PATH}:${GOPATH}/bin' >> ~/.bashrc && \
    source ~/.bashrc
```

Download SingularityCE from a release

You can download SingularityCE from one of the releases. To see a full list, visit the GitHub release page. After deciding on a release to install, you can run the following commands to proceed with the installation.

```
$ export VERSION=4.3.0 && # adjust this as necessary \
    wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/singularity-
↪ce-${VERSION}.tar.gz && \
    tar -xzf singularity-ce-${VERSION}.tar.gz && \
    cd singularity-ce-${VERSION}
```

Checkout Code from Git

The following commands will install SingularityCE from the GitHub repo to `/usr/local`. This method will work for >=v4.3.0. To install an older tagged release see older versions of the docs.

When installing from source, you can decide to install from either a **tag**, a **release branch**, or from the **main branch**.

- **tag**: GitHub tags form the basis for releases, so installing from a tag is the same as downloading and installing a specific release. Tags are expected to be relatively stable and well-tested.
- **release branch**: A release branch represents the latest version of a minor release with all the newest bug fixes and enhancements (even those that have not yet made it into a point release). For instance, to install v3.10 with the latest bug fixes and enhancements checkout `release-3.10`. Release branches may be less stable than code in a tagged point release.
- **main branch**: The `main` branch contains the latest, bleeding edge version of SingularityCE. This is the default branch when you clone the source code, so you don't have to check out any new branches to install it. The `main` branch changes quickly and may be unstable.

To ensure that the SingularityCE source code is downloaded to the appropriate directory use these commands.

```
$ git clone --recurse-submodules https://github.com/sylabs/singularity.git && \
    cd singularity && \
    git checkout --recurse-submodules v4.3.0
```

Compile Singularity

SingularityCE uses a custom build system called `makeit`. `mconfig` is called to generate a `Makefile` and then `make` is used to compile and install.

To support the SIF image format, automated networking setup etc., and older Linux distributions without user namespace support, Singularity must be `make install``ed as root or with ``sudo`, so it can install the `libexec/singularity/bin/starter-setuid` binary with root ownership and setuid permissions for privileged operations. If you need to install as a normal user, or do not want to use setuid functionality *see below* (page 21).

```
$ ./mconfig && \
    make -C ./builddir && \
    sudo make -C ./builddir install
```

By default SingularityCE will be installed in the `/usr/local` directory hierarchy. You can specify a custom directory with the `--prefix` option, to `mconfig` like so:

```
$ ./mconfig --prefix=/opt/singularity
```

This option can be useful if you want to install multiple versions of SingularityCE, install a personal version of SingularityCE on a shared system, or if you want to remove SingularityCE easily after installing it.

For a full list of `mconfig` options, run `mconfig --help`. Here are some of the most common options that you may need to use when building SingularityCE from source.

- `--without-libsubid`: Disable libsubid support, on distributions where it is not available. This prevents subuid/subgid mappings being retrieved from a source other than `/etc/subuid` / `/etc/subgid` files.
- `--sysconfdir`: Install read-only config files in sysconfdir. This option is important if you need the `singularity.conf` file or other configuration files in a custom location.

- `--localstatedir`: Set the state directory where containers are mounted. This is a particularly important option for administrators installing SingularityCE on a shared file system. The `--localstatedir` should be set to a directory that is present on each individual node.
- `-b`: Build SingularityCE in a given directory. By default this is `./builddir`.
- **`--reproducible`: Enable support for reproducible builds. Ensures** that the compiled binaries do not include any temporary paths, the source directory path, etc. This disables support for building plugins.

Unprivileged (non-setuid) Installation

If you need to install SingularityCE as a non-root user, or do not wish to allow the use of a setuid root binary, you can configure SingularityCE with the `--without-suid` option to mconfig:

```
$ ./mconfig --without-suid --prefix=/home/dave/singularity-ce && \
    make -C ./builddir && \
    make -C ./builddir install
```

If you have already installed SingularityCE you can disable the setuid flow by setting the option `allow setuid = no` in `etc/singularity/singularity.conf` within your installation directory.

When SingularityCE does not use setuid all container execution will use a user namespace. This requires support from your operating system kernel, and imposes some limitations on functionality. You should review the *requirements* (page 48) and *limitations* (page 49) in the *user namespace* (page 48) section of this guide.

Relocatable Installation

Since SingularityCE 3.8, an unprivileged (non-setuid) installation is relocatable. As long as the structure inside the installation directory (`--prefix`) is maintained, it can be moved to a different location and SingularityCE will continue to run normally.

Relocation of a default setuid installation is not supported, as restricted location / ownership of configuration files is important to security.

Source bash completion file

To enjoy bash shell completion with SingularityCE commands and options, source the bash completion file:

```
$ . /usr/local/etc/bash_completion.d/singularity
```

Add this command to your `~/.bashrc` file so that bash completion continues to work in new shells. (Adjust the path if you installed SingularityCE to a different location.)

3.1.4 Build and install an RPM

If you use RHEL, RHEL derivatives, or SUSE, building and installing a Singularity RPM allows your SingularityCE installation to be more easily managed, upgraded and removed. You can build an RPM directly from the release tarball.

Note: Be sure to download the correct asset from the GitHub releases page. It should be named `singularity-ce-<version>.tar.gz`.

After installing the *dependencies* (page 18) and installing *Go* (page 19) as detailed above, you are ready to download the tarball and build and install the RPM.

```
$ export VERSION=4.3.0 && # adjust this as necessary \
    wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/singularity-
↪ce-${VERSION}.tar.gz && \
    rpmbuild -tb singularity-ce-${VERSION}.tar.gz && \
    sudo rpm -ivh ~/rpmbuild/RPMS/x86_64/singularity-ce-$VERSION-1.el9.x86_64.rpm && \
    rm -rf ~/rpmbuild singularity-ce-$VERSION*.tar.gz
```

If you encounter a failed dependency error for golang but installed it from source, build with this command:

```
rpmbuild -tb --nodeps singularity-ce-${VERSION}.tar.gz
```

Options to `mconfig` can be passed using the familiar syntax to `rpmbuild`. For example, if you want to force the local state directory to `/mnt` (instead of the default `/var`) you can do the following:

```
rpmbuild -tb --define='_localstatedir /mnt' singularity-ce-$VERSION.tar.gz
```

Note: It is very important to set the local state directory to a directory that physically exists on nodes within a cluster when installing SingularityCE in an HPC environment with a shared file system.

Build an RPM from Git source

Alternatively, to build an RPM from a branch of the Git repository you can clone the repository, directly `make` an rpm, and use it to install Singularity:

```
$ ./mconfig && \
make -C builddir rpm && \
sudo rpm -ivh ~/rpmbuild/RPMS/x86_64/singularity-ce-4.3.0.el9.x86_64.rpm # or whatever␣
↪version you built
```

To build an rpm with an alternative install prefix set `RPMPREFIX` on the make step, for example:

```
$ make -C builddir rpm RPMPREFIX=/usr/local
```

For finer control of the rpmbuild process you may wish to use `make dist` to create a tarball that you can then build into an rpm with `rpmbuild -tb` as above.

3.1.5 Remove an old version

In a standard installation of SingularityCE (when building from source), the command `sudo make -C builddir install` lists all the files as they are installed. You must remove all of these files and directories to completely remove SingularityCE.

```
$ sudo rm -rf \
    /usr/local/libexec/singularity \
    /usr/local/var/singularity \
    /usr/local/etc/singularity \
    /usr/local/bin/singularity \
    /usr/local/bin/run-singularity \
    /usr/local/etc/bash_completion.d/singularity
```

If you anticipate needing to remove SingularityCE, it might be easier to install it in a custom directory using the `--prefix` option to `mconfig`. In that case SingularityCE can be uninstalled simply by deleting the parent directory. Or it may be useful to install SingularityCE *using a package manager* (page 22) so that it can be updated and/or uninstalled with ease in the future.

3.1.6 Testing & Checking the Build Configuration

After installation you can perform a basic test of Singularity functionality by executing a simple container from the Sylabs Cloud library:

```
$ singularity exec library://alpine cat /etc/alpine-release
3.10.0
```

See the user guide for more information about how to use SingularityCE.

singularity buildcfg

Running `singularity buildcfg` will show the build configuration of an installed version of SingularityCE, and lists the paths used by SingularityCE. Use `singularity buildcfg` to confirm paths are set correctly for your installation, and troubleshoot any 'not-found' errors at runtime.

```
$ singularity buildcfg
PACKAGE_NAME=singularity-ce
PACKAGE_VERSION=4.3.0
BUILDDIR=/home/myuser/singularity/builddir
PREFIX=/usr/local
EXECPREFIX=/usr/local
BINDIR=/usr/local/bin
SBINDIR=/usr/local/sbin
LIBEXECDIR=/usr/local/libexec
DATAROOTDIR=/usr/local/share
DATADIR=/usr/local/share
SYSCONFDIR=/usr/local/etc
SHAREDSTATEDIR=/usr/local/com
LOCALSTATEDIR=/usr/local/var
RUNSTATEDIR=/usr/local/var/run
INCLUDEDIR=/usr/local/include
DOCDIR=/usr/local/share/doc/singularity-ce
```

(continues on next page)

(continued from previous page)

```
INFODIR=/usr/local/share/info
LIBDIR=/usr/local/lib
LOCALEDIR=/usr/local/share/locale
MANDIR=/usr/local/share/man
SINGULARITY_CONFDIR=/usr/local/etc/singularity
SESSIONDIR=/usr/local/var/singularity/mnt/session
PLUGIN_ROOTDIR=/usr/local/libexec/singularity/plugin
SINGULARITY_CONF_FILE=/usr/local/etc/singularity/singularity.conf
SINGULARITY_SUID_INSTALL=1
```

Note that the `LOCALSTATEDIR` and `SESSIONDIR` should be on local, non-shared storage.

The list of files installed by a successful `setuid` installation of SingularityCE can be found in the *appendix, installed files section* (page 59).

Test Suite

The SingularityCE codebase includes a test suite that is run during development using CI services.

If you would like to run the test suite locally you can run the test targets from the `builddir` directory in the source tree:

- `make check` runs source code linting and dependency checks
- `make test` runs basic unit and integration tests
- `make e2e-test` runs end-to-end tests, which exercise a large number of operations by calling the SingularityCE CLI with different execution profiles.

Note: Running the full test suite requires a `docker` installation, and `nc` in order to test docker and instance/networking functionality.

SingularityCE must be installed in order to run the full test suite, as it must run the CLI with setuid privilege for the `starter-suid` binary.

Warning: `sudo` privilege is required to run the full tests, and you should not run the tests on a production system. We recommend running the tests in an isolated development or build environment.

3.2 Installation on Windows or Mac

Linux container runtimes like SingularityCE cannot run natively on Windows or Mac because of basic incompatibilities with the host kernel. (Contrary to a popular misconception, macOS does not run on a Linux kernel. It runs on a kernel called Darwin originally forked from BSD.)

To run SingularityCE on a Windows or macOS computer, a Linux virtual machine (VM) is required. There are various ways to configure a VM on both Windows and macOS. On WIndows, we recommend the Windows Subsystem for Linux (WSL2), and macOS, we recommend Lima.

3.2.1 Windows

Recent builds of Windows 10, and all builds of Windows 11, include version 2 of the Windows Subsystem for Linux. WSL2 provides a Linux virtual machine that is tightly integrated with the Windows environment. The default Linux distribution used by WSL2 is Ubuntu. It is straightforward to install SingularityCE inside WSL2 Ubuntu, and use all of its features.

Follow the WSL2 installation instructions to enable WSL2 with the default Ubuntu 22.04 environment. On Windows 11 and the most recent builds of Windows 10 this is as easy as opening an administrator command prompt or Powershell window and entering:

```
wsl --install
```

Follow the prompts. A restart is required, and when you open the 'Ubuntu' app for the first time you'll be asked to set a username and password for the Linux environment.

You can install SingularityCE from source, or from the Ubuntu packages at the GitHub releases page. To quickly install the 4.0.0 package use the following commands inside the WSL2 Ubuntu window:

```
$ wget https://github.com/sylabs/singularity/releases/download/v4.0.0/singularity-ce_4.0.
↪0-jammy_amd64.deb
$ sudo apt install ./singularity-ce_4.0.0-jammy_amd64.deb
```

The `singularity` command will now be available in your WSL2 environment:

```
$ singularity exec library://ubuntu echo "Hello World!"
INFO:    Downloading library image
28.4MiB / 28.4MiB␣
↪[================================================================================]␣
↪100 % 5.6 MiB/s 0s
Hello World!
```

GPU Support

WSL2 supports using an NVIDIA GPU from the Linux environment. To use a GPU from SingularityCE in WSL2, you must first install `libnvidia-container-tools`, following the instructions in the libnvidia-container documentation:

```
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg --dearmor -o /
↪usr/share/keyrings/nvidia-container-toolkit-keyring.gpg \
curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-
↪toolkit.list | \
  sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-
↪keyring.gpg] https://#g' | \
  sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list \
sudo apt-get update
sudo apt-get install -y nvidia-container-toolkit
```

Once this process has been completed, GPU containers can be run under WSL2 using the `--nv` and `--nvccli` flags together:

```
$ singularity pull docker://tensorflow/tensorflow:latest-gpu

$  singularity run --nv --nvccli tensorflow_latest-gpu.sif
INFO:    Setting 'NVIDIA_VISIBLE_DEVICES=all' to emulate legacy GPU binding.
```

(continues on next page)

(continued from previous page)

```
INFO:    Setting --writable-tmpfs (required by nvidia-container-cli)
________                                ________________
___  __/_____________________________  ____/__  /________      __
__  /  _  _ \_  __ \_  ___/  __ \_  ___/_  /_   __  /_  __ \_ | /| / /
_  /   /  __/  / / /(__  )/ /_/ /  /   _  __/   _  / / /_/ /_ |/ |/ /
/_/    \___//_/ /_//____/ \____//_/    /_/      /_/  \____/____/|__/
You are running this container as user with ID 1000 and group 1000,
which should map to the ID and group for your user on the Docker host. Great!
Singularity> python
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow as tf
>>> tf.config.list_physical_devices('GPU')
2022-03-25 11:42:25.672088: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:922]␣
↪could not open file to read NUMA node: /sys/bus/pci/devices/0000:01:00.0/numa_node
Your kernel may have been built without NUMA support.
2022-03-25 11:42:25.713295: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:922]␣
↪could not open file to read NUMA node: /sys/bus/pci/devices/0000:01:00.0/numa_node
Your kernel may have been built without NUMA support.
2022-03-25 11:42:25.713892: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:922]␣
↪could not open file to read NUMA node: /sys/bus/pci/devices/0000:01:00.0/numa_node
Your kernel may have been built without NUMA support.
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Note that the `--nvccli` flag is required to enable container setup using the `nvidia-container-cli` utility. SingularityCE's simpler library binding approach (`--nv` only) is not sufficient for GPU support under WSL2.

3.2.2 Mac

To install SingularityCE on macOS, we recommend using the lima VM platform, available on Homebrew.

If you don't already have Homebrew installed, you can install it as follows:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

Follow the instructions at the end of the installation process. In particular, make sure to add the relevant lines to your shell configuration:

```
$ (echo; echo 'eval "$(/home/linuxbrew/.linuxbrew/bin/brew shellenv)"') >> $HOME/.profile
$ eval "$(/home/linuxbrew/.linuxbrew/bin/brew shellenv)"
```

Once Homebrew is installed, install lima:

```
$ brew install lima
```

As part of the SingularityCE distribution (starting with version 4), we have provided an example template for using SingularityCE with lima. The example is available under the `examples/lima` directory in the SingularityCE source bundle, and can also be downloaded directly from the code repository.

The template is named `singularity-ce.yml`, and:

- Is based on AlmaLinux 9.

- Supports both Intel and Apple Silicon (ARM64) Macs.
- Installs the latest stable release of SingularityCE that has been published to the Fedora EPEL repositories.

Once you have obtained the template file, use it to start a lima VM:

```
$ limactl start ./singularity-ce.yml
```

You will be presented with an interactive menu:

```
$ limactl start ./singularity-ce.yml
? Creating an instance "singularity-ce"  [Use arrows to move, type to filter]
> Proceed with the current configuration
  Open an editor to review or modify the current configuration
  Choose another template (docker, podman, archlinux, fedora, ...)
  Exit
```

Choose the `Proceed with the current configuration` option, and lima will proceed to configure the VM according to the specifications in the template file. This can take a couple of minutes.

Once lima is done with the configuration step, you can enter the VM interactively and run SingularityCE commands:

```
$ limactl shell singularity-ce
[myuser@lima-singularity-ce myuser]$ singularity run library://alpine
INFO:    Downloading library image
2.8MiB / 2.8MiB␣
↪[=======================================================================================]␣
↪100 % 0.0 b/s 0s
Singularity> cat /etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.15.5
PRETTY_NAME="Alpine Linux v3.15"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
Singularity>
```

Your home directory is shared into the lima VM by default. However, since macOS places home directories under `/Users` (rather than `/home`), SingularityCE will not mount your home directory in the container unless you explicitly specify your macOS homedir, as shown here:

```
$ limactl shell singularity-ce
[myuser@lima-singularity-ce myuser]$ singularity run -H /Users/myuser library://alpine
INFO:    Using cached image
Singularity> ls
Applications Documents    Library      Music        Public
Desktop      Downloads    Movies       Pictures
```

You can also run SingularityCE using lima directly from the macOS command-line:

```
$ limactl shell singularity-ce singularity run library://alpine
INFO:    Using cached image
Singularity>
```

Or, with homedir mounting:

```
$ limactl shell singularity-ce singularity run -H /Users/myuser library://alpine
INFO:    Using cached image
Singularity>
```

To stop the lima VM:

```
$ limactl stop singularity-ce
```

To delete the lima VM:

```
$ limactl delete singularity-ce
```

3.2.3 SingularityCE Docker Image

It is also possible to run SingularityCE inside Docker, or another compatible OCI container runtime. This may be convenient if you have Docker Desktop, or a similar solution, already installed on your PC or Mac.

Docker containers for SingularityCE are maintained at https://quay.io/repository/singularity/singularity.

Note: These containers are maintained by a third party. They are not part of the SingularityCE project, nor are they reviewed by Sylabs.

An example of a suitable `compose.yaml` file to start up SingularityCE in a Docker container is given below. Note that privileged operation is needed to successfully run SingularityCE nested inside of Docker. Change the version number on the `image:` line to your preferred release.

```
services:
  singularity:
    image: quay.io/singularity/singularity:v3.11.4-slim
    stdin_open: true
    tty: true
    privileged: true
    volumes:
      - .:/root
    entrypoint: ["/bin/sh"]
```

Singularity in Docker can have various disadvantages, but basic container operations will work. Currently, the intended use case is continuous integration, meaning that you should be able to build a Singularity container using this Docker Compose file. For more information see issue#5 and the image's source repo

CHAPTER

FOUR

SINGULARITYCE CONFIGURATION FILES

As a SingularityCE Administrator, you will have access to various configuration files, that will let you manage container resources, set security restrictions and configure network options etc, when installing SingularityCE across the system. All of these files can be found in `/usr/local/etc/singularity` by default for installations from source (though the location may differ based on options passed during the installation). For installations from RPM or Deb packages you will find the configuration files in `/etc/singularity`. This section will describe the configuration files and the various parameters contained by them.

4.1 singularity.conf

Most of the configuration options are set using the file `singularity.conf` that defines the global configuration for SingularityCE across the entire system. Using this file, system administrators can influence the behavior of SingularityCE and restrict the functionality that users can access. As a security measure, for `setuid` installations of SingularityCE, `singularity.conf` must be owned by root and must not be writable by users or SingularityCE will refuse to run. This is not the case for `non-setuid` installations that will only ever execute with user privilege and thus do not require such limitations.

The options set via `singularity.conf` are listed below. Options are grouped together based on relevance. The actual order of options within `singularity.conf` may differ.

4.1.1 Setuid and Capabilities

`allow setuid`: To use all features of SingularityCE containers, SingularityCE will need to have access to some privileged system calls. SingularityCE achieves this by using a helper binary with the `setuid` bit enabled. The `allow-setuid` option lets you enable/disable users ability to utilize these binaries within SingularityCE. By default, it is set to "yes", but when disabled, various SingularityCE features will not function. Please see *Unprivileged Installations* (page 49) for more information about running SingularityCE without `setuid` enabled.

`root default capabilities`: In its default, non-OCI-mode, SingularityCE allows the specification of capabilities kept by default when the root user runs a container. Options include:

- full: all capabilities are maintained, this gives the same behavior as the `--keep-privs` option.
- file: only capabilities granted for root in `etc/singularity/capability.json` are maintained.
- no: no capabilities are maintained, this gives the same behavior as the `--no-privs` option.

The root user can modify the capabilities granted to individual containers when they are launched through the `--add-caps` and `drop-caps` flags.

In OCI-mode, SingularityCE follows the behavior of other OCI runtimes, and will always grant a default set of capabilities to the container. The `root default capabilities` option does not apply.

Please see Linux Capabilities in the user guide for more information.

4.1.2 Loop Devices

SingularityCE uses loop devices to facilitate the mounting of container file systems from SIF and other images.

`max loop devices`: This option allows an admin to limit the total number of loop devices SingularityCE will consume at a given time.

`shared loop devices`: This allows containers running the same image to share a single loop device. This minimizes loop device usage and helps optimize kernel cache usage. Enabling this feature can be particularly useful for MPI jobs.

4.1.3 Default runtime (native vs. OCI)

Starting with version 4.0, SingularityCE includes a fully-supported OCI-mode, allowing you to run OCI containers using `crun` or `runc` as the low-level runtime, for true OCI runtime compatibility. (See the OCI-mode section in the user guide for more information.)

By default, SingularityCE will use its native runtime unless the `run` / `shell` / `exec` / `pull` commands are invoked with the `--oci` flag. However, administrators who wish to configure their installation of SingularityCE to use OCI-mode by default can do so by adding `oci mode = yes` to their configuration file:

`oci mode`: Enable OCI-mode by default. (default: `no`)

Note: By default, OCI-mode will attempt to use OCI-SIF images to locally store and mount containers. (See the discussion of OCI-SIF in the user guide for more information.) If system does not meet the requirements for using OCI-SIF, OCI mode will fall back to a filesystem-based strategy: the OCI container will be unpacked into a temporary sandbox dir and run from there.

Administrators who wish to disable this fallback behavior can do so via the `tmp sandbox = no` option discussed *below* (page 36).

4.1.4 Namespace Options

`allow pid ns`: This option determines if users can leverage the PID namespace when running their containers through the `--pid` flag.

Note: Using the PID namespace can confuse the process tracking of some resource managers, as well as some MPI implementations.

`allow user ns`: This option controls the creation of user namespaces. If set to `no`, this will prevent execution of containers with the `--userns` or `--fakeroot` flags, and unprivileged installations of SingularityCE.

`allow uts ns`: Likewise, this option determines whether users can make use of the UTS namespace when running their containers. If sets to `no`, it disables the `--uts` and `--hostname` flags.

4.1.5 Configuration Files

SingularityCE can automatically create or modify several system files within containers to ease usage.

Note: These options will have no effect if the file does not exist within the container, or overlay or underlay support are enabled.

`config passwd`: This option determines if SingularityCE should automatically append an entry to `/etc/passwd` for the user running the container.

`config group`: This option determines if SingularityCE should automatically append the calling user's group entries to the containers `/etc/group`.

`config resolv_conf`: This option determines if SingularityCE should automatically bind the host's `/etc/resolv.conf` within the container.

4.1.6 Session Directory and System Mounts

`sessiondir max size`: In order for the SingularityCE runtime to run a container it needs to create a temporary in-memory `sessiondir` as a location to assemble various components of the container, including mounting filesystems over the base image. In addition, this sessiondir will hold files that are written to the container when `--writable-tmpfs` is used, plus isolated temporary filesystems in `--contain` mode. The default value from SingularityCE 3.11 is 64MiB. If users commonly run containers with `--writable-tmpfs`, `--contain`, or in `--oci` mode, this value should be increased to accommodate their workflows. The tmpfs will grow to the specified maximum size, as required. Memory is not allocated ahead of usage.

`mount proc`: This option determines if SingularityCE should automatically bind mount `/proc` within the container.

`mount sys`: This option determines if SingularityCE should automatically bind mount `/sys` within the container.

`mount dev`: Should be set to "YES", if you want SingularityCE to automatically bind mount a complete `/dev` tree within the container. If set to `minimal`, then only `/dev/null`, `/dev/zero`, `/dev/random`, `/dev/urandom`, and `/dev/shm` will be included.

`mount devpts`: This option determines if SingularityCE will mount a new instance of `devpts` when there is a `minimal /dev` directory as explained above, or when the `--contain` option is passed.

Note: This requires either a kernel configured with `CONFIG_DEVPTS_MULTIPLE_INSTANCES=y`, or a kernel version at or newer than `4.7`.

`mount home`: When this option is enabled, SingularityCE will automatically determine the calling user's home directory and attempt to mount it into the container.

`mount tmp`: When this option is enabled, SingularityCE will automatically bind mount `/tmp` and `/var/tmp` into the container from the host. If the `--contain` option is passed, SingularityCE will create both locations within the `sessiondir` or within the directory specified by the `--workdir` option if that is passed as well.

`mount hostfs`: This option will cause SingularityCE to probe the host for all mounted filesystems and bind those into containers at runtime.

`mount slave`: SingularityCE automatically mounts a handful host system directories to the container by default. This option determines if filesystem changes on the host should automatically be propagated to those directories in the container.

Note: This should be set to `yes` when autofs mounts occurring on the host system should be reflected up in the container.

`memory fs type`: This option allows admins to choose the temporary filesystem used by SingularityCE. Temporary filesystems are primarily used for system directories like `/dev` when the host system directory is not mounted within the container.

Note: For Cray CLE 5 and 6, up to CLE 6.0.UP05, there is an issue (kernel panic) when Singularity uses tmpfs, so on affected systems it's recommended to set this value to `ramfs` to avoid a kernel panic.

4.1.7 Bind Mount Management

`bind path`: This option is used to define a list of files or directories to automatically be made available when SingularityCE runs a container. In order to successfully mount listed paths the file or directory must exist within the container, or SingularityCE must be configured with either overlay or underlay support enabled.

Note: This option is ignored when containers are invoked with the `--contain` option.

You can define the a bind point where the source and destination are identical:

```
bind path = /etc/localtime
```

Or you can specify different source and destination locations using:

```
bind path = /etc/singularity/default-nsswitch.conf:/etc/nsswitch.conf
```

`user bind control`: This allows admins to decide if users can define bind points at runtime. By Default, this option is set to `YES`, which means users can specify bind points, scratch and tmp locations.

4.1.8 Limiting Container Execution

Warning: If unprivileged user namespace creation is allowed on a system, a user can supply and use their own unprivileged installation of Singularity or another container runtime to bypass container limits. They may also be able to use standard system tools such as `unshare`, `nsenter`, and FUSE mounts to access / execute arbitrary containers without installing any runtime.

The 'limit container' and 'allow container' directives are not effective if unprivileged user namespaces are enabled. They are only effectively applied when Singularity is running using the native runtime in setuid mode, and unprivileged container execution is not possible on the host.

You must disable unprivileged user namespace creation on the host if you rely on the these directives to limit container execution. This will disable OCI mode, which is unprivileged and cannot enforce these limits.

There are several ways to limit container execution as an admin listed below. If stricter controls are required, check out the *Execution Control List* (page 40).

`limit container owners`: This restricts container execution to only allow containers that are owned by the specified user.

Note: This feature will only apply when SingularityCE is running in SUID mode and the user is non-root. By default this is set to `NULL`.

`limit container groups`: This restricts container execution to only allow containers that are owned by the specified group.

Note: This feature will only apply when SingularityCE is running in SUID mode and the user is non-root. By default this is set to `NULL`.

`limit container paths`: This restricts container execution to only allow containers that are located within the specified path prefix.

Note: This feature will only apply when SingularityCE is running in SUID mode and the user is non-root. By default this is set to `NULL`.

`allow container ${type}`: This option allows admins to limit the types of image formats that can be leveraged by users with SingularityCE.

- `allow container sif` permits / denies execution of unencrypted SIF containers.
- `allow container encrypted` permits / denies execution of SIF containers with an encrypted root filesystem.
- `allow container squashfs` permits / denies execution of bare SquashFS image files. E.g. Singularity 2.x images.
- `allow container extfs` permits / denies execution of bare extfs image files.
- `allow container dir` permits / denies execution of sandbox directory containers. Also applies to SIF / squashfs / extfs images when mounted to a directory by FUSE binaries run as the user, or automatically extracted to a directory.

Note: These limitations do not apply to the root user.

Note: When a SIF / squashfs / extfs container image is mounted using FUSE, or extracted to disk, the `allow container dir` setting applies. In contrast to kernel mounts, FUSE mounted container images are mounted at a directory under the full control of the user, who may also manipulate the behavior of the FUSE binary.

4.1.9 Disabling Kernel Filesystem Mounts

When running in setuid mode, SingularityCE will mount extfs and squashfs filesystems using the kernel's filesystem drivers. These mounts are performed for standalone or SIF container images, overlay images or partitions, that use extfs or squashfs formats.

Options in `singularity.conf` allow these mounts to be disabled, to e.g. work around a kernel vulnerability that cannot be patched in a timely manner. Singularity will then attempt to use `squashfuse` or `fuse2fs` to mount container images in user space. If it is not possible to perform a FUSE mount, a container image will be extracted to a sandbox directory for execution.

Note that disabling kernel mounts will result in a significant loss of functionality in setuid mode. Container execution restrictions cannot be effectively applied, and not all overlay configurations are supported.

`allow kernel squashfs`: Defaults to yes. When set to no, SingularityCE will not mount squashfs filesystems using the kernel squashfs driver. If possible, `squashfuse_ll` will be used to mount a squashfs container image in user space. If `squashfuse_ll` is not available, or fails, the image will be extracted to a directory for execution.

`allow kernel extfs`: Defaults to yes. When set to no, SingularityCE will not mount extfs filesystems using the kernel extfs driver. If possible, `fuse2fs` will be used to mount an extfs container image in user space. If `fuse2fs` is not available, or fails, the image will be extracted to a directory for execution.

4.1.10 Networking Options

The `--network` option can be used to specify a CNI networking configuration that will be used when running a container with network virtualization.

The `--netns-path` option (from SingularityCE 4.2) can be used to join an existing network namespace when running a container.

Unrestricted use of CNI network configurations, and joining arbitrary existing network namespaces, requires root privilege as certain configurations may disrupt the host networking environment or allow access to restricted network resources.

SingularityCE allows specific users or groups to be granted the ability to run containers with administrator specified CNI configurations, and to join network namespaces exposed at specified paths.

`allow net users`: A list of non-root users that are permitted to use the CNI configurations specified in the `allow net networks` directive, and can join existing network namespaces listed in the `allow netns paths` directive. By default only root may use CNI configurations, or join existing network namespaces, except in the case of a fakeroot execution where only the `40_fakeroot.conflist` CNI configuration is used.

`allow net groups`: A list of non-root groups that are permitted to use the CNI configurations specified in the `allow net networks` directive, and can join existing network namespaces listed in the `allow netns paths` directive. By default only root may use CNI configurations, or join existing network namespaces, except in the case of a fakeroot execution where only the `40_fakeroot.conflist` CNI configuration is used.

`allow net networks`: Specify the names of CNI network configurations that may be used by users and groups listed in the `allow net users` / `allow net groups` directives. Thus feature only applies when SingularityCE is running in SUID mode and the user is non-root.

`allow netns paths`: Specify the paths to network namespaces that may be joined by users and groups listed in the `allow net users` / `allow net groups` directives. This restriction only applies when Singularity is running in SUID mode and the user is non-root.

4.1.11 GPU Options

SingularityCE provides integration with GPUs in order to facilitate GPU based workloads seamlessly. Both options listed below are particularly useful in GPU only environments. For more information on using GPUs with SingularityCE checkout *GPU Library Configuration* (page 42).

`always use nv`: Enabling this option will cause every action command (`exec/shell/run/instance`) to be executed with the `--nv` option implicitly added.

`always use rocm`: Enabling this option will cause every action command (`exec/shell/run/instance`) to be executed with the `--rocm` option implicitly added.

4.1.12 Supplemental Filesystems

`enable fusemount`: This will allow users to mount fuse filesystems inside containers using the `--fusemount` flag.

`enable overlay`: This option will allow SingularityCE to create bind mounts at paths that do not exist within the container image. This option can be set to `try`, which will try to use an overlayfs. If it fails to create an overlayfs in this case the bind path will be silently ignored.

`enable underlay`: This option will allow SingularityCE to create bind mounts at paths that do not exist within the container image, just like `enable overlay`, but instead using an underlay. This is suitable for systems where overlay is not possible or not working. If the overlay option is available and working, it will be used instead.

4.1.13 CNI Configuration and Plugins

`cni configuration path`: This option allows admins to specify a custom path for the CNI configuration that SingularityCE will use for Network Virtualization.

`cni plugin path`: This option allows admins to specify a custom path for SingularityCE to access CNI plugin executables. Check out the Network Virtualization section of the user guide for more information.

4.1.14 External Binaries

SingularityCE calls a number of external binaries for full functionality. The paths for certain critical binaries can be set in `singularity.conf`. At build time, `mconfig` will set initial values for these, by searching on the `$PATH` environment variable. You can override which external binaries are called by changing the value in `singularity.conf`.

`cryptsetup path`: Path to the cryptsetup executable, used to work with encrypted containers. Must be owned by root for security reasons.

`ldconfig path`: Path to the ldconfig executable, used to find GPU libraries. Must be owned by root for security reasons.

`nvidia-container-cli path`: Path to the nvidia-container-cli executable, used to find GPU libraries and configure the container when running with the `--nvccli` option. Must be owned by root for security reasons.

For the following additional binaries, if the `singularity.conf` entry is left blank, then `$PATH` will be searched at runtime.

`go path`: Path to the go executable, used to compile plugins.

`mksquashfs path`: Path to the mksquashfs executable, used to create SIF and SquashFS containers.

`mksquashfs procs`: Allows the administrator to specify the number of CPUs that mksquashfs may use when building an image. The fewer processors the longer it takes. To use all available CPU's set this to 0.

`mksquashfs mem`: Allows the administrator to set the maximum amount of memory that mksquashfs nay use when building an image. e.g. 1G for 1gb or 500M for 500mb. Restricting memory can have a major impact on the time it takes mksquashfs to create the image. NOTE: This functionality did not exist in squashfs-tools prior to version 4.3. If using an earlier version you should not set this.

`unsquashfs path`: Path to the unsquashfs executable, used to extract SIF and SquashFS containers.

4.1.15 Concurrent Downloads

SingularityCE 3.9 and above will pull `library://` container images using multiple concurrent downloads of parts of the image. This speeds up downloads vs using a single stream. The defaults are generally appropriate for the Sylabs Cloud, but may be tuned for your network conditions, or if you are pulling from a different library server.

`download concurrency`: specifies how many concurrent streams when downloading (pulling) an image from cloud library.

`download part size`: specifies the size of each part (bytes) when concurrent downloads are enabled.

`download buffer size`: specifies the transfer buffer size (bytes) when concurrent downloads are enabled.

4.1.16 Cgroups Options

`systemd cgroups`: specifies whether to use systemd to manage container cgroups. Required (with cgroups v2) for unprivileged users to apply resource limits on containers. If set to `no`, SingularityCE will directly manage cgroups via the cgroupfs.

4.1.17 Disabling temporary sandbox dirs

Some execution flows will extract the contents of an image into a temporary local sandbox dir prior to execution. Examples include: using a user namespace in native mode when FUSE is not available, as well as using OCI-mode in an environment that does not support OCI-SIF (see the discussion of OCI-mode *above* (page 30)).

Administrators who wish to disable this behavior, and prevent SingularityCE from extracting the contents of images to temporary sandbox dirs, may do so by adding `tmp sandbox = no` to their configuration file:

`tmp sandbox`: Allow extraction of image contents to temporary sandbox dir. (default: `yes`)

4.1.18 Deprecated Experimental Options

`sif fuse`: If set to `yes`, always attempt to mount a SIF image using `squashfuse` when running in unprivileged / user namespace flows. This is equivalent to always specifying the experimental *–sif-fuse* flag. **Deprecated in SingularityCE 4.1**, as a FUSE mount will be attempted by default in these circumstances. The option has no effect in 4.1, and is retained only for configuration file compatibility with prior versions.

4.1.19 Updating Configuration Options

In order to manage this configuration file, SingularityCE has a `config global` command group that allows you to get, set, reset, and unset values through the CLI. It's important to note that these commands must be run with elevated privileges because the `singularity.conf` can only be modified by an administrator.

Example

In this example we will changing the `bind path` option described above. First we can see the current list of bind paths set within our system configuration:

```
$ sudo singularity config global --get "bind path"
/etc/localtime,/etc/hosts
```

Now we can add a new path and verify it was successfully added:

```
$ sudo singularity config global --set "bind path" /etc/resolv.conf
$ sudo singularity config global --get "bind path"
/etc/resolv.conf,/etc/localtime,/etc/hosts
```

From here we can remove a path with:

```
$ sudo singularity config global --unset "bind path" /etc/localtime
$ sudo singularity config global --get "bind path"
/etc/resolv.conf,/etc/hosts
```

If we want to reset the option to the default at installation, then we can reset it with:

```
$ sudo singularity config global --reset "bind path"
$ sudo singularity config global --get "bind path"
/etc/localtime,/etc/hosts
```

And now we are back to our original option settings. You can also test what a change would look like by using the `--dry-run` option in conjunction with the above commands. Instead of writing to the configuration file, it will output what would have been written to the configuration file if the command had been run without the `--dry-run` option:

```
$ sudo singularity config global --dry-run --set "bind path" /etc/resolv.conf
# SINGULARITY.CONF
# This is the global configuration file for Singularity. This file controls
[...]
# BIND PATH: [STRING]
# DEFAULT: Undefined
# Define a list of files/directories that should be made available from within
# the container. The file or directory must exist within the container on
# which to attach to. you can specify a different source and destination
# path (respectively) with a colon; otherwise source and dest are the same.
# NOTE: these are ignored if singularity is invoked with --contain.
bind path = /etc/resolv.conf
bind path = /etc/localtime
bind path = /etc/hosts
[...]
$ sudo singularity config global --get "bind path"
/etc/localtime,/etc/hosts
```

Above we can see that `/etc/resolv.conf` is listed as a bind path in the output of the `--dry-run` command, but did not affect the actual bind paths of the system.

4.2 cgroups.toml

The cgroups (control groups) functionality of the Linux kernel allows you to limit and meter the resources used by a process, or group of processes. Using cgroups you can limit memory and CPU usage. You can also rate limit block IO, network IO, and control access to device nodes.

There are two versions of cgroups in common use. Cgroups v1 sets resource limits for a process within separate hierarchies per resource class. Cgroups v2, the default in newer Linux distributions, implements a unified hierarchy, simplifying the structure of resource limits on processes.

- v1 documentation: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
- v2 documentation: https://www.kernel.org/doc/Documentation/cgroup-v2.txt

SingularityCE 3.9 and above can apply resource limitations to systems configured for both cgroups v1 and the v2 unified hierarchy. Resource limits are specified using a TOML file that represents the `resources` section of the OCI runtime-spec: https://github.com/opencontainers/runtime-spec/blob/master/config-linux.md#control-groups

On a cgroups v1 system the resources configuration is applied directly. On a cgroups v2 system the configuration is translated and applied to the unified hierarchy.

Under cgroups v1, access restrictions for device nodes are managed directly. Under cgroups v2, the restrictions are applied by attaching eBPF programs that implement the requested access controls.

4.2.1 Examples

To apply resource limits to a container, use the `--apply-cgroups` flag, which takes a path to a TOML file specifying the cgroups configuration to be applied:

```
$ singularity shell --apply-cgroups /path/to/cgroups.toml my_container.sif
```

Note: The `--apply-cgroups` option requires cgroups v2 to be used without root privileges.

Limiting memory

To limit the amount of memory that your container uses to 500MB (524288000 bytes), set a `limit` value inside the `[memory]` section of your cgroups TOML file:

```
[memory]
    limit = 524288000
```

Start your container, applying the toml file, e.g.:

```
$ singularity run --apply-cgroups path/to/cgroups.toml library://alpine
```

Limiting CPU

CPU usage can be limited using different strategies, with limits specified in the `[cpu]` section of the TOML file.

shares

This corresponds to a ratio versus other cgroups with cpu shares. Usually the default value is `1024`. That means if you want to allow to use 50% of a single CPU, you will set `512` as value.

```
[cpu]
    shares = 512
```

A cgroup can get more than its share of CPU if there are enough idle CPU cycles available in the system, due to the work conserving nature of the scheduler, so a contained process can consume all CPU cycles even with a ratio of 50%. The ratio is only applied when two or more processes conflicts with their needs of CPU cycles.

quota/period

You can enforce hard limits on the CPU cycles a cgroup can consume, so contained processes can't use more than the amount of CPU time set for the cgroup. `quota` allows you to configure the amount of CPU time that a cgroup can use per period. The default is 100ms (100000us). So if you want to limit amount of CPU time to 20ms during period of 100ms:

```
[cpu]
    period = 100000
    quota = 20000
```

cpus/mems

You can also restrict access to specific CPUs (cores) and associated memory nodes by using `cpus/mems` fields:

```
[cpu]
    cpus = "0-1"
    mems = "0-1"
```

Where container has limited access to CPU 0 and CPU 1.

Note: It's important to set identical values for both `cpus` and `mems`.

Limiting IO

To control block device I/O, applying limits to competing container, use the `[blockIO]` section of the TOML file:

```
[blockIO]
    weight = 1000
    leafWeight = 1000
```

`weight` and `leafWeight` accept values between `10` and `1000`.

`weight` is the default weight of the group on all the devices until and unless overridden by a per device rule.

`leafWeight` relates to weight for the purpose of deciding how heavily to weigh tasks in the given cgroup while competing with the cgroup's child cgroups.

To apply limits to specific block devices, you must set configuration for specific device major/minor numbers. For example, to override `weight/leafWeight` for `/dev/loop0` and `/dev/loop1` block devices, set limits for device major 7, minor 0 and 1:

```
[blockIO]
    [[blockIO.weightDevice]]
        major = 7
        minor = 0
        weight = 100
        leafWeight = 50
    [[blockIO.weightDevice]]
        major = 7
        minor = 1
        weight = 100
        leafWeight = 50
```

You can also limit the IO read/write rate to a specific absolute value, e.g. 16MB per second for the `/dev/loop0` block device. The `rate` is specified in bytes per second.

```
[blockIO]
    [[blockIO.throttleReadBpsDevice]]
        major = 7
        minor = 0
        rate = 16777216
    [[blockIO.throttleWriteBpsDevice]]
        major = 7
        minor = 0
        rate = 16777216
```

Other limits

SingularityCE can apply all resource limits that are valid in the OCI runtime-spec `resources` section, including `unified` cgroups v2 constraints. It is most compatible, however, to use the cgroups v1 limits, which will be translated to v2 format when applied on a cgroups v2 system.

See https://github.com/opencontainers/runtime-spec/blob/master/config-linux.md#control-groups for information about the available limits. Note that SingularityCE uses TOML format for the configuration file, rather than JSON.

4.3 ecl.toml

The execution control list that can be used to restrict the execution of SIF files by signing key is defined here. You can authorize the containers by validating both the location of the SIF file in the filesystem and by checking against a list of signing entities.

Warning: If unprivileged user namespace creation is allowed on a system, a user can supply and use their own unprivileged installation of Singularity or another container runtime to bypass container limits. They may also be able to use standard system tools such as `unshare`, `nsenter`, and FUSE mounts to access / execute arbitrary containers without installing any runtime.

The ECL is not effective if unprivileged user namespaces are enabled. It is only effectively applied when Singularity is running using the native runtime in setuid mode, and unprivileged container execution is not possible on the host.

You must disable unprivileged user namespace creation on the host if you rely on the ECL limit container execution. This will disable OCI mode, which is unprivileged and cannot enforce these limits.

Warning: The ECL configuration applies to SIF container images only. To lock down execution fully you should disable execution of other container types (squashfs/extfs/dir) via the `singularity.conf` file `allow container` settings.

```
[[execgroup]]
  tagname = "group2"
  mode = "whitelist"
  dirpath = "/tmp/containers"
  keyfp = ["7064B1D6EFF01B1262FED3F03581D99FE87EAFD1"]
```

Only the containers running from and signed with above-mentioned path and keys will be authorized to run.

Three possible list modes you can choose from:

Whitestrict: The SIF must be signed by all of the keys mentioned.

Whitelist: As long as the SIF is signed by one or more of the keys, the container is allowed to run.

Blacklist: Only the containers whose keys are not mentioned in the group are allowed to run.

Note: The ECL checks will use the new signature format introduced in SingularityCE 3.6.0. Containers signed with older versions of Singularity SingularityCE will not pass ECL checks.

To temporarily permit the use of legacy insecure signatures, set `legacyinsecure = true` in `ecl.toml`.

4.3.1 Managing ECL public keys

Since SingularityCE 3.7.0 a global keyring is used for ECL signature verification. This keyring can be administered using the `--global` flag for the following commands:

- `singularity key import` (root user only)
- `singularity key pull` (root user only)
- `singularity key remove` (root user only)
- `singularity key export`
- `singularity key list`

Note: For security reasons, it is not possible to import private keys into this global keyring because it must be accessible by users and is stored in the file `SYSCONFDIR/singularity/global-pgp-public`.

4.4 GPU Library Configuration

When a container includes a GPU enabled application, SingularityCE (with the `--nv` or `--rocm` options) can properly inject the required Nvidia or AMD GPU driver libraries into the container, to match the host's kernel. The GPU `/dev` entries are provided in containers run with `--nv` or `--rocm` even if the `--contain` option is used to restrict the in-container device tree.

Compatibility between containerized CUDA/ROCm/OpenCL applications and host drivers/libraries is dependent on the versions of the GPU compute frameworks that were used to build the applications. Compatibility and usage information is discussed in the 'GPU Support' section of the user guide

4.4.1 NVIDIA GPUs / CUDA

The `nvliblist.conf` configuration file is used to specify libraries and executables that need to be injected into the container when running SingularityCE with the `--nv` Nvidia GPU support option. The provided `nvliblist.conf` is suitable for CUDA 11, but may need to be modified if you need to include additional libraries, or further libraries are added to newer versions of the Nvidia driver/CUDA distribution.

When adding new entries to `nvliblist.conf` use the bare filename of executables, and the `xxxx.so` form of libraries. Libraries are resolved via `ldconfig -p`, and exectuables are found by searching `$PATH`.

Experimental nvidia-container-cli Support

The nvidia-container-cli tool is Nvidia's officially support method for configuring containers to use a GPU. It is targeted at OCI container runtimes.

SingularityCE 3.9 introduces an experimental `--nvccli` option, which will call out to `nvidia-container-cli` for container GPU setup, rather than use the `nvliblist.conf` approach.

To use `--nvccli` a root-owned `nvidia-container-cli` binary must be present on the host. The binary that is run is controlled by the `nvidia-container-cli` directive in `singularity.conf`. During installation of SingularityCE, the `./mconfig` step will set the correct value in `singularity.conf` if `nvidia-container-cli` is found on the `$PATH`. If the value of `nvidia-container-cli path` is empty, SingularityCE will look for the binary on `$PATH` at runtime.

Note: To prevent use of `nvidia-container-cli` via the `--nvccli` flag, you may set `nvidia-container-cli path` to `/bin/false` in `singularity.conf`.

`nvidia-container-cli` is run as the `root` user during setuid operation of SingularityCE. The container starter process grants a number of Linux capabilities to `nvidia-container-cli`, which are required for it to configure the container for GPU operation. The operations performed by `nvidia-container-cli` are broadly similar to those which SingularityCE carries out when setting up a GPU container from `nvliblist.conf`.

4.4.2 AMD Radeon GPUs / ROCm

The `rocmliblist.conf` file is used to specify libraries and executables that need to be injected into the container when running SingularityCE with the `--rocm` Radeon GPU support option. The provided `rocmliblist.conf` is suitable for ROCm 4.0, but may need to modified if you need to include additional libraries, or further libraries are added to newer versions of the ROCm distribution.

When adding new entries to `rocmlist.conf` use the bare filename of executables, and the `xxxx.so` form of libraries. Libraries are resolved via `ldconfig -p`, and exectuables are found by searching `$PATH`.

4.4.3 GPU liblist format

The `nvliblist.conf` and `rocmliblist` files list the basename of executables and libraries to be bound into the container, without path information.

Binaries are found by searching `$PATH`:

```
# put binaries here
# In shared environments you should ensure that permissions on these files
# exclude writing by non-privileged users.
rocm-smi
rocminfo
```

Libraries should be specified without version information, i.e. `libname.so`, and are resolved using `ldconfig`.

```
# put libs here (must end in .so)
libamd_comgr.so
libcomgr.so
libCXLActivityLogger.so
```

If you receive warnings that binaries or libraries are not found, ensure that they are in a system path (binaries), or available in paths configured in `/etc/ld.so.conf` (libraries).

4.5 capability.json

4.5.1 Native runtime / non-OCI-Mode

In SingularityCE's default configuration, without `--oci`, a container started by root receives all capabilities, while a container started by a non-root user receives no capabilities.

Additionally, SingularityCE provides support for granting and revoking Linux capabilities on a user or group basis. The `capability.json` file is maintained by SingularityCE in order to manage these capabilities. The `capability` command group allows you to `add`, `drop`, and `list` capabilities for users and groups.

> **Warning:** In SingularityCE's default setuid and non-OCI mode, containers are only isolated in a mount namespace. A user namespace, which limits the scope of capabilities, is not used by default.
>
> Therefore, it is extremely important to recognize that **granting users Linux capabilities with the `capability` command group is usually identical to granting those users root level access on the host system**. Most if not all capabilities will allow users to "break out" of the container and become root on the host. This feature is targeted toward special use cases (like cloud-native architectures) where an admin/developer might want to limit the attack surface within a container that normally runs as root. This is not a good option in multi-tenant HPC environments

where an admin wants to grant a user special privileges within a container. For that and similar use cases, the *fakeroot feature* (page 49) is a better option.

For example, let us suppose that we have decided to grant a user (named `pinger`) capabilities to open raw sockets so that they can use `ping` in a container where the binary is controlled via capabilities.

To do so, we would issue a command such as this:

```
$ sudo singularity capability add --user pinger CAP_NET_RAW
```

This means the user `pinger` has just been granted permissions (through Linux capabilities) to open raw sockets within SingularityCE containers.

We can check that this change is in effect with the `capability list` command.

```
$ sudo singularity capability list --user pinger
CAP_NET_RAW
```

To take advantage of this new capability, the user `pinger` must also request the capability when executing a container with the `--add-caps` flag. `pinger` would need to run a command like this:

```
$ singularity exec --add-caps CAP_NET_RAW \
  library://sylabs/tests/ubuntu_ping:v1.0 ping -c 1 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=52 time=73.1 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 73.178/73.178/73.178/0.000 ms
```

If we decide that it is no longer necessary to allow the user `pinger` to open raw sockets within SingularityCE containers, we can revoke the appropriate Linux capability like so:

```
$ sudo singularity capability drop --user pinger CAP_NET_RAW
```

Now if `pinger` tries to use `CAP_NET_RAW`, SingularityCE will not give the capability to the container and `ping` will fail to create a socket:

```
$ singularity exec --add-caps CAP_NET_RAW \
  library://sylabs/tests/ubuntu_ping:v1.0 ping -c 1 8.8.8.8
WARNING: not authorized to add capability: CAP_NET_RAW
ping: socket: Operation not permitted
```

The `capability add` and `drop` subcommands will also accept the case insensitive keyword `all` to grant or revoke all Linux capabilities to a user or group.

For more information about individual Linux capabilities check out the man pages or use the `capability avail` command to output available capabilities with a description of their behaviors.

4.5.2 OCI-Mode

When containers are run in OCI-mode, by a non-root user, initialization is always performed inside a user namespace. The capabilities granted to a container are specific to this user namespace. For example, `CAP_SYS_ADMIN` granted to an OCI-mode container does not give the user the ability to mount a filesystem outside of the container's user namespace.

Because of this isolation of capabilities users can add and drop capabilities, using `--add-caps` and `--drop-caps`, without the need for the administrator to have granted permission to do so with the `singularity capabilities` command. The `capability.json` file is not consulted.

OCI-mode containers do not inherit the user's own capabilities, but instead run with a default set of capabilities that matches other OCI runtimes.

- CAP_NET_RAW
- CAP_NET_BIND_SERVICE
- CAP_AUDIT_READ
- CAP_AUDIT_WRITE
- CAP_DAC_OVERRIDE
- CAP_SETFCAP
- CAP_SETPCAP
- CAP_SETGID
- CAP_SETUID
- CAP_MKNOD
- CAP_CHOWN
- CAP_FOWNER
- CAP_FSETID
- CAP_KILL
- CAP_SYS_CHROOT

When the container is entered as the root user (e.g. with `--fakeroot`), these default capabilities are added to the effective, permitted, and bounding sets.

When the container is entered as a non-root user, these default capabilities are added to the bounding set.

4.6 seccomp-profiles

Secure Computing (seccomp) Mode is a feature of the Linux kernel that allows an administrator to filter system calls being made from a container. Profiles made up of allowed and restricted calls can be passed to different containers. *Seccomp* provides more control than *capabilities* alone, giving a smaller attack surface for an attacker to work from within a container.

You can set the default action with `defaultAction` for a non-listed system call. Example: `SCMP_ACT_ALLOW` filter will allow all the system calls if it matches the filter rule and you can set it to `SCMP_ACT_ERRNO` which will have the thread receive a return value of *errno* if it calls a system call that matches the filter rule. The file is formatted in a way that it can take a list of additional system calls for different architecture and SingularityCE will automatically take syscalls related to the current architecture where it's been executed. The `include/exclude-> caps` section will include/exclude the listed system calls if the user has the associated capability.

Use the `--security` option to invoke the container like:

```
$ sudo singularity shell --security seccomp:/home/david/my.json my_container.sif
```

For more insight into security options, network options, cgroups, capabilities, etc, please check the Userdocs and it's Appendix.

4.7 remote.yaml

System-wide remote endpoints are defined in a configuration file typically located at `/usr/local/etc/singularity/remote.yaml` (this location may vary depending on installation parameters) and can be managed by administrators with the `remote` command group.

4.7.1 Remote Endpoints

Sylabs introduced the online Sylabs Cloud to enable users to Create, Secure, and Share their container images with others.

SingularityCE allows users to login to an account on the Sylabs Cloud, or configure SingularityCE to use an API compatible container service such as a local installation of SingularityCE Enterprise, which provides an on-premise private Container Library, Remote Builder and Key Store.

Note: A fresh installation of SingularityCE is automatically configured to connect to the public Sylabs Cloud services.

Examples

Use the `remote` command group with the `--global` flag to create a system-wide remote endpoint:

```
$ sudo singularity remote add --global company-remote https://enterprise.example.com
INFO:    Remote "company-remote" added.
INFO:    Global option detected. Will not automatically log into remote.
```

Conversely, to remove a system-wide endpoint:

```
$ sudo singularity remote remove --global company-remote
INFO:    Remote "company-remote" removed.
```

Note: Once users log in to a system wide endpoint, a copy of the endpoint will be listed in a their `~/.singularity/remote.yaml` file. This means modifications or removal of the system-wide endpoint will not be reflected in the users configuration unless they remove the endpoint themselves.

Exclusive Endpoint

SingularityCE 3.7 introduces the ability for an administrator to make a remote the only usable remote for the system by using the `--exclusive` flag:

```
$ sudo singularity remote use --exclusive company-remote
INFO:    Remote "company-remote" now in use.
$ singularity remote list
Cloud Services Endpoints
========================

NAME            URI                     ACTIVE  GLOBAL  EXCLUSIVE  INSECURE
SylabsCloud     cloud.sylabs.io         NO      YES     NO         NO
company-remote  enterprise.example.com  YES     YES     YES        NO
myremote        enterprise.example.com  NO      NO      NO         NO

Keyservers
==========

URI                       GLOBAL  INSECURE  ORDER
https://keys.example.com  YES     NO        1*

* Active cloud services keyserver
```

Insecure (HTTP) Endpoints

From SingularityCE 3.9, if you are using a endpoint that exposes its service discovery file over an insecure HTTP connection only, it can be added by specifying the `--insecure` flag:

```
$ sudo singularity remote add --global --insecure test http://test.example.com
INFO:    Remote "test" added.
INFO:    Global option detected. Will not automatically log into remote.
```

This flag controls HTTP vs HTTPS for service discovery only. The protocol used to access individual library, build and keyserver URLs is set by the service discovery file.

Additional Information

For more details on the `remote` command group and managing remote endpoints, please check the Remote Userdocs.

4.7.2 Keyserver Configuration

By default, SingularityCE will use the keyserver correlated to the active cloud service endpoint. This behavior can be changed or supplemented via the `keyserver add` and `keyserver remove` commands. These commands allow an administrator to create a global list of key servers used to verify container signatures by default.

For more details on the `keyserver` command group and managing keyservers, please see the Keyserver Management section of the user guide.

CHAPTER

FIVE

USER NAMESPACES & FAKEROOT

User namespaces are an isolation feature that allow processes to run with different user identifiers and/or privileges inside that namespace than are permitted outside. A user may have a `uid` of `1001` on a system outside of a user namespace, but run programs with a different `uid` with different privileges inside the namespace.

User namespaces are used with containers to make it possible to set up a container without privileged operations, and so that a normal user can act as root inside a container to perform administrative tasks, without being root on the host outside.

SingularityCE uses user namespaces in the following situations:

- When the `setuid` workflow is disabled or SingularityCE was installed without root.
- When a container is run with the `--userns` option.
- When `--fakeroot` is used to impersonate a root user when building or running a container.
- When the *–oci* runtime mode is used.

5.1 User Namespace Requirements

To allow unprivileged creation of user namespaces a kernel >=3.8 is required, with >=3.18 being recommended due to security fixes for user namespaces.

To use a persistent overlay directory with `--overlay` when running unprivileged, a kernel >=5.11 is required.

Additionally, some Linux distributions require that unprivileged user namespace creation is enabled using a `sysctl` or kernel command line parameter. Please consult your distribution documentation or vendor to confirm the steps necessary to 'enable unprivileged user namespace creation'.

5.1.1 Debian

```
sudo sh -c 'echo kernel.unprivileged_userns_clone=1 \
    >/etc/sysctl.d/90-unprivileged_userns.conf'
sudo sysctl -p /etc/sysctl.d /etc/sysctl.d/90-unprivileged_userns.conf
```

5.2 Unprivileged Installations

As detailed in the *non-setuid installation* (page 21) section, SingularityCE can be compiled or configured with the `allow setuid = no` option in `singularity.conf` to not perform privileged operations using the `starter-setuid` binary.

When SingularityCE does not use `setuid` all container execution will use a user namespace. In this mode of operation, some features are not available, and there are impacts to the security/integrity guarantees when running SIF container images:

- SIF and other single file container images will be mounted using FUSE, falling back to extraction to a directory if a FUSE mount is not possible.
- The effectiveness of signing and verifying container images is reduced as, when mounted by a user controlled FUSE binary or extracted to a directory, modification is possible and verification of the image's original signature cannot be performed at runtime.
- Filesystem image, and SIF-embedded persistent overlays cannot be used. Directory overlays require kernel >=5.11.
- Encrypted containers cannot be used. SingularityCE mounts encrypted containers directly through the kernel, so that encrypted content is not extracted to disk. This requires the setuid workflow.
- Fakeroot functionality will rely on external setuid root `newuidmap` and `newgidmap` binaries which may be provided by the distribution.

5.3 –userns option

The `--userns` option to `singularity run/exec/shell` will start a container using a user namespace, avoiding the setuid privileged workflow for container setup even if SingularityCE was compiled and configured to use setuid by default.

The same limitations apply as in an unprivileged installation.

5.4 Fakeroot feature

Fakeroot (or commonly referred as rootless mode) allows an unprivileged user to run a container as a **"fake root"** user by leveraging user namespaces with user namespace UID/GID mapping.

User namespace UID/GID mapping allows a user to act as a different UID/GID in the container than they are on the host. A user can access a configured range of UIDs/GIDs in the container, which map back to (generally) unprivileged user UIDs/GIDs on the host. This allows a user to be `root (uid 0)` in a container, install packages etc., but have no privilege on the host.

5.4.1 Requirements

In addition to user namespace support, SingularityCE must manipulate `subuid` and `subgid` maps for the user namespace it creates. By default this happens transparently in the setuid workflow. With unprivileged installations of SingularityCE or where `allow setuid = no` is set in `singularity.conf`, SingularityCE attempts to use external setuid binaries `newuidmap` and `newgidmap`, so you need to install those binaries on your system.

5.4.2 Basics

To obtain the subuid and subgid mappings for a user, SingularityCE will either:

- Use libsubid, if compiled with support for this library.
- Use the `/etc/subuid` and `/etc/subgid` files directly, if compiled without libsubid support.

If compiled with libsubid support, the source of the mappings is controlled by the `subid` entry in `/etc/nsswitch.conf`. This will generally be `files`, in which case libsubid uses the local `/etc/subuid` and `/etc/subgid` files. Some distributions support using e.g. an LDAP source, via `sss`, when configured by the administrator.

When the `/etc/subuid` and `/etc/subgid` files are used as the source of mappings, a user must have an entry inside them to use the fakeroot feature. SingularityCE provides a *config fakeroot* (page 52) command to assist in managing these files, but it is important to understand how they work.

For user `foo` an entry in `/etc/subuid` might be:

```
foo:100000:65536
```

where `foo` is the username, `100000` is the start of the UID range that can be used by `foo` in a user namespace uid mapping, and `65536` number of UIDs available for mapping.

Same for `/etc/subgid`:

```
foo:100000:65536
```

Note: Some distributions add users to these files on installation, or when `useradd`, `adduser`, etc. utilities are used to manage local users.

The glibc nss name service switch mechanism does not currently support managing `subuid` and `subgid` mappings with external directory services such as LDAP. You must manage or provision mapping files direct to systems where fakeroot will be used.

Warning: SingularityCE requires that a range of at least `65536` IDs is used for each mapping. Larger ranges may be defined without error.

It is also important to ensure that the subuid and subgid ranges defined in these files don't overlap with each other, or any real UIDs and GIDs on the host system.

So if you want to add another user `bar`, `/etc/subuid` and `/etc/subgid` will look like:

```
foo:100000:65536
bar:165536:65536
```

Resulting in the following allocation:

User	Host UID	Sub UID/GID range
foo	1000	100000 to 165535
bar	1001	165536 to 231071

Inside a user namespace / container, `foo` and `bar` can now act as any UID/GID between 0 and 65536, but these UIDs are confined to the container. For `foo` UID 0 in the container will map to the host `foo` UID `1000` and `1 to 65536` will map to `100000-165535` outside of the container etc. This impacts the ownership of files, which will have different IDs inside and outside of the container.

Note: If you are managing large numbers of fakeroot mappings you may wish to specify users by UID rather than username in the `/etc/subuid` and `/etc/subgid` files. The man page for `subuid` advises:

"When large number of entries (10000-100000 or more) are defined in /etc/subuid, parsing performance penalty will become noticeable. In this case it is recommended to use UIDs instead of login names. Benchmarks have shown speed-ups up to 20x."

5.4.3 Filesystem considerations

Based on the above range, here we can see what happens when the user `foo` create files with `--fakeroot` feature:

Create file with container UID	Created host file owned by UID
0 (default)	1000
1 (daemon)	100000
2 (bin)	100001

Outside of the fakeroot container the user may not be able to remove directories and files created with a subuid, as they do not match with the user's UID on the host. The user can remove these files by using a container shell running with fakeroot.

5.4.4 Network configuration

With fakeroot, users can request a container network named `fakeroot`, other networks are restricted and can only be used by the real host root user. By default the `fakeroot` network is configured to use a network veth pair.

Warning: Do not change the `fakeroot` network type in `etc/singularity/network/40_fakeroot.conflist` without considering the security implications.

Note: Unprivileged installations of SingularityCE cannot use `fakeroot` network as it requires privilege during container creation to set up the network.

5.4.5 Configuration with `config fakeroot`

SingularityCE 3.5 and above provides a `config fakeroot` command that can be used by a root user to administer local system `/etc/subuid` and `/etc/subgid` files in a simple manner. This allows users to be granted the ability to use Singularity's fakeroot functionality without editing the files manually. The `config fakeroot` command will automatically ensure that generated subuid/subgid ranges are an appropriate size, and do not overlap.

`config fakeroot` must be run as the `root` user, or via `sudo singularity config fakeroot` as the `/etc/subuid` and `/etc/subgid` files form part of the system configuration, and are security sensitive. You may `--add` or `--remove` user subuid/subgid mappings. You can also `--enable` or `--disable` existing mappings.

Note: If you deploy SingularityCE to a cluster you will need to make arrangements to synchronize `/etc/subuid` and `/etc/subgid` mapping files to all nodes.

At this time, the glibc name service switch functionality does not support subuid or subgid mappings, so they cannot be defined in a central directory such as LDAP.

Adding a fakeroot mapping

Use the `-a/--add <user>` option to `config fakeroot` to create new mapping entries so that `<user>` can use the fakeroot feature of Singularity:

```
$ sudo singularity config fakeroot --add dave

# Show generated `/etc/subuid`
$ cat /etc/subuid
1000:4294836224:65536

# Show generated `/etc/subgid`
$ cat /etc/subgid
1000:4294836224:65536
```

The first subuid range will be set to the top of the 32-bit UID space. Subsequent subuid ranges for additional users will be created working down from this value. This minimizes the change of overlap with real UIDs on most systems.

Note: The `config fakeroot` command generates mappings specified using the user's uid, rather than their username. This is the preferred format for faster lookups when configuring a large number of mappings, and the command can be used to manipulate these by username.

Deleting, disabling, enabling mappings

Use the `-r/--remove <user>` option to `config fakeroot` to completely remove mapping entries. The `<user>` will no longer be able to use the fakeroot feature of Singularity:

```
$ sudo singularity config fakeroot --remove dave
```

Warning: If a fakeroot mapping is removed, the subuid/subgid range may be assigned to another user via `--add`. Any remaining files from the prior user that were created with this mapping will be accessible to the new user via fakeroot.

The -d/--disable and -e/--enable options will comment and uncomment entries in the mapping files, to temporarily disable and subsequently re-enable fakeroot functionality for a user. This can be useful to disable fakeroot for a user, but ensure the subuid/subgid range assigned to them is reserved, and not re-assigned to a different user.

```
# Disable dave
$ sudo singularity config fakeroot --disable dave

# Entry is commented
$ cat /etc/subuid
!1000:4294836224:65536

# Enable dave
$ sudo singularity config fakeroot --enable dave

# Entry is active
$ cat /etc/subuid
1000:4294836224:65536
```

5.5 Unprivileged Builds Without User Namespaces

Where local container builds need to be performed unprivileged, but user namespaces and / or subuid mapping cannot be enabled, limited support is provided via the use of `proot`. This functionality was introduced in {SingularityCE} 3.11.

`proot` is an optional dependency of SingularityCE that can be installed from community distribution repositories, or a static binary available from proot-me.github.io. The `proot` executable should be on the `PATH` in order for SingularityCE to use it.

When `singularity build` is run against a definition file by a non-root user, and without the `--fakeroot` option, SingularityCE will search the `PATH` for `proot`. If it is found, then the `%post` section of the build will run as an emulated root user. Commands run as the user who invoked `singularity build`, but `proot` will intercept system calls, so that the commands appear to be running as root.

Unprivileged builds with `proot` have limitations, as the emulation of the root user is not complete. These builds:

- Do not support arch / debootstrap / yum / zypper bootstraps. Use localimage, library, oras, or one of the docker/oci sources.
- Do not support `%pre` and `%setup` sections.
- Run the `%post` sections of a build in the container as an emulated root user.
- Run the `%test` section of a build as the non-root user, like singularity test.
- Are subject to any restrictions imposed in singularity.conf.
- Incur a performance penalty due to proot's ptrace based interception of syscalls.
- May fail if the `%post` script requires privileged operations that proot cannot emulate.

CHAPTER

SIX

SECURITY IN SINGULARITYCE

6.1 Security Policy

If you suspect you have found a vulnerability in SingularityCE we want to work with you so that it can be investigated, fixed, and disclosed in a responsible manner. Please follow the steps in our published Security Policy, which begins with contacting us privately via security@sylabs.io

Sylabs discloses vulnerabilities found in SingularityCE through public CVE reports, and notifications on our community channels. We encourage all users to monitor new releases of SingularityCE for security information. Security patches are applied to the latest open-source release.

SingularityPRO is a professionally curated and licensed version of SingularityCE that provides added security, stability, and support beyond that offered by the open source project. Security and bug-fix patches are backported to select versions of SingularityPRO, so that they can be deployed long-term where required. PRO users receive security fixes as detailed in the Sylabs Security Policy.

6.2 Background

SingularityCE grew out of the need to implement a container platform that was suitable for use on shared systems, such as HPC clusters. In these environments multiple people access a shared resource. User accounts, groups, and standard file permissions limit their access to data, devices, and prevent them from disrupting or accessing others' work.

To provide security in these environments a container needs to run as the user who starts it on the system. Before the widespread adoption of the Linux user namespace, only a privileged user could perform the operations which are needed to run a container. A default Docker installation uses a root-owned daemon to start containers. Users can ask the daemon to start a container on their behalf. However, coordinating a daemon with other job-scheduling software is difficult and, since the daemon is privileged, users can ask it to carry out actions that they wouldn't normally have permission to do.

When a user runs a container with SingularityCE, by default it is started as a normal process running under the user's account. Standard file permissions and other security controls based on user accounts, groups, and processes apply.

The exact way in which a container is configured and executed depends on whether it is run:

- Using the default native runtime, in setuid mode.
- Using the native runtime, in non-setuid / unprivileged mode.
- In OCI-mode.

6.3 Default Native Runtime

When a container is run with the default native runtime (not OCI-mode), a standard installation of SingularityCE will use a setuid root starter executable for container setup. Optionally, SingularityCE can be built or configured to run unprivileged. Unprivileged execution performs container setup within an unprivileged user namespace.

6.3.1 Setuid Mode

Using a setuid binary to run container setup operations was essential to support containers on older Linux distributions, such as CentOS 6, that were previously common in HPC and enterprise.

On more modern distributions, where unprivileged user namespace creation is permitted, setuid mode is still used by default because:

- Many HPC sites disable unprivileged user namespace creation due to their specific security risk model, or restrictions on kernel updates.
- Full support for using supplementary groups from the host system is not possible within user namespaces, impacting creation of files in some filesystem layouts that are common for large shared projects.
- There are preformance advantages for filesystem mounts performed via the kernel, instead of in userspace with FUSE.
- Privileged operations are required to handle LUKS2 encrypted containers.
- Privileged operations are required for container network configuration using Container Network Interface (CNI) plugins.
- Setuid mode allows execution limits configured by an administrator to be enforced, on systems where unprivileged user namespace creation is disabled.

To safely execute containers in setuid mode, SingularityCE uses a number of Linux kernel features. The container file system is mounted using the `nosuid` option, and processes are started with the `PR_NO_NEW_PRIVS` flag set. This means that even if you run `sudo` inside your container, you won't be able to change to another user, or gain root privileges by other means.

If you do require the additional isolation of the network, devices, PIDs etc. provided by other runtimes, SingularityCE can make use of additional namespaces and functionality such as seccomp and cgroups.

If you are concerned about potential security impacts of performing kernel filesystem mounts, or are unable to patch kernel filesystem vulnerabilities in a timely manner, you may wish to *disable them via singularity.conf* (page 33).

6.3.2 Unprivileged / User Namespace Mode

SingularityCE supports running containers without setuid, using user namespaces. It can be compiled with the `--without-setuid` option, or `allow setuid = no` can be set in `singularity.conf` to disable setuid mode and execute all containers via a user namespace. With this configuration, all container setup operations run as the user who starts the `singularity` program. However, there are some disadvantages:

- SIF and other single file container images will be mounted using FUSE, falling back to extraction to a directory if a FUSE mount is not possible.
- FUSE mounts may result in a small performance penalty.
- Running containers from a directory can dramatically impact the speed of execution for workloads accessing large numbers of small files (such as python application startup). This is because containers extracted to a directory do not benefit from the reduced metadata load on the filesystem that using an image file normally provides.

- The effectiveness of signing and verifying container images is reduced as, when mounted by a user controlled FUSE binary or extracted to a directory, modification is possible and verification of the image's original signature cannot be performed.
- Encryption is not supported. SingularityCE leverages kernel LUKS2 mounts to run encrypted containers without decrypting their content to disk.
- Some sites hold the opinion that vulnerabilities in kernel user namespace code could have greater impact than vulnerabilities confined to a single piece of setuid software. Therefore they are reluctant to enable unprivileged user namespace creation.
- Limitations on container execution by location, valid signatures, user/group cannot be effectively enforced. A user who can create a user namespace unprivileged would be able to trivially bypass any restrictions set for the system's SingularityCE installation.

Because of the points above, the default mode of operation of SingularityCE uses a setuid binary. Sylabs aims to reduce the circumstances that require this as new functionality is developed and reaches commonly deployed Linux distributions.

6.4 OCI-Mode

In OCI-Mode (`--oci`), SingularityCE always performs container setup within a user namespace. The setuid starter executable is not used, even when `allow setuid = yes` is set in `singularity.conf`.

Containers can be run directly from SIF files as long as the kernel is new enough to support FUSE mounts from user namespaces. Otherwise containers will be extracted to a directory before execution (unless this option *has been disabled* (page 36)).

Unprivileged users executing a container in OCI-Mode can access other uid/gids, can disable the `nosuid` flag on container mounts, and can grant capabilities to the container. However, these actions are always limited to the scope of the user namespace in which the container is created. On the host, all operations are mapped to the user's own uid/gid or those in the subuid/subgid map that an administrator has configured for the user. Increased capabilities, and other expanded permissions, do not apply outside of the container on the host.

6.5 Security Implications of Unprivileged User Namespaces

Warning: If you rely on the ECL or other container execution limits, you must disable unprivileged user namespace creation on your systems.

When unprivileged user namespace creation is allowed on a system, a user can supply and use their own unprivileged installation of Singularity or another container runtime. They may also be able to use standard system tools such as `unshare`, `nsenter`, and FUSE mounts to access / execute arbitrary containers without installing any runtime. Both of these approaches will allow users to bypass any restrictions that have been set in a system-wide installation of SingularityCE. These include:

- The `allow container` and `limit container` directives in `singularity.conf`.
- The Execution Control List, which restricts execution of SIF container images via signature checks.

Note also that SingularityCE's *–oci* mode is an unprivileged runtime that requires unprivileged user namespace creation. It does not implement the container restrictions that cannot be effectively enforced when unprivileged user namespaces are available.

If your primary security concern is that of restricting the containers which users can execute, you should use singularity in setuid mode, and ensure unprivileged user namespace creation is disabled on the host.

6.6 Singularity Image Format (SIF)

SingularityCE uses SIF as its default container format. A SIF container is a single file, which makes it easy to manage and distribute. Inside the SIF file, the container filesystem is held in a SquashFS object. By default, we mount the container filesystem directly using SquashFS. On a network filesystem this means that reads from the container are data-only. Metadata operations happen locally, speeding up workloads with many small files.

Holding the container image in a single file also enable unique security features. The container filesystem is immutable, and can be signed. The signature travels in the SIF image itself so that it is always possible to verify that the image has not been tampered with or corrupted.

We use private PGP keys to create a container signature, and the public key in order to verify the container. Verification of signed containers happens automatically in `singularity pull` commands against the Sylabs Cloud Container Library. A Keystore in the Sylabs Cloud makes it easier to share and obtain public keys for container verification.

A container may be signed once, by a trusted individual who approves its use. It could also be signed with multiple keys to signify it has passed each step in a CI/CD QA & Security process. In setuid mode, SingularityCE can be configured with an execution control list (ECL). The ECL requires the presence of one or more valid signatures, to limit execution to approved containers on systems that have unprivileged user namespace creation disabled.

In SingularityCE 3.4 and above, the root filesystem of a container (stored in the squashFS partition of SIF) can be encrypted. As a result, everything inside the container becomes inaccessible without the correct key or passphrase. The content of the container is private, even if the SIF file is shared in public.

Encryption and decryption are performed using the Linux kernel's LUKS2 feature. This is the same technology routinely used for full disk encryption. The encrypted container is mounted directly through the kernel. Unlike other container formats, an encrypted container is not decrypted to disk in order to run it.

6.7 Plugins

As discussed in the SingularityCE User Guide, plugins provide a way to augment Singularity with additional functionality. Before using the `singularity plugin compile` or `singularity plugin install` commands to compile or add a new plugin to your SingularityCE installation, make sure that you trust the origin of the plugin, and that you are certain it does not contain any malicious code.

For further information on verifying the contents of SIF files using cryptographic signatures, see the "Sign and Verify" section of the SingularityCE User Guide.

6.8 Configuration & Runtime Options

System administrators who manage SingularityCE can use configuration files to set security restrictions, grant or revoke a user's capabilities, manage resources and authorize containers etc.

Configuration files and their parameters are *documented for administrators here* (page 29).

When running a container as root, SingularityCE can apply hardening rules using seccomp and apparmor. See the 'Security Options' section of the user guide.

Limits on resource usage by containers can be enforced using cgroups. On systems that use cgroups v1, only the root user can set resource limits. On systems that use cgroups v2 and systemd, all users can apply resource limits as long as the system is configured for delegation.

By default, EL9, Ubuntu 22.04, Debian 11, Fedora 31 and newer use cgroups v2 and are configured for delegation so that unprivileged users will be able to use the `--apply-cgroups` and other resource limit flags of SingularityCE without further configuration.

On EL8 and Ubuntu 20.04 it is possible to setup a compatible configuration by following the 'Enabling cgroup v2' and 'Enabling CPU, CPUSET, and I/O delegation' steps at the rootless containers website

See the 'Limiting Container Resources' section of the user guide for more details of how to apply cgroups limits to containers at runtime.

CHAPTER

SEVEN

INSTALLED FILES

An installation of SingularityCE 4.3.0, performed as root via `sudo make install` consists of the following files, with ownership and permissions required to use the *setuid* workflow:

```
# Main executables
bin root:root 755 (drwxr-xr-x)
bin/singularity root:root 755 (-rwxr-xr-x)
bin/run-singularity root:root 755 (-rwxr-xr-x)

# Configuration files
etc root:root 755 (drwxr-xr-x)
etc/singularity root:root 755 (drwxr-xr-x)
etc/singularity/singularity.conf root:root 644 (-rw-r--r--)
etc/singularity/remote.yaml root:root 644 (-rw-r--r--)
etc/singularity/network root:root 755 (drwxr-xr-x)
etc/singularity/network/00_bridge.conflist root:root 644 (-rw-r--r--)
etc/singularity/network/10_ptp.conflist root:root 644 (-rw-r--r--)
etc/singularity/network/20_ipvlan.conflist root:root 644 (-rw-r--r--)
etc/singularity/network/30_macvlan.conflist root:root 644 (-rw-r--r--)
etc/singularity/network/40_fakeroot.conflist root:root 644 (-rw-r--r--)
etc/singularity/capability.json root:root 644 (-rw-r--r--)
etc/singularity/ecl.toml root:root 644 (-rw-r--r--)
etc/singularity/seccomp-profiles root:root 755 (drwxr-xr-x)
etc/singularity/seccomp-profiles/default.json root:root 644 (-rw-r--r--)
etc/singularity/nvliblist.conf root:root 644 (-rw-r--r--)
etc/singularity/rocmliblist.conf root:root 644 (-rw-r--r--)
etc/singularity/cgroups root:root 755 (drwxr-xr-x)
etc/singularity/cgroups/cgroups.toml root:root 644 (-rw-r--r--)
etc/singularity/global-pgp-public root:root 644 (-rw-r--r--)

# Runtime executables
libexec root:root 755 (drwxr-xr-x)
libexec/singularity root:root 755 (drwxr-xr-x)
libexec/singularity/bin root:root 755 (drwxr-xr-x)
libexec/singularity/bin/conmon root:root 755 (-rwxr-xr-x)
libexec/singularity/bin/singularity-buildkitd root:root 755 (-rwxr-xr-x)
libexec/singularity/bin/squashfuse_ll root:root 755 (-rwxr-xr-x)
libexec/singularity/bin/starter root:root 755 (-rwxr-xr-x)
libexec/singularity/bin/starter-suid root:root 4755 (-rwsr-xr-x)

# CNI network plugins
```

(continues on next page)

(continued from previous page)

```
libexec/singularity/cni root:root 755 (drwxr-xr-x)
libexec/singularity/cni/bandwidth root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/bridge root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/dhcp root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/firewall root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/host-device root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/host-local root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/ipvlan root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/loopback root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/macvlan root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/portmap root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/ptp root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/sbr root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/static root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/tap root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/tuning root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/vlan root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/vrf root:root 755 (-rwxr-xr-x)

# Documentation (man pages / bash completions)
share root:root 755 (drwxr-xr-x)
share/bash-completion root:root 755 (drwxr-xr-x)
share/bash-completion/completions root:root 755 (drwxr-xr-x)
share/bash-completion/completions/singularity root:root 644 (-rw-r--r--)
share/man root:root 755 (drwxr-xr-x)
share/man/man1 root:root 755 (drwxr-xr-x)
share/man/man1/singularity.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-build.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-cache.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-cache-clean.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-cache-list.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-capability.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-capability-add.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-capability-avail.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-capability-drop.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-capability-list.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-config.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-config-fakeroot.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-config-global.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-delete.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-exec.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-inspect.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-instance.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-instance-list.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-instance-start.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-instance-stats.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-instance-stop.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-key.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-key-export.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-key-import.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-key-list.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-key-newpair.1 root:root 644 (-rw-r--r--)
```

(continues on next page)

(continued from previous page)

```
share/man/man1/singularity-key-pull.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-key-push.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-key-remove.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-key-search.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-keyserver.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-keyserver-add.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-keyserver-list.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-keyserver-login.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-keyserver-logout.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-keyserver-remove.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-attach.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-create.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-delete.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-exec.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-kill.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-mount.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-pause.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-resume.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-run.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-start.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-state.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-umount.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-oci-update.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-overlay.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-overlay-create.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-plugin.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-plugin-compile.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-plugin-create.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-plugin-disable.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-plugin-enable.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-plugin-inspect.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-plugin-install.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-plugin-list.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-plugin-uninstall.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-pull.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-push.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-registry.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-registry-list.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-registry-login.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-registry-logout.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-remote.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-remote-add.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-remote-get-login-password.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-remote-list.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-remote-login.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-remote-logout.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-remote-remove.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-remote-status.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-remote-use.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-run.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-run-help.1 root:root 644 (-rw-r--r--)
```

(continues on next page)

(continued from previous page)

```
share/man/man1/singularity-search.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-shell.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-sif.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-sif-add.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-sif-del.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-sif-dump.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-sif-header.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-sif-info.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-sif-list.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-sif-new.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-sif-setprim.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-sign.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-test.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-verify.1 root:root 644 (-rw-r--r--)
share/man/man1/singularity-version.1 root:root 644 (-rw-r--r--)

# Container state directories
var root:root 755 (drwxr-xr-x)
var/singularity root:root 755 (drwxr-xr-x)
var/singularity/mnt root:root 755 (drwxr-xr-x)
var/singularity/mnt/session root:root 755 (drwxr-xr-x)
```

CHAPTER

EIGHT

LICENSE

This documentation is subject to the following 3-clause BSD license:

```
Copyright (c) 2017, SingularityWare, LLC. All rights reserved.
Copyright (c) 2018-2025, Sylabs, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

* Neither the name of the copyright holder nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```