SingularityCE Admin Guide

Release 3.9

SingularityCE Project Contributors

Jan 10, 2022

CONTENTS

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

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.

1.1 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.

SingularityCE favors an 'integration over isolation' approach to containers. By default only the mount namespace is isolated for containers, so that they have their own filesystem view. Access to hardware such as GPUs, high speed networks, and shared filesystems is easy and does not require special configuration. 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. Where more complete isolation is important, SingularityCE can use additional Linux namespaces and other security and resource limits to accomplish this.

1.2 SingularityCE Security

Note: See also the *security section* (page 41) of this guide, for more detail.

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

- The 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.
- 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.

To support the SIF image format, automated networking setup etc., and older Linux distributions without user namespace support, Singularity runs small amounts of privileged container setup code via a `starter-setuid` binary. This is a 'setuid root' binary, so that SingularityCE can perform filesystem loop mounts and other operations that need privilege. The setuid flow is the default mode of operation, but *can be disabled* (page 13) on build, or in the `singularity.conf` configuration file if required.

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

See the *non-setuid installation section* (page 13) for further detail on how to install SingularityCE to run in non-setuid mode.

1.3 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 7) 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 13) section of this guide for detail of how to install as a non-root user, and how this affects the functionality of SingularityCE.

1.3.1 Install Dependencies

On Red Hat Enterprise Linux or CentOS install the following dependencies:

```
$ sudo yum update -y && \
    sudo yum groupinstall -y 'Development Tools' && \
    sudo yum install -y \
    openssl-devel \
    libuuid-devel \
    libseccomp-devel \
    wget \
    squashfs-tools \
    cryptsetup
```

On Ubuntu or Debian install the following dependencies:

```
$ sudo apt-get update && sudo apt-get install -y \
    build-essential \
    uuid-dev \
```

(continues on next page)

(continued from previous page)

```
    libgpgme-dev \
    squashfs-tools \
    libseccomp-dev \
    wget \
    pkg-config \
    git \
    cryptsetup-bin
```

1.3.2 Install Go

SingularityCE v3 is written primarily in Go, and you will need Go 1.16 or above installed to compile it from source. Versions of Go packaged by your distribution may not be new enough to build SingularityCE.

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=1.17.6 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
```

1.3.3 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=3.9.2 && # 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}
```

1.3.4 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.

- `--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`.

1.4 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<https://github.com/sylabs/singularity/releases>`_** 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.

1.5 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 19) section where the files and configuration options are discussed.

1.6 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

TWO

INSTALLING SINGULARITYCE

This section will guide you through the process of installing SingularityCE 3.9.2 via several different methods. (For instructions on installing earlier versions of SingularityCE please see earlier versions of the docs.)

2.1 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.

2.1.1 System Requirements

SingularityCE requires ~140MiB 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.

External Binaries

Singularity depends on a number of external binaries for full functionality. From SingularityCE 3.9, the methods that are used to find these binaries have been standardized as below.

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

- `true`
- `mkfs.ext3` is used to create overlay images.
- `cp`
- `dd`
- `newuidmap` and `newgidmap` are distribution provided setuid binaries used to configure subuid/gid mappings for `--fakeroot` in non-setuid installs.

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` or `yum`, `rpm`, `curl` for EL derived RPM based distributions.
- `uname`, `zypper`, `SUSEConnect` for SLES derived RPM based distributions.

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. CentOS 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. CentOS / 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 the Linux `overlay` filesystem driver to construct a container root filesystem that combines files from different locations. 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 / (sub)uid/gid mapping

When SingularityCE is run using the *fakeroot* (page 36) option 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

Lustre and GPFS 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 or GPFS filesystem. SIF containers on Lustre / GPFS 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 or GPFS filesystem.
- When using `--fakeroot` to build or run a container, your `TMPDIR/SINGULARITY_TMPDIR` should not be a Lustre or GPFS location.
- You should not run a sandbox container with `--fakeroot` from a Lustre or GPFS location.

2.1.2 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<https://github.com/sylabs/singularity/releases>`_** 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.

2.1.3 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 15) before executing the installation commands. You will also need to install some dependencies and install Go.

Install Dependencies

On Red Hat Enterprise Linux or CentOS install the following dependencies:

```
$ sudo yum update -y && \
    sudo yum groupinstall -y 'Development Tools' && \
    sudo yum install -y \
    openssl-devel \
    libuuid-devel \
    libseccomp-devel \
    wget \
    squashfs-tools \
    cryptsetup
```

On Ubuntu or Debian install the following dependencies:

```
$ sudo apt-get update && sudo apt-get install -y \
    build-essential \
    uuid-dev \
    libgpgme-dev \
    squashfs-tools \
    libseccomp-dev \
    wget \
    pkg-config \
    git \
    cryptsetup-bin
```

Note: You can build SingularityCE (3.5+) without `cryptsetup` available, but will not be able to use encrypted containers without it installed on your system.

Install Go

SingularityCE v3 is written primarily in Go, and you will need Go 1.16 or above installed to compile it from source.

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=1.17.6 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=3.9.2 && # 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 >=v3.9.2. 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 **master 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 v3.2 with the latest bug fixes and enhancements checkout `release-3.2`. Release branches may be less stable than code in a tagged point release.
- **master branch**: The `master` 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 `master` 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 https://github.com/sylabs/singularity.git && \
    cd singularity && \
    git checkout v3.9.2
```

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 13).

```
$ ./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.

- `--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`.

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 35) and *limitations* (page 36) in the *user namespace* (page 35) 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.)

2.1.4 Build and install an RPM

If you use RHEL, CentOS or SUSE, building and installing a Singularity RPM allows your SingularityCE installation be more easily managed, upgraded and removed. In SingularityCE >=v3.0.1 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 11) and installing *Go* (page 12) as detailed above, you are ready to download the tarball and build and install the RPM.

```
$ export VERSION=3.9.2 && # 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.el7.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-3.9.2.el7.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.

2.1.5 Remove an old version

In a standard installation of SingularityCE 3.0.1 and beyond (when building from source), the command `sudo make 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 14) so that it can be updated and/or uninstalled with ease in the future.

2.1.6 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
3.9.2
```

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
PACKAGE_VERSION=3.9.2
BUILDDIR=/home/dtrudg/Sylabs/Git/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
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
```

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 44).

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 unit-test` runs basic unit tests
- `make integration-test` runs 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.

2.2 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.)

For this reason, the SingularityCE community maintains a set of Vagrant Boxes via Vagrant Cloud, one of Hashicorp's open source tools. The current versions can be found under the sylabs organization.

2.2.1 Windows

Install the following programs:

- Git for Windows
- VirtualBox for Windows
- Vagrant for Windows
- Vagrant Manager for Windows

2.2.2 Mac

SingularityCE is available via Vagrant (installable with Homebrew or manually)

To use Vagrant via Homebrew:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
$ brew install --cask virtualbox vagrant vagrant-manager
```

2.2.3 SingularityCE Vagrant Box

Run Git Bash (Windows) or open a terminal (Mac) and create and enter a directory to be used with your Vagrant VM.

```
$ mkdir vm-singularity-ce && \
    cd vm-singularity-ce
```

If you have already created and used this folder for another VM, you will need to destroy the VM and delete the Vagrantfile.

```
$ vagrant destroy && \
    rm Vagrantfile
```

Then issue the following commands to bring up the Virtual Machine. (Substitute a different value for the `$VM` variable if you like.)

```
$ export VM=sylabs/singularity-ce-3.8-ubuntu-bionic64 && \
    vagrant init $VM && \
    vagrant up && \
    vagrant ssh
```

You can check the installed version of SingularityCE with the following:

```
vagrant@vagrant:~$ singularity version
3.9.2
```

Of course, you can also start with a plain OS Vagrant box as a base and then install SingularityCE using one of the above methods for Linux.

2.2.4 SingularityCE Docker Image

It is possible to use a Dockerized Singularity, here is a sample `compose.yaml` (Singularity version 3.7.4) for use with Docker Compose:

```
services:
  singularity:
    image: quay.io/singularity/singularity:v3.7.4-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

THREE

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.

3.1 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.

3.1.1 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 36) for more information about running SingularityCE without `setuid` enabled.

`root default capabilities`: SingularityCE allows the specification of capabilities kept by the root user when running a container by default. Options include:

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

Note: The root user can manage the capabilities granted to individual containers when they are launched through the `--add-caps` and `drop-caps` flags. Please see Linux Capabilities in the user guide for more information.

3.1.2 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.

3.1.3 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.

3.1.4 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.

3.1.5 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. This option specifies how large the default `sessiondir` should be (in MB). It should be set large enough to accommodate files that will be created in a `--writable-tmpfs`, or the empty `/tmp` and other paths provided when `--contain` is used.

`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.

3.1.6 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.

3.1.7 Limiting Container Execution

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 28).

`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 EXT image files.
- `allow container dir` permits / denies execution of sandbox directory containers.

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

This behavior differs from SingularityCE versions before 3.9, where the `allow container squashfs/extfs` directives also applied to the filesystem embedded in a SIF image.

3.1.8 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. Unrestricted use of CNI network configurations requires root privilege, as certain configurations may disrupt the host networking environment.

SingularityCE 3.8 allows specific users or groups to be granted the ability to run containers with administrator specified CNI configurations.

`allow net users`: Allow specified root administered CNI network configurations to be used by the specified list of users. By default only root may use CNI configuration, except in the case of a fakeroot execution where only

40_fakeroot.conflist is used. This feature only applies when SingularityCE is running in SUID mode and the user is non-root.

`allow net groups`: Allow specified root administered CNI network configurations to be used by the specified list of users. By default only root may use CNI configuration, except in the case of a fakeroot execution where only 40_fakeroot.conflist is used. This feature only applies when SingularityCE is running in SUID mode and the user is non-root.

`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.

3.1.9 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 29).

`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.

3.1.10 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.

3.1.11 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.

3.1.12 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.

3.1.13 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.

3.1.14 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.

3.2 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.

Note: SingularityCE does not currently support applying native cgroups v2 `unified` resource limit specifications. Use the cgroups v1 limits, which will be translated to v2 format when applied on a cgroups v2 system.

3.2.1 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:

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

Note: The `--apply-cgroups` option can only be used with 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.:

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

After that, you can verify that the container is only using 500MB of memory. This example assumes that there is only one running container. If you are running multiple containers you will find multiple cgroups trees under the `singularity` directory.

```
# cgroups v1
$ cat /sys/fs/cgroup/memory/singularity/*/memory.limit_in_bytes
  524288000

# cgroups v2 - note translation of memory.limit_in_bytes -> memory.max
$ cat /sys/fs/cgroup/singularity/*/memory.max
524288000
```

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
```

(continues on next page)

(continued from previous page)

```
        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, **except** native `unified` cgroups v2 constraints. Use the cgroups v1 limits, which will be translated to v2 format when applied on a cgroups v1 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.

3.3 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: 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`.

3.3.1 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`.

3.4 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

3.4.1 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`.

3.4.2 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`.

3.4.3 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).

3.5 capability.json

Warning: 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 36) is a better option.

SingularityCE provides full support for admins to grant and revoke 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.

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.

3.6 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.

3.7 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.

3.7.1 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.

3.7.2 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 `add-keyserver` and `remove-keyserver` 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 `remote` command group and managing keyservers, please check the Remote Userdocs.

CHAPTER

FOUR

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 setup 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 3 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.

4.1 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 (3.18 also adds OverlayFS support which is used by Singularity).

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'.

4.1.1 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
```

4.1.2 RHEL/CentOS 7

From 7.4, kernel support is included but must be enabled with:

```
sudo sh -c 'echo user.max_user_namespaces=15000 \
    >/etc/sysctl.d/90-max_net_namespaces.conf'
sudo sysctl -p /etc/sysctl.d /etc/sysctl.d/90-max_net_namespaces.conf
```

4.2 Unprivileged Installations

As detailed in the *non-setuid installation* (page 13) 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:

- All containers must be run from sandbox directories. SIF images are extracted to a sandbox directory on the fly, preventing verification at runtime, and potentially allowing external modification of the container at runtime.
- Filesystem image, and SIF-embedded persistent overlays cannot be used.
- 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.

4.3 –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.

4.4 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.

4.4.1 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.

4.4.2 Basics

Fakeroot relies on `/etc/subuid` and `/etc/subgid` files to find configured mappings from real user and group IDs, to a range of otherwise vacant IDs for each user on the host system that can be remapped in the user namespace. A user must have an entry in these system configuration files to use the fakeroot feature. SingularityCE provides a *config fakeroot* (page 38) 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."

4.4.3 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.

4.4.4 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 setup the network.

4.4.5 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
```

(continues on next page)

(continued from previous page)

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

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

CHAPTER

FIVE

SECURITY IN SINGULARITYCE

5.1 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.

5.2 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 request that the daemon starts a container on their behalf. However, coordinating a daemon with other schedulers 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, 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. In a default installation SingularityCE uses a setuid starter binary to perform only the specific tasks needed to setup the container.

5.3 Setuid & User Namespaces

Using a setuid binary to run container setup operations is essential to support containers on older Linux distributions, such as CentOS 6, that were previously common in HPC and enterprise. Newer distributions have support for 'unprivileged user namespace creation'. This means a normal user can create a user namespace, in which most setup operations needed to run a container can be run, unprivileged.

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 enable this. In this mode *all* operations run as the user who starts the `singularity` program. However, there are some disadvantages:

- SIF and other single file container images cannot be mounted directly. The container image must be extracted to a directory on disk to run. This impact the speed of execution. Workloads accessing large numbers of small files (such as python application startup) do not benefit from the reduced metadata load on the filesystem an image file provides.
- Replacing direct kernel mounts with a FUSE approach is likely to cause a significant reduction in performance.
- The effectiveness of signing and verifying container images is reduced as, when 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.

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.

5.4 Runtime & User Privilege Model

While other runtimes have aimed to safely sandbox containers executing as the `root` user, so that they cannot affect the host system, SingularityCE has adopted an alternative security model:

- Containers should be run as an unprivileged user.
- The user should never be able to elevate their privileges inside the container to gain control over the host.
- All permission restrictions on the user outside of a container should apply inside the container.
- Favor integration over isolation. Allow a user to use host resources such as GPUs, network file systems, high speed interconnects easily. The process ID space, network etc. are not isolated in separate namespaces by default.

To accomplish this, 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.

5.5 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. SingularityCE can be configured with an execution control list (ECL), which requires the presence of one or more valid signatures, to limit execution to approved containers.

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.

5.6 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.

For example, the *Execution Control List* (page 28) file allows restricting usage of SIF containers based on their signature and the key used to sign them.

Configuration files and their parameters are *documented for administrators here* (page 19).

When running a container as root, Singularity can apply hardening rules using cgroups, seccomp, apparmor. See the 'security options' section of the user guide, and *cgroups.toml documentation* (page 25).

CHAPTER

SIX

INSTALLED FILES

An installation of SingularityCE 3.9.2, performed as root via `sudo make install` consists of the following files, with ownership and permissions required to use the *setuid* workflow:

```
# Container session / state
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)

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

# Helper executables
libexec/singularity root:root 755 (drwxr-xr-x)
libexec/singularity/plugin root:root 755 (drwxr-xr-x)
libexec/singularity/bin root:root 755 (drwxr-xr-x)
libexec/singularity/bin/starter root:root 755 (-rwxr-xr-x)
libexec/singularity/bin/starter-suid root:root 4755 (-rwsr-xr-x)
libexec/singularity/cni root:root 755 (drwxr-xr-x)
libexec/singularity/cni/firewall root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/portmap root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/tuning root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/host-device root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/vrf root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/dhcp root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/host-local root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/static root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/loopback root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/ptp root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/bandwidth root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/ipvlan root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/macvlan root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/sbr root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/bridge root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/vlan root:root 755 (-rwxr-xr-x)

# Singularity configuration
etc/singularity root:root 755 (drwxr-xr-x)
etc/singularity/network root:root 755 (drwxr-xr-x)
etc/singularity/network/10_ptp.conflist root:root 644 (-rw-r--r--)
```

(continues on next page)

(continued from previous page)

```
etc/singularity/network/20_ipvlan.conflist root:root 644 (-rw-r--r--)
etc/singularity/network/00_bridge.conflist root:root 644 (-rw-r--r--)
etc/singularity/network/40_fakeroot.conflist root:root 644 (-rw-r--r--)
etc/singularity/network/30_macvlan.conflist 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/remote.yaml root:root 644 (-rw-r--r--)
etc/singularity/singularity.conf root:root 644 (-rw-r--r--)
etc/singularity/global-pgp-public root:root 644 (-rw-r--r--)
etc/singularity/capability.json 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/ecl.toml root:root 644 (-rw-r--r--)
etc/singularity/nvliblist.conf root:root 644 (-rw-r--r--)

# Bash completion configuration
etc/bash_completion.d root:root 755 (drwxr-xr-x)
etc/bash_completion.d/singularity root:root 644 (-rw-r--r--)
```

CHAPTER

SEVEN

LICENSE

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

```
Copyright (c) 2017, SingularityWare, LLC. All rights reserved.
Copyright (c) 2018-2021, 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.
```