Singularity Container Documentation

Release 3.6

Admin Docs

May 25, 2021

CONTENTS

Welcome to the Singularity Admin Guide!

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

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

CHAPTER

ONE

ADMIN QUICK START

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

For additional help or support contact the Sylabs team.

1.1 Architecture of Singularity

Singularity 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.

Singularity 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. User home directories, `/tmp` space, and installation specific mounts make it simple for users to benefit from the reproducibility of containerized applications without major changs to their existing workflows. Where more complete isolation is important, Singularity can use additional Linux namespaces and other security and resource limits to accomplish this.

1.2 Singularity Security

Singularity 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 `PR_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 Singularity can perform filesystem loop mounts and other operations that need privilege. The setuid flow is the default mode of operation, but *can be disabled* on build, or in the `singularity.conf` configuration file if required.

Note: Running Singularity 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* for further detail on how to install singularity to run in non-setuid mode.

1.3 Installation from Source

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

To install Singularity directly from source, follow the procedure below. Other methods are discussed in the *Installation* section.

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

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 \
    libgpgme-dev \
    squashfs-tools \
    libseccomp-dev \
    wget \
    pkg-config \
```

(continues on next page)

(continued from previous page)

```
    git \
    cryptsetup-bin
```

1.3.2 Install Go

Singularity v3 is written primarily in Go, and you will need Go 1.13 or above installed to compile it from source. Versions of Go packaged by your distribution may not be new enough to build Singularity.

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=1.13.5 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 Singularity from a GitHub release

You can download Singularity 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=3.6.4 && # adjust this as necessary \
    wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/singularity-
↪${VERSION}.tar.gz && \
    tar -xzf singularity-${VERSION}.tar.gz && \
    cd singularity
```

1.3.4 Compile & Install Singularity

Singularity 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 Singularity 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 Singularity on a shared system, or if you want to remove Singularity 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 Singularity 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 Singularity on a shared file system. The `--localstatedir` should be set to a directory that is present on each individual node.
- `-b`: Build Singularity in a given directory. By default this is `./builddir`.

1.4 Configuration

Singularity 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`.

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

`singularity.conf` contains the majority of options controlling the runtime behaviour of Singularity. Additional files control security, network, and resource configuration. Head over to the *Configuration files* section where the files and configuration options are discussed.

1.5 Test Singularity

You can run a quick test of Singularity 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 Singularity.

CHAPTER

TWO

INSTALLING SINGULARITY

This section will guide you through the process of installing Singularity 3.6.4 via several different methods. (For instructions on installing earlier versions of Singularity please see earlier versions of the docs.)

2.1 Installation on Linux

Singularity 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

Singularity 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 Singularity requires that the kernel supports:

- **OverlayFS mounts** - (minimum kernel >=3.18) Required for full flexiblity 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.

RHEL & CentOS 6 do not support these features, but Singularity can be used with some limitations.

Filesystem support / limitations

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

- We strongly recommend installing Singularity on local disk on each compute node.
- If Singularity 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 Singularity build with `./mconfig`.

Disk usage at the `--localstatedir` location is neglible (<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 Singularity, 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 Singularity 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 Singularity is run using the *fakeroot* option it creates a user namespace for the container, and UIDs / GIDs in that user namepace 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.

Singularity cache / atomic rename

Singularity 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 Singularity version 3.6 and above the cache is concurrency safe. Parallel runs of Singularity 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` filesytems 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 Singularity 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 Singularity 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 Before you begin

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

2.1.3 Install from Source

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

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 Singularity (3.5+) without `cryptsetup` available, but will not be able to use encrypted containers without it installed on your system.

Install Go

Singularity v3 is written primarily in Go, and you will need Go 1.13 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=1.13.5 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 Singularity from a release

You can download Singularity 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=3.6.4 && # adjust this as necessary \
    wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/singularity-
↪${VERSION}.tar.gz && \
    tar -xzf singularity-${VERSION}.tar.gz && \
    cd singularity
```

Checkout Code from Git

The following commands will install Singularity from the GitHub repo to `/usr/local`. This method will work for >=v3.6.4. 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 Singularity. 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 Singularity source code is downloaded to the appropriate directory use these commands.

```
$ git clone https://github.com/sylabs/singularity.git && \
    cd singularity && \
    git checkout v3.6.4
```

Compile Singularity

Singularity 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*.

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

By default Singularity 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 Singularity, install a personal version of Singularity on a shared system, or if you want to remove Singularity 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 Singularity 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 Singularity on a shared file system. The `--localstatedir` should be set to a directory that is present on each individual node.
- `-b`: Build Singularity in a given directory. By default this is `./builddir`.

Unprivileged (non-setuid) Installation

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

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

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

When singularity 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* and *limitations* in the *user namespace* section of this guide.

Source bash completion file

To enjoy bash shell completion with Singularity 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 Singularity 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 Singularity installation be more easily managed, upgraded and removed. In Singularity >=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-<version>.tar.gz*.

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

```
$ export VERSION=3.6.4 && # adjust this as necessary \
    wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/singularity-
↪${VERSION}.tar.gz && \
    rpmbuild -tb singularity-${VERSION}.tar.gz && \
    sudo rpm -ivh ~/rpmbuild/RPMS/x86_64/singularity-$VERSION-1.el7.x86_64.rpm && \
    rm -rf ~/rpmbuild singularity-$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-${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-$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 Singularity 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-3.6.4.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 Singularity 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 Singularity.

```
$ sudo rm -rf \
    /usr/local/libexec/singularity \
    /usr/local/var/singularity \
    /usr/local/etc/singularity \
    /usr/local/bin/singularity \
```

(continues on next page)

(continued from previous page)

```
    /usr/local/bin/run-singularity \
    /usr/local/etc/bash_completion.d/singularity
```

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

2.1.6 Distribution packages of Singularity

Note: Packaged versions of Singularity in Linux distribution repos are maintained by community members. They may be older releases of Singularity, as it can take time to package and distribute new versions. For the latest upstream versions of Singularity it is recommended that you build from source using one of the methods detailed above.

Install the CentOS/RHEL package using yum

The EPEL (Extra Packages for Enterprise Linux) repos contain Singularity rpms that are regularly updated. To install Singularity from the epel repos, first install the epel-release package and then install Singularity. For instance, on CentOS 6/7/8 do the following:

```
$ sudo yum update -y && \
    sudo yum install -y epel-release && \
    sudo yum update -y && \
    sudo yum install -y singularity
```

2.1.7 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 Singularity.

singularity buildcfg

Running `singularity buildcfg` will show the build configuration of an installed version of Singularity, and lists the paths used by Singularity. 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=3.6.4
BUILDDIR=/home/dtrudg/Sylabs/Git/singularity/builddir
PREFIX=/usr/local
EXECPREFIX=/usr/local
BINDIR=/usr/local/bin
```

(continues on next page)

(continued from previous page)

```
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 Singularity can be found in the *appendix, installed files section*.

Test Suite

The Singularity 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 singularity 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.

Singularity 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 Singularity 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 Singularity 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.

Sylabs has also developed a beta version of Singularity Desktop for Mac, which runs Singularity in a lightweight virtual machine, in a transparent manner.

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

To use Singularity Desktop for macOS (Beta Preview):

Download a Mac installer package here.

Singularity is also available via Vagrant (installable with Homebrew or manually) or with the Singularity Desktop for macOS (Alpha Preview).

To use Vagrant via Homebrew:

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

2.2.3 Singularity 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 && \
    cd vm-singularity
```

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-3.5-ubuntu-bionic64 && \
    vagrant init $VM && \
    vagrant up && \
    vagrant ssh
```

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

```
vagrant@vagrant:~$ singularity version
3.6.4
```

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

CHAPTER

THREE

SINGULARITY CONFIGURATION FILES

As a Singularity 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 Singularity across the system. All these files can be found in `/usr/local/etc/singularity` by default (though its location will obviously differ based on options passed during the installation). This page will describe the following configuration files and the various parameters contained by them. They are usually self documenting but here are several things to pay special attention to:

3.1 singularity.conf

Most of the configuration options are set using the file `singularity.conf` that defines the global configuration for Singularity across the entire system. Using this file, system administrators can have direct say as to what functions the users can utilize. As a security measure, for `setuid` installations of Singularity it must be owned by root and must not be writable by users or Singularity will refuse to run. This is not the case for `non-setuid` installations that will only ever execute with user priviledge and thus do not require such limitations. The options for this configuration are listed below. Options are grouped together based on relevance, the order of options within `singularity.conf` differs.

3.1.1 Setuid and Capabilities

`ALLOW SETUID`: To use all features of Singularity containers, Singularity will need to have access to some privileged system calls. One way singularity achieves this is by using binaries with the `setuid` bit enabled. This variable lets you enable/disable users ability to utilize these binaries within Singularity. By default, it is set to "yes", but when disabled, various Singularity features will not function. Please see *Unprivileged Installations* for more information about running Singularity without `setuid` enabled.

`ROOT DEFAULT CAPABILITIES`: Singularity 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

Singularity uses loop devices to facilitate the mounting of container filesystems from SIF images.

`MAX LOOP DEVICES`: This option allows an admin to limit the total number of loop devices Singularity 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: For some HPC systems, using the PID namespace has the potential of confusing some resource managers as well as some MPI implementations.

3.1.4 Configuration Files

Singularity allows for the automatic configuration of several system configuration files within containers to ease usage across systems.

Note: These options will do nothing unless the file or directory path exists within the container or Singularity has either overlay or underlay support enabled.

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

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

`CONFIG RESOLV_CONF`: This option determines if Singularity 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 Singularity runtime to create a container it needs to create a `sessiondir` to manage various components of the container, including mounting filesystems over the base image filesystem. This option specifies how large the default `sessiondir` should be (in MB) and will only affect users who use the `--contain` options without also specifying a location to perform default read/writes to via the `--workdir` or `--home` options.

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

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

`MOUNT DEV`: Should be set to "YES", if you want Singularity to automatically bind mount */dev* within the container. If set to 'minimal', then only 'null', 'zero', 'random', 'urandom', and 'shm' will be included.

`MOUNT DEVPTS`: This option determines if Singularity 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, Singularity will automatically determine the calling user's home directory and attempt to mount it into the container.

`MOUNT TMP`: When this option is enabled, Singularity will automatically bind mount `/tmp` and `/var/tmp` into the container from the host. If the `--contain` option is passed, Singularity 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 Singularity to probe the host for all mounted filesystems and bind those into containers at runtime.

`MOUNT SLAVE`: Singularity automatically mounts a handful host system directories to the container by default. This option determines if filesystem changes on the host should automatically be propogated to those directories in the container.

Note: This should be set to `yes` when autofs mounts in the system should show up in the container.

`MEMORY FS TYPE`: This option allows admins to choose the temporary filesystem used by Singularity. 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 for defining a list of files or directories to automatically be made available when Singularity runs a container. In order to successfully mount listed paths the file or directory path must exist within the container, or Singularity has 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.

`LIMIT CONTAINER OWNERS`: This restricts container execution to only allow conatiners that are owned by the specified user.

Note: This feature will only apply when Singularity 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 conatiners that are owned by the specified group.

Note: This feature will only apply when Singularity 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 Singularity 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 Singularity. Formats include `squashfs` which is used by SIF and v2.x Singularity images, `extfs` which is used for writable overlays and some legacy Singularity images, `dir` which is used by sandbox images and `encrypted` which is only used by SIF images to encrypt filesystem contents.

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

3.1.8 GPU Options

Singularity 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 Singularity checkout GPU Library Configuration.

`ALWAYS USE NV ${TYPE}`: Enabling this option will cause every action command (`exec/shell/run/instance`) to be executed with the `--nv` option implicitly added.

`ALWAYS USE ROCM ${TYPE}`: Enabling this option will cause every action command (`exec/shell/run/instance`) to be executed with the `--rocm` option implicitly added.

3.1.9 Supplemental Filesystems

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

`ENABLE OVERLAY`: This option will allow Singularity 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 Singularity 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.

3.1.10 External Tooling Paths

Internally, Singularity leverages several pieces of tooling in order to provide a wide breadth of features for users. Locations for these tools can be customized by system admins and referenced with the options below:

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

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

`MKSQUASHFS PATH`: This allows an admin to specify the location of `mksquashfs` if it is not installed in a standard location. If set, `mksquashfs` at this path will be used instead of a `mksquashfs` found in `PATH`.

`CRYPTSETUP PATH`: The location for `cryptsetup` is recorded by Singularity at build time and will use that value if this is undefined. This option allows an admin to set the path of `cryptsetup` if it is located in a custom location and will override the value recorded at build time.

3.1.11 Updating Configuration Options

In order to manage this configuration file, Singularity 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 priveledges 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

Cgroups or Control groups let you implement metering and limiting on the resources used by processes. You can limit memory, CPU. You can block IO, network IO, set SEL permissions for device nodes etc.

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

3.2.1 Examples

When you are limiting resources, apply the settings in the TOML file by using the path as an argument to the `--apply-cgroups` option like so:

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

3.2.2 Limiting memory

To limit the amount of memory that your container uses to 500MB (524288000 bytes):

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

Start your container like so:

```
$ sudo singularity instance start --apply-cgroups path/to/cgroups.toml my_container.sif␣
↪instance1
```

After that, you can verify that the container is only using 500MB of memory. (This example assumes that `instance1` is the only running instance.)

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

Do not forget to stop your instances after configuring the options.

Similarly, the remaining examples can be tested by starting instances and examining the contents of the appropriate subdirectories of `/sys/fs/cgroup/`.

3.2.3 Limiting CPU

Limit CPU resources using one of the following strategies. The `cpu` section of the configuration file can limit memory with the following:

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 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`.

3.2.4 Limiting IO

You can limit and monitor access to I/O for block devices. Use the `[blockIO]` section of the configuration file to do this like so:

```
[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 override `weight/leafWeight` for `/dev/loop0` and `/dev/loop1` block devices you would do something like this:

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

You could limit the IO read/write rate to 16MB per second for the `/dev/loop0` block device with the following configuration. 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
```

3.3 ecl.toml

The execution control list 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.

```
[[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 Singularity 3.6.0. Containers signed with older versions of Singularity Singularity will not pass ECL checks.

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

3.4 GPU Library Configuration

When a container includes a GPU enabled application, Singularity (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

If the nvidia-container-cli tool is installed on the host system, it will be used to locate any Nvidia libraries and binaries on the host system.

If `nvidia-container-cli` is not present, the `nvliblist.conf` file is used to specify libraries and executables that need to be injected into the container when running Singularity with the `--nv` Nvidia GPU support option. The default `nvliblist.conf` is suitable for CUDA 10.1, 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.

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 Singularity with the `--rocm` Radeon GPU support option. The default `rocmliblist.conf` is suitable for ROCm 2.10, but may need to modified if you need to include additional libraries, or further libraries are added to newer versions of the ROCm distribution.

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

Note: 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* is a better option.

Singularity provides full support for admins to grant and revoke Linux capabilities on a user or group basis. The `capability.json` file is maintained by Singularity 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 Singularity 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 Singularity 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`, Singularity 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 Singularity 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

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

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

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 with the `--global` flag so that they are easily available for users.

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

3.7.1 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
[sudo] password for dave:
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
[sudo] password for dave:
INFO:    Remote "company-remote" removed.
```

Note: Once users login 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.

For more insight into the `remote` command group, using remote endpoints, etc, 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.

Singularity uses user namespaces in 3 situations:

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

Note: CentOS/RHEL 7 doesn't provide a package for `newuidmap` and `newgidmap`, so you will need to compile/install **shadow-utils** by yourself.

Singularity expects to find these binaries in one of those standard paths: `/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin`

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 usernamespace. A user must have an entry in these system configuration files to use the fakeroot feature. Singularity provides a *config fakeroot* 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: Singularity 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 eachother, 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 Singularity cannot use `fakeroot` network as it requires privilege during container creation to setup the network.

4.4.5 Configuration with `config fakeroot`

Singularity 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 approriate 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 Singularity 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 definied in a central directory such as LDAP.

Adding a fakeroot mapping

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

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

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

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

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

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

Deleting, disabling, enabling mappings

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

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

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

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

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

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

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

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

CHAPTER

FIVE

SECURITY IN SINGULARITY CONTAINERS

Containers are all the rage today for many good reasons. They are light weight, easy to spin-up and require reduced IT management resources as compared to hardware VM environments. More importantly, container technology facilitates advanced research computing by granting the ability to package software in highly portable and reproducible environments encapsulating all dependencies, including the operating system. But there are still some challenges to container security.

Singularity, which is a container paradigm created by necessity for scientific and application driven workloads, addresses some core missions of containers : Mobility of Compute, Reproducibility, HPC support, and **Security**. This document intends to inform admins of different security features supported by Singularity.

5.1 Singularity Runtime

The Singularity runtime enforces a unique security model that makes it appropriate for *untrusted users* to run *untrusted containers* safely on multi-tenant resources. Because the Singularity runtime dynamically writes UID and GID information to the appropriate files within the container, the user remains the same *inside* and *outside* the container, i.e., if you're an unprivileged user while entering the container, you'll remain an unprivileged user inside the container. A privilege separation model is in place to prevent users from escalating privileges once they are inside of a container. The container file system is mounted using the `nosuid` option, and processes are spawned with the `PR_NO_NEW_PRIVS` flag. Taken together, this approach provides a secure way for users to run containers and greatly simplifies things like reading and writing data to the host system with appropriate ownership.

It is also important to note that the philosophy of Singularity is *Integration* over *Isolation*. Most container run times strive to isolate your container from the host system and other containers as much as possible. Singularity, on the other hand, assumes that the user's primary goals are portability, reproducibility, and ease of use and that isolation is often a tertiary concern. Therefore, Singularity only isolates the mount namespace by default, and will also bind mount several host directories such as `$HOME` and `/tmp` into the container at runtime. If needed, additional levels of isolation can be achieved by passing options causing Singularity to enter any or all of the other kernel namespaces and to prevent automatic bind mounting. These measures allow users to interact with the host system from within the container in sensible ways.

5.2 Singularity Image Format (SIF)

Sylabs addresses container security as a continuous process. It attempts to provide container integrity throughout the distribution pipeline.. i.e., at rest, in transit and while running. Hence, the SIF has been designed to achieve these goals.

A SIF file is an immutable container runtime image. It is a physical representation of the container environment itself. An important component of SIF that elicits security feature is the ability to cryptographically sign a container, creating a signature block within the SIF file which can guarantee immutability and provide accountability as to who signed it. Singularity follows the OpenPGP standard to create and manage these keys. After building an image within Singularity, users can `singularity sign` the container and push it to the Library along with its public PGP key(Stored in *Keystore*) which later can be verified (`singularity verify`) while pulling or downloading the image. This feature in particular protects collaboration within and between systems and teams.

5.2.1 SIF Encryption

In Singularity 3.4 and above the container root filesystem that resides in the squashFS partition of a SIF can be encrypted, rendering it's contents inaccessible without a secret. Unlike other platforms, where encrypted layers must be assembled into an unencrypted runtime directory on disk, Singularity mounts the encrypted root file system directly from the SIF using Kernel dm-crypt/LUKS functionality, so that the content is not exposed on disk. Singularity containers provide a comparable level of security to LUKS2 full disk encryption commonly deployed on Linux server and desktop systems.

As with all matters of security, a layered approach must be taken and the system as a whole considered. For example, it is possible that decrypted memory pages could be paged out the system swap file or device, which could result in decrypted information being stored at rest on physical media. Operating system level mitigations such as encrypted swap space may be required depending on the needs of your application.

Encryption and decryption of containers requires `cryptsetup` version 2. The SIF root filesystem will be encrypted using the default LUKS cipher on the host. The current default cipher used by `cryptsetup` for LUKS2 in mainstream Linux distributions is `aes-xts-plain64` with a 256 bit key size. The default key derivation function for LUKS2 is `argon2i`.

Singularity currently supports 2 types of secrets for encrypted containers:

- *Passphrase*: a text passphrase is passed directly to `cryptsetup` for LUKS encryption of the root fs.
- *Asymmetric RSA keypair*: a randomly generated 256-bit secret is used to perform LUKS encryption of the rootfs. This secret is encrypted with a user-provided RSA public key, and the ciphertext stored in the SIF file. At runtime the RSA private key must be provided to decrypt the secret and allow decryption of the root filesystem to use the container.

Note: You can verify the default LUKS2 cipher and PBKDF on your system by running `cryptsetup --help`.

`cryptsetup` sets a memory cost for the `argon2i` PBKDF based on the RAM available in the system used for encryption, up to a maximum of 1GiB. Encrypted containers created on systems with >2GiB RAM may be unusable on systems with <1GiB of free RAM.

5.3 Admin Configurable Files

Singularity Administrators have the ability to access various configuration files, that will let them set security restrictions, grant or revoke a user's capabilities, manage resources and authorize containers etc. One such file interesting in this context is ecl.toml which allows blacklisting and whitelisting of containers. You can find all the configuration files and their parameters documented here.

5.3.1 cgroups support

Starting v3.0, Singularity added native support for `cgroups`, allowing users to limit the resources their containers consume without the help of a separate program like a batch scheduling system. This feature helps in preventing DoS attacks where one container seizes control of all available system resources in order to stop other containers from operating properly. To utilize this feature, a user first creates a configuration file. An example configuration file is installed by default with Singularity to provide a guide. At runtime, the `--apply-cgroups` option is used to specify the location of the configuration file and cgroups are configured accordingly. More about cgroups support here.

5.3.2 `--security` options

Singularity supports a number of methods for specifying the security scope and context when running Singularity containers. Additionally, it supports new flags that can be passed to the action commands; `shell`, `exec`, and `run` allowing fine grained control of security. Details about them are documented here.

5.4 Security in SCS

Singularity Container Services (SCS) consist of a Remote Builder, a Container Library, and a Keystore. Taken together, the Singularity Container Services provide an end-to-end solution for packaging and distributing applications in secure and trusted containers.

5.4.1 Remote Builder

As mentioned earlier, the Singularity runtime prevents executing code with root-level permissions on the host system. But building a container requires elevated privileges that most production environments do not grant to users. The Remote Builder solves this challenge by allowing unprivileged users a service that can be used to build containers targeting one or more CPU architectures. System administrators can use the system to monitor which users are building containers, and the contents of those containers. Starting with Singularity 3.0, the CLI has native integration with the Build Service from version 3.0 onwards. In addition, a web GUI interface to the Build Service also exists, which allows users to build containers using only a web browser.

Note: Please see the *Fakeroot feature* which is a secure option for admins in multi-tenant HPC environments and similar use cases where they might want to grant a user special privileges inside a container.

5.4.2 Container Library

The Container Library enables users to store and share Singularity container images based on the Singularity Image Format (SIF). A web front-end allows users to create new projects within the Container Library, edit documentation associated with container images, and discover container images published by their peers.

5.4.3 Key Store

The Key Store is a key management system offered by Sylabs that utilizes OpenPGP implementation to facilitate sharing and maintaining of PGP public keys used to sign and verify Singularity container images. This service is based on the OpenPGP HTTP Keyserver Protocol (HKP), with several enhancements:

- The Service requires connections to be secured with Transport Layer Security (TLS).
- The Service implements token-based authentication, allowing only authenticated users to add or modify PGP keys.
- A web front-end allows users to view and search for PGP keys using a web browser.

5.4.4 Security Considerations of Cloud Services:

1. Communications between users, the auth service and the above-mentioned services are secured via TLS.
2. The services support authentication of users via authentication tokens.
3. There is no implicit trust relationship between Auth and each of these services. Rather, each request between the services is authenticated using the authentication token supplied by the user in the associated request.
4. The services support MongoDB authentication as well as TLS/SSL.

Note: SingularityPRO is a professionally curated and licensed version of Singularity that provides added security, stability, and support beyond that offered by the open source project. Subscribers receive advanced access to security patches through regular updates so, when a CVE is announced publicly PRO subscribers are already using patched software.

Security is not a check box that one can tick and forget. It's an ongoing process that begins with software architecture, and continues all the way through to ongoing security practices. In addition to ensuring that containers are run without elevated privileges where appropriate, and that containers are produced by trusted sources, users must monitor their containers for newly discovered vulnerabilities and update when necessary just as they would with any other software. Sylabs is constantly probing to find and patch vulnerabilities within Singularity, and will continue to do so.

CHAPTER

SIX

INSTALLED FILES

An installation of Singularity 3.6.4, 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/cni root:root 755 (drwxr-xr-x)
libexec/singularity/cni/portmap root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/host-local root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/firewall root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/bandwidth root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/host-device root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/loopback root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/tuning root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/bridge root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/dhcp root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/ipvlan root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/flannel root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/static root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/vlan root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/macvlan root:root 755 (-rwxr-xr-x)
libexec/singularity/cni/ptp root:root 755 (-rwxr-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)

# Singularity configuration
etc/singularity root:root 755 (drwxr-xr-x)
etc/singularity/capability.json root:root 644 (-rw-r--r--)
etc/singularity/remote.yaml root:root 644 (-rw-r--r--)
etc/singularity/network root:root 755 (drwxr-xr-x)
etc/singularity/network/20_ipvlan.conflist root:root 644 (-rw-r--r--)
```

(continues on next page)

(continued from previous page)

```
etc/singularity/network/30_macvlan.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/10_ptp.conflist root:root 644 (-rw-r--r--)
etc/singularity/ecl.toml root:root 644 (-rw-r--r--)
etc/singularity/rocmliblist.conf root:root 644 (-rw-r--r--)
etc/singularity/seccomp-profiles root:root 755 (drwxr-xr-x)
etc/singularity/seccomp-profiles/default.json root:root 644 (-rw-r--r--)
etc/singularity/nvliblist.conf root:root 644 (-rw-r--r--)
etc/singularity/singularity.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/actions root:root 755 (drwxr-xr-x)
etc/singularity/actions/exec root:root 755 (-rwxr-xr-x)
etc/singularity/actions/shell root:root 755 (-rwxr-xr-x)
etc/singularity/actions/run root:root 755 (-rwxr-xr-x)
etc/singularity/actions/start root:root 755 (-rwxr-xr-x)
etc/singularity/actions/test root:root 755 (-rwxr-xr-x)

# 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) 2018-2020, 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.
```