Singularity Container Documentation

Release 3.3

Admin Docs

May 25, 2021

CONTENTS

1 Admin Quick Start 1

1.1 Installation . . . 1

1.1.1 Install Dependencies . . . 1

1.1.2 Download and Build the RPM . . . 1

1.1.3 Setting `localstatedir` . . . 2

1.2 Configuration . . . 2

1.3 Singularity Architecture . . . 2

1.4 Singularity Security . . . 2

1.4.1 Security of the Container Runtime . . . 2

1.4.2 Security of the Container Itself . . . 3

1.4.3 Administrator Control of Users' Containers . . . 3

1.4.4 Fakeroot feature . . . 3

1.4.4.1 Basics . . . 4

1.4.4.2 Filesystem consideration . . . 5

1.4.4.3 Network consideration . . . 5

1.5 Updating Singularity . . . 5

1.6 Uninstalling Singularity . . . 5

2 Singularity Configuration Files 7

2.1 singularity.conf . . . 7

2.1.1 Limiting containers . . . 8

2.2 cgroups.toml . . . 8

2.2.1 Examples . . . 8

2.2.1.1 Limiting memory . . . 8

2.2.1.2 Limiting CPU . . . 9

2.2.1.3 Limiting IO . . . 9

2.3 ecl.toml . . . 10

2.4 nvliblist.conf . . . 11

2.5 capability.json . . . 11

2.6 seccomp-profiles . . . 12

3 Security in Singularity Containers 13

3.1 Singularity Runtime . . . 13

3.2 Singularity Image Format (SIF) . . . 14

3.3 Admin Configurable Files . . . 14

3.3.1 cgroups support . . . 14

3.3.2 `--security` options . . . 14

3.4 Security in SCS . . . 15

3.4.1 Remote Builder . . . 15

3.4.2 Container Library . . . 15

3.4.3 Key Store 15
3.4.4 Security Considerations of Cloud Services: 15

CHAPTER

ONE

ADMIN QUICK START

This document will cover installation and administration points of Singularity on a Linux host. This will also cover an overview of *configuring Singularity*, *Singularity architecture*, and *the Singularity security model*.

For any additional help or support contact the Sylabs team, or send a email to support@sylabs.io.

1.1 Installation

This section will explain how to install Singularity from an RPM. If you want more information on installation, including alternate installation procedures and options for other operating systems, see the user guide installation page.

1.1.1 Install Dependencies

Before we build the RPM, we need to install some dependencies:

```
$ sudo yum -y update && sudo yum -y install \
    wget \
    rpm-build \
    git \
    gcc \
    libuuid-devel \
    openssl-devel \
    libseccomp-devel \
    squashfs-tools \
    epel-release
$ sudo yum -y install golang
```

1.1.2 Download and Build the RPM

The Singularity tarball for building the RPM is available on the Github release page.

```
$ export VERSION=3.0.2  # this is the singularity version, change as you need

$ wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/singularity-$
↪{VERSION}.tar.gz && \
    rpmbuild -tb singularity-${VERSION}.tar.gz && \
    sudo rpm --install -vh ~/rpmbuild/RPMS/x86_64/singularity-${VERSION}-1.el7.x86_64.
↪rpm && \
    rm -rf ~/rpmbuild singularity-${VERSION}*.tar.gz
```

1.1.3 Setting `localstatedir`

The local state directories used by `singularity` at runtime will be placed under the supplied `prefix` option. This will cause issues if that directory tree is read-only or if it is shared between several hosts or nodes that might run `singularity` simultaneously.

In such cases, you should specify the `localstatedir` option. This will override the `prefix` option, instead placing the local state directories within the path explicitly provided. Ideally this should be within the local filesystem, specific to only a single host or node.

In the case of a cluster, admins must ensure that the `localstatedir` exists on all nodes with `root:root` ownership and `0755` permissions

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

1.2 Configuration

There are several ways to configuring Singularity. Head over to the *Configuration files* section where most of the conf files and setting of configuration options are discussed.

1.3 Singularity Architecture

The architecture of Singularity allows containers to be executed as if they were native programs or scripts on a host system.

As a result, integration with 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.

1.4 Singularity Security

1.4.1 Security of the Container Runtime

The Singularity security model is unique among container platforms. The bottom line? **Untrusted users** (those who don't have root access and aren't getting it) can run **untrusted containers** (those that have not been vetted by admins) **safely**. There are a few pieces of the model to consider.

First, Singularity's design forces a user to have the same UID and GID context inside and outside of the container. This is accomplished by dynamically writing entries to `/etc/passwd` and `/etc/groups` at runtime. This design makes it trivially easy for a user inside the container to safely read and write data to the host system with correct ownership, and it's also a cornerstone of the Singularity security context.

Second, Singularity mounts the container file system with the `nosuid` flag and executes processes within the container with the `PR_SET_NO_NEW_PRIVS` bit set. Combined with the fact that the user is the same inside and outside of the container, this prevents a user from escalating privileges.

Taken together, this design means your users can run whatever containers they want, and you don't have to worry about them damaging your precious system.

1.4.2 Security of the Container Itself

A malicious container may not be able to damage your system, but it could still do harm in the user's space without escalating privileges.

Starting in Singularity 3.0, containers may be cryptographically signed when they are built and verified at runtime via PGP keys. This allows a user to ensure that a container is a bit-for-bit reproduction of the container produced by the original author before they run it. As long as the user trusts the individual or company that created the container, they can run the container without worrying.

Key signing and verification is made easy using the Sylabs Keystore infrastructure. Join the party! And get more information about signing and verifying in the Singularity user guide.

1.4.3 Administrator Control of Users' Containers

Singularity provides several ways for administrators to control the specific containers that users can run.

- Admins can set directives in the `singularity.conf` file to limit container access.
  - *limit container owners*: Only allow containers to be used when they are owned by a given user (default empty)
  - *limit container groups*: Only allow containers to be used when they are owned by a given group (default empty)
  - *limit container paths*: Only allow containers to be used that are located within an allowed path prefix (default empty)
  - *allow container squashfs*: Limit usage of image containing squashfs filesystem (default yes)
  - *allow container extfs*: Limit usage of image containing ext3 filesystem (default yes)
  - *allow container dir*: Limit usage of directory image (default yes)
- Admins can also whitelist or blacklist containers through the ECL (Execution Control List) located in `ecl.toml`. This method is available in >=3.0:

  > This file describes execution groups in which SIF (default format since 3.0) images are checked for authorized loading/execution. The decision is made by validating both the location of the SIF file and by checking against a list of signing entities.

1.4.4 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 namespace UID/GID mapping.

Note: This feature requires a Linux kernel >= 3.8, but the recommended version is >= 3.18

Some distributions doesn't enable user namespace by default, so you will need to enable it to use fakeroot:

```
$ sudo sysctl -w user.max_user_namespaces=10000
```

Note: If the above command doesn't work, please refer to the documentation of your distribution documentation to figure out how to enable user namespace

For unprivileged installation of Singularity or if `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 package for `newuidmap` and `newgidmap`, so you will need to compile/install **shadow-utils** by yourself.

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

1.4.4.1 Basics

Fakeroot relies on `/etc/subuid` and `/etc/subgid` to find the use fakeroot mappings, which means that users added in those files could use the fakeroot feature, user mappings must be added in files `/etc/subuid` and `/etc/subgid`, here a valid entry for user `foo`:

For `/etc/subuid`:

```
foo:100000:65536
```

where `foo` is the username, `100000` is the start of UID range and `65536` the range count.

Same for `/etc/subgid`:

```
foo:100000:65536
```

where `foo` is the username, `100000` is the start of GID range and `65536` the range count.

Note: Some distributions already adds the main user by default in those files.

Warning: All entries with a range count different from 65536 are not considered valid by Singularity.

It's also important to ensure that the start range doesn't overlap with existing UID/GID on your 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	UID/GID range
foo	1000	100000 to 165535
bar	1001	165536 to 231071

It allows unprivileged users to change current UID/GID to any UID/GID between 0 and 65536 inside container. It also impacts files and directories ownership depending of UID/GID set in container during file/directory creation.

1.4.4.2 Filesystem consideration

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

1.4.4.3 Network consideration

With fakeroot, users can request a container network named `fakeroot`, other networks are restricted and can only be used by root user. This network is configured to use a network veth pair, it's strongly advised to not change the network type in `network/40_fakeroot.conflist` file for security reasons.

Warning: Unprivileged installation could not use `fakeroot` network as it requires privileges to setup the network.

1.5 Updating Singularity

Updating Singularity is just like installing it, but with the `--upgrade` flag instead of `--install`. Make sure you pick the latest tarball from the Github release page.

```
$ export VERSION=3.0.2  # the newest singularity version, change as you need

$ wget https://github.com/sylabs/singularity/releases/download/v${VERSION}/singularity-$
↪{VERSION}.tar.gz && \
    rpmbuild -tb singularity-${VERSION}.tar.gz && \
    sudo rpm --upgrade -vh ~/rpmbuild/RPMS/x86_64/singularity-${VERSION}-1.el7.x86_64.
↪rpm && \
    rm -rf ~/rpmbuild singularity-${VERSION}*.tar.gz
```

1.6 Uninstalling Singularity

If you install Singularity using RPM, you can uninstall it again in just a one command: (Just use `sudo`, or do this as root)

```
$ sudo rpm --erase singularity
```

CHAPTER

TWO

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:

2.1 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, it must be owned by root and must not be writable by users or Singularity will refuse to run.

The following are some of the configurable options:

`ALLOW SETUID`: To use containers, your users will have 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 (e.g. mounting of the Singularity image file format).

`USER BIND CONTROL`: This allows admins to enable/disable users to define bind points at runtime. By Default, its "YES", which means users can specify bind points, scratch and tmp locations.

`BIND PATH`: Used for setting of automatic *bind points* entries. You can define a list of files/directories that should be made available from within the container. If the file exists within the container on which to attach to use the path like:

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

You can specify different source and destination locations using:

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

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

`MOUNT HOME`: To automatically determine the calling of user's home directory and attempt to mount it's base path into the container.

2.1.1 Limiting containers

There are several ways in which you can limit the running of containers in your system:

> `LIMIT CONTAINER OWNERS`: Only allow containers to be used that are owned by a given user.
>
> `LIMIT CONTAINER GROUPS`: Only allow containers to be used that are owned by a given group.
>
> `LIMIT CONTAINER PATHS`: Only allow containers to be used that are located within an allowed path prefix.

Note: These features will only apply when Singularity is running in SUID mode and the user is non-root. By default they all are set to *NULL*.

The `singularity.conf` file is well documented and most information can be gleaned by consulting it directly.

2.2 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.

2.2.1 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
```

2.2.1.1 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/`.

2.2.1.2 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`.

2.2.1.3 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
```

2.3 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 file system 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.

2.4 nvliblist.conf

When a container includes a GPU enabled application and libraries, Singularity (with the `--nv` option) can properly inject the required Nvidia GPU driver libraries into the container, to match the host's kernel. This config file is the place where it searches for NVIDIA libraries in your host system. However, `nvliblist.conf` will be ignored in case of having nvidia-container-cli installed, which will be used to locate any nvidia libraries and binaries on the host system.

For GPU and CUDA support –nv option works like:

```
$ singularity exec --nv ubuntu.sif gpu_program.exec
$ singularity run --nv docker://tensorflow/tensorflow:gpu_latest
```

You can also mention libraries/binaries and they will be mounted into the container when the `--nv` option is passed.

2.5 capability.json

Singularity provides full support for granting and revoking Linux capabilities on a user or group basis. By default, all Linux capabilities are dropped when a user enters the container system. When you decide to add/revoke some capabilities, you can do so using the `Singularity capability` options: `Add`, `Drop` and `List`.

For example, if you do:

```
$ sudo singularity capability add --user david CAP_SYS_RAWIO
```

You've let the user David to perform I/O port operations, perform a range of device-specific operations on other devices etc. To perform the same for a group of users do:

```
$ sudo singularity capability add --group mygroup audit_write
```

Use `drop` in the same format for revoking their capabilities.

To see a list of all users and their capabilities, simply do:

```
$ sudo singularity capability list --all
```

capability.json is the file maintained by Singularity where the `capability` commands create/delete entries accordingly.

To know more about the capabilities you can add do:

```
$ singularity capability add --help
```

Note: The above commands can only be issued by root user(admin).

The –add-caps and related options will let the user request the capability when executing a container.

2.6 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.

CHAPTER

THREE

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.

3.1 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.

3.2 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.

With a new development to SIF, the root file system that resides in the squashFS partition of SIF can be encrypted, rendering it's contents inaccessible without a secret. This feature will make it necessary for users to provide a password or key file to run the container. It also ensures that no other user on the system will be able to look at your container files. Since it is all encrypted, it can defend against intruders manipulating the image while in transit.

Unlike other container platforms where the build step requires a number of layers to be read and written into another layer involving the creation of intermediate containers, Singularity executes it in a single step resulting in a `.sif` file thereby reducing the attack surface and eliminating any chances of injecting malicious content during building and running of containers.

3.3 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.

3.3.1 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.

3.3.2 `--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.

3.4 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.

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.4.4 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.