Singularity Container Documentation

Release 3.0

User Docs

Nov 08, 2018

CONTENTS

CHAPTER

ONE

QUICK START

This guide is intended for running Singularity on a computer where you have root (administrative) privileges.

If you need to request an installation on your shared resource, see the requesting an installation help page for information to send to your system administrator.

For any additional help or support contact the Sylabs team: https://www.sylabs.io/contact/

1.1 Quick Installation Steps

You will need a Linux system to run Singularity.

See the installation page for information about installing older versions of Singularity.

1.1.1 Install system dependencies

You must first install development libraries to your host. Assuming Ubuntu (apply similar to RHEL derivatives):

```
$ sudo apt-get update && sudo apt-get install -y \
    build-essential \
    libssl-dev \
    uuid-dev \
    libgpgme11-dev
```

1.1.2 Install Go

Singularity 3.0 is written primarily in Go, and you will need Go installed to compile it from source.

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

First, visit the Go download page and pick the appropriate Go archive (>=1.11.1). Copy the link address and download with `wget` like so:

```
$ export VERSION=1.11 OS=linux ARCH=amd64
$ cd /tmp
$ wget https://dl.google.com/go/go$VERSION.$OS-$ARCH.tar.gz
```

Then extract the archive to `/usr/local`

```
$ sudo tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

Finally, 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
```

1.1.3 Clone the Singularity repository

Go is a bit finicky about where things are placed. Here is the correct way to build Singularity from source.

```
$ mkdir -p $GOPATH/src/github.com/sylabs
$ cd $GOPATH/src/github.com/sylabs
$ git clone https://github.com/sylabs/singularity.git
$ cd singularity
```

1.1.4 Install Go dependencies

Dependencies are managed using Dep. You can use go get to install it like so:

```
$ go get -u -v github.com/golang/dep/cmd/dep
```

1.1.5 Compile the Singularity binary

Now you are ready to build Singularity. Dependencies will be automatically downloaded. You can build Singularity using the following commands:

```
$ cd $GOPATH/src/github.com/sylabs/singularity
$ ./mconfig
$ make -C builddir
$ sudo make -C builddir install
```

Singularity must be installed as root to function properly.

1.2 Overview of the Singularity Interface

Singularity's command line interface allows you to build and interact with containers transparently. You can run programs inside a container as if they were running on your host system. You can easily redirect IO, use pipes, pass arguments, and access files, sockets, and ports on the host system from within a container.

The `help` command gives an overview of Singularity options and subcommands as follows:

```
$ singularity help

Linux container platform optimized for High Performance Computing (HPC) and
Enterprise Performance Computing (EPC)

Usage:
  singularity [global options...]

Description:
  Singularity containers provide an application virtualization layer enabling
  mobility of compute via both application and environment portability. With
```

(continues on next page)

(continued from previous page)

```
  Singularity one is capable of building a root file system that runs on any
  other Linux system where Singularity is installed.

Options:
  -d, --debug              print debugging information (highest verbosity)
  -h, --help               help for singularity
  -q, --quiet              suppress normal output
  -s, --silent             only print errors
  -t, --tokenfile string   path to the file holding your sylabs
                           authentication token (default
                           "/home/david/.singularity/sylabs-token")
  -v, --verbose            print additional information

Available Commands:
  build       Build a new Singularity container
  capability  Manage Linux capabilities on containers
  exec        Execute a command within container
  help        Help about any command
  inspect     Display metadata for container if available
  instance    Manage containers running in the background
  keys        Manage OpenPGP key stores
  pull        Pull a container from a URI
  push        Push a container to a Library URI
  run         Launch a runscript within container
  run-help    Display help for container if available
  search      Search the library
  shell       Run a Bourne shell within container
  sign        Attach cryptographic signatures to container
  test        Run defined tests for this particular container
  verify      Verify cryptographic signatures on container
  version     Show application version

Examples:
  $ singularity help <command>
      Additional help for any Singularity subcommand can be seen by appending
      the subcommand name to the above command.


For additional help or support, please visit https://www.sylabs.io/docs/
```

Information about subcommand can also be viewed with the `help` command.

```
$ singularity help verify
Verify cryptographic signatures on container

Usage:
  singularity verify [verify options...] <image path>

Description:
  The verify command allows a user to verify cryptographic signatures on SIF
  container files. There may be multiple signatures for data objects and
  multiple data objects signed. By default the command searches for the primary
  partition signature. If found, a list of all verification blocks applied on
  the primary partition is gathered so that data integrity (hashing) and
  signature verification is done for all those blocks.

Options:
```

(continues on next page)

(continued from previous page)

```
  -g, --groupid uint32   group ID to be verified
  -h, --help             help for verify
  -i, --id uint32        descriptor ID to be verified
  -u, --url string       key server URL (default "https://keys.sylabs.io")


Examples:
  $ singularity verify container.sif


For additional help or support, please visit https://www.sylabs.io/docs/
```

Singularity uses positional syntax (i.e. the order of commands and options matters).

Global options affecting the behavior of all commands follow the main `singularity` command. Then sub commands are passed followed by their options and arguments.

For example, to pass the `--debug` option to the main `singularity` command and run Singularity with debugging messages on:

```
$ singularity --debug run library://sylabsed/examples/lolcow
```

To pass the `--containall` option to the `run` command and run a Singularity image in an isolated manner:

```
$ singularity run --containall library://sylabsed/examples/lolcow
```

Singularity 2.4 introduced the concept of command groups. For instance, to list Linux capabilities for a particular user, you would use the `list` command in the `capabilities` command group like so:

```
$ singularity capability list --user dave
```

Container authors might also write help docs specific to a container or for an internal module called an `app`. If those help docs exist for a particular container, you can view them like so.

```
$ singularity help container.sif  # See the container's help, if provided

$ singularity help --app foo container.sif  # See the help for foo, if provided
```

1.3 Download pre-built images

You can use the `search` command to locate groups, collections, and containers of interest on the Container Library .

```
$ singularity search alp
No users found for 'alp'

Found 1 collections for 'alp'
    library://jchavez/alpine

Found 5 containers for 'alp'
    library://jialipassion/official/alpine
            Tags: latest
    library://dtrudg/linux/alpine
            Tags: 3.2 3.3 3.4 3.5 3.6 3.7 3.8 edge latest
    library://sylabsed/linux/alpine
```

(continues on next page)

(continued from previous page)

```
            Tags: 3.6 3.7 latest
    library://library/default/alpine
            Tags: 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 latest
    library://sylabsed/examples/alpine
            Tags: latest
```

You can use the pull and build commands to download pre-built images from an external resource like the Container Library or Docker Hub.

When called on a native Singularity image like those provided on the Container Library, `pull` simply downloads the image file to your system.

```
$ singularity pull library://sylabsed/linux/alpine
```

You can also use `pull` with the `docker://` uri to reference Docker images served from a registry. In this case `pull` does not just download an image file. Docker images are stored in layers, so `pull` must also combine those layers into a usable Singularity file.

```
$ singularity pull docker://godlovedc/lolcow
```

Pulling Docker images reduces reproducibility. If you were to pull a Docker image today and then wait six months and pull again, you are not guaranteed to get the same image. If any of the source layers has changed the image will be altered. If reproducibility is a priority for you, try building your images from the Container Library.

You can also use the `build` command to download pre-built images from an external resource. When using `build` you must specify a name for your container like so:

```
$ singularity build ubuntu.sif library://ubuntu

$ singularity build lolcow.sif docker://godlovedc/lolcow
```

Unlike `pull`, `build` will convert your image to the latest Singularity image format after downloading it.

`build` is like a "Swiss Army knife" for container creation. In addition to downloading images, you can use `build` to create images from other images or from scratch using a definition file. You can also use `build` to convert an image between the container formats supported by Singularity.

1.4 Interact with images

You can interact with images in several ways. It is not actually necessary to `pull` or `build` an image to interact with it. The commands listed here will work with image URIs in addition to accepting a local path to an image.

For these examples we will use a `lolcow_latest.sif` image that can be pulled from the Container Library like so.

```
$ singularity pull library://sylabsed/examples/lolcow
```

1.4.1 Shell

The shell command allows you to spawn a new shell within your container and interact with it as though it were a small virtual machine.

```
$ singularity shell lolcow_latest.sif

Singularity lolcow_latest.sif:~>
```

The change in prompt indicates that you have entered the container (though you should not rely on that to determine whether you are in container or not).

Once inside of a Singularity container, you are the same user as you are on the host system.

```
Singularity lolcow_latest.sif:~> whoami
david

Singularity lolcow_latest.sif:~> id
uid=1000(david) gid=1000(david) groups=1000(david),4(adm),24(cdrom),27(sudo),30(dip),
↪46(plugdev),116(lpadmin),126(sambashare)
```

`shell` also works with the `library://`, `docker://`, and `shub://` URIs. This creates an ephemeral container that disappears when the shell is exited.

```
$ singularity shell library://sylabsed/examples/lolcow
```

1.4.2 Executing Commands

The exec command allows you to execute a custom command within a container by specifying the image file. For instance, to execute the `cowsay` program within the `lolcow_latest.sif` container:

```
$ singularity exec lolcow_latest.sif cowsay moo
 _____
< moo >
 -----
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

`exec` also works with the `library://`, `docker://`, and `shub://` URIs. This creates an ephemeral container that executes a command and disappears.

```
$ singularity exec library://sylabsed/examples/lolcow cowsay "Fresh from the library!"
 _________________________
< Fresh from the library! >
 -------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

1.4.3 Running a container

Singularity containers contain runscripts. These are user defined scripts that define the actions a container should perform when someone runs it. The runscript can be triggered with the run command, or simply by calling the container as though it were an executable.

```
$ singularity run lolcow_latest.sif
 _______________________________________
/ You have been selected for a secret \
\ mission.                            /
 ---------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

$ ./lolcow_latest.sif
 ______________________________________
/ Q: What is orange and goes "click, \
\ click?" A: A ball point carrot.    /
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

`run` also works with the `library://`, `docker://`, and `shub://` URIs. This creates an ephemeral container that runs and then disappears.

```
$ singularity run library://sylabsed/examples/lolcow
 _____________________________________
/ Is that really YOU that is reading \
\ this?                              /
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

1.4.4 Working with Files

Files on the host are reachable from within the container.

```
$ echo "Hello from inside the container" > $HOME/hostfile.txt

$ singularity exec lolcow_latest.sif cat $HOME/hostfile.txt

Hello from inside the container
```

This example works because `hostfile.txt` exists in the user's home directory. By default Singularity bind mounts `/home/$USER`, `/tmp`, and `$PWD` into your container at runtime.

You can specify additional directories to bind mount into your container with the `--bind` option. In this example, the `data` directory on the host system is bind mounted to the `/mnt` directory inside the container.

```
$ echo "Drink milk (and never eat hamburgers)." > /data/cow_advice.txt

$ singularity exec --bind /data:/mnt lolcow_latest.sif cat /mnt/cow_advice.txt
Drink milk (and never eat hamburgers).
```

Pipes and redirects also work with Singularity commands just like they do with normal Linux commands.

```
$ cat /data/cow_advice.txt | singularity exec lolcow_latest.sif cowsay
 ________________________________________
< Drink milk (and never eat hamburgers). >
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

1.5 Build images from scratch

Singularity v3.0 produces immutable images in the Singularity Image File (SIF) format. This ensures reproducible and verifiable images and allows for many extra benefits such as the ability to sign and verify your containers.

However, during testing and debugging you may want an image format that is writable. This way you can `shell` into the image and install software and dependencies until you are satisfied that your container will fulfill your needs. For these scenarios, Singularity also supports the `sandbox` format (which is really just a directory).

For more details about the different build options and best practices, read about the Singularity flow.

1.5.1 Sandbox Directories

To build into a `sandbox` (container in a directory) use the `build --sandbox` command and option:

```
$ sudo singularity build --sandbox ubuntu/ library://ubuntu
```

This command creates a directory called `ubuntu/` with an entire Ubuntu Operating System and some Singularity metadata in your current working directory.

You can use commands like `shell`, `exec` , and `run` with this directory just as you would with a Singularity image. If you pass the `--writable` option when you use your container you can also write files within the sandbox directory (provided you have the permissions to do so).

```
$ sudo singularity exec --writable ubuntu touch /foo

$ singularity exec ubuntu/ ls /foo
/foo
```

1.5.2 Converting images from one format to another

The `build` command allows you to build a container from an existing container. This means that you can use it to convert a container from one format to another. For instance, if you have already created a sandbox (directory) and want to convert it to the default immutable image format (squashfs) you can do so:

```
$ singularity build new-sif sandbox
```

Doing so may break reproducibility if you have altered your sandbox outside of the context of a definition file, so you are advised to exercise care.

1.5.3 Singularity Definition Files

For a reproducible, production-quality container you should build a SIF file using a Singularity definition file. This also makes it easy to add files, environment variables, and install custom software, and still start from your base of choice (e.g., the Container Library).

A definition file has a header and a body. The header determines the base container to begin with, and the body is further divided into sections that do things like install software, setup the environment, and copy files into the container from the host system.

Here is an example of a definition file:

```
BootStrap: library
From: ubuntu:16.04

%post
    apt-get -y update
    apt-get -y install fortune cowsay lolcat

%environment
    export LC_ALL=C
    export PATH=/usr/games:$PATH

%runscript
    fortune | cowsay | lolcat

%labels
    Author GodloveD
```

To build a container from this definition file (assuming it is a file named lolcow.def), you would call build like so:

```
$ sudo singularity build lolcow.sif lolcow.def
```

In this example, the header tells Singularity to use a base Ubuntu 16.04 image from the Container Library.

The `%post` section executes within the container at build time after the base OS has been installed. The `%post` section is therefore the place to perform installations of new applications.

The `%environment` section defines some environment variables that will be available to the container at runtime.

The `%runscript` section defines actions for the container to take when it is executed.

And finally, the `%labels` section allows for custom metadata to be added to the container.

This is a very small example of the things that you can do with a definition file. In addition to building a container from the Container Library, you can start with base images from Docker Hub and use images directly from official repositories such as Ubuntu, Debian, CentOS, Arch, and BusyBox. You can also use an existing container on your host system as a base.

If you want to build Singularity images but you don't have administrative (root) access on your build system, you can build images using the Remote Builder.

This quickstart document just scratches the surface of all of the things you can do with Singularity!

If you need additional help or support, contact the Sylabs team: https://www.sylabs.io/contact/

CHAPTER

TWO

BUILD A CONTAINER

`build` is the "Swiss army knife" of container creation. You can use it to download and assemble existing containers from external resources like the Container Library and Docker Hub. You can use it to convert containers between the formats supported by Singularity. And you can use it in conjunction with a Singularity definition file to create a container from scratch and customized it to fit your needs.

2.1 Overview

The `build` command accepts a target as input and produces a container as output.

The target defines the method that `build` uses to create the container. It can be one of the following:

- URI beginning with **library://** to build from the Container Library
- URI beginning with **docker://** to build from Docker Hub
- URI beginning with **shub://** to build from Singularity Hub
- path to a **existing container** on your local machine
- path to a **directory** to build from a sandbox
- path to a Singularity definition file

`build` can produce containers in two different formats that can be specified as follows.

- compressed read-only **Singularity Image File (SIF)** format suitable for production (default)
- writable **(ch)root directory** called a sandbox for interactive development (`--sandbox` option)

Because `build` can accept an existing container as a target and create a container in either supported format you can convert existing containers from one format to another.

2.2 Downloading an existing container from the Container Library

You can use the build command to download a container from the Container Library.

```
$ sudo singularity build lolcow.simg library://sylabs-jms/testing/lolcow
```

The first argument (`lolcow.simg`) specifies a path and name for your container. The second argument (`library://sylabs-jms/testing/lolcow`) gives the Container Library URI from which to download. By default the container will be converted to a compressed, read-only SIF. If you want your container in a writable format use the `--sandbox` option.

2.3 Downloading an existing container from Docker Hub

You can use `build` to download layers from Docker Hub and assemble them into Singularity containers.

```
$ sudo singularity build lolcow.sif docker://godlovedc/lolcow
```

2.4 Creating writable `--sandbox` directories

If you wanted to create a container within a writable directory (called a sandbox) you can do so with the `--sandbox` option. It's possible to create a sandbox without root privileges, but to ensure proper file permissions it is recommended to do so as root.

```
$ sudo singularity build --sandbox lolcow/ library://sylabs-jms/testing/lolcow
```

The resulting directory operates just like a container in a SIF file. To make changes within the container, use the `--writable` flag when you invoke your container. It's a good idea to do this as root to ensure you have permission to access the files and directories that you want to change.

```
$ sudo singularity shell --writable lolcow/
```

2.5 Converting containers from one format to another

If you already have a container saved locally, you can use it as a target to build a new container. This allows you convert containers from one format to another. For example if you had a sandbox container called `development/` and you wanted to convert it to SIF container called `production.sif` you could:

```
$ sudo singularity build production.sif development/
```

Use care when converting a sandbox directory to the default SIF format. If changes were made to the writable container before conversion, there is no record of those changes in the Singularity definition file rendering your container non-reproducible. It is a best practice to build your immutable production containers directly from a Singularity definition file instead.

2.6 Building containers from Singularity definition files

Of course, Singularity definition files can be used as the target when building a container. For detailed information on writing Singularity definition files, please see the Container Definition docs. Let's say you already have the following container definition file called `lolcow.def`, and you want to use it to build a SIF container.

```
Bootstrap: docker
From: ubuntu:16.04

%post
    apt-get -y update
    apt-get -y install fortune cowsay lolcat

%environment
    export LC_ALL=C
    export PATH=/usr/games:$PATH
```

(continues on next page)

(continued from previous page)

```
%runscript
    fortune | cowsay | lolcat
```

You can do so with the following command.

```
$ sudo singularity build lolcow.sif lolcow.def
```

The command requires `sudo` just as installing software on your local machine requires root privileges.

2.7 Build options

2.7.1 --builder

Singularity 3.0 introduces the option to perform a remote build. The `--builder` option allows you to specify a URL to a different build service. For instance, you may need to specify a URL to build to an on premises installation of the remote builder. This option must be used in conjunction with `--remote`.

2.7.2 --detached

When used in combination with the `--remote` option, the `--detached` option will detach the build from your terminal and allow it to build in the background without echoing any output to your terminal.

2.7.3 --force

The `--force` option will delete and overwrite an existing Singularity image without presenting the normal interactive prompt.

2.7.4 --json

The `--json` option will force Singularity to interpret a given definition file as a json.

2.7.5 --library

This command allows you to set a different library. (The default library is "https://library.sylabs.io")

2.7.6 --notest

If you don't want to run the `%test` section during the container build, you can skip it with the `--notest` option. For instance, maybe you are building a container intended to run in a production environment with GPUs. But perhaps your local build resource does not have GPUs. You want to include a `%test` section that runs a short validation but you don't want your build to exit with an error because it cannot find a GPU on your system.

2.7.7 --remote

Singularity 3.0 introduces the ability to build a container on an external resource running a remote builder. (The default remote builder is located at "https://build.sylabs.io".)

2.7.8 --sandbox

Build a sandbox (chroot directory) instead of the default SIF format.

2.7.9 --section

Instead of running the entire definition file, only run a specific section or sections. This option accepts a comma delimited string of definition file sections. Acceptable arguments include `all`, `none` or any combination of the following: `setup`, `post`, `files`, `environment`, `test`, `labels`.

Under normal build conditions, the Singularity definition file is saved into a container's meta-data so that there is a record showing how the container was built. Using the `--section` option may render this meta-data useless, so use care if you value reproducibility.

2.7.10 --update

You can build into the same sandbox container multiple times (though the results may be unpredictable and it is generally better to delete your container and start from scratch).

By default if you build into an existing sandbox container, the `build` command will prompt you to decide whether or not to overwrite the container. Instead of this behavior you can use the `--update` option to build _into_ an existing container. This will cause Singularity to skip the header and build any sections that are in the definition file into the existing container.

The `--update` option is only valid when used with sandbox containers.

2.8 More Build topics

- If you want to **customize the cache location** (where Docker layers are downloaded on your system), specify Docker credentials, or any custom tweaks to your build environment, see build environment.
- If you want to make internally **modular containers**, check out the getting started guide here
- If you want to **build your containers** on the Remote Builder, (because you don't have root access on a Linux machine or want to host your container on the cloud) check out this site

CHAPTER

THREE

BIND PATHS AND MOUNTS

If enabled by the system administrator, Singularity allows you to map directories on your host system to directories within your container using bind mounts. This allows you to read and write data on the host system with ease.

3.1 Overview

When Singularity 'swaps' the host operating system for the one inside your container, the host file systems becomes inaccessible. But you may want to read and write files on the host system from within the container. To enable this functionality, Singularity will bind directories back into the container via two primary methods: system-defined bind paths and user-defined bind paths.

3.2 System-defined bind paths

The system administrator has the ability to define what bind paths will be included automatically inside each container. Some bind paths are automatically derived (e.g. a user's home directory) and some are statically defined (e.g. bind paths in the Singularity configuration file). In the default configuration, the directories `$HOME` , `/tmp` , `/proc` , `/sys` , `/dev`, and `$PWD` are among the system-defined bind paths.

3.3 User-defined bind paths

If the system administrator has enabled user control of binds, you will be able to request your own bind paths within your container.

The Singularity action commands (`run`, `exec` ,''shell'', and `instance start` will accept the `--bind/-B` command-line option to specify bind paths, and will also honor the `$SINGULARITY_BIND` (or `$SINGULARITY_BINDPATH`) environment variable. The argument for this option is a comma-delimited string of bind path specifications in the format `src[:dest[:opts]]`, where `src` and `dest` are paths outside and inside of the container respectively. If `dest` is not given, it is set equal to `src`. Mount options (`opts`) may be specified as `ro` (read-only) or `rw` (read/write, which is the default). The `--bind/-B` option can be specified multiple times, or a comma-delimited string of bind path specifications can be used.

3.3.1 Specifying bind paths

Here's an example of using the `--bind` option and binding `/data` on the host to `/mnt` in the container (`/mnt` does not need to already exist in the container):

```
$ ls /data
bar  foo

$ singularity exec --bind /data:/mnt my_container.sif ls /mnt
bar  foo
```

You can bind multiple directories in a single command with this syntax:

```
$ singularity shell --bind /opt,/data:/mnt my_container.sif
```

This will bind `/opt` on the host to `/opt` in the container and `/data` on the host to `/mnt` in the container.

Using the environment variable instead of the command line argument, this would be:

```
$ export SINGULARITY_BIND="/opt,/data:/mnt"

$ singularity shell my_container.sif
```

Using the environment variable `$SINGULARITY_BIND`, you can bind paths even when you are running your container as an executable file with a runscript. If you bind many directories into your Singularity containers and they don't change, you could even benefit by setting this variable in your `.bashrc` file.

3.3.2 A note on using `--bind` with the `--writable` flag

To mount a bind path inside the container, a *bind point* must be defined within the container. The bind point is a directory within the container that Singularity can use as a destination to bind a directory on the host system.

Starting in version 3.0, Singularity will do its best to bind mount requested paths into a container regardless of whether the appropriate bind point exists within the container. Singularity can often carry out this operation even in the absence of the "overlay fs" feature.

However, binding paths to non-existent points within the container can result in unexpected behavior when used in conjuction with the `--writable` flag, and is therefore disallowed. If you need to specify bind paths in combination with the `--writable` flag, please ensure that the appropriate bind points exist within the container. If they do not already exist, it will be necessary to modify the container and create them.

CHAPTER

FOUR

PERSISTENT OVERLAYS

Persistent overlay directories allow you to overlay a writable file system on an immutable read-only container for the illusion of read-write access.

4.1 Overview

A persistent overlay is a directory that "sits on top" of your compressed, immutable SIF container. When you install new software or create and modify files the overlay directory stores the changes.

If you want to use a SIF container as though it were writable, you can create a directory to use as a persistent overlay. Then you can specify that you want to use the directory as an overlay at runtime with the `--overlay` option.

You can use a persistent overlays with the following commands:

- `run`
- `exec`
- `shell`
- `instance.start`

4.2 Usage

To use a persistent overlay, you must first have a container.

```
$ sudo singularity build ubuntu.sif library://ubuntu
```

Then you must create a directory. (You can also use the `--overlay` option with a legacy writable ext3 image.)

```
$ mkdir my_overlay
```

Now you can use this overlay directory with your container. Note that it is necessary to be root to use an overlay directory.

```
$ sudo singularity shell --overlay my_overlay/ ubuntu.sif

Singularity ubuntu.sif:~> touch /foo

Singularity ubuntu.sif:~> apt-get update && apt-get install -y vim

Singularity ubuntu.sif:~> which vim
```

(continues on next page)

(continued from previous page)

```
/usr/bin/vim

Singularity ubuntu.sif:~> exit
```

You will find that your changes persist across sessions as though you were using a writable container.

```
$ sudo singularity shell --overlay my_overlay/ ubuntu.sif

Singularity ubuntu.sif:~> ls /foo
/foo

Singularity ubuntu.sif:~> which vim
/usr/bin/vim

Singularity ubuntu.sif:~> exit
```

If you mount your container without the `--overlay` directory, your changes will be gone.

```
$ sudo singularity shell ubuntu.sif

Singularity ubuntu.sif:~> ls /foo
ls: cannot access 'foo': No such file or directory

Singularity ubuntu.sif:~> which vim

Singularity ubuntu.sif:~> exit
```

CHAPTER

FIVE

SIGNING AND VERIFYING CONTAINERS

Singularity 3.0 introduces the abilities to create and manage PGP keys and use them to sign and verify containers. This provides a trusted method for Singularity users to share containers. It ensures a bit-for-bit reproduction of the original container as the author intended it.

5.1 Verifying containers from the Container Library

The `verify` command will allow you to verify that a container has been signed using a PGP key. To use this feature with images that you pull from the container library, you must first generate an access token to the Sylabs Cloud. If you don't already have a valid access token, follow these steps:

1. Go to : https://cloud.sylabs.io/
2. Click "Sign in to Sylabs" and follow the sign in steps.
3. Click on your login id (same and updated button as the Sign in one).
4. Select "Access Tokens" from the drop down menu.
5. Click the "Manage my API tokens" button from the "Account Management" page.
6. Click "Create".
7. Click "Copy token to Clipboard" from the "New API Token" page.
8. Paste the token string into your `~/.singularity/sylabs-token` file.

Now you can verify containers that you pull from the library, ensuring they are bit-for-bit reproductions of the original image.

```
$ singularity pull library://alpine

$ singularity verify alpine_latest.sif
Verifying image: alpine_latest.sif
Data integrity checked, authentic and signed by:
    Sylabs Admin <support@sylabs.io>, KeyID 51BE5020C508C7E9
```

In this example you can see that **Sylabs Admin** has signed the container.

5.2 Signing your own containers

5.2.1 Generating and managing PGP keys

To sign your own containers you first need to generate one or more keys.

If you attempt to sign a container before you have generated any keys, Singularity will guide you through the interactive process of creating a new key. Or you can use the `newpair` subcommand in the `key` command group like so:.

```
$ singularity keys newpair
Enter your name (e.g., John Doe) : Dave Godlove
Enter your email address (e.g., john.doe@example.com) : d@sylabs.io
Enter optional comment (e.g., development keys) : demo
Generating Entity and OpenPGP Key Pair... Done
Enter encryption passphrase :
```

The `list` subcommand will show you all of the keys you have created or saved locally.‘

```
$ singularity keys list
Public key listing (/home/david/.singularity/sypgp/pgp-public):

0) U: Dave Godlove (demo) <d@sylabs.io>
   C: 2018-10-08 15:25:30 -0400 EDT
   F: 135E426D67D8416DE1D6AC7FFED5BBA38EE0DC4A
   L: 4096
   --------
```

In the output above, the letters stand for the following:

- U: User
- C: Creation date and time
- F: Fingerprint
- L: Key length

After generating your key you can optionally push it to the Keystore using the fingerprint like so:

```
$ singularity keys push 135E426D67D8416DE1D6AC7FFED5BBA38EE0DC4A
public key `135E426D67D8416DE1D6AC7FFED5BBA38EE0DC4A` pushed to server successfully
```

This will allow others to verify images that you have signed.

If you delete your local public PGP key, you can always locate and download it again like so.

```
$ singularity keys search Godlove
Search results for 'Godlove'

Type bits/keyID     Date       User ID
--------------------------------------------------------------------------------
pub  4096R/8EE0DC4A 2018-10-08 Dave Godlove (demo) <d@sylabs.io>
--------------------------------------------------------------------------------

$ singularity keys pull 8EE0DC4A
1 key(s) fetched and stored in local cache /home/david/.singularity/sypgp/pgp-public
```

But note that this only restores the *public* key (used for verifying) to your local machine and does not restore the *private* key (used for signing).

5.2.2 Signing and validating your own containers

Now that you have a key generated, you can use it to sign images like so:

```
$ singularity sign my_container.sif
Signing image: my_container.sif
Enter key passphrase:
Signature created and applied to my_container.sif
```

Because your public PGP key is saved locally you can verify the image without needing to contact the Keystore.

```
$ singularity verify my_container.sif
Verifying image: my_container.sif
Data integrity checked, authentic and signed by:
    Dave Godlove (demo) <d@sylabs.io>, KeyID FED5BBA38EE0DC4A
```

If you've pushed your key to the Keystore you can also verify this image in the absence of a local key. To demonstrate this, first delete your local keys, and then try to use the `verify` command again.

```
$ rm ~/.singularity/sypgp/*

$ singularity verify my_container.sif
Verifying image: my_container.sif
INFO:    key missing, searching key server for KeyID: FED5BBA38EE0DC4A...
INFO:    key retreived successfully!
Store new public key 135E426D67D8416DE1D6AC7FFED5BBA38EE0DC4A? [Y/n] y
Data integrity checked, authentic and signed by:
    Dave Godlove (demo) <d@sylabs.io>, KeyID FED5BBA38EE0DC4A
```

Answering yes at the interactive prompt will store the Public key locally so you will not have to contact the Keystore again the next time you verify your container.

CHAPTER

SIX

SECURITY OPTIONS

Singularity 3.0 introduces many new security related options to the container runtime. This document will describe the new methods users have for specifying the security scope and context when running Singularity containers.

6.1 Linux Capabilities

Singularity provides full support for granting and revoking Linux capabilities on a user or group basis. For example, let us suppose that an admin has decided to grant a user capabilities to open raw sockets so that they can use `ping` in a container where the binary is controlled via capabilities (i.e. a recent version of CentOS).

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

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

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

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

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

To take advantage of this new capability, the user `david` must also request the capability when executing a container with the `--add-caps` flag like so:

```
$ singularity exec --add-caps CAP_NET_RAW library://centos 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=128 time=18.3 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 18.320/18.320/18.320/0.000 ms
```

If the admin decides that it is no longer necessary to allow the user `dave` to open raw sockets within Singularity containers, they can revoke the appropriate Linux capability like so:

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

The `capabiltiy add` and `drop` subcommands will also accept the case insensitive keyword `all` to grant or revoke all Linux capabilities to a user or group. Similarly, the `--add-caps` option will accept the `all` keyword. Of course appropriate caution should be exercised when using this keyword.

6.2 Security related action options

Singularity 3.0 introduces many new flags that can be passed to the action commands; `shell`, `exec`, and `run` allowing fine grained control of security.

6.2.1 --add-caps

As explained above, `--add-caps` will "activate" Linux capabilities when a container is initiated, providing those capabilities have been granted to the user by an administrator using the `capability add` command. This option will also accept the case insensitive keyword `all` to add every capability granted by the administrator.

6.2.2 --allow-setuid

The SetUID bit allows a program to be executed as the user that owns the binary. The most well-known SetUID binaries are owned by root and allow a user to execute a command with elevated privileges. But other SetUID binaries may allow a user to execute a command as a service account.

By default SetUID is disallowed within Singularity containers as a security precaution. But the root user can override this precaution and allow SetUID binaries to behave as expected within a Singularity container with the `--allow-setuid` option like so:

```
$ sudo singularity shell --allow-setuid some_container.sif
```

6.2.3 --keep-privs

It is possible for an admin to set a different set of default capabilities or to reduce the default capabilities to zero for the root user by setting the `root default capabilities` parameter in the `singularity.conf` file to `file` or `no` respectively. If this change is in effect, the root user can override the `singularity.conf` file and enter the container with full capabilities using the `--keep-privs` option.

```
$ sudo singularity exec --keep-privs library://centos 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=128 time=18.8 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 18.838/18.838/18.838/0.000 ms
```

6.2.4 --drop-caps

By default, the root user has a full set of capabilities when they enter the container. You may choose to drop specific capabilities when you initiate a container as root to enhance security.

For instance, to drop the ability for the root user to open a raw socket inside the container:

```
$ sudo singularity exec --drop-caps CAP_NET_RAW library://centos ping -c 1 8.8.8.8
ping: socket: Operation not permitted
```

The `drop-caps` option will also accept the case insensitive keyword `all` as an option to drop all capabilities when entering the container.

6.2.5 --security

The --security flag allows the root user to leverage security modules such as SELinux, AppArmor, and seccomp within your Singularity container. You can also change the UID and GID of the user within the container at runtime.

For instance:

```
$ sudo whoami
root

$ sudo singularity exec --security uid:1000 my_container.sif whoami
david
```

To use seccomp to blacklist a command follow this procedure. (It is actually preferable from a security standpoint to whitelist commands but this will suffice for a simple example.) Note that this example was run on Ubuntu and that Singularity was installed with the libseccomp-dev and pkg-config packages as dependencies.

First write a configuration file. An example configuration file is installed with Singularity, normally at /usr/local/etc/singularity/seccomp-profiles/default.json. For this example, we will use a much simpler configuration file to blacklist the mkdir command.

```
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "archMap": [
        {
            "architecture": "SCMP_ARCH_X86_64",
            "subArchitectures": [
                "SCMP_ARCH_X86",
                "SCMP_ARCH_X32"
            ]
        }
    ],
    "syscalls": [
        {
            "names": [
                "mkdir"
            ],
            "action": "SCMP_ACT_KILL",
            "args": [],
            "comment": "",
            "includes": {},
            "excludes": {}
        }
    ]
}
```

We'll save the file at /home/david/no_mkdir.json. Then we can invoke the container like so:

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

Singularity> mkdir /tmp/foo
Bad system call (core dumped)
```

Note that attempting to use the blacklisted mkdir command resulted in a core dump.

The full list of arguments accepted by the --security option are as follows:

```
--security="seccomp:/usr/local/etc/singularity/seccomp-profiles/default.json"
--security="apparmor:/usr/bin/man"
--security="selinux:context"
--security="uid:1000"
--security="gid:1000"
--security="gid:1000:1:0" (multiple gids, first is always the primary group)
```

CHAPTER

SEVEN

NETWORK VIRTUALIZATION

Singularity 3.0 introduces full integration with cni , and several new features to make network virtualization easy.

A few new options have been added to the action commands (`exec`, `run`, and `shell`) to facilitate these features, and the `--net` option has been updated as well. These options can only be used by root.

7.1 --dns

The `--dns` option allows you to specify a comma separated list of DNS servers to add to the `/etc/resolv.conf` file.

```
$ nslookup sylabs.io | grep Server
Server:             127.0.0.53

$ sudo singularity exec --dns 8.8.8.8 ubuntu.sif nslookup sylabs.io | grep Server
Server:             8.8.8.8

$ sudo singularity exec --dns 8.8.8.8 ubuntu.sif cat /etc/resolv.conf
nameserver 8.8.8.8
```

7.2 --hostname

The `--hostname` option accepts a string argument to change the hostname within the container.

```
$ hostname
ubuntu-bionic

$ sudo singularity exec --hostname hal-9000 my_container.sif hostname
hal-9000
```

7.3 --net

Passing the `--net` flag will cause the container to join a new network namespace when it initiates. New in Singularity 3.0, a bridge interface will also be set up by default.

```
$ hostname -I
10.0.2.15
```

(continues on next page)

(continued from previous page)

```
$ sudo singularity exec --net my_container.sif hostname -I
10.22.0.4
```

7.4 --network

The `--network` option can only be invoked in combination with the `--net` flag. It accepts a comma delimited string of network types. Each entry will bring up a dedicated interface inside container.

```
$ hostname -I
172.16.107.251 10.22.0.1

$ sudo singularity exec --net --network ptp ubuntu.sif hostname -I
10.23.0.6

$ sudo singularity exec --net --network bridge,ptp ubuntu.sif hostname -I
10.22.0.14 10.23.0.7
```

When invoked, the `--network` option searches the singularity configuration directory (commonly `/usr/local/etc/singularity/network/`) for the cni configuration file corresponding to the requested network type(s). Several configuration files are installed with Singularity by default corresponding to the following network types:

- bridge
- ptp
- ipvlan
- macvlan

Administrators can also define custom network configurations and place them in the same directory for the benefit of users.

7.5 --network-args

The `--network-args` option provides a convenient way to specify arguments to pass directly to the cni plugins. It must be used in conjuction with the `--net` flag.

For instance, let's say you want to start an NGINX server on port 80 inside of the container, but you want to map it to port 8080 outside of the container:

```
$ sudo singularity instance start --writable-tmpfs \
    --net --network-args "portmap=8080:80/tcp" docker://nginx web2
```

The above command will start the Docker Hub official NGINX image running in a background instance called `web2`. The NGINX instance will need to be able to write to disk, so we've used the `--writable-tmpfs` argument to allocate some space in memory. The `--net` flag is necessary when using the `--network-args` option, and specifying the `portmap=8080:80/tcp` argument which will map port 80 inside of the container to 8080 on the host.

Now we can start NGINX inside of the container:

```
$ sudo singularity exec instance://web2 nginx
```

And the `curl` command can be used to verify that NGINX is running on the host port 8080 as expected.

```
$ curl localhost:8080
10.22.0.1 - - [16/Oct/2018:09:34:25 -0400] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0"
↪"-"
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

For more information about cni, check the cni specification.

CHAPTER

EIGHT

LIMITING CONTAINER RESOURCES WITH CGROUPS

Starting in Singularity 3.0, users have the ability to limit container resources using cgroups.

8.1 Overview

Singularity cgroups support can be configured and utilized via a TOML file. An example file is typically installed at `/usr/local/etc/singularity/cgroups/cgroups.toml`. You can copy and edit this file to suit your needs. Then when you need to limit your container 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
```

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

8.2 Examples

8.2.1 Limiting memory

To limit the amount of memory that your container uses to 500MB (524288000 bytes), follow this example. First, create a `cgroups.toml` file like this and save it in your home directory.

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

Start your container like so:

```
$ sudo singularity instance start --apply-cgroups /home/$USER/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
```

After you are finished with this example, be sure to cleanup your instance with the following command.

```
$ sudo singularity instance stop instance1
```

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

8.2.2 Limiting CPU

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

8.2.2.1 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.

8.2.2.2 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
```

8.2.2.3 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`.

For more information about limiting CPU with cgroups, see the following external links:

- Red Hat resource management guide section 3.2 CPU
- Red Hat resource management guide section 3.4 CPUSET
- Kernel scheduler documentation

8.2.3 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
```

To limit the IO read/write rate to 1000 IO per second (IOPS) on `/dev/loop0` block device, you can do the following. The rate is specified in IOPS.

```
[blockIO]
    [[blockIO.throttleReadIOPSDevice]]
        major = 7
        minor = 0
        rate = 1000
    [[blockIO.throttleWriteIOPSDevice]]
        major = 7
        minor = 0
        rate = 1000
```

For more information about limiting IO, see the following external links:

- Red Hat resource management guide section 3.1 blkio
- Kernel block IO controller documentation
- Kernel CFQ scheduler documentation

8.2.3.1 Limiting device access

You can limit read, write, or creation of devices. In this example, a container is configured to only be able to read from or write to `/dev/null`.

```
[[devices]]
    access = "rwm"
    allow = false
[[devices]]
    access = "rw"
    allow = true
    major = 1
    minor = 3
    type = "c"
```

For more information on limiting access to devices the Red Hat resource management guide section 3.5 DEVICES.