Singularity Container Documentation

Release 2.5.2

Admin Docs

Jul 16, 2018

CONTENTS

CHAPTER

ONE

ADMINISTRATION QUICKSTART

This document will cover installation and administration points of Singularity for multi-tenant HPC resources and will not cover usage of the command line tools, container usage, or example use cases.

1.1 Installation

There are two common ways to install Singularity, from source code and via binary packages. This document will explain the process of installation from source, and it will depend on your build host to have the appropriate development tools and packages installed. For Red Hat and derivatives, you should install the following `yum` group to ensure you have an appropriately setup build server:

```
$ sudo yum groupinstall "Development Tools"
```

1.1.1 Downloading the source

You can download the source code either from the latest stable tarball release or via the GitHub master repository. Here is an example downloading and preparing the latest development code from GitHub:

```
$ mkdir ~/git

$ cd ~/git

$ git clone https://github.com/singularityware/singularity.git

$ cd singularity

$ ./autogen.sh
```

Once you have downloaded the source, the following installation procedures will assume you are running from the root of the source directory.

1.1.2 Source Installation

The following example demonstrates how to install Singularity into `/usr/local`. You can install Singularity into any directory of your choosing, but you must ensure that the location you select supports programs running as `SUID`. It is common for people to disable `SUID` with the mount option `nosuid` for various network mounted file systems. To ensure proper support, it is easiest to make sure you install Singularity to a local file system.

Assuming that `/usr/local` is a local file system:

```
$ ./configure --prefix=/usr/local --sysconfdir=/etc

$ make

$ sudo make install
```

Note: The `make install` above must be run as root to have Singularity properly installed. Failure to install as root will cause Singularity to not function properly or have limited functionality when run by a non-root user.

Also note that when you configure, `squashfs-tools` is **not** required, however it is required for full functionality. You will see this message after the configuration:

```
mksquashfs from squash-tools is required for full functionality
```

If you choose not to install `squashfs-tools`, you will hit an error when your users try a pull from Docker Hub, for example.

1.1.3 Prefix in special places (–localstatedir)

As with most autotools-based build scripts, you are able to supply the `--prefix` argument to the configure script to change where Singularity will be installed. Care must be taken when this path is not a local filesystem or has atypical permissions. The local state directories used by Singularity at runtime will also be placed under the supplied `--prefix` and this will cause malfunction if the tree is read-only. You may also experience issues if this directory is shared between several hosts/nodes that might run Singularity simultaneously.

In such cases, you should specify the `--localstatedir` variable in addition to `--prefix`. This will override the prefix, 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. For example, the Makefile contains this variable by default:

```
CONTAINER_OVERLAY = ${prefix}/var/singularity/mnt/overlay
```

By supplying the configure argument `--localstatedir=/some/other/place` Singularity will instead be built with the following. Note that `${prefix}/var` that has been replaced by the supplied value:

```
CONTAINER_OVERLAY = /some/other/place/singularity/mnt/overlay
```

In the case of cluster nodes, you will need to ensure the following directories are created on all nodes, with `root:root` ownership and `0755` permissions:

```
${localstatedir}/singularity/mnt

${localstatedir}/singularity/mnt/container

${localstatedir}/singularity/mnt/final

${localstatedir}/singularity/mnt/overlay

${localstatedir}/singularity/mnt/session
```

Singularity will fail to execute without these directories. They are normally created by the install make target; when using a local directory for `--localstatedir` these will only be created on the node `make` is run on.

1.1.4 Building an RPM directly from the source

Singularity includes all of the necessary bits to properly create an RPM package directly from the source tree, and you can create an RPM by doing the following:

```
$ ./configure

$ make dist

$ rpmbuild -ta singularity-*.tar.gz
```

Near the bottom of the build output you will see several lines like:

```
...
Wrote: /home/gmk/rpmbuild/SRPMS/singularity-2.3.el7.centos.src.rpm

Wrote: /home/gmk/rpmbuild/RPMS/x86_64/singularity-2.3.el7.centos.x86_64.rpm

Wrote: /home/gmk/rpmbuild/RPMS/x86_64/singularity-devel-2.3.el7.centos.x86_64.rpm

Wrote: /home/gmk/rpmbuild/RPMS/x86_64/singularity-debuginfo-2.3.el7.centos.x86_64.rpm

...
```

You will want to identify the appropriate path to the binary RPM that you wish to install, in the above example the package we want to install is `singularity-2.3.el7.centos.x86_64.rpm` , and you should install it with the following command:

```
$ sudo yum install /home/gmk/rpmbuild/RPMS/x86_64/singularity-2.3.el7.centos.x86_64.
↪rpm
```

Note: If you want to have the binary RPM install the files to an alternative location, you should define the environment variable 'PREFIX' (below) to suit your needs, and use the following command to build:

```
$ PREFIX=/opt/singularity

$ rpmbuild -ta --define="_prefix $PREFIX" --define "_sysconfdir $PREFIX/etc" --define
↪"_defaultdocdir $PREFIX/share" singularity-*.tar.gz
```

We recommend you look at our *security admin guide* to get further information about container privileges and mounting.

CHAPTER

TWO

SECURITY

2.1 Container security paradigms

First some background. Most container platforms operate on the premise, **trusted users running trusted containers**. This means that the primary UNIX account controlling the container platform is either "root" or user(s) that root has deputized (either via `sudo` or given access to a control socket of a root owned daemon process). Singularity on the other hand, operates on a different premise because it was developed for HPC type infrastructures where you have users, none of which are considered trusted. This means the paradigm is considerably different as we must support **untrusted users running untrusted containers**.

2.2 Untrusted users running untrusted containers!

This simple phrase describes the security perspective Singularity is designed with. And if you additionally consider the fact that running containers at all typically requires some level of privilege escalation, means that attention to security is of the utmost importance.

2.2.1 Privilege escalation is necessary for containerization!

As mentioned, there are several containerization system calls and functions which are considered "privileged" in that they must be executed with a certain level of capability/privilege. To do this, all container systems must employ one of the following mechanisms:

1. **Limit usage to root:** Only allow the root user (or users granted `sudo`) to run containers. This has the obvious limitation of not allowing arbitrary users the ability to run containers, nor does it allow users to run containers as themselves. Access to data, security data, and securing systems becomes difficult and perhaps impossible.
2. **Root owned daemon process:** Some container systems use a root owned daemon background process which manages the containers and spawns the jobs within the container. Implementations of this typically have an IPC control socket for communicating with this root owned daemon process and if you wish to allow trusted users to control the daemon, you must give them access to the control socket. This is the Docker model.
3. **SetUID:** Set UID is the "old school" UNIX method for running a particular program with escalated permission. While it is widely used due to it's legacy and POSIX requirement, it lacks the ability to manage fine grained control of what a process can and can not do; a SetUID root program runs as root with all capabilities that comes with root. For this reason, SetUID programs are traditional targets for hackers.
4. **User Namespace:** The Linux kernel's user namespace may allow a user to virtually become another user and run a limited set privileged system functions. Here the privilege escalation is managed via the Linux kernel which takes the onus off of the program. This is a new kernel feature and thus requires new kernels and not all distributions have equally adopted this technology.

5. **Capability Sets:** Linux handles permissions, access, and roles via capability sets. The root user has these capabilities automatically activated while non-privileged users typically do not have these capabilities enabled. You can enable and disable capabilities on a per process and per file basis (if allowed to do so).

2.2.2 How does Singularity do it?

Singularity must allow users to run containers as themselves which rules out options 1 and 2 from the above list. Singularity supports the rest of the options to following degrees of functionally:

- **User Namespace:** Singularity supports the user namespace natively and can run completely unprivileged ("root-less") since version 2.2 (October 2016) but features are severely limited. You will not be able to use container "images" and will be forced to only work with directory (sandbox) based containers. Additionally, as mentioned, the user namespace is not equally supported on all distribution kernels so don't count on legacy system support and usability may vary.
- **SetUID:** This is the default usage model for Singularity because it gives the most flexibility in terms of supported features and legacy compliance. It is also the most risky from a security perspective. For that reason, Singularity has been developed with transparency in mind. The code is written with attention to simplicity and readability and Singularity increases the effective permission set only when it is necessary, and drops it immediately (as can be seen with the `-debug` run flag). There have been several independent audits of the source code, and while they are not definitive, it is a good assurance.
- **Capability Sets:** This is where Singularity is headed as an alternative to SetUID because it allows for much finer grained capability control and will support all of Singularity's features. The downside is that it is not supported equally on shared file systems.

2.3 Where are the Singularity priviledged components

When you install Singularity as root, it will automatically setup the necessary files as SetUID (as of version 2.4, this is the default run mode). The location of these files is dependent on how Singularity was installed and the options passed to the `configure` script. Assuming a default `./configure` run which installs files into `--prefix` of `/usr/local` you can find the SetUID programs as follows:

```
$ find /usr/local/libexec/singularity/ -perm -4000

/usr/local/libexec/singularity/bin/start-suid

/usr/local/libexec/singularity/bin/action-suid

/usr/local/libexec/singularity/bin/mount-suid
```

Each of the binaries is named accordingly to the action that it is suited for, and generally, each handles the required privilege escalation necessary for Singularity to operate. What specifically requires escalated privileges?

1. Mounting (and looping) the Singularity container image
2. Creation of the necessary namespaces in the kernel
3. Binding host paths into the container

Removing any of these SUID binaries or changing the permissions on them would cause Singularity to utilize the non-SUID workflows. Each file with `*-suid` also has a non-suid equivalent:

```
/usr/local/libexec/singularity/bin/start

```

(continues on next page)

(continued from previous page)

```
/usr/local/libexec/singularity/bin/action

/usr/local/libexec/singularity/bin/mount
```

While most of these workflows will not properly function without the SUID components, we have provided these fall back executables for sites that wish to limit the SETUID capabilities to the bare essentials/minimum. To disable the SetUID portions of Singularity, you can either remove the above `*-suid` files, or you can edit the setting for `allow suid` at the top of the `singularity.conf` file, which is typically located in `$PREFIX/etc/singularity/singularity.conf`.

```
# ALLOW SETUID: [BOOL]

# DEFAULT: yes

# Should we allow users to utilize the setuid program flow within Singularity?

# note1: This is the default mode, and to utilize all features, this option

# will need to be enabled.

# note2: If this option is disabled, it will rely on the user namespace

# exclusively which has not been integrated equally between the different

# Linux distributions.

allow setuid = yes
```

You can also install Singularity as root without any of the SetUID components with the configure option `--disable-suid` as follows:

```
$ ./configure --disable-suid --prefix=/usr/local

$ make

$ sudo make install
```

2.4 Can I install Singularity as a user?

Yes, but don't expect all of the functions to work. If the SetUID components are not present, Singularity will attempt to use the "user namespace". Even if the kernel you are using supports this namespace fully, you will still not be able to access all of the Singularity features.

2.5 Container permissions and usage strategy

As a system admin, you want to set up a configuration that is customized for your cluster or shared resource. In the following paragraphs, we will elaborate on this container permissions strategy, giving detail about which users are allowed to run containers, along with image curation and ownership.

These settings can all be found in the Singularity configuration file which is installed to `$PREFIX/etc/singularity/singularity.conf`. When running in a privileged mode, the configuration file **MUST** be owned by root and thus the system administrator always has the final control.

2.5.1 controlling what kind of containers are allowed

Singularity supports several different container formats:

- **squashfs:** Compressed immutable (read only) container images (default in version 2.4)
- **extfs:** Raw file system writable container images
- **dir:** Sandbox containers (chroot style directories)

Using the Singularity configuration file, you can control what types of containers Singularity will support:

```
# ALLOW CONTAINER ${TYPE}: [BOOL]

# DEFAULT: yes

# This feature limits what kind of containers that Singularity will allow

# users to use (note this does not apply for root).

allow container squashfs = yes

allow container extfs = yes

allow container dir = yes
```

2.5.2 limiting usage to specific container file owners

One benefit of using container images is that they exist on the filesystem as any other file would. This means that POSIX permissions are mandatory. Here you can configure Singularity to only "trust" containers that are owned by a particular set of users.

```
# LIMIT CONTAINER OWNERS: [STRING]

# DEFAULT: NULL

# Only allow containers to be used that are owned by a given user. If this

# configuration is undefined (commented or set to NULL), all containers are

# allowed to be used. This feature only applies when Singularity is running in

# SUID mode and the user is non-root.

#limit container owners = gmk, singularity, nobody
```

Note: If you are in a high risk security environment, you may want to enable this feature. Trusting container images to users could allow a malicious user to modify an image either before or while being used and cause unexpected behavior from the kernel (e.g. a DOS attack). For more information, please see: https://lwn.net/Articles/652468/

2.5.3 limiting usage to specific paths

The configuration file also gives you the ability to limit containers to specific paths. This is very useful to ensure that only trusted or blessed container's are being used (it is also beneficial to ensure that containers are only being used on

performant file systems).

```
# LIMIT CONTAINER PATHS: [STRING]

# DEFAULT: NULL

# Only allow containers to be used that are located within an allowed path

# prefix. If this configuration is undefined (commented or set to NULL),

# containers will be allowed to run from anywhere on the file system. This

# feature only applies when Singularity is running in SUID mode and the user is

# non-root.

#limit container paths = /scratch, /tmp, /global
```

2.6 Logging

Singularity offers a very comprehensive auditing mechanism via the system log. For each command that is issued, it prints the UID, PID, and location of the command. For example, let's see what happens if we shell into an image:

```
$ singularity exec ubuntu true

$ singularity shell --home $HOME:/ ubuntu

Singularity: Invoking an interactive shell within container...


ERROR  : Failed to execv() /.singularity.d/actions/shell, continuing to /bin/sh: No␣
↪such file or directory

ERROR  : What are you doing gmk, this is highly irregular!

ABORT  : Retval = 255
```

We can then peek into the system log to see what was recorded:

```
Oct  5 08:51:12 localhost Singularity: action-suid (U=1000,P=32320)> USER=gmk, IMAGE=
↪'ubuntu', COMMAND='exec'

Oct  5 08:53:13 localhost Singularity: action-suid (U=1000,P=32311)> USER=gmk, IMAGE=
↪'ubuntu', COMMAND='shell'

Oct  5 08:53:13 localhost Singularity: action-suid (U=1000,P=32311)> Failed to␣
↪execv() /.singularity.d/actions/shell, continuing to /bin/sh: No such file or␣
↪directory

Oct  5 08:53:13 localhost Singularity: action-suid (U=1000,P=32311)> What are you␣
↪doing gmk, this is highly irregular!

Oct  5 08:53:13 localhost Singularity: action-suid (U=1000,P=32311)> Retval = 255
```

2.6.1 A peek into the SetUID program flow

We can also add the `--debug` argument to any command itself at runtime to see everything that Singularity is doing. In this case we can run Singularity in debug mode and request use of the PID namespace so we can see what Singularity is doing there:

```
$ singularity --debug shell --pid ubuntu

Enabling debugging

Ending argument loop

Singularity version: 2.3.9-development.gc35b753

Exec'ing: /usr/local/libexec/singularity/cli/shell.exec

Evaluating args: '--pid ubuntu'
```

(snipped to PID namespace implementation)

```
DEBUG   [U=1000,P=30961]   singularity_runtime_ns_pid()             Using PID␣
↪namespace: CLONE_NEWPID

DEBUG   [U=1000,P=30961]   singularity_runtime_ns_pid()             Virtualizing PID␣
↪namespace

DEBUG   [U=1000,P=30961]   singularity_registry_get()               Returning NULL␣
↪on 'DAEMON_START'

DEBUG   [U=1000,P=30961]   prepare_fork()                           Creating parent/
↪child coordination pipes.

VERBOSE [U=1000,P=30961]   singularity_fork()                       Forking child␣
↪process

DEBUG   [U=1000,P=30961]   singularity_priv_escalate()              Temporarily␣
↪escalating privileges (U=1000)

DEBUG   [U=0,P=30961]      singularity_priv_escalate()              Clearing␣
↪supplementary GIDs.

DEBUG   [U=0,P=30961]      singularity_priv_drop()                  Dropping␣
↪privileges to UID=1000, GID=1000 (8 supplementary GIDs)

DEBUG   [U=0,P=30961]      singularity_priv_drop()                  Restoring␣
↪supplementary groups

DEBUG   [U=1000,P=30961]   singularity_priv_drop()                  Confirming we␣
↪have correct UID/GID

VERBOSE [U=1000,P=30961]   singularity_fork()                       Hello from␣
↪parent process

DEBUG   [U=1000,P=30961]   install_generic_signal_handle()          Assigning␣
↪generic sigaction()s

DEBUG   [U=1000,P=30961]   install_generic_signal_handle()          Creating generic␣
↪signal pipes
```

(continues on next page)

(continued from previous page)

```
DEBUG   [U=1000,P=30961]   install_sigchld_signal_handle()          Assigning␣
↪SIGCHLD sigaction()

DEBUG   [U=1000,P=30961]   install_sigchld_signal_handle()          Creating sigchld␣
↪signal pipes

DEBUG   [U=1000,P=30961]   singularity_fork()                       Dropping␣
↪permissions

DEBUG   [U=0,P=30961]      singularity_priv_drop()                  Dropping␣
↪privileges to UID=1000, GID=1000 (8 supplementary GIDs)

DEBUG   [U=0,P=30961]      singularity_priv_drop()                  Restoring␣
↪supplementary groups

DEBUG   [U=1000,P=30961]   singularity_priv_drop()                  Confirming we␣
↪have correct UID/GID

DEBUG   [U=1000,P=30961]   singularity_signal_go_ahead()            Sending go-ahead␣
↪signal: 0

DEBUG   [U=1000,P=30961]   wait_child()                             Parent process␣
↪is waiting on child process

DEBUG   [U=0,P=1]          singularity_priv_drop()                  Dropping␣
↪privileges to UID=1000, GID=1000 (8 supplementary GIDs)

DEBUG   [U=0,P=1]          singularity_priv_drop()                  Restoring␣
↪supplementary groups

DEBUG   [U=1000,P=1]       singularity_priv_drop()                  Confirming we␣
↪have correct UID/GID

VERBOSE [U=1000,P=1]       singularity_fork()                       Hello from child␣
↪process

DEBUG   [U=1000,P=1]       singularity_wait_for_go_ahead()          Waiting for go-
↪ahead signal

DEBUG   [U=1000,P=1]       singularity_wait_for_go_ahead()          Received go-
↪ahead signal: 0

VERBOSE [U=1000,P=1]       singularity_registry_set()               Adding value to␣
↪registry: 'PIDNS_ENABLED' = '1'
```

(snipped to end)

```
DEBUG   [U=1000,P=1]       envar_set()                              Unsetting␣
↪environment variable: SINGULARITY_APPNAME

DEBUG   [U=1000,P=1]       singularity_registry_get()               Returning value␣
↪from registry: 'COMMAND' = 'shell'

LOG     [U=1000,P=1]       main()                                   USER=gmk, IMAGE=
↪'ubuntu', COMMAND='shell'
```

(continues on next page)

(continued from previous page)

```
INFO    [U=1000,P=1]       action_shell()                    Singularity:␣
↪Invoking an interactive shell within container...


DEBUG   [U=1000,P=1]       action_shell()                    Exec'ing /.
↪singularity.d/actions/shell

Singularity ubuntu:~>
```

Not only do I see all of the configuration options that I (probably forgot about) previously set, I can trace the entire flow of Singularity from the first execution of an action (shell) to the final shell into the container. Each line also describes what is the effective UID running the command, what is the PID, and what is the function emitting the debug message.

2.6.2 A peek into the "rootless" program flow

The above snippet was using the default SetUID program flow with a container image file named "ubuntu". For comparison, if we also use the `--userns` flag, and snip in the same places, you can see how the effective UID is never escalated, but we have the same outcome using a sandbox directory (chroot) style container.

```
$ singularity -d shell --pid --userns ubuntu.dir/

Enabling debugging

Ending argument loop

Singularity version: 2.3.9-development.gc35b753

Exec'ing: /usr/local/libexec/singularity/cli/shell.exec

Evaluating args: '--pid --userns ubuntu.dir/'
```

(snipped to PID namespace implementation, same place as above)

```
DEBUG   [U=1000,P=32081]   singularity_runtime_ns_pid()      Using PID␣
↪namespace: CLONE_NEWPID

DEBUG   [U=1000,P=32081]   singularity_runtime_ns_pid()      Virtualizing PID␣
↪namespace

DEBUG   [U=1000,P=32081]   singularity_registry_get()        Returning NULL␣
↪on 'DAEMON_START'

DEBUG   [U=1000,P=32081]   prepare_fork()                    Creating parent/
↪child coordination pipes.

VERBOSE [U=1000,P=32081]   singularity_fork()                Forking child␣
↪process

DEBUG   [U=1000,P=32081]   singularity_priv_escalate()       Not escalating␣
↪privileges, user namespace enabled

DEBUG   [U=1000,P=32081]   singularity_priv_drop()           Not dropping␣
↪privileges, user namespace enabled

VERBOSE [U=1000,P=32081]   singularity_fork()                Hello from␣
↪parent process
```

(continues on next page)

(continued from previous page)

```
DEBUG   [U=1000,P=32081]   install_generic_signal_handle()          Assigning␣
↪generic sigaction()s

DEBUG   [U=1000,P=32081]   install_generic_signal_handle()          Creating generic␣
↪signal pipes

DEBUG   [U=1000,P=32081]   install_sigchld_signal_handle()          Assigning␣
↪SIGCHLD sigaction()

DEBUG   [U=1000,P=32081]   install_sigchld_signal_handle()          Creating sigchld␣
↪signal pipes

DEBUG   [U=1000,P=32081]   singularity_signal_go_ahead()            Sending go-ahead␣
↪signal: 0

DEBUG   [U=1000,P=32081]   wait_child()                             Parent process␣
↪is waiting on child process

DEBUG   [U=1000,P=1]       singularity_priv_drop()                  Not dropping␣
↪privileges, user namespace enabled

VERBOSE [U=1000,P=1]       singularity_fork()                       Hello from child␣
↪process

DEBUG   [U=1000,P=1]       singularity_wait_for_go_ahead()          Waiting for go-
↪ahead signal

DEBUG   [U=1000,P=1]       singularity_wait_for_go_ahead()          Received go-
↪ahead signal: 0

VERBOSE [U=1000,P=1]       singularity_registry_set()               Adding value to␣
↪registry: 'PIDNS_ENABLED' = '1'
```

(snipped to end)

```
DEBUG   [U=1000,P=1]       envar_set()                              Unsetting␣
↪environment variable: SINGULARITY_APPNAME

DEBUG   [U=1000,P=1]       singularity_registry_get()               Returning value␣
↪from registry: 'COMMAND' = 'shell'

LOG     [U=1000,P=1]       main()                                   USER=gmk, IMAGE=
↪'ubuntu.dir', COMMAND='shell'

INFO    [U=1000,P=1]       action_shell()                           Singularity:␣
↪Invoking an interactive shell within container...


DEBUG   [U=1000,P=1]       action_shell()                           Exec'ing /.
↪singularity.d/actions/shell

Singularity ubuntu.dir:~> whoami

gmk

Singularity ubuntu.dir:~>
```

Here you can see that the output and functionality is very similar, but we never increased any privilege and none of the `*-suid` program flow was utilized. We had to use a chroot style directory container (as images are not supported with the user namespace, but you can clearly see that the effective UID never had to change to run this container.

Note: Singularity can natively create and manage chroot style containers just like images! The above image was created using the command: `singularity build ubuntu.dir docker://ubuntu:latest`

2.7 Summary

Singularity supports multiple modes of operation to meet your security needs. For most HPC centers, and general usage scenarios, the default run mode is most effective and featurefull. For the security critical implementations, the user namespace workflow maybe a better option. It becomes a balance security and functionality (the most secure systems do nothing).

CHAPTER

THREE

THE SINGULARITY CONFIG FILE

When Singularity is running via the SUID pathway, the configuration **must** be owned by the root user otherwise Singularity will error out. This ensures that the system administrators have direct say as to what functions the users can utilize when running as root. If Singularity is installed as a non-root user, the SUID components are not installed, and the configuration file can be owned by the user (but again, this will limit functionality). The Configuration file can be found at `$SYSCONFDIR/singularity/singularity.conf`. The template in the repository is located at `etc/singularity.conf`. It is generally self documenting but there are several things to pay special attention to:

3.1 Parameters

3.1.1 ALLOW SETUID (boolean, default='yes')

This parameter toggles the global ability to execute the SETUID (SUID) portion of the code if it exists. As mentioned earlier, if the SUID features are disabled, various Singularity features will not function (e.g. mounting of the Singularity image file format). You can however disable SUID support **iff** (if and only if) you do not need to use the default Singularity image file format and if your kernel supports user namespaces and you choose to use user namespaces.

Note: As of the time of this writing, the user namespace is rather buggy

3.1.2 ALLOW PID NS (boolean, default='yes')

While the PID namespace is a neat feature, it does not have much practical usage in an HPC context so it is recommended to disable this if you are running on an HPC system where a resource manager is involved as it has been known to cause confusion on some kernels with enforcement of user limits. Even if the PID namespace is enabled by the system administrator here, it is not implemented by default when running containers. The user will have to specify they wish to implement un-sharing of the PID namespace as it must fork a child process.

3.1.3 ENABLE OVERLAY (boolean, default='no')

The overlay file system creates a writable substrate to create bind points if necessary. This feature is very useful when implementing bind points within containers where the bind point may not already exist so it helps with portability of containers. Enabling this option has been known to cause some kernels to panic as this feature maybe present within a kernel, but has not proved to be stable as of the time of this writing (e.g. the Red Hat 7.2 kernel).

3.1.4 CONFIG PASSWD, GROUP, RESOLV_CONF (boolean, default='yes')

All of these options essentially do the same thing for different files within the container. This feature updates the described file (`/etc/passwd`, `/etc/group` , and `/etc/resolv.conf` respectively) to be updated dynamically as the container is executed. It uses binds and modifies temporary files such that the original files are not manipulated.

3.1.5 MOUNT PROC,SYS,DEV,HOME,TMP (boolean, default='yes')

These configuration options control the mounting of these file systems within the container and of course can be overridden by the system administrator (e.g. the system admin decides not to include the /dev tree inside the container). In most useful cases, these are all best to leave enabled.

3.1.6 MOUNT HOSTFS (boolean, default='no')

This feature will parse the host's mounted file systems and attempt to replicate all mount points within the container. This maybe a desirable feature for the lazy, but it is generally better to statically define what bind points you wish to encapsulate within the container by hand (using the below "bind path" feature).

3.1.7 BIND PATH (string)

With this configuration directive, you can specify any number of bind points that you want to extend from the host system into the container. Bind points on the host file system must be either real files or directories (no special files supported at this time). If the overlayFS is not supported on your host, or if `enable overlay = no` in this configuration file, a bind point must exist for the file or directory within the container. The syntax for this consists of a bind path source and an optional bind path destination separated by a colon. If no bind path destination is specified, the bind path source is used also as the destination.

3.1.8 USER BIND CONTROL (boolean, default='yes')

In addition to the system bind points as specified within this configuration file, you may also allow users to define their own bind points inside the container. This feature is used via multiple command line arguments (e.g. `--bind`, `--scratch` , and `--home`) so disabling user bind control will also disable those command line options. Singularity will automatically disable this feature if the host does not support the prctl option `PR_SET_NO_NEW_PRIVS`. In addition, `enable overlay` must be set to `yes` and the host system must support overlayFS (generally kernel versions 3.18 and later) for users to bind host directories to bind points that do not already exist in the container.

3.1.9 AUTOFS BUG PATH (string)

With some versions of autofs, Singularity will fail to run with a "Too many levels of symbolic links" error. This error happens by way of a user requested bind (done with -B/–bind) or one specified via the configuration file. To handle this, you will want to specify those paths using this directive. For example:

```
autofs bug path = /share/PI
```

3.2 Logging

In order to facilitate monitoring and auditing, Singularity will syslog() every action and error that takes place to the `LOCAL0` syslog facility. You can define what to do with those logs in your syslog configuration.

3.3 Loop Devices

Singularity images have `ext3` file systems embedded within them, and thus to mount them, we need to convert the raw file system image (with variable offset) to a block device. To do this, Singularity utilizes the `/dev/loop*` block devices on the host system and manages the devices programmatically within Singularity itself. Singularity also uses the `LO_FLAGS_AUTOCLEAR` loop device `ioctl()` flag which tells the kernel to automatically free the loop device when there are no more open file descriptors to the device itself. Earlier versions of Singularity managed the loop devices via a background watchdog process, but since version 2.2 we leverage the `LO_FLAGS_AUTOCLEAR` functionality and we forego the watchdog process. Unfortunately, this means that some older Linux distributions are no longer supported (e.g. RHEL <= 5). Given that loop devices are consumable (there are a limited number of them on a system), Singularity attempts to be smart in how loop devices are allocated. For example, if a given user executes a specific container it will bind that image to the next available loop device automatically. If that same user executes another command on the same container, it will use the loop device that has already been allocated instead of binding to another loop device. Most Linux distributions only support 8 loop devices by default, so if you find that you have a lot of different users running Singularity containers, you may need to increase the number of loop devices that your system supports by doing the following: Edit or create the file `/etc/modprobe.d/loop.conf` and add the following line:

```
options loop max_loop=128
```

After making this change, you should be able to reboot your system or unload/reload the loop device as root using the following commands:

```
# modprobe -r loop

# modprobe loop
```

CHAPTER

FOUR

CONTAINER CHECKS

New to Singularity 2.4 is the ability to, on demand, run container "checks," which can be anything from a filter for sensitive information, to an analysis of content on the filesystem. Checks are installed with Singularity, managed by the administrator, and available to the user.

4.1 What is a check?

Broadly, a check is a script that is run over a mounted filesystem, primary with the purpose of checking for some security issue. This process is tightly controlled, meaning that the script names in the checks folder are hard coded into the script check.sh. The flow of checks is the following:

- the user calls `singularity check container.img` to invoke check.exec
- specification of ‘‘–low‘‘(3), ‘‘–med‘‘(2), or ‘‘–high‘‘(1) sets the level to perform. The level is a filter, meaning that a level of 3 will include 3,2,1, and a level of 1 (high) will only call checks of high priority.
- specification of `-t/--tag` will allow the user (or execution script) to specify a kind of check. This is primarily to allow for extending the checks to do other types of things. For example, for this initial batch, these are all considered `default` checks. The check.help displays examples of how the user specifies a tag:

```
# Perform all default checks, these are the same

$ singularity check ubuntu.img

$ singularity check --tag default ubuntu.img


# Perform checks with tag "clean"

$ singularity check --tag clean ubuntu.img
```

4.1.1 Adding a Check

A check should be a bash (or other) script that will perform some action. The following is required: **Relative to SINGULARITY_ROOTFS** The script must perform check actions relative to `SINGULARITY_ROOTFS`. For example, in python you might change directory to this location:

```
import os

base = os.environ["SINGULARITY_ROOTFS"]

os.chdir(base)
```

or do the same in bash:

```
cd $SINGULARITY_ROOTFS

ls $SINGULARITY_ROOTFS/var
```

Since we are doing a mount, all checks must be static relative to this base, otherwise you are likely checking the host system.

Verbose The script should indicate any warning/message to the user if the check is found to have failed. If pass, the check's name and status will be printed, with any relevant information. For more thorough checking, you might want to give more verbose output.

Return Code The script return code of "success" is defined in check.sh, and other return codes are considered not success. When a non success return code is found, the rest of the checks continue running, and no action is taken. We might want to give some admin an ability to specify a check, a level, and prevent continuation of the build/bootstrap given a fail. **Check.sh** The script level, path, and tags should be added to check.sh in the following

format:

```
##########################################################################################

# CHECK SCRIPTS

##########################################################################################


#         [SUCCESS] [LEVEL]  [SCRIPT]                                                    ␣
↪                          [TAGS]

execute_check    0    HIGH  "bash $SINGULARITY_libexecdir/singularity/helpers/checks/
↪1-hello-world.sh"       security

execute_check    0     LOW  "python $SINGULARITY_libexecdir/singularity/helpers/
↪checks/2-cache-content.py"   clean

execute_check    0    HIGH  "python $SINGULARITY_libexecdir/singularity/helpers/
↪checks/3-cve.py"            security
```

The function `execute_check` will compare the level (`[LEVEL]`) with the user specified (or default) `SINGULARITY_CHECKLEVEL` and execute the check only given it is under the specified threshold, and (not yet implemented) has the relevant tag. The success code is also set here with `[SUCCESS]`. Currently, we aren't doing anything with `[TAGS]` and thus perform all checks.

4.2 How to tell users?

If you add a custom check that you want for your users to use, you should tell them about it. Better yet, tell us about it so it can be integrated into the Singularity software for others to use.

CHAPTER

FIVE

TROUBLESHOOTING

This section will help you debug (from the system administrator's perspective) Singularity.

5.1 Not installed correctly, or installed to a non-compatible location

Singularity must be installed by root into a location that allows for `SUID` programs to be executed (as described above in the installation section of this manual). If you fail to do that, you may have user's reporting one of the following error conditions:

```
ERROR  : Singularity must be executed in privileged mode to use images

ABORT  : Retval = 255
```

```
ERROR  : User namespace not supported, and program not running privileged.

ABORT  : Retval = 255
```

```
ABORT  : This program must be SUID root

ABORT  : Retval = 255
```

If one of these errors is reported, it is best to check the installation of Singularity and ensure that it was properly installed by the root user onto a local file system.

CHAPTER

SIX

INSTALLATION ENVIRONMENTS

6.1 Singularity on HPC

One of the architecturally defined features in Singularity is that it can execute containers like they are native programs or scripts on a host computer. As a result, integration with schedulers is simple and runs exactly as you would expect. All standard input, output, error, pipes, IPC, and other communication pathways that locally running programs employ are synchronized with the applications running locally within the container. Additionally, because Singularity is not emulating a full hardware level virtualization paradigm, there is no need to separate out any sandboxed networks or file systems because there is no concept of user-escalation within a container. Users can run Singularity containers just as they run any other program on the HPC resource.

6.1.1 Workflows

We are in the process of developing Singularity Hub, which will allow for generation of workflows using Singularity containers in an online interface, and easy deployment on standard research clusters (e.g., SLURM, SGE). Currently, the Singularity core software is installed on the following research clusters, meaning you can run Singularity containers as part of your jobs:

- The Sherlock cluster at Stanford University
- SDSC Comet and Gordon (XSEDE)
- MASSIVE M1 M2 and M3 (Monash University and Australian National Merit Allocation Scheme)

6.1.1.1 Integration with MPI

Another result of the Singularity architecture is the ability to properly integrate with the Message Passing Interface (MPI). Work has already been done for out of the box compatibility with Open MPI (both in Open MPI v2.1.x as well as part of Singularity). The Open MPI/Singularity workflow works as follows:

1. mpirun is called by the resource manager or the user directly from a shell
2. Open MPI then calls the process management daemon (ORTED)
3. The ORTED process launches the Singularity container requested by the mpirun command
4. Singularity builds the container and namespace environment
5. Singularity then launches the MPI application within the container
6. The MPI application launches and loads the Open MPI libraries
7. The Open MPI libraries connect back to the ORTED process via the Process Management Interface (PMI)
8. At this point the processes within the container run as they would normally directly on the host.

This entire process happens behind the scenes, and from the user's perspective running via MPI is as simple as just calling mpirun on the host as they would normally. Below are example snippets of building and installing OpenMPI into a container and then running an example MPI program through Singularity.

6.1.1.2 Tutorials

Using Host libraries: GPU drivers and OpenMPI BTLs

6.1.1.3 MPI Development Example

What are supported Open MPI Version(s)? To achieve proper container'ized Open MPI support, you should use Open MPI version 2.1. There are however three caveats:

1. Open MPI 1.10.x may work but we expect you will need exactly matching version of PMI and Open MPI on both host and container (the 2.1 series should relax this requirement)
2. Open MPI 2.1.0 has a bug affecting compilation of libraries for some interfaces (particularly Mellanox interfaces using libmxm are known to fail). If your in this situation you should use the master branch of Open MPI rather than the release.
3. Using Open MPI 2.1 does not magically allow your container to connect to networking fabric libraries in the host. If your cluster has, for example, an infiniband network you still need to install OFED libraries into the container. Alternatively you could bind mount both Open MPI and networking libraries into the container, but this could run afoul of glib compatibility issues (its generally OK if the container glibc is more recent than the host, but not the other way around)

6.1.1.4 Code Example using Open MPI 2.1.0 Stable

```
$ # Include the appropriate development tools into the container (notice we are␣
↪calling

$ # singularity as root and the container is writable)

$ sudo singularity exec -w /tmp/Centos-7.img yum groupinstall "Development Tools"

$

$ # Obtain the development version of Open MPI

$ wget https://www.open-mpi.org/software/ompi/v2.1/downloads/openmpi-2.1.0.tar.bz2

$ tar jtf openmpi-2.1.0.tar.bz2

$ cd openmpi-2.1.0

$

$ singularity exec /tmp/Centos-7.img ./configure --prefix=/usr/local

$ singularity exec /tmp/Centos-7.img make

$

$ # Install OpenMPI into the container (notice now running as root and container is␣
↪writable)
```

(continues on next page)

(continued from previous page)

```
$ sudo singularity exec -w -B /home /tmp/Centos-7.img make install

$

$ # Build the OpenMPI ring example and place the binary in this directory

$ singularity exec /tmp/Centos-7.img mpicc examples/ring_c.c -o ring

$

$ # Install the MPI binary into the container at /usr/bin/ring

$ sudo singularity copy /tmp/Centos-7.img ./ring /usr/bin/

$

$ # Run the MPI program within the container by calling the MPIRUN on the host

$ mpirun -np 20 singularity exec /tmp/Centos-7.img /usr/bin/ring
```

6.1.1.5 Code Example using Open MPI git master

The previous example (using the Open MPI 2.1.0 stable release) should work fine on most hardware but if you have an issue, try running the example below (using the Open MPI Master branch):

```
$ # Include the appropriate development tools into the container (notice we are␣
↪calling

$ # singularity as root and the container is writable)

$ sudo singularity exec -w /tmp/Centos-7.img yum groupinstall "Development Tools"

$

$ # Clone the OpenMPI GitHub master branch in current directory (on host)

$ git clone https://github.com/open-mpi/ompi.git

$ cd ompi

$

$ # Build OpenMPI in the working directory, using the tool chain within the container

$ singularity exec /tmp/Centos-7.img ./autogen.pl

$ singularity exec /tmp/Centos-7.img ./configure --prefix=/usr/local

$ singularity exec /tmp/Centos-7.img make

$

$ # Install OpenMPI into the container (notice now running as root and container is␣
↪writable)
```

(continues on next page)

(continued from previous page)

```
$ sudo singularity exec -w -B /home /tmp/Centos-7.img make install
$
$ # Build the OpenMPI ring example and place the binary in this directory
$ singularity exec /tmp/Centos-7.img mpicc examples/ring_c.c -o ring
$
$ # Install the MPI binary into the container at /usr/bin/ring
$ sudo singularity copy /tmp/Centos-7.img ./ring /usr/bin/
$
$ # Run the MPI program within the container by calling the MPIRUN on the host
$ mpirun -np 20 singularity exec /tmp/Centos-7.img /usr/bin/ring


Process 0 sending 10 to 1, tag 201 (20 processes in ring)
Process 0 sent to 1
Process 0 decremented value: 9
Process 0 decremented value: 8
Process 0 decremented value: 7
Process 0 decremented value: 6
Process 0 decremented value: 5
Process 0 decremented value: 4
Process 0 decremented value: 3
Process 0 decremented value: 2
Process 0 decremented value: 1
Process 0 decremented value: 0
Process 0 exiting
Process 1 exiting
Process 2 exiting
Process 3 exiting
Process 4 exiting
```

(continues on next page)

(continued from previous page)

```
Process 5 exiting

Process 6 exiting

Process 7 exiting

Process 8 exiting

Process 9 exiting

Process 10 exiting

Process 11 exiting

Process 12 exiting

Process 13 exiting

Process 14 exiting

Process 15 exiting

Process 16 exiting

Process 17 exiting

Process 18 exiting

Process 19 exiting
```

6.2 Image Environment

6.2.1 Directory access

By default Singularity tries to create a seamless user experience between the host and the container. To do this, Singularity makes various locations accessible within the container automatically. For example, the user's home directory is always bound into the container as is /tmp and /var/tmp. Additionally your current working directory (cwd/pwd) is also bound into the container iff it is not an operating system directory or already accessible via another mount. For almost all cases, this will work flawlessly as follows:

```
$ pwd

/home/gmk/demo

$ singularity shell container.img

Singularity/container.img> pwd

/home/gmk/demo

Singularity/container.img> ls -l debian.def

-rw-rw-r--. 1 gmk gmk 125 May 28 10:35 debian.def
```

(continues on next page)

(continued from previous page)

```
Singularity/container.img> exit

$
```

For directory binds to function properly, there must be an existing target endpoint within the container (just like a mount point). This means that if your home directory exists in a non-standard base directory like "/foobar/username" then the base directory "/foobar" must already exist within the container. Singularity will not create these base directories! You must enter the container with the option `--writable` being set, and create the directory manually.

6.2.1.1 Current Working Directory

Singularity will try to replicate your current working directory within the container. Sometimes this is straight forward and possible, other times it is not (e.g. if the base dir of your current working directory does not exist). In that case, Singularity will retain the file descriptor to your current directory and change you back to it. If you do a 'pwd' within the container, you may see some weird things. For example:

```
$ pwd

/foobar

$ ls -l

total 0

-rw-r--r--. 1 root root 0 Jun  1 11:32 mooooo

$ singularity shell ~/demo/container.img

WARNING: CWD bind directory not present: /foobar

Singularity/container.img> pwd

(unreachable)/foobar

Singularity/container.img> ls -l

total 0

-rw-r--r--. 1 root root 0 Jun  1 18:32 mooooo

Singularity/container.img> exit

$
```

But notice how even though the directory location is not resolvable, the directory contents are available.

6.2.2 Standard IO and pipes

Singularity automatically sends and receives all standard IO from the host to the applications within the container to facilitate expected behavior from the interaction between the host and the container. For example:

```
$ cat debian.def | singularity exec container.img grep 'MirrorURL'

MirrorURL "http://ftp.us.debian.org/debian/"

$

Making changes to the container (writable)

By default, containers are accessed as read only. This is both to enable parallel␣
↪container execution (e.g. MPI). To enter a container using exec, run, or shell you␣
↪must pass the --writable flag in order to open the image as read/writable.
```

6.2.3 Containing the container

By providing the argument `--contain` to `exec`, `run` or `shell` you will find that shared directories are no longer shared. For example, the user's home directory is writable, but it is non-persistent between non-overlapping runs.

6.3 License

```
Redistribution and use in source and binary forms, with or without

modification, are permitted provided that the following conditions are met:


(1) Redistributions of source code must retain the above copyright notice,

this list of conditions and the following disclaimer.


(2) 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.


(3) Neither the name of the University of California, Lawrence Berkeley

National Laboratory, U.S. Dept. of Energy 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 OWNER OR CONTRIBUTORS BE LIABLE
```

(continues on next page)

(continued from previous page)

```
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.

You are under no obligation whatsoever to provide any bug fixes, patches, or
upgrades to the features, functionality or performance of the source code
("Enhancements") to anyone; however, if you choose to make your Enhancements
available either publicly, or directly to Lawrence Berkeley National
Laboratory, without imposing a separate written license agreement for such
Enhancements, then you hereby grant the following license: a  non-exclusive,
royalty-free perpetual license to install, use, modify, prepare derivative
works, incorporate into other computer software, distribute, and sublicense
such enhancements or derivative works thereof, in binary and source code form.

If you have questions about your rights to use or distribute this software,
please contact Berkeley Lab's Innovation & Partnerships Office at
IPO@lbl.gov.

NOTICE.  This Software was developed under funding from the U.S. Department of
Energy and the U.S. Government consequently retains certain rights. As such,
the U.S. Government has been granted for itself and others acting on its
behalf a paid-up, nonexclusive, irrevocable, worldwide license in the Software
to reproduce, distribute copies to the public, prepare derivative works, and
perform publicly and display publicly, and to permit other to do so.
```

6.3.1 In layman terms...

In addition to the (already widely used and very free open source) standard BSD 3 clause license, there is also wording specific to contributors which ensures that we have permission to release, distribute and include a particular contribu-

tion, enhancement, or fix as part of Singularity proper. For example any contributions submitted will have the standard BSD 3 clause terms (unless specifically and otherwise stated) and that the contribution is comprised of original new code that the contributor has authority to contribute.

CHAPTER

SEVEN

APPENDIX

7.1 Using Host libraries: GPU drivers and OpenMPI BTLs

Note: Much of the GPU portion of this tutorial is deprecated by the `--nv` option that automatically binds host system driver libraries into your container at runtime. See the `exec` command for an example.

Singularity does a fantastic job of isolating you from the host so you don't have to muck about with `LD_LIBRARY_PATH`, you just get exactly the library versions you want. However, in some situations you need to use library versions that match host exactly. Two common ones are NVIDIA gpu driver user-space libraries, and OpenMPI transport drivers for high performance networking. There are many ways to solve these problems. Some people build a container and copy the version of the libs (installed on the host) into the container.

7.1.1 What We will learn today

This document describes how to use a bind mount, symlinks and ldconfig so that when the host libraries are updated the container does not need to be rebuilt.

Note this tutorial is tested with Singularity commit 945c6ee343a1e6101e22396a90dfdb5944f442b6, which is part of the (current) development branch, and thus it should work with version 2.3 when that is released. The version of OpenMPI used is 2.1.0 (versions above 2.1 should work).

7.1.2 Environment

In our environment we run CentOS 7 hosts with:

1. slurm located on `/opt/slurm-<version>` and the slurm user `slurm`
2. Mellanox network cards with drivers installed to `/opt/mellanox` (Specifically we run a RoCEv1 network for Lustre and MPI communications)
3. NVIDIA GPUs with drivers installed to `/lib64`
4. OpenMPI (by default) for MPI processes

7.1.3 Creating your image

Since we are building an ubuntu image, it may be easier to create an ubuntu VM to create the image. Alternatively you can follow the recipe here.

Use the following def file to create the image.

```
Bootstrap: debootstrap

MirrorURL: http://us.archive.ubuntu.com/ubuntu/

OSVersion: xenial

Include: apt



%post

apt install -y software-properties-common

apt-add-repository -y universe

apt update

apt install -y wget

mkdir /usr/local/openmpi || echo "Directory exists"

mkdir /opt/mellanox || echo "Directory exists"

mkdir /all_hostlibs || echo "Directory exists"

mkdir /desired_hostlibs || echo "Directory exists"

mkdir /etc/libibverbs.d || echo "Directory exists"

echo "driver mlx4" > /etc/libibverbs.d/mlx4.driver

echo "driver mlx5" > /etc/libibverbs.d/mlx5.driver

adduser slurm || echo "User exists"

wget https://gist.githubusercontent.com/l1ll1/89b3f067d5b790ace6e6767be5ea2851/raw/
↪422c8b5446c6479285cd29d1bf5be60f1b359b90/desired_hostlibs.txt -O /tmp/desired_
↪hostlibs.txt

cat /tmp/desired_hostlibs.txt | xargs -I{} ln -s /all_hostlibs/{} /desired_hostlibs/{}

rm /tmp/desired_hostlibs.txt
```

The mysterious `wget` line gets a list of all the libraries that the CentOS host has in `/lib64` that we think its safe to use in the container. Specifically these are things like nvidia drivers.

```
libvdpau_nvidia.so

libnvidia-opencl.so.1

libnvidia-ml.so.1

libnvidia-ml.so

libnvidia-ifr.so.1

```

(continues on next page)

(continued from previous page)

```
libnvidia-ifr.so

libnvidia-fbc.so.1

libnvidia-fbc.so

libnvidia-encode.so.1

libnvidia-encode.so

libnvidia-cfg.so.1

libnvidia-cfg.so

libicudata.so.50

libicudata.so

libcuda.so.1

libcuda.so

libGLX_nvidia.so.0

libGLESv2_nvidia.so.2

libGLESv1_CM_nvidia.so.1

libEGL_nvidia.so.0

libibcm.a

libibcm.so

libibcm.so.1

libibcm.so.1.0.0

libibdiag-2.1.1.so

libibdiag.a

libibdiag.la

libibdiag.so

libibdiagnet_plugins_ifc-2.1.1.so

libibdiagnet_plugins_ifc.a

libibdiagnet_plugins_ifc.la

libibdiagnet_plugins_ifc.so

libibdmcom-2.1.1.so

libibdmcom.a
```

(continues on next page)

(continued from previous page)

```
libibdmcom.la

libibdmcom.so

libiberty.a

libibis-2.1.1.so.3

libibis-2.1.1.so.3.0.3

libibis.a

libibis.la

libibis.so

libibmad.a

libibmad.so

libibmad.so.5

libibmad.so.5.5.0

libibnetdisc.a

libibnetdisc.so

libibnetdisc.so.5

libibnetdisc.so.5.3.0

libibsysapi-2.1.1.so

libibsysapi.a

libibsysapi.la

libibsysapi.so

libibumad.a

libibumad.so

libibumad.so.3

libibumad.so.3.1.0

libibus-1.0.so.5

libibus-1.0.so.5.0.503

libibus-qt.so.1

libibus-qt.so.1.3.0
```

(continues on next page)

(continued from previous page)

```
libibverbs.a

libibverbs.so

libibverbs.so.1

libibverbs.so.1.0.0

liblustreapi.so

libmlx4-rdmav2.so

libmlx4.a

libmlx5-rdmav2.so

libmlx5.a

libnl.so.1

libnuma.so.1

libosmcomp.a

libosmcomp.so

libosmcomp.so.3

libosmcomp.so.3.0.6

libosmvendor.a

libosmvendor.so

libosmvendor.so.3

libosmvendor.so.3.0.8

libpciaccess.so.0

librdmacm.so.1

libwrap.so.0
```

Also note:

1. in `hostlibs.def` we create a slurm user. Obviously if your `SlurmUser` is different you should change this name.
2. We make directories for `/opt` and `/usr/local/openmpi`. We're going to bindmount these from the host so we get all the bits of OpenMPI and Mellanox and Slurm that we need.

7.1.4 Executing your image

On our system we do:

```
SINGULARITYENV_LD_LIBRARY_PATH=/usr/local/openmpi/2.1.0-gcc4/lib:/opt/munge-0.5.11/
↪lib:/opt/slurm-16.05.4/lib:/opt/slurm-16.05.4/lib/slurm:/desired_hostlibs:/opt/
↪mellanox/mxm/lib/

export SINGULARITYENV_LD_LIBRARY_PATH
```

then

```
srun  singularity exec -B /usr/local/openmpi:/usr/local/openmpi -B /opt:/opt -B /
↪lib64:/all_hostlibs hostlibs.img <path to binary>
```

7.2 Building an Ubuntu image on a RHEL host

This recipe describes how to build an Ubuntu image using Singularity on a RHEL compatible host.

Note: This tutorial is intended for Singularity release 2.1.2, and reflects standards for that version.

In order to do this, you will need to first install the 'debootstrap' package onto your host. Then, you will create a definition file that will describe how to build your Ubuntu image. Finally, you will build the image using the Singularity commands 'create' and `bootstrap`.

7.2.1 Preparation

This recipe assumes that you have already installed Singularity on your computer. If you have not, follow the instructions here to install. After Singularity is installed on your computer, you will need to install the 'debootstrap' package. The 'debootstrap' package is a tool that will allow you to create Debian-based distributions such as Ubuntu. In order to install 'debootstrap', you will also need to install 'epel-release'. You will need to download the appropriate RPM from the EPEL website. Make sure you download the correct version of the RPM for your release.

```
# First, wget the appropriate RPM from the EPEL website (https://dl.fedoraproject.org/
↪pub/epel/)

# In this example we used RHEL 7, so we downloaded epel-release-latest-7.noarch.rpm

$ wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm


# Then, install your epel-release RPM

$ sudo yum install epel-release-latest-7.noarch.rpm


# Finally, install debootstrap

$ sudo yum install debootstrap
```

7.2.1.1 Creating the Definition File

You will need to create a definition file to describe how to build your Ubuntu image. Definition files are plain text files that contain Singularity keywords. By using certain Singularity keywords, you can specify how you want your image

to be built. The extension '.def' is recommended for user clarity. Below is a definition file for a minimal Ubuntu image:

```
DistType "debian"

MirrorURL "http://us.archive.ubuntu.com/ubuntu/"

OSVersion "trusty"



Setup

Bootstrap



Cleanup

The following keywords were used in this definition file:
```

- DistType: DistType specifies the distribution type of your intended operating system. Because we are trying to build an Ubuntu image, the type "debian" was chosen.
- MirrorURL: The MirrorURL specifies the download link for your intended operating system. The Ubuntu archive website is a great mirror link to use if you are building an Ubuntu image.
- OSVersion: The OSVersion is used to specify which release of a Debian-based distribution you are using. In this example we chose "trusty" to specify that we wanted to build an Ubuntu 14.04 (Trusty Tahr) image.
- Setup: Setup creates some of the base files and components for an OS and is highly recommended to be included in your definition file.
- Bootstrap: Bootstrap will call apt-get to install the appropriate package to build your OS.
- Cleanup: Cleanup will remove temporary files from the installation.

While this definition file is enough to create a working Ubuntu image, you may want increased customization of your image. There are several Singularity keywords that allow the user to do things such as install packages or files. Some of these keywords are used in the example below:

```
DistType "debian"

MirrorURL "http://us.archive.ubuntu.com/ubuntu/"

OSVersion "trusty"


Setup

Bootstrap


InstallPkgs python

InstallPkgs wget

RunCmd wget https://bootstrap.pypa.io/get-pip.py

```

(continues on next page)

(continued from previous page)

```
RunCmd python get-pip.py

RunCmd ln -s /usr/local/bin/pip /usr/bin/pip

RunCmd pip install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/
↪tensorflow-0.9.0-cp27-none-linux_x86_64.whl


Cleanup
```

Before going over exactly what image this definition file specifies, the remaining Singularity keywords should be introduced.

- InstallPkgs: InstallPkgs allows you to install any packages that you want on your newly created image.
- InstallFile: InstallFile allows you to install files from your computer to the image.
- RunCmd: RunCmd allows you to run a command from within the new image during the installation.
- RunScript: RunScript adds a new line to the runscript invoked by the Singularity subcommand 'run'. See the run page for more information.

Now that you are familiar with all of the Singularity keywords, we can take a closer look at the example above. As with the previous example, an Ubuntu image is created with the specified DistType, MirrorURL, and OSVersion. However, after Setup and Bootstrap, we used the InstallPkgs keyword to install 'python' and 'wget'. Then we used the RunCmd keyword to first download the pip installation wheel, and then to install 'pip'. Subsequently, we also used RunCmd to pip install `TensorFlow`. Thus, we have created a definition file that will install 'python', 'pip', and 'Tensorflow' onto the new image.

7.2.1.2 Creating your image

Once you have created your definition file, you will be ready to actually create your image. You will do this by utilizing the Singularity 'create' and 'bootstrap' subcommands. The process for doing this can be seen below:

Note: We have saved our definition file as "ubuntu.def"

```
# First we will create an empty image container called ubuntu.img

$ sudo singularity create ubuntu.img

Creating a sparse image with a maximum size of 1024MiB...

INFO   : Using given image size of 1024

Formatting image (/sbin/mkfs.ext3)

Done. Image can be found at: ubuntu.img


# Next we will bootstrap the image with the operating system specified in our␣
↪definition file

$ sudo singularity bootstrap ubuntu.img ubuntu.def

W: Cannot check Release signature; keyring file not available /usr/share/keyrings/
↪ubuntu-archive-keyring.gpg
```

(continues on next page)

(continued from previous page)

```
I: Retrieving Release

I: Retrieving Packages

I: Validating Packages

I: Resolving dependencies of required packages...

I: Resolving dependencies of base packages...

I: Found additional base dependencies: gcc-4.8-base gnupg gpgv libapt-pkg4.12␣
↪libreadline6 libstdc++6 libusb-0.1-4 readline-common ubuntu-keyring

I: Checking component main on http://us.archive.ubuntu.com/ubuntu...

I: Retrieving adduser 3.113+nmu3ubuntu3

I: Validating adduser 3.113+nmu3ubuntu3

I: Retrieving apt 1.0.1ubuntu2

I: Validating apt 1.0.1ubuntu2

snip...

Downloading pip-8.1.2-py2.py3-none-any.whl (1.2MB)

100% |################################| 1.2MB 1.1MB/s

Collecting setuptools

Downloading setuptools-24.0.2-py2.py3-none-any.whl (441kB)

100% |################################| 450kB 2.7MB/s

Collecting wheel

Downloading wheel-0.29.0-py2.py3-none-any.whl (66kB)

100% |################################| 71kB 9.9MB/s

Installing collected packages: pip, setuptools, wheel

Successfully installed pip-8.1.2 setuptools-24.0.2 wheel-0.29.0

At this point, you have successfully created an Ubuntu image with 'python', 'pip',␣
↪and 'TensorFlow' on your RHEL computer.

Tips and Tricks

Here are some tips and tricks that you can use to create more efficient definition␣
↪files:
```

7.2.1.3 Use here documents with RunCmd

Using here documents with conjunction with RunCmd can be a great way to decrease the number of RunCmd keywords that you need to include in your definition file. For example, we can substitute a here document into the previous example:

```
DistType "debian"

MirrorURL "http://us.archive.ubuntu.com/ubuntu/"

OSVersion "trusty"


Setup

Bootstrap


InstallPkgs python

InstallPkgs wget

RunCmd /bin/sh <<EOF

wget https://bootstrap.pypa.io/get-pip.py

python get-pip.py

ln -s /usr/local/bin/pip /usr/bin/pip

pip install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-
↪0.9.0-cp27-none-linux_x86_64.whl

EOF


Cleanup
```

As you can see, using a here document allowed us to decrease the number of RunCmd keywords from 4 to 1. This can be useful when your definition file has a lot of RunCmd keywords and can also ease copying and pasting command line recipes from other sources.

7.2.1.4 Use InstallPkgs with multiple packages

The InstallPkgs keyword is able to install multiple packages with a single keyword. Thus, another way you can increase the efficiency of your code is to use a single InstallPkgs keyword to install multiple packages, as seen below:

```
DistType "debian"

MirrorURL "http://us.archive.ubuntu.com/ubuntu/"

OSVersion "trusty"


Setup

```

(continues on next page)

(continued from previous page)

```
Bootstrap


InstallPkgs python wget

RunCmd /bin/sh <<EOF

wget https://bootstrap.pypa.io/get-pip.py

python get-pip.py

ln -s /usr/local/bin/pip /usr/bin/pip

pip install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-
↪0.9.0-cp27-none-linux_x86_64.whl

EOF


Cleanup
```

Using a single InstallPkgs keyword to install both 'python' and 'wget' allowed to decrease the number of InstallPkgs keywords we had to use in our definition file. This slimmed down our definition file and helped reduce clutter.