Security Requirements of Containerization

Ian A. Palmer*
Massachusetts Institute of Technology
Cambridge, MA 02139
iapalm@mit.edu

Finnian P. Jacobson-Schulte*
Massachusetts Institute of Technology
Cambridge, MA 02139
fjacob@mit.edu

Abstract

As containerization software grows in popularity, questions about its security come to the forefront. These security concerns are even more prevalent on shared machines used by untrusting users, as is the case with many High Performance Computing (HPC) systems. We outline a general purpose containerization security policy for these shared HPC systems. We then examine an open source HPC containerization software, Singularity, and discuss the security guarantees that they claim to have in place. Finally we analyze the security of several different configurations of the singularity software, judging each configuration against our outlined security policy, and highlighting apparent vulnerabilities. Our goal in this is to shed a light on the security issues present in containerization, and outline policies to ensure security for developers using these packages.

1 Introduction

Containerization is an operating-system level technology which allows a user to host multiple isolated user space instances simultaneously. Developers often times write code in specialized environments, consisting of different configuration files, libraries and dependencies necessary for the code to run successfully. When a developer then tries to run the code on another machine or operating system, issues such as bugs and errors may arise. Containerization attempts to abstract away the host operating system, bundling the code together with its configuration files, libraries and any other dependencies into a lightweight portable package called a *container* or *image*. These containers can then be run on any machine with a containerization software package installed. The idea of containerization has been around since the early 1980s with the introduction of `chroot`. In recent years containerization has become increasingly popular as a more lightweight alternative to virtual machines. The most widely used distributions in industry include Docker and Kubernetes.

With the rise in use of containerization software, one big question comes to mind: *How can security guarantees be worked into the containerization software?* In many circumstances it is the case that security guarantees are the afterthought of industry developers. Only once software is written, working correctly and ready to deploy is the security assessed and implemented. Containerization has the potential to change this. If security guarantees can be baked in to the containerization software in use, these security guarantees will be present from the beginning. This will give a baseline for the security of the system in development, with the goal of stopping preventable security mistakes from taking place.

This paper focuses on containerization in a specific setting, that of High Performance Computing (HPC). High performance computing is the use of machines with large resource pools and greater performance than consumer computers. HPC enables large scale research, including simulations, machine learning, and cryptography. Unlike many consumer systems, HPC systems are often simultaneously used by many users for different applications. These users do not necessarily trust each other, so HPC systems must protect users from intentional or accidental attacks from other users.

*Final project for MIT 6.857: Computer and Network Security (Spring 2020)

Containerization is extremely useful in the field of HPC, where a main goal is research reproducability. Creating container images of code and its dependencies allows researchers to run and verify other researchers experiments, without worry of unwanted affects due to differing runtime environments.

In this paper, we first outline a general security policy for containerization software specifically for HPC systems. We then explore the security policies present in an open source HPC containerization software, Singularity. Finally, we perform a security analysis of the Singularity software, determining whether their security goals are met. We aim to show vulnerabilities in the different configurations of the software and look into which configuration, if any, best adheres to our outlined security policy.

1.1 Preliminaries

This section contains several key definitions pertaining to containerization, high performance computing and security that will be used throughout the description of the general security policy as well as the overview and analysis of Singularity's security.

Container Images. The basic unit used for execution in containerization software is container images. These images consist of the the software to be run in the container, along with all of its dependencies, libraries and configuration files. Different layers of this image can be pulled from different sources to create the desired runtime environment of the container owner. Images are stored as lightweight files on the disk, and are controlled with user permissions the same way a normal file in the file system is.

User Permissions. When different users are accessing the same filesystem, user permissions are enforced to control what each individual user is allowed to do. User permissions restrict the read, write and execute abilities of users for each file in the filesystem. This limits what code the users are allowed to run on the system. One special user of the system is the *root* user. This user is in control of assigning the permissions of the other users in the system.

High Performance Computing (HPC). The system that we will be most concerned with when designing our security policy is High Performance Computing (HPC) systems. These are shared systems used by multiple users who do not necessarily trust each other. Some challenges that arise in HPC systems are maintaining privacy and isolation when multiple users all have access to the same system, and making sure that system resources are shared fairly among users.

1.2 Previous Work

As a starting point for our analysis, we will take a look at the current literature on containerization security. We find three main types of relevant document; specific software documentation and their security policies, articles explaining positives and negatives of containerization in the field of security, and specific security analyses of containerization software packages. This section gives a brief overview of these findings and points the reader to sources to read more about the specific examples.

A clear starting point for discovering what current containerization software is prioritizing with respect to security is looking at their documentation. The Singularity security documentation will be discussed in more detail in 3, and the full documentation can be found at [5]. We also took a look at Docker's security policy, the full details of which can be found at [2]. The main source of security that is explored in their documentation is the intrinsic security of the kernel, and how that security transfers to security in the containerization process. This seems to work well for Docker's intended use, however when moving over to the world of high performance computing, allowing for multiple users on the same system means more guarantees are needed for security.

Another look into previous work done in containerization security is found in articles published by different sources highlighting current vulnerabilities as well as advantages and disadvantages pertaining to containerization. Three of the most helpful articles that we use in our work on the HPC containerization security policy are [1; 3; 6]. These articles focus on a few key aspects of containerization security, mainly user access and permissions, kernel root users, and container images. Our security policy focuses heavily on these three aspects of the system, with various modifications to adapt to the HPC environment. Finally, the articles discuss how having security policies baked in to the containerization software packages being used in industry, which should be a priority for developers. Having this automated sense of security can prevent many of the issues that arise when

treating system security as an afterthought. Because of this, having a general purpose containerization security policy is crucial to make sure that containerization being used in this way is meeting its goal.

Finally we look at several independent security analyses of various containerization software packages. One that is particularly useful for shining a light on some current issues in containerization software is Konakandla's analysis in [4]. Konakandla analyzes each component of the containerization software independently, and then performs a pipeline level security analysis. Results from that paper show how Docker containers are not secure using the default configuration. Vulnerabilities include no memory limits on processes being run in containers as well as the ability for different containers running on the same machine to communicate with each other. Docker, however, was designed to be run on individual machines, rather than on HPC systems with multiple users. Docker's allowance for root permissions escalation makes it unusable on HPC systems, and is something that we closely consider when designing our security policy. Moving our attention to Singularity, they make the claim that their software is for "untrusted users running untrusted containers"[5]. Their documentation further goes on to say that "There have been several independent audits of the source code, and while they are not definitive, it is a good assurance." [5]. However, we are not able to locate these audits to check the security analysis against our own security guidelines. Therefore, after outlining our HPC containerization security policy we look into Singularity ourselves and see what configuration is necessary for the software to meet our security guarantees.

Using these current containerization software security docs and analyses, as well as examples of security advantages and disadvantages as inspiration, we are now ready to outline our containerization security policy for HPC systems.

2 Containerization Security Policy

Now that we have seen the basics of containerization and high performance computing, as well as some examples of what work has previously been done on containerization security, we are ready to outline our security policy. Our policy consists of security guarantees that we believe are necessary for HPC systems running containerization software to remain secure. We first give an overview of the different actors of the system and the different states that the system can be in. We then give the our policy, a set of requirements that containerization software on HPC systems must meet.

2.1 Actors

There are four main actors, or users, in HPC systems. They are the root user, container owners, other users (not owners of a specific container) and non-users. We will give a quick description of each user and then describe which actions they are and are not allowed to take in the system, and how this affects security.

Root User. The root user of a HPC system is a special account that is used for system administration. This user is allowed to modify the file system, user permissions, resource allocation and all other attributes of the system. The root user is allowed to read write and execute files in the file system. The root user can also start and stop containers currently running on the system.

Container Owner. The container owner is a user currently running the container on the machine. In HPC systems there may be multiple of these type of users, as multiple users may have the ability to run container images on the system. The owner of a container should be able to start and stop their containers, as well as execute code within the container consistent with their permissions set by the root user. They should be able to read, write and execute other files and container images in the file system, again consistent with their user permissions. They should not be able to have any affect on other containers running in the system that they are not the owner of.

Other User. Other users of the system are users who have access to the system but are either not currently running a container or do not have the ability to run containers. Other users should be able to read, write and execute files in the system consistent with their permissions set by the root user. They should not be able to interact with containers that are currently running unless allowed by their user permissions. They should not be allowed to suspend other users currently running containers.

Non-User. Non-users of the system are actors who do are not given access to the system. They should no be able to learn anything more about the system than they would if the system was not

running the containerization software. Non-users should not be able to have any affect on containers being run by other users in the system.

2.2 System States

There are several different states that our system can be in while running a containerization software package. These are; when no containers are running, when a container/containers is/are running and we are outside of the container, and when a container is running and we are inside of the container. We will give a brief description of each state and how it relates to the security of the system.

No containers currently running.
When no containers are currently running on the system, the system should act in the same way as it would if the containerization software were not running, with the exception that users with the correct permissions can now begin executing containers. This is similar to file execution on the original system. The system in this state should look the same to all the actors described above as it does when the containerization software is not running. This guarantees that just having the containerization software on the system does not introduce any new security vulnerabilities.

Container(s) running, outside of the container.
When a container is running on the system, the system outside of the container should look the same as when the container was not running. The dependencies and libraries that are being used and run inside containers should not have an affect on the outside system. The dependencies and libraries should not be able to be used by code executed outside of the container. Any user who is currently running a container should have no escalated privileges outside of the container, their privileges should stay the same as those assigned by the root user.

Inside a running container.
Inside of a container running on the system, the user may have the appearance of being the root user, however this sense of privilege escalation cannot allow for the user to make changes to the system that they would not be able to make outside of the container. This state should not be able to interact with other running containers unless allowed for by the permissions set by the root user and the owners of the containers.

2.3 Security Requirements

Now that we have seen the different actors of HPC systems and the different states that the system can be in, we can outline the security requirements that a containerization software package needs to meet to be used in a secure way on a HPC system. For each requirement we will give the requirement and then give a brief overview and what it adds to the overall security of the software.

Container images must be trusted by the system in order to be run. Container images are the building blocks of containers, and can be made up of pieces pulled from many different sources. Often the image layers are composed of many open source components. To guarantee that the code used in the container images will not have negative affects on or compromise the system, they must have proof of authentication from their source. Furthermore, the source must be listed as a trusted source of the system to allow for the image to be used. A simple way to enforce this authentication would be using an existentially unforgeable digital signature scheme, allowing the creators of images to digitally sign the images when putting them out for use by developers.

Users privileges should not be escalated when they are running containers. This requirement was mentioned multiple times above in 2.1 and 2.2 and is one of the most important security requirements for containerization in HPC environments. The main reason that Docker and other containerization software packages like it cannot be used on HPC machines is because they escalate container owner privileges to root status for certain actions. In a HPC system multiple users may not trust each other and want their code and containers to be protected. Ensuring that no privilege escalation takes place guarantees that users are only able to act in the way that the root user would allow them to act in the system without the containerization software.

Users of the system must be held accountable for the actions they complete on the system. Since there will be multiple untrusted users interacting with the containerization software on the system, there needs to be some way to track what users complete what actions. Requiring that users do not get elevated privileges when running containers makes this much easier. Maintaining that users

hold consistent privileges will allow the system to track what actions the user takes. This includes downloading container images as well as starting, stopping and executing code in containers.

Users should not be able to use more system resources when running their containers than is set by the root user. As mentioned in 2.1, the root user is allowed to set resource constraints on the other users of the system. These constraints regulate the amount of memory, computing power, and any other system resources that specific users programs and containers are allowed to consume. This helps protect the system against attacks such as denial of service to other users in the system.

Containers should maintain a certain level of isolation in the system. One of the biggest goals of containerization is to create isolated runtime environments for developers to run their code in. This goal becomes even more important when the system is shared, as now the code does not only need to be isolated from a single users other parts of the system, but from other users and their programs and containers as well. When users are running containers they want them isolated from other users. Container code should not be able to interact with other users code, and vice versa. The only interaction with currently running containers should be through actions specifically allowed by the users permissions. As an added security measure container owners can choose to encrypt their container, causing only themselves and other users of their choosing to be able to decrypt and run or execute code within the container.

3 Singularity Security Implementation

Singularity offers several different security options as part of its software package. They vary in the guarantees they provide and the level of permissions they require. Singularity's security options can be divided into two categories: configuration parameters and compile-time parameters.

Configuration Parameters. Configuration parameters are specified during the installation of Singularity. These parameters determine some of the high-level behaviors of Singularity. Here, we analyze the SetUID parameter and capability sets.

Compile-Time Parameters. Some security options are available to users as they compile images. These include container encryption, source verification, and the `fakeroot` option.

3.1 SetUID

Mounting container images, creating user namespaces within the kernel, and binding filesystem paths between the user namespace and container image all require root permissions. Installing Singularity as the root user and enabling SetUID creates two binaries in the installation directory. One of these binaries, `starter`, has a permissions octal of `0755`, meaning that it is readable and executable by all users. The other binary, `starter-suid`, is owned by `root` and has a permissions octal of `4755`. The first octal value of `4` indicates that any users executing the file will execute it with the permissions of its owner, which in this case is `root`.

This is the fundamental difference in the program flow between an installation with the SetUID flag enabled and one without the flag enabled. Without the `starter-suid` binary, Singularity is unable to perform system calls required to mount container images. Users without that feature enabled are limited to sandbox containers, which are folders as opposed to singular image files. These sandbox images do not require mounting. Singularity will also be dependent on user namespace features, which are not universally implemented between all Linux distributions.

After installation, this option can be configured by modifying the `singularity.conf` file. However, this could pose a security risk, as containers that are already mounted would not be affected.

Programs that rely on SetUID are traditional targets for hackers. The SetUID flag allows any user to execute the program with the permissions of the file's owner. As a result, adversaries can use techniques like buffer overflows to execute arbitrary code with root permissions, effectively giving them control over the system. That could be a concern with Singularity installations, especially those in environments processing sensitive/classified information. However, no record of any vulnerabilities based on a SetUID attack on Singularity have been found. In addition, there are many programs that employ SetUID controls without posing a security risk, including the Unix program `ping`. As a result, enabling the SetUID flag is not a major cause for concern, but administrators should be aware of the potential vulnerabilities.

3.2 Capability Sets

Capability sets, like the SetUID flag, are a native Unix feature. Unlike the SetUID flag, however, capability sets are not a POSIX requirement and are therefore not universally implemented in all Unix kernels. Capability sets allow for a more fine-grained control of program permissions. In that way, the philosophy behind their development was to split the functions of the root user into many different capabilities such that if an adversary gained control of one capability, they would be able to do less damage than if they had gained root permissions.

Some of the most common capabilities include `CAP_NET_RAW`, which allows processes to use sockets, `CAP_CHOWN`, which allows processes to alter file UIDs and GIDs, and `CAP_KILL`, which allows processes to bypass permissions checks when sending kill signals. When adding or removing capabilities, Singularity simply executes the corresponding kernel code. Capability flags are stored in a file's extended attributes in the filesystem. Granting a capability from the Singularity interface `singularity capability` is fairly simple, but requires root access. This prevents a non-privileged user from granting themselves capabilities which can then be used to gain root access.

Delegating root access via capability sets is generally considered to be more secure than setting the SetUID flag. However, it is still vulnerable to a number of attacks in which an adversary can use the granted privilege to "break out" and gain complete root access to the machine. As a result, administrators should use caution when granting capabilities to user containers.

3.3 Container Encryption

Singularity allows users to optionally encrypt their containers with a passphrase or RSA keypair. Encrypting containers prevents users without the passphrase or secret key from mounting or executing commands within the container.

Singularity prevents the contents from becoming unintentionally revealed by performing the decryption entirely within kernel space. This ensures that the decrypted container is never located in the user namespace. In addition, Singularity supports several increased security measures with regards to encryption. Users can supply passphrase files, which prevents the key from being recorded in the `bash` history. RSA keys are accepted in the PKCS #1 format, which is resilient for the purposes of encrypting a container image file.

The container encryption scheme is not perfect, however, and vulnerabilities exist in this scheme, as shown in 4.2.

3.4 Sign-and-Verify

Singularity allows users to host container images in online repositories, so it provides a utility to authenticate the source of external images. It implements this using PGP signatures. The signature is computed for the root filesystem partition of the container, which means that this signature scheme is potentially vulnerable to attacks.

3.5 Fakeroot

In general, users without root permissions cannot construct container images from recipe files. The exception to this is using the `fakeroot` argument, which creates containers in which the user appears to have root permissions for most purposes.

Singularity containers built in that way use the UID and GID mappings between the host namespace and the container namespace located in `/etc/subuid` and `/etc/subgid`. Users can then operate as root within container namespaces. However, they will not be able to perform any actions inside of the container that they would not be able to perform otherwise, as their privileges will be limited by their true UID.

4 Singularity Security Analysis

To fully evaluate the security guarantees of Singularity, we attempted to demonstrate vulnerabilities in each of the security schemes.

4.1 Capability Sets

While capability sets are a powerful tool to selectively allow users and processes to perform privileged actions, they are often vulnerable to privilege escalation attacks. For example, consider a HPC cluster with a non-root user `alice`. There are a number of legitimate reasons to request capabilities, and perhaps `alice` needs to request the `CAP_SETUID` capability in order to control the permissions with which routines within the container will run. The system administrator would grant her this capability with the following command:

```
sudo singularity capability add --user alice CAP_SETUID
```

Alice can then build a simple container containing a Python runtime, named `python-container.sif`. Alice can then execute this one line of code:

```
singularity exec --add-caps CAP_SETUID python-container.sif
    python3 -c 'import os; os.setuid(0); os.system("/bin/bash")'
```

This code gives Alice access to a bash prompt. Running the `id` command reveals that Alice is now running bash as root:

```
id
> uid=0(root) gid=1000(alice)
```

She can also perform other root-only operations. `cat /etc/shadow` outputs the contents of the system shadow file to the console, as expected, even though `stat /etc/shadow` indicates it has a permissions octal of `0640`.

This attack works because delegating the ability to change the SetUID flag to Alice allows her to spawn a child process with root permissions, then use that process to assume root permissions herself. This has violated our security scheme - an unprivileged user (Alice) has assumed root privileges in the system.

This attack is not limited to a Python attack vector - similar attacks are possible with almost any modern programming language. This attack can even work through programs like Tar. To do this, Alice is granted the `CAP_SETUID` capability like before. To read `/etc/shadow`, Alice can first create a tarball of the file in her home directory:

```
pwd
> /home/alice
tar -cvf shadow.tar /etc/shadow
> tar: Removing leading ‘/’ from member names
> /etc/shadow
ls -al
> ./
> ../
> shadow.tar
```

Alice can then extract the tarball to produce a copy of `/etc/shadow`.

```
tar -xvf shadow.tar
> etc/shadow
```

There are multiple capabilities that are susceptible to privilege escalation attacks. These include `CAP_SETUID`, `CAP_SETFCAP`, `CAP_SETGID`, and `CAP_CHOWN`, among others.

Capability sets, then, must be used with caution in a Singularity installation. If a vulnerable capability is given to users, it is essentially equivalent to granting them root access. However, using capability sets can be preferable to SetUID controls because it allows for more flexibility. Our recommendation is that capability sets should be granted to users in place of SetUID controls when the system administrator fully understands the risks and delegates only those capabilities not vulnerable to privilege escalation attacks.

4.2 Container Encryption

Container encryption prevents users from mounting the container image without the secret passphrase or key, but it does not necessarily protect all information about the container from unprivileged users.

An encrypted container can be created with the following commands. Note the use of an environment variable prevents the passphrase from being hardcoded, which could be stored in the bash history.

```
export SINGULARITY_ENCRYPTION_PASSPHRASE=$(cat passphrase.txt)
sudo -E singularity build --encrypt analyze-encryption/encrypted-
        container.sif ubuntu-container.def
```

Here, `ubuntu-container.def` is a simple image recipe file with the following contents:

```
Bootstrap: docker
From: ubuntu:16.04

%post
    apt-get update
    apt-get install -y fortune
```

This metadata is not protected by container encryption, however. In fact, none of the container metadata, including the definition file and other preset environment variables, are encrypted or protected in any way. Here, everyone can see that the container has the `fortune` package installed, even though the container itself is encrypted.

In a more security-critical situation, container developers might want to store passphrases and other key information in environment variables that are built into their containers. A developer might create the following defintion file:

```
Bootstrap: docker
From: ubuntu:16.04

%post
        apt-get update
        apt-get install -y git

%help
        This container is secret!

%environment
        SECRET_KEY=b0d3229952dff54161
        SECRET_PASSPHRASE=68572020
```

And use this command to build the container:

```
sudo -E singularity build --encrypt analyze-encryption/encrypted-
container-vulnerable.sif ubuntu-container-secret.def
```

Any unprivileged user, then, can run this command to reveal all of the information in the definition file.

```
singularity inspect --deffile encrypted-container-vulnerable.sif
```

The definition file is built into the container image, too, so deleting the definition file does not preserve the secret.

Singularity claims in their documentation that metadata encryption will be included in a future version. Now, however, we believe this represents a significant potential security vulnerability in Singularity. To mitigate the risks of unencrypted metadata revealing sensitive information, users should follow a few general guidelines:

- Always assume that definition files are public information.
- If any sensitive configuration must be done to the environment, it should not be done during compile-time.
- Containers should not be shared outside their original level of security. For example, a classified container should under no circumstances be shared with a user without a clearance.

These recommendations are essentially cautions against trusting container encryption to completely hide all of a container's contents and metadata. Encryption is a very good option when used correctly, as it prevents unprivileged users from running code inside the container.

4.3 Sign-and-Verify

Singularity's sign-and-verify implementation is vulnerable, much like its encryption implementation. Just as Singularity only encrypts the root partition of the container filesystem, only the root partition is used to generate signatures. As a result, it is possible for an adversary to provide a container that has falsified metadata.

As an example, suppose Alice wants to use an Ubuntu-based container with Python pre-installed. At the same time, Eve is attempting to install software on victims' machines without their knowledge. Eve could compile an image based off of the following definition file, which contains a malicious software package:

```
Bootstrap: docker
From: ubuntu:16.04

%post
        apt-get update
        apt-get install -y python3
        gcc malicious-code.c -o malicious-program
```

Eve could then modify the metadata contents to remove the mention of the malicious program. To Alice, this definition file appears to meet all of her needs. She can then attempt to verify the container:

```
singularity verify ubuntu-python.sif
> Container is signed by 1 key(s):
>
> Verifying partition: FS:
> b0d3229952dff54161b0d3229952dff54161
> [REMOTE]  Eve <eve@foo.bar>
> [OK]      Data integrity verified
>
> INFO:    Container verified: ubuntu-python.sif
```

To Alice, it appears that the container is verified and the definition file contains only the software she wants. However, Eve's malicious program will be included with the image.

To prevent this attack on the sign-and-verify system, it is necessary for both the signer and verifier to supply the `--all` argument to the sign/verify operation. When both parties use this option, it allows a much stronger guarantee of authenticity. It is not sufficient for only the verifier to use this option, though. The signer can choose to withhold signatures from certain partitions (including the compiled definition file), and all Alice will see when she attempts to verify the container with the `--all` argument is a warning:

```
WARNING: Missing signature for SIF descriptor 1 (Def.FILE)
```

For the sign-and-verify system to be secure, it must require all parties to sign/verify all partitions. Singularity claims this is coming in a future version, but until then we identify this as a major security concern. In the meantime, system administrators must weigh the risks of using external container images in secure environments. In most cases, we believe the most secure option is for users to build containers themselves from recipe files. If sharing containers within a secure environment, users should always sign/verify all partitions before trusting a container. This ensures only trusted code is allowed to run in a sensitive high-performance computing environment.

4.4 Analysis Overview

As described in our security policy, it is essential for Singularity to make certain security guarantees. Here, we have also demonstrated that certain configurations of Singularity can violate the security policy laid out in 2. Specifically, we state in 2.3 that users must not be able to escalate their privileges when running containers. Our attack using capability sets violates this principle. Additionally, we cannot always trust containers because we may not be able to verify all of their partitions. This violates our requirement of trust.

System administrators, then, bear the primary responsibility of configuring Singularity in a way that prevents users from violating our security principles. We make the following recommendations about the configuration of Singularity to best implement our security policy:

- If the containers will be used to process sensitive/classified information, then Singularity should be installed without SetUID capabilities and containers should be built with the `--fakeroot` option. This prevents any possibility of privilege escalation, as the program will run entirely in the user namespace.
- If the system is not processing sensitive/classified information, then Singularity may be installed with the SetUID flag enabled.
- Users should only be delegated capabilities if the administrator fully understands the capability and the risks it poses to security.
- Users should not use containers retrieved from a remote source if any partition cannot be verified.
- Containers should be encrypted with a strong passphrase or RSA keypair.

5 Conclusion

In this paper we outline a security policy recommending security requirements for HPC containerization software. We then give an overview of the current security implementations in an open source HPC containerization software, Singularity. Finally, we give an analysis of Singularities security implementation, demonstrating vulnerabilities in the different security configurations that they make available to their users. With the rise in popularity of containerization over the past several years, security should be one of the first considerations. Baking security into containerization software packages allows developers to be reassured when deploying their code that software guarantees are met, and were not treated as an afterthought. In the field of scientific research on HPC systems having containerization capabilities greatly reduces the difficulty of reproducability. Having the security guarantees outlined in our security policy ensures that the users can be confident that their containers are safely being run on the system. Singularity has several policies in place to work towards this goal. However there are still vulnerabilities present in different configurations of the software making different users and containers potentially vulnerable.

6 Acknowledgements

We would like to thank the MIT 6.857: Computer and Network Security course staff, especially Professors Yael Kalai and Ron Rivest for exposing us to the necessary tools to design our security policy as well as perform our security analysis. We would also like to thank them for their time and effort in giving us helpful feedback throughout the process of writing this paper.

References

[1] Bocetta, S. (March 22, 2019). *The 4 Most Vulnerable Areas of Container Security in 2019*. ContainerJournal.com. April 7, 2020. https://containerjournal.com

[2] Docker Docs (2019). *Docker Security*. April 7, 2020. docs.docker.com/engine/security/security/

[3] Jerbi, A. (Aug. 4, 2016). *5 Keys to Conquering Container Security*. May 7, 2020. https://www.infoworld.com/

[4] Konakandla, M. (2016). *Security Analysis of Docker*. April 7, 2020. OWASP AppsecUSA '16.

[5] Singularity Version 2.5.2 Docs (July 18, 2018). *Security*. April 7, 2020. singularity.lbl.gov/docs-security

[6] Tozzi, C. (Aug. 16, 2018). *3 Container Security Advantages and 3 Security Challenges*. May 7, 2020. https://containerjournal.com/