AN ANALYSIS OF SECURITY VULNERABILITIES IN CONTAINER IMAGES FOR SCIENTIFIC DATA ANALYSIS

Bhupinder Kaur

A thesis
in
The Department
of
Computer Science and Software Engineering (CSSE)

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

November 2020

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Bhupinder Kaur**

Entitled: **An Analysis of Security Vulnerabilities in Container Images for Scientific Data Analysis**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

_____ Examiner

_____ Examiner

_____ Examiner

_____ Supervisor

Approved _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____

Rama Bhat, Ph.D.,ing., FEIC, FCSME, FASME, Interim Dean
Faculty of Engineering and Computer Science

Abstract

An Analysis of Security Vulnerabilities in Container Images for Scientific Data Analysis

Bhupinder Kaur

Software containers greatly facilitate the deployment and reproducibility of scientific data analyses on high-performance computing clusters (HPC). However, container images often contain outdated or unnecessary software packages, which increases the number of security vulnerabilities in the images and widens the attack surface of the infrastructure. This thesis presents a vulnerability analysis of container images for scientific data analysis. We compare results obtained with four vulnerability scanners, focusing on the use case of neuroscience data analysis, and quantifying the effect of image update and minification on the number of vulnerabilities. We find that container images used for neuroscience data analysis contain hundreds of vulnerabilities, that software updates remove about two thirds of these vulnerabilities, and that removing unused packages is also effective. We conclude with recommendations on how to build container images with a reduced amount of vulnerabilities.

Acknowledgments

I would like to thank both my supervisors, Dr. Tristan Glatard and Dr. Aiman Hanna, for their unwavering guidance, support, valuable feedbacks, and patience throughout my thesis. I am glad and consider myself lucky that I have such experienced, motivated, and organized professors. I would also like to extend my gratitude to Mathieu Dugré for his help, efforts, and valuable inputs. I also had great pleasure of working with all my lab members, specially, Martin, Valérie, and Ali, you guys are so helpful and great colleagues. Many thanks to my family and friends for giving their emotional support and believing in me. Finally, big thanks to the Concordia University for providing me the chance to work with such supportive and knowledgeable supervisors, and giving me all the facilities and infrastructure to complete this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Containers are very lightweight, flexible, and more resource-efficient than virtual machines. They provide an operating system (OS) level virtualization, which removes the overhead of having an extra OS layer. However, containers tightly integrate with the host due to the sharing of the kernel, which raises security concerns. In this chapter, we provide a brief introduction of containers, mainly discussing two popular containerization techniques Docker and Singularity.

1.1 Containers

OS-level virtualization initially started in 1979 when Chroot was introduced. Chroot is a Unix system call, which changes the root directory of a process and its children processes. A chrooted process cannot access files outside the designated directory. As the time passed, this virtualization got evolved and took different forms of FreeBSD jails, Linux VServer project, and Solaris containers. However, in 2008, with the introduction of Linux containers (LXC) OS-level virtualization reached to an another level of popularity. LXC gained advantage over other Linux-based projects due to the integration of virtualization features into the upstream Linux kernel, which removed the need of applying patches to the Linux kernel and recompilation [2]. LXC relies on Linux kernel's control groups (cgroups) and namespaces. Cgroups control

the usage of system resources (memory, CPU, network, etc) among a group of processes. Namespaces define what a process can see by providing an isolated view of the operating system. For example, PID namespace isolate processes running in a container [3].

1.1.1 Docker

Docker was introduced in 2013 and has gained popularity since then. Docker is built on top of LXC. It automates application deployment, in addition to providing isolated environment. Docker allows software engineers to develop an application only once, on one system, and then use it on other systems without much effort. With the help of Docker, developers pack the application with all its dependencies, libraries, and tools, which can be run on any Linux server later. Docker containers share boot file-system and kernel with the host, which raises security concerns. Another issue is that Docker is not supported by traditional high-performance computing (HPC) resources. This is mainly due to the default configuration of Docker, which runs containers as root. Consequently, it becomes hard for system administrators to keep a record of which user is doing what [4, 5].

Docker Daemon

Docker engine is known as Docker daemon. It is a service that runs on the host operating system and is responsible for managing all Docker containers. It also allocates the host's resources to Docker containers. Like LXC, Docker daemon also uses *cgroups* and *namespaces* of Linux, and hence only support Linux operating system. Docker daemon requires root access to operate, and every container run by Docker is spawned as a child of root-owned Docker daemon. Therefore, users can gain escalated privileges by coercing Docker daemon, which creates serious security risks in shared environments. As a consequence, Docker is not supported in multi-user environments.

Docker Images

A Docker image is a package which contains an application along with all required dependencies and libraries. All the steps of creating a particular image can be saved in a Docker file. Using Dockerfile, there are two ways to create a Docker image. The first is to use an existing Docker image as a base image and then build a new image from that one, by making changes in the filesystem, which is saved as a new layer of the Docker image. The second way is to build a new Docker image from scratch. Docker images are organized as a series of *layers* which are stacked on top of each other. Each layer consists of a filesystem diff that is introduced due to the changes made on the layer below it. All layers of the Docker image are identified by unique *layerIDs*. All these layers stacked together gives a unified and complete view of a Docker image. These layers are compressed into a single image and can be pushed to the Docker Hub.

Docker Hub

Docker Hub is a repository used by Docker to keep all Docker images in one central place. Docker images on Docker Hub can be private or public. Docker Hub also proposes automated builds and webhooks. Docker webhook is an HTTP call-back triggered by a *push* on external code repository. Automated builds, also known as autobuilds, automatically build Docker images from the source code located on external code repository (e.g. GitHub, BitBucket, etc.) and then host them automatically on Docker Hub. During the setup of Docker automated builds, a list of branches and tags, that the user wants to build into Docker images is created. When source code is pushed to a code repository (like GitHub) for these listed image tags, *push* uses Docker webhook to build a new Docker image [6]. The created image is then automatically pushed to Docker Hub. After the automated build, Docker sends HTTP request to a reachable Docker host on the Internet to notify it about the availability of new image, which pulls the built image and restarts container on the new image through Docker webhooks [7, 8].

1.1.2 Singularity

Singularity was introduced in 2016. It offers mobility of compute by facilitating portable environments through a single image file, which makes Singularity containers different from Docker's layered containers. Once Singularity image is created, it is hashed by using SHA256 hashing. Hence, it cannot be changed once it is created, consequently, it can be used for reproducing the results of scientific experiments. Singularity was introduced with the main goal of providing support for multi-user environment [9], where Docker failed to provide such support. Main features provided by Singularity are mobility of compute and reproducibility.

Mobility of compute is defined as creating and maintaining a workflow on a local machine that can be easily ported on other hosts, Linux distributions, and/or cloud service providers. To achieve this, Singularity packs everything that is required for the application to run in a distributable image. Then that image can be copied, archived, and shared. Moreover, Singularity containers are easily portable across different versions of the C library and different kernel implementations [10].

Same features of Singularity, which are used for mobility of compute, facilitate reproducibility as well. Hash of the image is also stored along with the image [10].

Singularity Images

Singularity takes snapshot, locks, and archives the developed application, which is known as Singularity image [11]. Due to image hashing, Singularity does not contain image layers, unlike Docker images. While publishing experimental results, authors can also publish Singularity image along with its hash, which allow other researchers to verify results.

Singularity Hub

Singularity Hub is a central repository provided by Singularity to keep Singularity images and their hash. It is designed to facilitate reproducible research and publications. Singularity Hub can be easily cited in the publications and hence other researchers can easily replicate conducted experiments. Singularity Hub connects with GitHub for automatic builds of Singularity images from the source code present at GitHub.

1.2 Thesis Outline

This thesis consists of four main parts divided into chapters. Chapter 2 provides literature survey on the security of containers. In accordance with that, it provides a list of security requirements that every OS-level virtualization technique should fulfill, and hence based on these requirements it investigates security of containers. This chapter also points out strong features and security weaknesses of containers. Chapter 3 presents various experiments done to analyze security of Docker and Singularity images used in two containerization frameworks used in neuroscience. Additionally, it provides results of these experiments. Finally, Chapter 4 concludes this thesis and highlights the future work.

Chapter 2

Security Review of Containers

In spite of the popularity of containers, there are various security concerns as well. Here, we focus our investigation on the security of Singularity and Docker containers because of three reasons. First, they are popular among containers. Second, security becomes a barrier in the adoption of containers in production. Third, both are already used in various environments, so it is easy to practically investigate their security. In this chapter, we study a set of security requirements, that each OS-level virtualization solution should provide. Additionally, this chapter investigates how these requirements are fulfilled by Singularity and Docker containers, and list their security implications. For this study, we categorize security into two broad categories: internal security provided by containers, and external security provided by the Linux operating system.

2.1 Container's Internal Security

We examine the security of containers based on the requirements given by Reshetova et al., 2014 [2] for comparing the security of a number of OS-level virtualization techniques. According to this research, an OS-level virtualization technique should satisfy the following requirements: process isolation, filesystem isolation, device isolation, IPC isolation, network isolation and limiting of resources inside containers. Apart

from that, in this section, we provide vulnerabilities that are added to containers by container images [12, 13].

2.1.1 Process Isolation

The main goal of process isolation is to prevent one container process from seeing or interfering with another container process. In other words, it limits the permissions and visibility of a container process to processes running in other containers. Containers achieve this goal through the use of namespaces in the Linux kernel. For this purpose, Docker uses *PID namespaces* which isolate a process with a particular process ID from the host and other containers. Provided that, it becomes difficult for an attacker to see processes running in other containers, hence harder to attack them [14]. In contrast to Docker, the default configuration of Singularity does not isolate PID namespaces of containers from the host. The reason behind that is Singularity's main goal is to provide mobility of compute and reproducibility, and not full isolation. However, this configuration can be changed to separate the *PID* namespaces either by using a command line or environment variable setting.

2.1.2 Filesystem Isolation

Filesystem isolation is required to prevent illegitimate access to filesystems of the host and containers. Docker uses *filesystem namespaces*, also called *mount namespaces*, for achieving this goal. Filesystem namespaces hide filesystems of the host and containers from other containers. However, some of the kernel filesystems are not namespaced, so Docker containers mount them for operation. For example, */sys, /proc/sys, /proc/sysrq - trigger, /proc/irq, and /proc/bus* are not namespaced [15]. This causes security concerns as Docker containers are directly able to access host filesystems. Consequently, Docker provides two filesystem protection mechanisms. First, Docker gives read-only permissions to containers for these filesystems. Second, containers are not allowed to remount any filesystem within containers. Additionally, Docker offers a *copy-on-write* filesystem mechanism, which allows containers to write to their specific filesystems and changes are not visible to other containers. Similarly,

files in Singularity containers are also isolated using *filesystem* namespaces.

2.1.3 Device Isolation

Applications and kernel access devices through special files known as device nodes. It is very crucial to limit the set of devices nodes that containers can access. This is primarily because an attacker can own the whole system by gaining access to some important device nodes such as /dev/mem, /dev/sd*, or /dev/tty. Docker uses the *Device Whitelist Controller* [16] feature of control groups to limit the set of devices that a Docker container can see and use. Additionally, Docker starts containers with *nodev* which prevents the use of already created device nodes inside the image. Furthermore, Docker does not allow containers to create new device nodes. However, some of the important device nodes cannot be namespaced such as */dev/mem, /dev/sd**, and kernel modules. Direct access to these device nodes by containers possess serious security concerns. Besides that, if a Docker container is executed in *privileged* mode then it gets access to all devices. Conversely, in default configuration of Singularity, all host devices are visible inside the container because the user is same inside and outside the container.

2.1.4 Inter-Process Communication (IPC) Isolation

IPC is a set of objects through which processes communicate with each other, such as shared memory segments, semaphores, and message queues. IPC isolation is needed to prevent containers from accessing or modifying data belonging to other containers, which is transmitted through these objects. Docker utilizes *IPC namespaces* to assign an IPC namespace to each container. Process in one IPC namespace cannot read or write IPC resources of another IPC namespace. The default configuration of Singularity does not provide IPC isolation.

2.1.5 Network Isolation

Isolation of container's network is very important to prevent network-based attacks such as address resolution protocol (ARP) spoofing and Man-in-the-Middle (MitM) Attack [17]. ARP spoofing is an attack which associates the attacker's Media Access Control (MAC) address with the Internet Protocol (IP) address of another host. ARP is a stateless protocol, hence hosts save all ARP replies even if they had not sent any ARP request for it [18], which becomes a source of attack. So, the host that has spoofed ARP response is not able to verify whether it belongs to legitimate host or attacker, and hence it starts sending packets at attacker's MAC address. In Singularity, by default, the container shares the network with the host. This is because Singularity tries to virtualize "as few as possible" namespaces. Whereas, Docker uses *network* namespaces [19] to isolate the network of Docker containers. Therefore, each Docker container has its different IP address, IP routing tables, network devices, etc. Consequently, Docker containers interact through the network interfaces with each other as well as with the host [20]. On the contrary, all containers share the same network bridge, which makes Docker vulnerable to ARP spoofing attack. In addition to that, bridge of Docker network forwards all incoming packets without any kind of filtering, which makes it vulnerable to Mac flooding [14]. In Docker, the network is used for two purposes : (i) to control Docker daemon remotely, (ii) to distribute images, which include pulling and saving images on Docker Hub.

Docker Daemon Remote Control

Docker daemon is controlled remotely through a socket (which is by default a Unix socket but can be changed to TCP socket), so Docker commands can be performed from another host remotely [21]. By accessing this socket, an attacker can pull and run containers in privileged mode, hence getting root access to the host.

Vulnerable Image Distribution Process

Docker images are present in compressed format on Docker Hub. So, Docker daemon pulls, uncompresses, and then runs containers from these images. Here, Docker daemon can be related to the package manager and Docker Hub to the software repository. Hence, Docker daemon also has an attack surface similar to the package manager. Vulnerabilities include storing, processing potentially untrusted code in Docker images by Docker daemon. The source code can be tampered during transfer or at the source. If some part of the network is compromised, an attacker can replace an image with malicious image, and that image gets downloaded on the host. As the image is in a compressed format, the attacker can cleverly craft the image (i.e. all zeros). If so, it has the potential of filling whole storage on the host after decompression, hence causing denial-of-service (DoS) attack. Other possible attacks are code injection or replay attacks. Additionally, the malicious image can be uploaded to the Docker Hub by an adversary. That image can be downloaded by millions of users infecting millions of machines.

For mitigation of these attacks, Docker introduced content trust which allows signing images before pushing to Docker Hub. However, this content trust can be disabled and thus disabling image signature check. Another issue is related to automated build and webhooks, where compromised GitHub account can lead to the execution of a malicious code. According to the experiment performed in [7] the malicious code was put in production in a very short time i.e within 5 minutes and 30 seconds of commit on GitHub. The content trust provides an environment where a single entity is trusted but in this case, trust is divided among several external entities.

2.1.6 Limiting of Resources

A DoS attack occurs when intended users are not able to use the system or network resources [22]. To launch DoS, the attacker floods targeted host or network traffic with superfluous requests to overload systems. Consequently, the target crashes or its resources get exhausted, hence disrupting normal execution of the system. To solve

this issue, both Docker and Singularity use *cgroups*. Cgroups restrict the amount of resources (CPU, memory, and disk I/O) that are used by containers, thus not allowing one container to consume all resources.

2.1.7 Vulnerabilities in Container Images

There can be vulnerabilities present inside the image itself when it is downloaded from the image repository. According to [23] over 30% official Docker images had high-priority common vulnerabilities and exposures (CVE) identifiers (IDs) and around 64% had high or medium level CVE vulnerabilities at the time of this work. This research work also states that Docker images with the *latest* tag also had vulnerabilities. These vulnerabilities are due to outdated packages contained in images, which may be a consequence of the use of old base image or due to pulling of outdated code during build.

Docker introduced Docker security scanning through which users can scan images to check whether they contain vulnerability or not. However, this scanning is limited to only private repositories and it is a paid service. The scan traverses all layers of the image and then identifies software packages in those layers. Further, it checks vulnerabilities in these software components by taking their Secure Hash Algorithms (SHAs) and then comparing against a standard list of CVEs. This scan can take up to 24 hours depending on image sizes. Additionally, this scanning technique does not detect malware, virus, or vulnerabilities which are not mentioned in the standard CVE database [24].

2.2 Security provided by Kernel

Out-of-the-box security provided by the Linux kernel, which secures the host from containers, is known as host hardening. Linux provides Linux capabilities and Linux Security Modules (LSM) to harden the security of the host system. Linux capabilities divide the privileges of superuser into pieces, and assign a subset of these capabilities

to specific processes. Whereas, LSM provides a framework for Linux to support various security modules. Currently, three security modules are officially integrated with Linux kernel which includes SELinux, AppArmor, and Seccomp. Out of these three, only the first two are supported by Docker. Docker only integrates with Seccomp if LXC are used. Below, we provide the details of these host hardening techniques.

2.2.1 Linux Capabilities

According to the official documentation of Linux [25], Unix systems traditionally categorized processes as *privileged* processes and *unprivileged* processes. Privileged processes are root users with zero user id (UID), whereas unprivileged users are normal users with nonzero UID. Privileged processes are exempted from permission check, whereas unprivileged processes are liable to full permission checks. Linux divides privileges of superuser into different pieces, known as capabilities, which can be independently enabled or disabled. As a result, Docker can disable some of the capabilities of containers, thus improving the security of the system. As Docker containers share the kernel with the host, most of their tasks are done by the host. As a consequence, disabling some of the capabilities in Docker containers do not affect their functionality. For example, *CAP_NET_ADMIN* capability allows configuration of the network, which can be disabled in Docker containers because all network configurations are handled by Docker daemon. By default, most of Linux capabilities are disabled when Docker container is started in order to secure the host system from attackers [1]. Table 1 lists some of the capabilities that are disabled in Docker containers.

CAP_SETPCAP	Modify process capabilities
CAP_SYS_MODULE	Insert/Remove kernel modules
CAP_SYS_RAWIO	Modify kernel memory
CAP_SYS_PACCT	Configure process accounting
CAP_SYS_NICE	Modify priority of processes
CAP_SYS_RESOURCE	Override resource limits
CAP_SYS_TIME	Modify the system clock
CAP_SYS_TTY_CONF IG	Configure tty devices
CAP_AUDIT_WRITE	Write the audit log
CAP_AUDIT_CONTROL	Configure audit subsystem
CAP_MAC_OVERRIDE	Ignore kernel MAC policy
CAP_MAC_ADMIN	Configure MAC configuration
CAP_SYSLOG	Modify kernel printk behavior
CAP_NET_ADMIN	Configure the network
CAP_SYS_ADMIN	Catch all

Table 1: Some disabled capabilities in Docker containers [1].

2.2.2 Linux Security Modules

Below, we provide details of two LSMs that are currently supported by Docker.

SELinux

SELinux is security enhancement to the Linux system, which integrates Mandatory Access Control (MAC). MAC strongly separates all applications, which in turn decreases potential damage if an application is compromised. SELinux classifies active users or processes as subjects, and all system resources as objects. Everything in SELinux is controlled by labels [26]. System administrator writes SELinux policies, which control accesses of system objects by processes. These policies are of three types: Type Enforcement (TE), Multi-Category Security (MCS), and Multi-Level

Security (MLS). Out of these three, Docker uses only the first two types of policies [14]. Using TE, Docker protects the host from containers, by forcing container processes to read/write content that has a specific label [14]. MCS is used to protect containers from other containers. To achieve that, each container is assigned a unique MCS label. All the files that belong to a container are also labelled with the same MCS label. Therefore, kernel does not allow a container process to read/write to a file that have different MCS label. Hence isolating all containers.

AppArmor

Like SELinux, AppArmor is another MAC solution for enhancing security of Linux systems. It uses the concept of file system paths instead of labels. AppArmor mentions the file path in the binary of the application along with the allowed permissions on that file. Two modes are supported by AppArmor: enforcement mode and complain/learning mode [14]. Enforcement mode is used to enforce policies that are defined in the AppArmor profile, whereas complain/learning mode also allows violations of the profile policies. However, these violations are logged, and may be used later for developing new profiles. Docker provides an interface to systems that support AppArmor. This interface is used to load AppArmor profile for containers. Consequently, when containers are launched, a pre-defined AppArmor profile is loaded automatically, if administrator does not specify an AppArmor profile. These security profiles are loaded in enforcement mode by default, to make sure that the policies defined in the profile are enforced. Therefore, the important files of the host such as */sys/fs/cgroups/* and */sys/kernel/security/* remain protected from containers.

Limitations of LSMs

Security provided by these modules is limited due to the generic nature of profiles provided by these security modules [7]. For example, default SELinux profile assigns same domain to all Docker containers, which helps in protecting host from containers, but not containers from containers. Similarly, default AppArmor profile provide full access of network system and capabilities to Docker containers. To improve the

security of the host, a potential solution is to write container-specific profiles.

2.3 Conclusion

In this chapter, we examine various security threats that containers are prone to. We focus our investigation on the security of Docker and Singularity containers because of their popularity. In that context, we study a list of security requirements that every OS-level virtualization should fulfill. Later, we investigate security of Docker and Singularity containers based on these requirements. We broadly categories security of containers into two types: internal security provided by containers, and external security provided by the Linux kernel. Internal security includes process isolation, device isolation, IPC isolation, network isolation, limiting of resources in containers, and vulnerabilities in images. We then highlight solutions provided by containers to provide internal security, and later, we discuss how efficient these solutions are. Further, we discuss external security, which is security provided by the Linux kernel, to protect the host from containers, or containers from containers. External security includes Linux capabilities and LSM framework. Linux capabilities can be disabled/enabled as needed, and LSM framework is used to define various security profiles for containers. Inspite of all these efforts, it is clear that there is still a need of security enhancement in containers.

Chapter 3

Image Analysis

In this chapter, we analyze container images, which are used in two neuroscience frameworks. In accordance with that, these images are scanned using four popular scanners. Additionally, two different approaches, *image update* and *image minification* are followed in order to see their effect on the security vulnerabilities. Finally, this chapter discuss results of these experiments.

3.1 Image Scanning

Image scanning is a technique to detect image vulnerabilities, which reduces the risk of vulnerability exploitation inside containers. Scanning refers to the practice of collecting all information about the container image, investigating it for vulnerabilities, and finally producing a report to summarize scanning results. These reports can be used for security assessments.

3.1.1 Container Images

We scanned all container images available at the time of this study on two containerization frameworks used in neuroscience: BIDS apps [27] (26 images) and Boutiques [28] (18 images), totalling 44 container images (Table 2). At the time of the

study, BIDS apps had 27 images, out of which one wasn't available on DockerHub. Boutiques had 49 images, however, only 23 unique images were listed, out of which 3 couldn't be retrieved and 2 were already included in BIDS apps. All the final 26 images from BIDS apps were Docker images, whereas the 18 Boutiques images contained 12 Docker images and 6 Singularity images.

3.1.2 Image Scanners

We used four container image scanners : Anchore, Vuls, and Clair to scan Docker images, and Singularity Container Tools (Stools) to scan Singularity images.

Anchore

Anchore is an end-to-end, open-source container security platform. It analyzes container images and lists vulnerable OS packages, non-OS packages (Python, Java, Gem, and npm), and files. In our experiments, we used Anchore Engine version 0.5.0 through Docker image `anchore/anchore-engine:v0.5.0`, and Anchore vulnerability database version 0.0.11.

Abbrv	**Image**	**Distribution**
k	bids/hyperalignment	ubuntu:16.04
l	bids/niak	ubuntu:16.04
h	bids/fibredensityandcrosssection	ubuntu:14.04
g	bids/ndmg	ubuntu:14.04
f	bids/ndmg:v0.1.0	ubuntu:14.04
j	bids/oppni:v0.7.0-1	ubuntu:14.04
Q	bids/brainiak-srm	ubuntu:16.04
e	bids/tracula:v6.0.0-4	ubuntu:14.04
H	bids/rs_signal_extract:0.1	ubuntu:16.04
d	bids/example	ubuntu:14.04
i	bids/cpac:v1.0.1a_22	ubuntu:16.04
S	bids/mindboggle:0.0.4-1	debian:8

F	bids/nipypelines:0.3.0	debian:8
N	bt5e/ants:latest	centos:7
a	poldracklab/fmriprep:1.2.3	ubuntu:16.04
Z	bids/hcppipelines:v3.17.0-18	debian:8
b	poldracklab/fmriprep:unstable	ubuntu:16.04
c	bids/dparsf:v4.3.12	ubuntu:14.04
Y	poldracklab/mriqc:0.15.0	ubuntu:16.04
x*	shots47s/bids-fmriprep-1.2.3	ubuntu:16.04
R	gkiar/dwipreproc_fsl-5.0.11_minified	ubuntu:16.04
U	bids/freesurfer	ubuntu:14.04
I	bids/antscorticalthickness:v2.2.0-1	ubuntu:17.04
V	bids/mrtrix3_connectome	ubuntu:18.04
O	bigdatalabteam/hcp-prefreesurfer:exec-centos7-fslbuild-centos5-latest	centos:7
P	bigdatalabteam/hcp-prefreesurfer:exec-centos7.freesurferbuild-centos4-latest	centos:7
v*	aces/cbrain-containers-recipes:fsl_v6.0.1	ubuntu:16.04
D	bids/broccoli:v1.0.0	centos:6
w*	shots47s/bids-freesurfer-6.0	ubuntu:14.04
s*	c3genomics/genpipes	centos:7
W	bids/spm	ubuntu:14.04
X	bids/aa:v0.2.0	ubuntu:14.04
E	mcin/docker-fsl:latest	centos:7
K	mcin/ica-aroma:latest	centos:7
G	bids/baracus	ubuntu:14.04
T	camarasu/creaphase:0.3	centos:7
M	mcin/qeeg:latest	centos:7
L	bids/magetbrain	ubuntu:18.04
u*	MontrealSergiy/BEst	ubuntu:16.04
J	bids/afni_proc	ubuntu:17.1
C	gkiar/mask2boundary:v0.1.0	alpine:3.9.0
t*	bioinformatics-group/aqua-singularity-recipe	debian:9

A	gkiar/onevoxel:v0.3.0rc2	alpine:3.7.1
B	bids/rshrf:1.0.1	alpine:3.8.4

Table 2: List of images scanned. Image with asterisk (*) mark are Singularity image and others are all Docker images.

Vuls

Vuls is an open-source vulnerability scanner for Linux and FreeBSD. It offers both static and dynamic scanning, and both local and remote scanning. In our experiments, we used Vuls 0.9.0, executed through Docker image `vuls/vuls:0.9.0` in remote dynamic mode.

Clair

Clair is an open-source and extensible vulnerability scanner for Docker and appc container images, developed by CoreOS (now Container Linux), a Linux distribution to deploy container clusters. Clair has a client-server architecture, in which the server scans Docker images layer by layer and maintains a database of vulnerabilities. We used Clair through Clair-scanner, a tool to facilitate the testing of container images against a local Clair server. lair-scanner scans the image, prepares a list of vulnerabilities, compares that list against a whitelist, and flags vulnerabilities that are not present in the whitelist. In our experiments, we did not use a whitelist to filter scanning results in order to make a fair comparison between scanners. Clair-scanner maintains a Docker image with the up-to-date vulnerability database from a set of different sources. We used Clair version 2.0.6, executed through Docker image `arminc/clair-local-scan:v2.0.6`. For the vulnerability database, we used Docker image `arminc/clair-db:latest`, last updated on 2019-09-18.

Stools

Singularity Tools (Stools) are an extension of Clair for Singularity images. Stools exports Singularity images to `tar.gz` format, acting as a single layer Docker image to circumvent the Docker-specific requirements in the Clair API. In our experiments, we used Singularity Tools version 3.2.1 through Docker image `vanessa/stools-clair:v3.2.1`. Since Stools uses Clair internally for scanning, vulnerability databases used by Stools are same as mentioned for Clair. To scan Singularity images, we followed the steps mentioned in the Stools documentation.

3.1.3 Vulnerability Databases used by Scanners

Scanners refer to two types of vulnerability databases (Table 3). The first one is the Open Vulnerability and Assessment Language (OVAL) database, an international open standard that supports various OS distributions including Ubuntu, Debian and CentOS but not Alpine. The second one are vulnerability databases from specific OS distributions, such as Alpine-SecDB, Debian Security Bug Tracker, Ubuntu CVE Tracker, or Red Hat Security Data. In these databases, OS distributions often assign a status to each vulnerability, to keep track of required and available security fixes in different versions of the distribution. Vuls uses OVAL databases for all distributions except Alpine, whereas Anchore uses OVAL only for CentOS. On the contrary, Clair exclusively refers to distribution-specific databases, as distribution-specific databases are assumed to be more complete. It is also worth noting that there is no vulnerability data for Ubuntu 17.04 and 17.10 distributions in the OVAL database, since these distributions have reached end of life, meaning that images with these distributions cannot be scanned with Vuls. For CentOS images, Anchore and Clair give scanning results using Red Hat Security Advisory (RHSA) identifiers, whereas Vuls uses the Common Vulnerabilities and Exposures (CVE) identifiers used in OVAL. We mapped RHSA identifiers to corresponding CVE identifiers, to allow for a comparison between scanners. Also, all these scanners refer to National Vulnerability database (NVD), which is a vulnerability database launched by the National Institute of Standards and Technology (NIST), in order to get additional information about the vulnerabilities (e.g., its severity scores, its fix information, and its impact ratings).