DockerGate: Automated Seccomp Policy Generation for Docker Images

Subodh Dharmadhikari
ssdharma@ncsu.edu

Mohit Goyal
mgoyal@ncsu.edu

Abstract

Docker has become a popular technology used by cloud services because of the ease of deployment and process isolation provided to the apps. By default, docker has access to a majority of system calls that are made to the host kernel. This unrestricted access of system calls results in exposing a greater attack surface for sharing same kernel. Docker version 17.03 onwards includes support for a Seccomp profile that can allow or deny system calls that the applications inside a Docker container can make. However, it is impractical to manually define a Seccomp profile for a particular Docker image without innate knowledge of all executable code inside. In this paper we propose DockerGate, a tool that can statically analyze a docker image to identify the system calls in the executable binaries present in the docker image and generate a Seccomp policy. Our key insight here is that static analysis can be reliably used to figure out what system calls are required by each individual container. We evaluate DockerGate by using the tool to generate Seccomp policies for 110 images and running each image within a container with the policy applied. The policy generated should have a minimum number of system calls required for the successful execution of applications in the container, therefore, producing a least privileged Seccomp policy.

1 Introduction

Linux Containers have been gaining attention in production cloud environments in recent times. There have been a wide range of products that are leveraging the container technology (LXC [34], OpenVZ [47], Docker[45] etc.). Linux Containers are lightweight virtual machine processes [48] which share the common host kernel and isolate the data and process from host using kernel security features like namespaces, cgroups and mandatory access controls [32, 41].

Docker [39] is one of the technologies that leverages the use of Linux containers. The containers are based on LXC[34] and provide a virtualization framework for software level as well as hardware level [31]. By encapsulating all system dependencies in a single read-only file, Docker has made it easy to maintain and deploy an application on a large scale. There exist ready-to-use images [2] published by various vendors and community on Dockerhub [40].Based on a survey conducted in April 2017, around 12,947 companies are currently using Docker [38]. Moreover, there is research being done to introduce Docker for High Performance Computing [42]. Docker has also been considered as a solution to the problem of not having access to reproducible research, especially in Computer Systems Research [24, 22, 36].

One of the major reasons for using Docker containers is the security they are able to provide. Since, all the processes and data within a container are inaccessible directly, it reduces the attack surface to the bare minimum of a single container. Combe [26] provides a security perspective on Docker usage in deployment pipelines. He describes possible Adversary models, Vulnerabilities in Docker, Insecure Practices and Docker's Security features. Common attacks like escalation-of-privilege attacks [35] can be avoided as the data and processes are executing in an isolated environment with no access to host resources. The Docker Security open-source community [28] is also active in mitigating new threats on a regular basis. As of April 2017, there have been only 14 CVEs registered since 2014 [27]. However, it is more of a Cat-Mouse game as ongoing research keeps finding new vulnerability classes for the container ecosystem [25]

Docker containers use runC [37] to start a container, which takes care of managing all the components required for a container which consist of cgroups, namespaces and capabilities [4]. The most recent release for Docker introduced a feature to support Seccomp [3] profiles. The Seccomp profile allows restricting the usage of system calls made by processes within a container to its host kernel. The Seccomp profile is a simple file which can specify the action on a system call, whether to allow it or deny. When a container is launched with a Seccomp profile applied to it, it limits the number of system calls the container can make. If no explicit profiles are used for the container, a default profile is loaded implicitly

which blocks access to 44 system calls and allows over 300 system calls [3]. This default policy was created to be all-inclusive of every Docker Image so that their functionality is not hampered by Seccomp.

Many cloud hosting services like AWS, Digital Ocean etc. allow deployment and hosting of docker containers and related technologies [16]. Deployment of a Docker container allows it to use most host operating system resources, which mainly includes system calls, files and access to network. Users can deploy and run containers with different types of applications and services running inside it. Currently there exist no measures for the host to check which processes are being executed inside the containers. While being a security feature, this isolation poses a certain risk to the host. The hosting vendors have to trust the Docker Images and containers to keep the Kernel safe from any possible attack. If the docker containers are executed with an image-specific Seccomp policy, then the possibility of any process using any vulnerable system calls that the application is not using, will be blocked. This will help the host vendors keep the host operating system safe from being exploited by container operations. For example, one would not be able to execute System Call Fuzzers [6] from inside a container.

To generate image-specific Seccomp policies and protect the host, we propose **DockerGate**[1], a framework which can be used to generate a custom Seccomp profile for a Docker image. The Seccomp profile generated should contain the system calls which must be allowed to be accessed by the container processes. With the generation of an image-specific Seccomp profile, we hypothesize that limiting the number of system calls accessible to a container, we can reduce the attack surface on the host kernel without affecting the functionality of the containerized application processes. We evaluate DockerGate by running the framework on a random sample of 110 Docker images and testing the successful start-up and functioning of each container. Each policy produced allows significantly less system calls than the system calls allowed by the default Seccomp policy provided by Docker.

This paper makes the following contributions:

- *We propose DockerGate, an automated Seccomp policy generator for Docker Images.* Our approach involves statically analyzing all executable code in the Docker image and mapping the system calls that might be invoked from that code. Those system calls are aggregated to create a least-privileged Seccomp policy
- *We implement a proof-of-concept prototype for DockerGate that can be used to analyze Docker images based on Ubuntu [17].* By using various Linux-based Binary Analysis tools [11, 10, 12], we were able to profile the ELF binaries in each image and relate them to the libraries they are dynamically linked to. We then mapped those invoked library functions to the system calls each function required and generated the Seccomp profile.
- *We evaluate DockerGate on a random sample of 110 Docker images*. We used DockerGate to generate a Seccomp policy for each image. We then tested each policy by applying them to a container running the specific image that policy was generated for.

DockerGate produces a smaller, lesser privileged, image-specific Seccomp policy for each Docker image. This is able to reduce the attack surface for the host operating system while keeping the Docker container functioning properly.

The remainder of this paper proceeds as follows. Section 2 gives the background and motivation for the paper. Section 3 gives an overview of the design for DockerGate. Section 4 gives a detailed description of the Design for DockerGate Section 5 evaluates our solution. Section 6 discusses additional topics. Section 7 describes related work. Section 8 concludes.

2 Background

The following section describes the background for the paper and our motivation towards proposing DockerGate

2.1 Docker

Docker [39] is an application that helps software developers in deploying their applications across a wide variety of host operating systems without the need to manually configure the dependencies on each individual platform. All system dependencies like required packages and configuration is stored with the actual application in a read-only file called a Docker Image [2]. This allows the developers to configure the application once and deploy it multiple times on variety of platforms without worrying about the platform-specific issues. Whenever the application has to be deployed, the Docker Image is instantiated as a Docker Container [2], that is spawned on the host operating system by a Docker Daemon, that runs on the host itself. Other advantages of running an application in a Docker container are its performance and the process isolation.

Docker containers are considered lightweight. This can allow one host to spawn several containers at

[1] Source Code and Data Set available at `https://github.com/subodh-dharma/dockergate`

a time. Every container behaves as an independent virtual machine in itself, complete with its own file system. However, unlike a virtual machine, that has its own kernel, all Docker containers share the host kernel. So, all system calls being made by a Docker Container would be made to the host kernel. This gives a performance advantage to Containers over Virtual Machines [43]. Bui [23] compared and analyzed between virtual machines and containers in terms of architecture, working and some security features. His experiments concluded that docker containers provide more dense and secured virtual environments compared to virtual machines.

Docker Hub [40] and Docker Store are central repositories that maintain several official and community Docker images. Using these repositories, a Docker user can maintain his/her Docker image and use it on any host by simply pulling the image from Docker Hub onto the host machine. Docker Hub is a very popular Docker image sharing website with about 400 thousand public images available[23].

2.2 Seccomp

Seccomp [9] is a feature in Linux kernel that can force a process to make a one-way transit to a Secure Computing Mode where it is only allowed to make certain system calls (read(), write(), exit(), sigreturn()) on already open file descriptors. This is done to restrict untrusted processes to limited functionality. Seccomp-bpf [9] was developed as an extension to Seccomp that could be used with a configurable policy to filter system calls using Berkeley Packet Filter [55] rules. Out of the actions Seccomp supports, SCMP_ACT_ALLOW signifies that the system call should be allow through whereas SCMP_ACT_ERRNO returns an error when a system call, that is not allowed, is attempted by the protected process.

Since Docker containers make system calls to the kernel, Docker 1.10 was released with custom Seccomp profile support. By defining a list of system calls that are allowed/disallowed, the attack surface of the host kernel is decreased. The Seccomp profile is attached to a running Docker container when the container is started. The system calls made by the container are then filtered using a mechanism similar to Berkeley filter packets according to the rules defined in the seccomp policy. With this new feature, a default Seccomp policy was also introduced which is applied to each Docker container by default. This Seccomp policy blocks 44 vulnerable system calls and allows about 300 system calls [1].

2.3 Motivation

Seccomp support was added as a mechanism to Docker to limit the system calls that could be made by a Docker container. However, a policy is required to define which system calls should be allowed or blocked. While there is a default policy available, it does not provide least-privilege and exposes a larger attack surface. Since every image contains different executable code and has different requirements, we hypothesize that an image-specific profile could be generated that could provide a lesser-privilege to the container running the image by only allowing the system calls that are required and blocking the rest.

2.4 Threat Model

We consider that an attacker has gained complete control of a docker container using a vulnerability present in the application being run inside the container. The attackers goal is to impede the functioning of the host operating system or the kernel by calling a set of vulnerable system calls that are not required by the application being run by the container but might be allowed by the default policy. We do not consider the possibility that the vulnerable system calls could be required by the application itself. The Seccomp policy generated should be able to block the other system calls.

3 Overview

In DockerGate, we propose to analyze the docker image by performing static analysis on the binaries shipped within the images. Using the results of the static analysis we identify the system calls used and generate a seccomp profile. When a docker image is executed as a container, on a seccomp enabled container, it loads a default profile, which allows the container to access almost all system calls on the host. This results in a large attack surface on the host kernel.

However,performing static analysis on docker images presents several implementation challenges:

- *Use of Dynamic Linked Libraries :* In modern practices, binary programs dont perform a system call directly, instead they use libraries that are linked at runtime which act as wrapper functions to call the system calls.
- *Analyzing libraries that link to other libraries:* Libraries are often dynamically linked to other libraries which in turn are linked to other libraries. Recursively going through each library cost too much time and we had to come up with a solution that required to cache this analysis

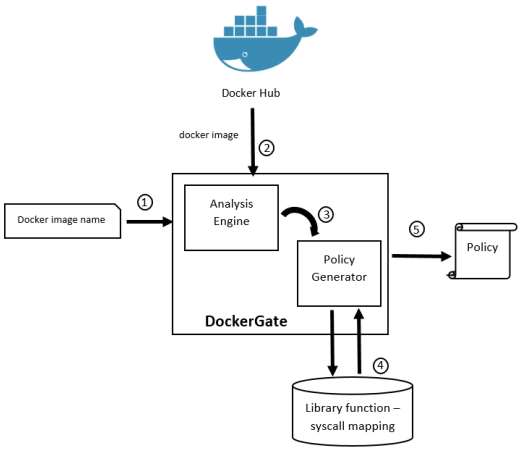

Figure 1: A high-level architecture of our approach

We identified a set of images suitable for testing from Docker Hub and analyzed the Dockerfiles for each image to determine the base image, and finalize on the data set. We used a web scraper to select random images from community image, similar to technique used by Shu et al [53] in developing and downloading data set for their experiments. We then used banyanops/collector [20] to pull these images and execute DockerGate specific scripts within the container. Based on the intermediate results of the DockerGate scripts and a global database of system call mappings we generate a policy using the JSON [8] format, as specified in docker docs [3].

Figure 1 shows major components of the DockerGate framework. An image name is provided to the DockerGate framework to generate the Seccomp policy. In the first stage, the *Analysis Engine* uses the Banyanops collector framework's ability to pull the docker image from Docker hub. The Banyanops framework is configured to execute scripts that perform static analysis within the container. In this stage, Banyanops iteratively creates a new container of the docker image and execute the custom scripts to be executed within the container. The intermediate output generated by these scripts is passed on to the *Policy Generator* submodule to determine the system calls performed by the binaries within the container.

In the Policy Generator, the intermediate output generated is checked for the library function calls it makes and it consults a predefined database of library functions. This database consists of records of standard glibc library [7]. Each library function is associated with the system calls it can perform. Based on the library functions called we determine a set of system calls that can be performed by the container. The framework also supports analysis of any custom libraries found within a image/container. After the analysis of the library the database is updated accordingly to include these function calls as well.

Finally, based on the executable binaries found in the container and the library functions used by them, a policy is generated in the Seccomp policy format where default action is SCMP_ACT_ERRNO and identified system calls are approved to be executed with action SCMP_ACT_ALLOW.

4 Design

DockerGate provides a solution to auto-generate a Seccomp policy for Docker containers by performing static analysis of the executable code contained within the Docker image. This section describes the design in detail.

4.1 Analysis Engine

To perform static analysis, DockerGate initially requires access to the files contained within the image. To access these files it is required to instantiate the docker image to a container. We developed custom Python/Shell scripts which are used for analysis of these files. Each script is executed in a separate container so that the results of one script do not affect the other.

A general work flow to analyze can be described as to perform a **docker pull** on an image then execute a **docker run** on the image and execute the custom script within the new container. We collect and store the results of the scripts on the host for further analysis.We used BanyanOps [20] to automate this process.

We configured the banyanops/collector framework to pull the image from Dockerhub, execute the custom scripts required to collect information on the files to be analyzed, extract the output from the docker container and save them to the host to use in the next stages of the DockerGate.

The custom scripts used to perform analysis have the following objectives:

- *Identify the shared libraries* : All the executables generally use library function calls of the shared libraries to interact with the system. To use any system calls, they use the wrapper functions in the form of library functions defined in the shared libraries. We developed a script to generate a list of libraries used by an executable. We use a simple approach of using ldd command line tool [10] to obtain the library dependencies of the binary files.

- *Identify the library function calls* : To identify the library function calls made by an executable we developed a script which uses nm command line

tool [11]. The nm command used with -D option generates a list of symbols which are not identifiable to the native execution platform.

The binary executable files are dynamically binded to the shared library. During execution the function calls in the binary are substituted by the corresponding native modules of execution defined in the shared library and hence in the object file they are defined as "Unidentified". This makes easy to analyze the object file or binary file and use this function call for further analysis.

- *Mapping library functions to system calls* : Libraries are defined as shared object files(.so) which contains the definition of the library functions to be loaded directly into the memory. We identified such files occurring in the container and determined the system calls for each function. We scanned all the libraries and consolidated the mapping into a single file to form a centralized database of such mapping. We used the text section of objdump output [12] to analyze and perform the mapping of library functions and system calls.

4.2 Policy Generator

The Policy Generator is final stage in the DockerGate framework. It makes use of the output files generated by the custom scripts executed in the docker containers using Banyanops as explained in previous section.

The Policy Generator uses the files generated by the nm tool and a centralized database that maps every library function call to the system calls it is required to perform. It generates a set of system calls based on the function calls recorded by our custom scripts . This set of system calls is stored in a JSON [8] file which can further be used as a Seccomp profile.

4.3 System Call Database

This is a global database generated and updated after analysis of every new shared library that is found in the docker image. The database is a JSON file which represents a list of function names format as shown in Figure 2.

The *function_name* is the name of the library function appearing in the shared library object file. The *callq* array is a list of function calls made within the same shared object or any statically linked shared object. The *syscalls* is the list of system calls appearing in the function definition directly or indirectly.

The system call database is generated from the analysis of the text section of the objdump output of the shared object file. However, the database consists of number of

```
"function_name" :
{
      "callq" : [ func_1, func_2, ···],
      "syscalls" : [sys1, sys2, ···]
}
```

Figure 2: System Call Database Structure

functions in the callq array which needs to be resolved to their respective system calls. Hence, to reduce the number of function in these callq array we resolve the functions in multiple passes by using the same database as input. Currently, we perform five passes to resolve the functions and obtain a detailed list of system calls used by the library function.

This database keeps growing with every Docker Image DockerGate analyzes. DockerGate maintains an index of all the libraries it has already analyzed. While it maintains a static database for a core set of library shared objects it analyzed in the beginning, each set of libraries a binary is linked to is added to this database. This is aimed to improve the performance of DockerGate for future runs as the mapping for a majority of shared-object libraries would already be available.

4.4 Workflow

Figure 3 shows the execution of the framework components in the container and the host system. The figure mainly covers the Analysis Engine in the DockerGate framework which performs most of its operations inside the container.

The Analysis begins by scanning all the files inside the containers using the file command i.e **Stage 1**, which filters out the files identified as ELF files [5]. ELF files are the executable binary files which contain the machine object code and references to external shared libraries used by the file.

In **Stage 2**, ELF files are processed using ldd command line tool to obtain a list of dynamically linked libraries, which are in the shared object (.so) format. These libraries are further analyzed in next stage viz Stage 3 to obtain the system calls performed in each function call. To optimize and increase the performance of this stage, we maintain a global data set of function to system call mapping in the form of System Call Database. We also maintain a global list of library names in the 'index' file which were analyzed previously for any other docker image. If any of the libraries obtained in Stage 2 exists in the 'index' file, the processing for the library is skipped else the library is processed in Stage 3.

In **Stage 3**, we use the **.text** and **.plt** section of the

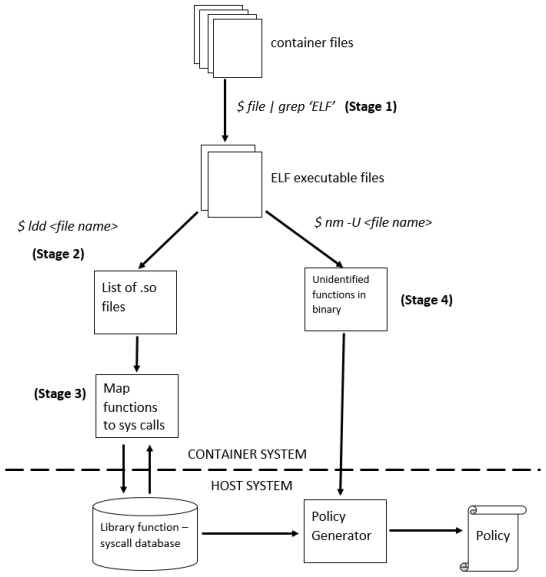

Figure 3: A detailed design

object dump (objdump) for the shared object identified in the previous stage. The **.text** section consists of all the functions defined in that library, while the **.plt** section contains stubs of the functions that are defined in the other dynamically linked-libraries [14]. The main goal of this stage is to identify the system calls performed by each function in the library. As observed by Rauti et al [52], in Linux Architecture, a system call in the object code can be identified by a SYSCALL command. When such SYSCALL command is interpreted the system executes the system call registered in the EAX register. Linux stores numeric value of the system call in EAX. Thus, to achieve our goal, we first look for SYSCALL command in the function definition, and we backtrack the commands until we find a 'mov' command with destination register as EAX. In general we look for a instruction which satisfies the syntax mov 0x00,$eax , where 0x00 can be a numeric value or any other register. If the 0x00 is a hex equivalent of the system call number, we have identified the system call used. If the 0x00 is a register e.g. r01, r02 etc. we further backtrack the program until we find an instruction with syntax mov 0x00, REG, where 0x00 must be a hex value and REG is one of the register encountered as source register in the origin mov instruction with EAX. The backtracking is performed recursively in backward direction in the program until the source of the mov instruction is a hex number.

Along with the direct system calls issued by each function, it can call other functions from within the same library or any external dynamically-linked library that could in-turn eventually be required to invoke a system call. We identify these calls by the jump instructions in the programs. Some of the prominently used jump instructions are *jmp, callq, jne, je* etc. When a function in the **.text** section calls a function from one of its linked libraries, it actually refers to the function stub in its **.plt** section. We leverage this fact and using basic text-processing, are able to recursively follow these function calls to the original set of system-calls, if any.

Further in Stage 3, after identifying the numeric codes of system calls, we convert them to the string equivalent or human readable format. We store this information for each function call into the System Call Database in the structure defined in Figure 2. As described previously, the System call database is refined by iterative approach to resolve the functions in the callq to be converted to system calls, thus keeping the database updated.

In **Stage 4**, we analyze the ELF files by checking which function calls are used by the particular binary files. In the object code of a binary file, the function calls to the library functions remain unidentified. Using the nm utility tool, with -D option, it generates a list of Unidentified symbols which are function calls to externally linked libraries. We use these functions calls and substitute them for the corresponding system calls by referring the System Call Database. This list of system calls is nothing but the system calls to be included in the Seccomp policy of the docker image which are required to be allowed to execute and hence tagged under SCMP_ACT_ALLOW.

5 Evaluation

5.1 Characterizing Data Set

Docker Hub currently host around 400,000 images. To perform experiments with DockerGate, it was required to limit our dataset to test and evaluate. Docker Hub hosts two types of images viz. Official images and Community Images. Official images are maintained by official owners of applications and community images are maintained by users and community members which contain some customized applications developed on top of official images.

To characterize the images we used the Dockerfiles of each image. Dockerfile for Official Images were obtained from the links consolidated in Dockers Official Library repository[9]. While Dockerfile(s) for community images were obtained using a web scraper. We used ScraPy [15], a web scraping framework, to scrape Docker Hub for image names. Using random English dictionary words as search keywords, we were able to get about 96,000 Docker image names. Because of limitations in resources, we randomly sampled 110 Docker image names from those 96,000 names. We conduct our further experiments on this sample of 110 Docker images.

Base Image	Number of docker images
openjdk	118
buildpack-deps	79
debian	78
alpine	73
scratch	49
php	42
python	15
microsoft/windowsservercore	12
microsoft/nanoserver	9
ubuntu	9

Table 1: Top 10 base images used in building a Docker Image

We analyzed these Dockerfiles to determine the nature of docker images. We first identified the base image of the docker image, which can be obtained from Dockerfiles first line starting with keyword FROM. We analyzed the official images and community images separately. The result of this analysis can be found in Table 1.

As can be observed from Table 1, out of total 536 Dockerfiles, 118 images have Openjdk base images. Since Docker images can be derived from each other, we investigated the base images for OpenJDK and buildpack-deps. It was found that all these images were based on either Ubuntu, Alpine or Debian images as their most lowest level base image. The majority of the images were based on Ubuntu and Debian. A complete list of the distribution of images can found in the Appendix 9.

Further we analyzed Dockerfiles of the community images obtained using web scraper. We randomly selected 400 images from the set and checked for the base image. It was observed that about 250 images had a base image of Ubuntu and Debian. While about 100 images were based on Alpine, the rest of them were distributed equally among Fedora, CentOS and Scratch(Not based on any prior Docker Image).

Since the Ubuntu and Debian images was more popular in usage for developing the custom docker images, we have considered only Ubuntu or Debian based images for the analysis and testing of DockerGate.

We then analyzed the average file composition of a Docker Image. Out of a random sample of 110 Docker images, we analyzed the file types for all the executable code in each. As shown in Table 3, it was found that ELF files and Python scripts made up most of the executable code in an average Docker Image. So, we decided to analyze ELF files to generate the Seccomp Policies. This was also supported by the fact that Python files were linked to ELF shared-object libraries. So by analyzing ELF files, we could provide some coverage for Python executable code as well.

Container Status	Number of docker images
Successful Execution	80
Failed Execution	30
Total	**110**

Table 2: Container status after loading them with DockerGate generated Seccomp policy

5.2 DockerGate Policy Evaluation

After issuing a docker run command, the launched container can be in three possible states.

- **Up** - the container is up and live, applications within the container are being executed as expected.
- **Created** - in this state it creates a writable container layer but doesnt start the container execution.
- **Exit** - the container is destroyed and all the resources are freed. Depending on the exit code, it determines if container was gracefully destroyed or with some error in the application. Exit code of 0 indicates a graceful exit, while any other code indicates an erroneous exit from container

We determine the generated Seccomp policy for docker images to be successful by applying the generated policy and attempting to create and execute a simple container. If the status of the container is Up or if it exits with exit code 0, we assume that the container was created successfully and was able to execute the entry-point application within the container. We also verified the success of a container by executing the basic commands within the container. If the result of these commands was successful then we considered the container might function normally. The commands that we tested were *uname, mkdir, /bin/sh & echo*. If the container exited with a non-zero code or entry point failed to execute or appeared to hang-up, we considered it a failure in the Seccomp policy. From a set of Ubuntu based docker images, we executed DockerGate on randomly selected 110 community images. The images contained applications ranging from simple binary files to complete Databases and Web Servers.

From the Table 2 we can see that 80 out of 110 containers exited in successful state. While 30 of

File Type	No. of Files (in percentage)
ELF Files	28
Python Scripts	35
Shell Scripts	14
Others	23

Table 3: Average File Composition for Executable code of a Docker Image

them ended in failure. Further we analyzed the failed images, in which we observed 17 images crashed during execution i.e. they exited the container with status greater than 0 {e.g. 1, 139, 159}. The remaining 13 images out of 30 failed images ended up in **Created** state. The Created state indicates the seccomp policy was successful in initiating the containers but not successful in execution. The reason we speculate is that the container didn't had any executable binaries but relied on script based languages or other native languages like JavaScript, Python or Java.

The Failed images also included 3 images which had failed to execute because the seccomp policy generated didn't include some of the system calls. This caused the container to exit erroneously.

To determine which system calls were used during the execution of the container we used a tool called SystemTap [30]. We developed a custom SystemTap script which would print the system calls being called during the execution of the container. The reason why we could trust System Tap in this case, as compared to strace is that, when a system tap script is executed, a custom Linux kernel module is created and inserted into the kernel. Based on the probe events defined in the script, when any such event is triggered corresponding action is executed in the kernel space. This allowed us to identify exactly which system calls had been used by any command, which in this case, was the Docker container.

We spawned the Docker container for same 110 images used above and obtained the system calls called during the process. In the 110 images above, we found that an average of 118 system calls are actually required for spawning and initiating the start up process within the container.

Thus from the above experiments conducted, we can derive a lower bound on the number of system calls a docker image with base Ubuntu image can have. With an average of **minimum 118 system calls** a basic Ubuntu image can be successfully executed, while our **DockerGate framework generates seccomp policy with average number of system calls equal to 213**.

The current default seccomp policy includes about over 300 system calls to be allowed to be executed, which exposes a large attack surface for the host kernel. Based on the result of experiments conducted, the default policy can be reduced to 213 system calls identified above and reduce the attack surface of the host kernel.

During the evaluation of the images we observed that 17 system calls were required to create a container from its image. Hence, these system calls were included by default when generating the policies. A full list of these system calls can be found in Appendix.

6 Discussion

Using static analysis to develop system wide policies has its limitations. The correctness of the policies depends upon the code coverage of the system. While DockerGate does analyze most of the ELF executables and shared objects, executable code like Java JAR files still remains out of scope for DockerGate. Since many images do use Java, this poses to be a threat to validity as the system calls these files make are left out of the policy.

In our ELF shared-object analysis, we have tracked the contents of the EAX register till just before the system call. However, while we do handle the "mov" instruction recursively as explained in Section 4, there were some cases where the contents of EAX were being modified by other instructions such as XOR and ADD. Such examples were fewer in nature. But we had to log them as exceptions that DockerGate could not handle.

We have focused solely on static analysis in DockerGate because of the reasons cited in Section 2. We believe that if a technique could be found that could execute all possible branches of the executable code in a Docker container, dynamic analysis could be combined with DockerGate to provide tighter policies. Such work has been done previously in Android Applications [13] and could be expanded to Docker containers. While static analysis gives a complete overview of what system calls are required, the dynamic analysis could remove the system calls present in dead-code or code that can never be reached during the execution of the application in the Docker container.

DockerGate can also be extended to produce AppArmour [21] profiles. AppArmour profiles can control the permissions the program in a container can have. These permissions range from read/write/execute abilities on certain files to network access. AppArmour and Seccomp profiles combined can further reduce the attack surface on the host kernel as an attacker would not be able to use an existing program outside the bounds defined by both the security modules.

7 Related Work

After the introduction of the containers, many developers have focused on container hardening which uses the kernel capabilities and features to protect and secure the containers. Various approaches have been proposed earlier where different kernel features can be used to secure the container. One such approach, as proposed by Rastogi et al. [51], is to divide a container into simpler containers to have limited functionality with least privilege, while preserving the overall functionality of the Image.

Provos [49] proposed *Systrace* to facilitate process-specific policy generation based on how the processes invoked system calls. This was a Dynamic Analysis technique that monitored the behaviour of the processes. Policy generation has also been attempted with SELinux [54, 33] and with Android Applications [18]. However, with Docker being a young technology, it hasn't gained much traction on research in academia. But there are many community developers who attempt to bring tools or solutions in various aspects of the Docker. Docker-Slim [29], is a similar project which primarily depends on dynamic analysis of the Docker containers. However, as discussed earlier in this paper, it is difficult to perform dynamic analysis because of the varied number of applications running inside the container. On the contrary, the static analysis scans all eligible files and identify all possible system calls made by the binaries executables present in the docker image. As seen in [18], the code coverage by dynamic analysis despite having an automation framework ranges from 8-10%.

One more kernel feature that can be exploited for container hardening is AppArmor. LiCShield [44] exploits this feature by generating rules for restricting the access of a docker container by monitoring the changes made by processes associated with container and convert it into Linux security module for AppArmor. The works of LiCShield was based on Docker 1.6 version which didnt had the support of Seccomp and AppArmor profiles, but were required to depend on host operating system to implement AppArmor. LiCShield was required to be supported by the host operating system and Docker was not involved in any way. They monitored and controlled the processes forked by Docker daemon.

There have been proposals [19] related to shipping a SELinux policy along with the docker image, where SELinux features are enabled on the host machine, and the SELinux policy is applied to the docker containers executed from this image. But this proposal requires that all the host operating systems must have SElinux policy installed and enabled which is by default is disabled for most Linux based systems. According to the study by Raj et al [46], using the security features like AppArmor and SELinux policies built in Docker, the various stages of DevOps pipeline can be secured and based on suitable profiles can secure the stages and deployments. Their experiments claim to mitigate certain security risk vectors.

Rajagopalan et al [50] proposed Authenticated System Calls which performed static analysis on executable binaries to replace the system calls with custom system calls. But this required modification in the host kernel which may affect the integrity of host kernel.

8 Conclusion

Coming up with least privilege policies to improve Container Security is a time-consuming and difficult task. DockerGate represents an initial step towards achieving tighter policies by leveraging the capability of static analysis on code. It is able to reduce the attack surface on the host kernel and keep the Docker containers functional at the same time. We believe better results can be produced when complete code coverage can be achieved by including all types of executable code. The same approach could be extended to creating Network-related policies for Docker container.

References

[1] Default docker seccomp policy. `https://github.com/moby/moby/blob/master/profiles/seccomp/default.json`.

[2] Docker images and containers. `https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/`.

[3] Docker seccomp policy. `https://docs.docker.com/engine/security/seccomp/`.

[4] Docker security. `https://docs.docker.com/engine/security/security/`.

[5] Elf file format. `http://man7.org/linux/man-pages/man5/elf.5.html`.

[6] Fuzzing dcoker containers with trinity. `https://www.binarysludge.com/2014/07/02/fuzzing-docker-containers-with-trinity/`.

[7] Glibc - core libraries for linux systems. `https://www.gnu.org/software/libc/libc.html`.

[8] Json. `http://www.json.org/`.

[9] Linux kernel feature - seccomp. `http://man7.org/linux/man-pages/man2/seccomp.2.html`.

[10] Linux tool - ldd. `https://linux.die.net/man/1/ldd`.

[11] Linux tool - nm. `https://linux.die.net/man/1/nm`.

[12] Linux tool - objdump. `https://linux.die.net/man/1/objdump`.

[13] monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html.

[14] Plt and got - the key to code sharing and dynamic libraries. https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html.

[15] Scrapy. https://scrapy.org/.

[16] The shortlist of docker hosting. https://blog.codeship.com/the-shortlist-of-docker-hosting/.

[17] Ubuntu. https://www.ubuntu.com.

[18] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[19] E. Bacis, S. Mutti, S. Capelli, and S. Paraboschi. Dockerpolicymodules: mandatory access control for docker containers. In *Communications and Network Security (CNS), 2015 IEEE Conference on*, pages 749–750. IEEE, 2015.

[20] banyanops. banyanops/collector. https://github.com/banyanops/collector.

[21] M. Bauer. Paranoid penguin: an introduction to novell apparmor. *Linux Journal*, 2006(148):13, 2006.

[22] C. Boettiger. An introduction to docker for reproducible research. In *ACM SIGOPS Operating Systems Review - Special Issue on Repeatability and Sharing of Experimental Artifacts Volume 49 Issue 1s*, pages 71–79. ACM.

[23] T. Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[24] C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren. Measuring reproducibility in computer systems research. *Department of Computer Science, University of Arizona, Tech. Rep*, 2014.

[25] T. Combe, A. Martin, and R. Di Pietro. Containers: Vulnerability analysis. Technical report, tech. report, Nokia Bell Labs.

[26] T. Combe, A. Martin, and R. Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016.

[27] CVE. Docker cve list. https://www.cvedetails.com/vulnerability-list/vendor_id-13534/product_id-28125/Docker-Docker.html, 2014.

[28] Docker. Docker open source community. https://github.com/docker/docker.github.io.

[29] docker slim. docker-slim. https://github.com/docker-slim/docker-slim.

[30] F. C. Eigler and R. Hat. Problem solving with systemtap. In *Proc. of the Ottawa Linux Symposium*, pages 261–268. Citeseer, 2006.

[31] J. Fink. Docker: a software as a service, operating system-level virtualization framework. *code4lib*, 25(36), apr 2014.

[32] A. Grattafiori. Understanding and hardening linux containers. *Whitepaper, NCC Group*, 2016.

[33] T. Harada, T. Horie, and K. Tanaka. Access policy generation system based on process execution history. In *Network Security Forum*, 2003.

[34] M. Helsley. Lxc: Linux container tools. Technical report, IBM Developer Works, feb 2009.

[35] J. Hertz. Abusing privileged and unprivileged linux containers. *NCCGroup*, 2016.

[36] L.-H. Hung, D. Kristiyanto, S. B. Lee, and K. Y. Yeung. Guidock: Using docker containers with a common graphics user interface to address the reproducibility of research. *PloS one*, 11(4):e0152686, 2016.

[37] S. Hykes. Introducing runc: a lightweight universal container runtime. *Docker Blog*, 2015.

[38] iDataLabs. Companies using docker. Technical report, iDataLabs, 2017.

[39] D. Inc. Docker. https://www.docker.com.

[40] D. Inc. Docker hub. https://hub.docker.com.

[41] D. Inc. Introduction to container security. *www.docker.com*, 2016.

[42] D. M. Jacobsen and R. S. Canon. Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*, 2015.

[43] S. Kyoung-Taek, H. Hyun-Seo, M. Il-Young, K. Oh-Young, and K. Byeong-Jun. Performance comparison analysis of linux container and virtual machine for building cloud. In *Advanced Science and Technology Letters, Vol.66 (Networking and Communication 2014)*, pages 105–111.

[44] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, and L. Foschini. Securing the infrastructure and the workloads of linux containers. In *Communications and Network Security (CNS), 2015 IEEE Conference on*, pages 559–567. IEEE, 2015.

[45] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[46] A. R. MP, A. Kumar, S. J. Pai, and A. Gopal. Enhancing security of docker using linux hardening techniques. In *Applied and Theoretical Computing and Communication Technology (iCATccT), 2016 2nd International Conference on*, pages 94–99. IEEE, 2016.

[47] OpenVZ. Openvz. `https://openvz.org`.

[48] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance evaluation of virtualization technologies for server consolidation. In *HP Laboratories*. HP Laboratories, 2007.

[49] N. Provos. Improving host security with system call policies. In *Usenix Security*, volume 3, page 19, 2003.

[50] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting. Authenticated system calls. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 358–367. IEEE, 2005.

[51] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel. Towards least privilege containers with cimplifier. *arXiv preprint arXiv:1602.08410*, 2016.

[52] S. Rauti, S. Laurén, S. Hosseinzadeh, J.-M. Mäkelä, S. Hyrynsalmi, and V. Leppänen. Diversification of system calls in linux binaries. In *International Conference on Trusted Systems*, pages 15–35. Springer, 2014.

[53] R. Shu, X. Gu, and W. Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280. ACM, 2017.

[54] B. T. Sniffen, D. R. Harris, and J. D. Ramsdell. Guided policy generation for application authors. In *SELinux Symposium*, 2006.

[55] V. J. Steven McCanne. The bsd packet filter: a new architecture for user-level packet capture. In *USENIX'93 Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 2–2.

9 Appendix

9.1 Default System Calls used in DockerGate Seccomp Policy

System Calls
capget
capset
chdir
futex
fchown
readdirent
getdents64
getpid
getppid
lstat
openat
prctl
setgid
setgroups
setuid
stat
rt_sigreturn

Table 4: Default System Calls included in the DockerGate generated seccomp policy for all Ubuntu based images

9.2 Base Image Analysis of Official Images on Docker Hub

Base Image	Number of docker images
openjdk	118
buildpack-deps	79
debian	78
alpine	73
scratch	49
php	42
python	15
microsoft/windowsservercore	12
microsoft/nanoserver	9
ubuntu	9
ruby	8
erlang	8
tomcat	6
docker	6
haxe	4
redmine	3
node	3
jruby	2
golang	2
pypy	2
sentry	2
rabbitmq	2
clojure	2
geonetwork	1
znc	1
Total	**536**

Table 5: Number of docker images

9.3 DockerGate Generated Policy Analysis on Community Images

Analysis of Docker Images with DockerGate and SystemTap. The following table shows the number of System calls captured by DockerGate framework and Number of System Calls capture during creation time using System Tap.The Remarks section explains the results of the container state after loading the seccomp profile generated and creation a container. Based on the Remarks the case is considered as SUCCESS or FAILED.

Table 6: Evaluation Statistics for policies generated using DockerGate

Image Name	DockerGate Policy	SystemTap Captured	Remarks	Result
adbrowne/dynamodb-eventstore	217	119	**Missed timer_settime() in policy**	FAILED
bkeithryder/ls	217	118	Container Crashed	FAILED
burl/docker-node-base	16	52	Container Created	FAILED
carverdamien/convoy	230	117	Entrypoint Failed	FAILED
ciena/cord-maas-bootstrap	220	118	Entrypoint Failed	FAILED
cloudposse/ubuntu-vps	221	119	Container is up and running	SUCCESS
davislaboratory/docker-rstudio-server	233	118	Entrypoint Failed	FAILED
diogoalves86/testes-docker	211	117	Container is up and running	SUCCESS
dockerinaction/ch6_htop	216	117	Container is up and running	SUCCESS
dolaterio/parrot	230	119	Container is up and running	SUCCESS
eastuni/apachemodjk	237	115	Container Created	FAILED
fergalmoran/dss.radio	231	119	Entrypoint Failed	FAILED
geoffreyplitt/docker-sabnzb	207	118	Server is up and running	SUCCESS
gooddingo/dockercon_autotest	216	120	Container is up and running	SUCCESS
granthbr/mesos-base	217	118	Container is up and running	SUCCESS
httpd	207	120	Server is up and running	SUCCESS
kubilus1/gendev	219	117	Container is up and running	SUCCESS
manubing/rpi-fluentd	232	117	Container is up and running	SUCCESS
mattinclude/certdragon	238	118	Server is up and running	SUCCESS
muralikflora/ubuntu	233	119	Container is up and running	SUCCESS
neiltheblue/simplenbdwrapper	238	124	Container is up and running	SUCCESS
ninthwalker/plexreport	17	45	Entrypoint Failed	FAILED
peterdemin/gotty-irssi	231	118	Server is up and running	SUCCESS
subodhdharma/dockertest	210	119	Container is up and running	SUCCESS
tomcat	208	119	Server is up and running	SUCCESS
vcatechnology/debian	202	120	Container is up and running	SUCCESS
zaraki673/docker-mysql	197	116	Container Created	FAILED
zooniverse/jiscmail-bounces	230	117	Entrypoint Failed	FAILED
guhuaping/ats-5.2.0-ubuntu	233	120	Container is up and running	SUCCESS
greeninja/fictional-computing-machine	225	117	Server is up and running	SUCCESS
sspickle/assessdb	220	120	Server is up and running	SUCCESS
simright/calculixir	208	118	Containers Crashed 139 Normal	PARTIAL
scottg489/diff-info-service	210	118	Container is up and running	SUCCESS
idgis/awstats	216	118	Server is up and running	SUCCESS
jtassin/transmission	216	119	Containers Crashed 1 - Normal	PARTIAL
ojixzzz/tiptopgallery	220	119	Server is up and running	SUCCESS
openstf/stf	214	119	Containers crashed 139 - Normal	PARTIAL
ajw107/docker-plex	215	120	Server is up and running	SUCCESS
1and1internet/ubuntu-16-apache-php-5.6-zend-2	214	119	Server is up and running	SUCCESS
ascdc/mirador	216	119	Container is up and running	SUCCESS
cloudnthings/rocketsay	216	117	Container Created	SUCCESS
cmptech/auto_ubuntu1610_nodejs_sharessl	151	118	Container is up and running	SUCCESS

djocker/sshd	221	122	Server is up and running	SUCCESS
grahamgilbert/ubuntu-ssh-s3fs	217	118	**Missed fork() in the policy**	FAILED
hornoo/docker-lab3.1	210	119	Entrypoint Failed	FAILED
kurron/docker-docker-compose	216	122	Containers crashed 1 Normal	SUCCESS
lambertod/todobackend-base	220	119	Containers crashed 0 Normal	SUCCESS
osbert/dash-listen	221	119	Container is up and running	SUCCESS
pivotalservices/cfops	216	118	Container is up and running	SUCCESS
smarp/committee	216	118	Containers crashed 159 Normal	SUCCESS
quantumobject/docker-xowa	223	119	Container is up and running	SUCCESS
tgraf/cilium	210	118	Container is up and running	SUCCESS
0xfireball/che-0xfireball-dotnet	217	119	Server is up and running	SUCCESS
acdaic4v/ubuntu-perl-base	210	122	Container is up and running	SUCCESS
bkadali/dnreasysql	216	119	Container is up and running	SUCCESS
dannysu/miniflux	209	125	Container is up and running	SUCCESS
drewrobb/bamboo	216	118	Container is up and running	SUCCESS
harryr/emptyflask	221	118	Container is up and running	SUCCESS
qautomatron/docker-browsermob-proxy	210	119	Container is up and running	SUCCESS
jkatzer/sandstorm	216	120	Container is up and running	SUCCESS
nishidayuya/docker-vagrant-ubuntu	221	119	Container is up and running	SUCCESS
andresriancho/django-uwsgi-nginx-ssh	223	124	Container is up and running	SUCCESS
antespi/docker-imap-devel	210	118	**Missed epoll_create() in policy**	FAILED
checkoutcrypto/site	219	119	Container is up and running	SUCCESS
julienvey/tempest-in-docker	17	118	Container Created	FAILED
leisurelink/alpine-consul	17	118	Container Created	FAILED
tacid/docker-ubuntu-sshd	241	121	Container is up and running	SUCCESS
elbaschid/headless-firefox	17	117	Container Created	FAILED
sublimino/mnemosyne	215	120	Command Line Test Success	SUCCESS
jangorecki/drd-pkg	120	116	Command Line Test Success	SUCCESS
universalcore/unicore-cms-fflangola	210	120	Command Line Test Success	SUCCESS
carolinedocker/caroline_db	202	117	Container crashed with 139	FAILED
spunjabi/docker-whale	218	119	Command Line Test Success	SUCCESS
cmathre/clay-test	218	120	Container is up and running	SUCCESS
automatikdonn/krakenceph	17	89	Container Created	FAILED
mohamedamjad/nginx-session-keeper	198	119	Command Line Test Success	SUCCESS
joshdover/kafka	17	113	Created	FAILED
spaziodati/neologism	220	122	Container is up and running	SUCCESS
asakaguchi/magellan-tdiary	17	115	Container Created	FAILED
bowlingx/docker-nginx-php	221	118	Command Line Test Success	SUCCESS
banterability/env-test-ops	218	120	Command Line Test Success	SUCCESS
satanas78/docker-whale	218	119	Command Line Test Success	SUCCESS
daveisrising/squad	221	117	Command Line Test Success	SUCCESS
jamalshahverdiev/docker-mynode	209	120	Command Line Test Success	SUCCESS
almagest/meteord-ffmpeg-gm	212	119	Container Crashed	FAILED
perspicaio/testapp	17	127	Container Created	FAILED
programlabbet/mini-ruby	17	104	Container Created	FAILED
sirile/scala-boot-test	17	108	Container Created	FAILED
gesellix/morgue	220	119	Container is up and running	SUCCESS
devries/bottle	214	117	Container is up and running	SUCCESS
ness2u/cockpit	220	120	Container is up and running	SUCCESS
thiagobraga/docker-composer	204	118	Container is up and running	SUCCESS
networkedinsights/ni-postgres	17	114	Container Created	FAILED

abienvenu/kyela	204	119	Command Line Test Success	SUCCESS
estherr/ubuntu_apache	210	120	Command Line Test Success	SUCCESS
chug/ubuntu13.04x64	17	105	Container Created	FAILED
adolphlwq/ubuntu_hadoop_pseudo	219	120	Container is up and running	SUCCESS
wywincl/ubuntu-cn	218	117	Container is up and running	SUCCESS
arnaudblancher/docker-ubuntu14-jmeter	219	119	Container is up and running	SUCCESS
alexisvincent/debian	208	122	Container is up and running	SUCCESS
simplexsys/debian-zulu-jdk	203	117	Container is up and running	SUCCESS
peasantspring/ubuntu-ping	218	119	Container is up and running	SUCCESS
alanfranz/fwd-debian-wheezy	197	116	Container is up and running	SUCCESS
canglaoshi/docker-ubuntu	219	120	Command Line Test Success	SUCCESS
davidsiaw/ubuntu-apache2-php5-fuel-mysql	221	117	Command Line Test Success	SUCCESS
derbyclub/ubuntu-apache	218	116	Container is up and running	SUCCESS
thenodi/docker-ubuntu1404-valet	218	122	Container is up and running	SUCCESS
abigail/debian-vagrant-puppet	208	120	Container is up and running	SUCCESS
britchey/ubuntuproject	210	118	Command Line Test Success	SUCCESS
sotoiwa/ubuntu	218	117	Container is up and running	SUCCESS