# DockerGate: Automated Seccomp Policy Generation for Docker Images

Mohit Goyal
mgoyal@ncsu.edu

Subodh Dharmadhikari
ssdharma@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 applications. 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 platform that can statically analyze a Docker image to identify the system calls in the executable binaries. Our key insight here is that by analyzing the reachable, executable code, tighter Seccomp profiles can be generated which would reduce the attack surface of the Docker Container.The static analysis framework was developed as a pipeline that first generates a graph of the filesystem and then traverses the graph, analyzing each node for system calls. We test DockerGate by generating Seccomp policies for 40 Docker Images and running the Images within a container with the Seccomp policy applied. We achieve an average size of 230 system calls for a Seccomp policy as opposed to the Default policy of 300 system calls. We also manage to achieve basic functionality for 39 out of 42 containers.

## 1 Introduction

Linux Containers are lightweight virtual machine processes 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 [23].

Docker [4] is a Linux Container Management technology that leverages the use of Linux containers. The containers are based on LXC[24] and provide a virtualization framework for software as well as hardware [22]. 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 [6] published by various vendors and community on Dockerhub [5].The availability of Docker framework for different operating systems as well as different levels of infrastructure ranging from desktop to cloud service providers makes it more popular, easy and flexible to install and use. Based on a survey conducted in April 2017, around 12,947 companies are currently using Docker [26]. Moreover, there is research being done to introduce Docker to High Performance Computing [27]. Docker has also been considered as a solution to the problem of not having access to reproducible research, especially in Computer Systems Research [19, 18].

Docker containers use runC [25] to start a container, which takes care of managing all the components required for a container which consist of cgroups, namespaces and capabilities [8]. In the most recent release of docker v1.10, Docker introduced a feature to support Seccomp [7] 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. As per the developers for Docker, Seccomp was introduced to reduce the attack surface exposed by the kernel via vulnerable system calls. A notable vulnerability [1] showed that keyctl system call was vulnerable and could cause denial of service and escalation of privilege. So, to protect the kernel from such vulnerabities being exploited by using a Docker container as a vector, Seccomp support added. A default profile is loaded implicitly which blocks access to 44 system calls and allows over 300 system calls [7]. 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 [15]. 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 [9] from inside a container.

To generate image-specific Seccomp policies and protect the host, we propose **DockerGate**[1], a platform 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 42 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.* By using Python APIs exposed by reverse-engineering tool Radare2, we analyze every executable binary and the required functions in their linked libraries to create a cumulative list of system calls that can be called.

- *We evaluate DockerGate on a random sample of 42 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 [4] is an application that helps software developers in deploying their applications across a wide variety of host operating systems without needing 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 [6]. 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 [6], 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 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 [28].

Docker Hub [5] and Docker Store are central repositories that maintain several official and community Docker images. Using these repositories, a Docker user

[1]Source code and Data set available at `https://github.com/mohitgoyal2011/dockergate2`

can maintain 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 [11] is a feature in Linux kernel that makes 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 [11] was developed as an extension to Seccomp that could be used with a configurable policy to filter system calls using Berkeley packet filter [31] 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 [3].

### 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 doesnt 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. For example, CVE-2017-15265 [2] describes a use-after-free vulnerability while issuing a ioctl to any sound device. This could lead to escalation of privilege for the attacker. Now, if this vulnerability were to be combined with a vulnerable web application that allowed the attacker to execute custom code within its context, the attacker could use this vulnerability to achieve root privileges. However, most applications may never need to make an ioctl call to a device. So, if we are able to identify the applications that don't need the ioctl system calls to function properly, we can remove that capability from the respective containers by blocking the ioctl calls within the Seccomp policy. This reduces the attack surface for the kernel and keeps the functionality intact at the same time. Lastly, it is assumed that a Docker user would never list down the system calls or capabilities on her own. As a matter of fact, this assumption was mentioned in the original issue opened [2] for the Seccomp support for Docker. It was also mentioned that automatic policy generation can be added later. But we haven't come across any pull requests or tickets that work towards the same. So, we consider our work as a first step in that direction.

### 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. This presents an interesting situation as the attacker gains the ability to arbitrarily execute code within the Docker Container. The attacker could now also use System Calls as a vector to gain access to the kernel. We define the attackers goal as 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 Docker Images 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 that is tighter than the default Seccomp profile.

However,performing static analysis on docker images presents several implementational 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.

[2] https://github.com/moby/moby/issues/17142

*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

*Integrating sophisticated binary analysis tools into the pipeline:* To ensure that complete analysis is performed, we use Radare2 [13] as our disassembler. While it has many advantages, the major challenge we faced was to integrate it within DockerGate and interact with the files within a Docker container. Fortunately, Radare2 and Docker consist of APIs that allowed us to do the same.

To create our dataset, we wrote a web scraper using Scrapy [14] that could query Docker Hub and scrape the names of 2000 random Docker Images. We filtered these Docker Images by ensuring these were community contributed images and had had a considerable number of ”pulls” from users. This ensured we were working on a relevant set of Docker Images that are being used in the world today. We then used DockerGate to analyze these Images. Eash image was first mounted on to a Docker Container and traversed for all executable files and linked libraries within the Container. Then each traversed file is copied from the container to the host and then analyzed for the system calls. Based on the intermediate results of the DockerGate scripts and the database of system call mappings we generate a policy using the JSON [10] format, as specified in docker docs [7].

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 ”Docker Run” to fetch the Docker Image from the Docker Repository and is then mounted on a container. The container is mounted with some additional Python scripts that traverse through the entire filesystem and generate a call graph. This call graph links all binaries to their linked libraries. Then, the call graph is traversed in the host and each file traversed is copied on to the host to be analyzed by Radare2. The Python script using Radare2 APIs basically checks every function within the binary for system calls by looking for direct system calls, or calls to library functions that would eventually make these system calls

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.

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.

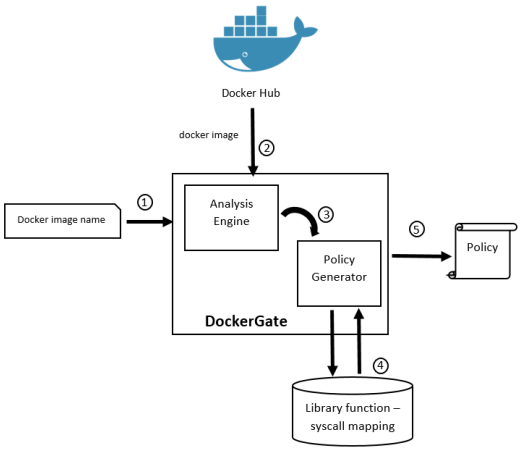


Figure 1: A high-level architecture of our approach

# 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. In this section, we describe the architecture of this platform and how it can be extended to perform other experiments with different types of files.

## 4.1 Overview

We developed DockerGate as a static analysis platform that could go through all the executable code in a Docker Image and can extract what system calls each executable requires. The process involves mounting the Docker Image onto a container, traversing through the filesystem and generating a call/linked graph and then individually analyzing each executable and their associated libraries. So the Analysis process can be divided into three phases :

- Phase 1: Call graph generation for Docker Image
- Phase 2: Traversing the Call graph and analyzing each file
- Phase 3: Seccomp policy is written

The following describes how each phase is executed and what tools are used in each phase and how the entire solution is implemented.

## 4.2 Phase 1: Call Graph Generation

The initial pass is done by mounting the Docker Image and doing a depth first type of filesystem traversal. Wherever an executable file is encountered, we add it as a blue node to the traversal graph. Then, we check

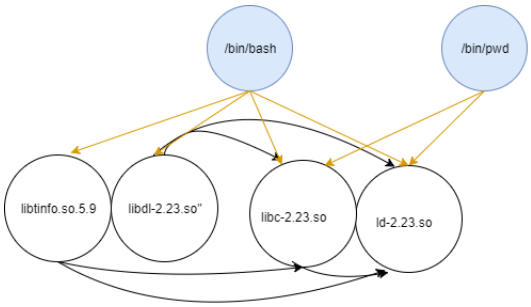


Figure 2: Example Call Graph

whether this executable file is dynamically linked and if yes, which libraries it is linked to. These libraries are also added to the graph as white nodes. These libraries are also recursively checked for other linked libraries. So we get a multiple component graph as can be 2.

The above is done by sharing a folder containing the file traversal code and graph generation code between the host and container. The folder contains a statically-linked version of Python with the Graphviz module installed. By including all of these required tools, there is no dependency for the code that is required from the container. This makes DockerGate platform-independent.

The final output graph is stored as a DOT file and is copied back to the host. Upon visualization of the graph, it can be seen that there are no directed paths from one ELF binary to another and the graphs are rather disconnected. We believe that once we include analysis of bash scripts and Python files, the connectivity will improve and we will be able to produce tighter Seccomp policies. As of now, we are analyzing all executables as we are assuming all of the code is reachable. However, for Docker Images that are Web Servers and are based on some base OS like Ubuntu or CentOS, many of the executables are present just as dead code and are almost never executed. So, we believe that if we follow the execution path or call-graph from the entrypoint of the Docker Image, we can produce Seccomp policies that only allow the system calls that are present in that path. So, unusable code like bash (in Tomcat) wont even be analyzed. The code is such that once we are able to find a suitable method that can analyze bash scripts and Python files, it can be added as a separate module to DockerGate.

### 4.3 Phase 2: Call-Graph Traversal and Analysis

Once the DOT graph of the file traversal is generated, the rest of the analysis happens in the host itself. We use Radare2 as our primary analysis tool. In the initial version of DockerGate, we had used based text analysis of the object files of the executable files. During the evaluation, we found that it had been missing several system calls and wasnt as sophisticated as one would like. So, in this iteration, we decided to switch to Radare2. Radare2 provides Python APIs that can dissect an executable file or linked library and provide the assembly code for every function. Using a combination of these APIs with text analysis, we search for all the system calls being called in that binary or library function. 3 shows how the analysis is done once an executable or ”blue” node is reached.

During the traversal, if DockerGate encounters a ”blue” node or an executable binary, it first ensures that all of its linked libraries have already been analyzed. So, it starts a depth first traversal from that node. Once, it encounters a white node (library) with all the dependencies analyzed or no dependencies, it invokes the Analysis module to disassemble this library and process its functions. Using Radare2 Python APIs, DockerGate is able to get a list of all the functions defined a library. A function can invoke a system call in two ways - by calling it directly or calling another function that eventually calls the system call.

For the first way, as observed by Rauti et al [30], in Linux Architecture, a system call in the object code can be identified by a SYSCALL command. When such a 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. We can also encounter a command sequence that may look like ” mov 0x04, $ecx; mov $ecx, $eax”. So, during the backtracking, we ensure that we track the correct register until we find a hardcoded value. The attentive reader might point out that there are other methods of manipulating a register like using *add*,*xor*. However, while manually analyzing the some of the libraries, we found very few instances of instructions other than *mov* being used to manipulate the value of EAX before invoking SYSCALL. While it is a limitation that threatens the completeness of our analysis, we find that it does not affect the generated policy in any manner as the affected System call is covered else where.

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 again use Radare2 to list down all the functions a particular function invokes. These called functions can be from within the same library or be present in the externally linked libraries. For the latter, since we have already analyzed those libraries, we get a list of

the system calls from the database by querying it for the linked-library hash and function name. If the function belongs to the same library and hasn't been analyzed, we recursively analyze the callee function first before resuming the analysis of the caller function. For example, if function A called function B and function B hadn't been seen before, we analyze function B and return the list of system calls to be appended to the system calls in function A's list.

Once all dependencies of a blue node have been analyzed, Radare2 is used to list all the functions being called by the executable binary. Since we already have all the system calls being called by those functions in the database, we simply query the database for those functions. The system calls returned by this query are added to the intermediate set of system calls DockerGate maintains to eventually be written as the Seccomp Policy. Each executable is analyzed in a similar fashion and the system calls invoked the binaries are added to the intermediate set.

We maintain an SQLLite3 [16] database to record all the analysis we have done. This is to ensure that if DockerGate encounters any library or binary that it has analyzed before, it wouldn't have to do the analysis again. We uniquely identify a binary or library by using a SHA 256 hash of the file as a primary key. This strategy proves effective in reducing the time spent in analyzing the executable code of a Docker Image. This is because a Docker Image can be based on another base Docker Image. So, a large subset of the libraries in the current Docker Image would have been inherited from the base Docker Image. If DockerGate has already analyzed the base, most of the analysis of the libraries would be present in the database already. And therefore, the analysis phase would involve simply looking up the database. For example, the Tomcat Docker Image is based on Ubuntu Docker Image. So, since DockerGate would had already analyzed Ubuntu, the analysis for Tomcat took less time than if we would have analyzed everything from the ground up. The figure shows the structure of the SQLLite3 database and an example of the data included.

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.

While we have currently implemented full graph traversal that will lead to the analysis of all executable code available in the Docker Image, we also plan on trying to further tighten the Seccomp policies by following

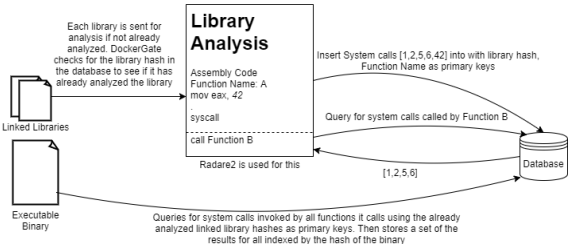


Figure 3: Analysis Phase. This is when a dynamically linked executable binary is reached during the traversal of the call graph. First its linked libraries are analyzed and then the executable binary itself.

a logical path of execution. We believe a lot of the executable code within a Docker Image would have a very low chance of being executed if it is not reachable from the defined entry point of the Docker. So, if we don't include the system calls invoked by the unreachable binaries, we can further reduce the System calls in the Seccomp profiles and reduce the attack surface for the container.

### 4.4 Phase 3: Policy Generation

The Policy Generator is final phase in the DockerGate framework. It generates the policy by going through the cumulative system calls required by all the traversed executable binaries. We use "ausyscall" to translate the system call number to the relevant system call name. The final policy is saved a a JSON file with the Default Action being DENY and the action for each of these system call being "ALLOW". An example of this can be seen in the following figure.

## 5 Evaluation

In this section, we describe how we evaluate the Seccomp policies generated by DockerGate. We first obtain Docker Image names from Docker Hub using a web scraper and then run each image in a Docker Container with the Seccomp policy applied.

### 5.1 Characterizing Data Set

Docker Hub currently hosts 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 [14], 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 2000 Docker image names from those 96,000 names. We ran DockerGate on 42 Docker Images.

The time it took to completely analyze an image varied from fifteen minutes to more than a day. This was governed by the size of the Docker Image and whether the libraries and binaries had been seen before or not. Since we ran DockerGate on a random sample, most of the libraries contained within the images hadn't been analyzed before.

The following sub-section provides the details about how the policies generated were evaluated.

## 5.2 DockerGate Policy Evaluation

After issuing a Docker run command, a 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 writeable 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 image 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. If the container exited with a non-zero code or appeared to hang-up, we considered it a failure in the Seccomp policy.

We also ran several common commands within each of the containers and checked the output of the commands in the container. The commands included "ls", "cat", "/bin/sh", "echo", "uname". While this list of commands isn't exhaustive, it does provide a basic sanity test for the applicability of our generated Seccomp profiles to the containers. The commands ran successfully within the containers that were UP and failed in the others.

| **Container Status** | **Number of docker images** |
|---|---|
| Exit Status 0 | 0 |
| Up (live container) | 39 |
| Created | 1 |
| Exit Status != 0 | 2 |
| **Total** | **42** |

Table 1: Container status after loading them with DockerGate generated Seccomp policy

The policies generated contained at an average of 230 system calls in the Seccomp policy. With an exception in an image burl/docker-node-base which generated a Seccomp policy of 16 system calls resulting in a failed state (Created). A summary of container states and success of the Seccomp policy to spawn a successful container can be given in 5.2:

From the Table 5.2 we can see that almost all policies were applied successfully except for three. They failed with an "Operation not permitted" indicating that some system calls were not included in the Seccomp profile, but were essential to the operation of the container.

To determine which system calls were used during the execution of the container we used a tool called SystemTap [21]. 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 about 20 images out of the ones used above and obtained the system calls called during the process. In the 20 images above, we found that an average of 118 system calls are actually required for spawning and initiating the startup 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 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 230.

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 230 system calls identified above and reduce the attack surface of the host kernel.

# 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. While we have included all executable code within the generated graph file, we currently do not have a way to statically analyze these files.

During the experiment, we found that even though the major and minor versions of a particular library might be the same, the content isn't always identical. For example, DockerGate may come across libc-2.23.so in multiple Docker Images. However, it is not necessary that the libc-2.23.so file found in the Docker Image for Ubuntu is the same for that found in Tomcat. So, we found that the same library was being processed multiple times despite having a database to prevent that. While each of them was a different file, we did not find any evidence of the functions being analyzed differently. This costed us time as many Docker Images had to be analyzed from the ground up. We did find many instances where the same library file was being used. But, we believe this can be improved upon and should be pursued further.

A binary can be dynamically linked to several libraries. When we search for an imported symbol or function called by the binary, we look through all the linked libraries. However, it is possible that the same function might be defined in multiple libraries. Our query consisted of all of the hashes of the linked libraries and the function name. But, if the query returned multiple results, we picked the first one. In our current evaluation, we haven't encountered a point where multiple results are being returned. But this is theoretically possible and our evaluation set is too small to check the possibility of the same.

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 believe even tighter Seccomp policies can be achieved if we only include the system calls invoked by a subset of the executable binaries. This subset should include only those binaries that can logically be reached from the entry-point of the Docker Container. While our infrastructure allows for such an experiment by tweaking the graph traversal, we couldn't go forward with it because of time constraints. However, we do plan to pursue this experiment in the future.

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 Android Applications [12] 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.

The use of Radare2 and Graphviz with DockerGate makes the platform more versatile. GraphViz allows us to add more attributes to the generated filesystem graph which can provide us more insight to the kind of application the Docker Image executes. For example, by analyzing the various connected components within the graph, we can create multiple Seccomp policies for the same Docker Image based on which application is being invoked. For example, if the Tomcat container is run as a web server, the Seccomp policy that is applied on the container can contain the system calls pertaining to the web server executables. If it is run in a configuration mode, the Seccomp policy can contain the system calls pertaining to the administrative tools that would be run. While this may be very non-deterministic, it may allow us to reduce the attack surface on some popular Docker Images.

Since Radare2 can be used to analyze branches within functions, we might be able to figure out whether a particular syscall instruction is reachable or not. It is possible that a binary may never follow that control flow and the system call in question may never be executed. This will produce tighter Seccomp policies. However, we feel this is an ambitious direction to take and it may not yield the results desired.

# 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 had been proposed earlier where different kernel features can be

used to secure the container. Seccomp is one such feature on which DockerGate proposes its solution.

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 [20], 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. The project aims to generate Seccomp profiles by monitoring the interaction between a user and the Docker Container's HTTP APIs (if it has any). Since it uses Dynamic analysis, it depends upon how much code the user is able to trigger. This served to be one of the major limitations of this project.

One more kernel feature that can be exploited for container hardening is AppArmor. LiCShield [29] 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. The monitored and controlled the processes forked by docker daemon.

There have been proposals [17] 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.

# 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] Cve-2016 0728. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0728.

[2] Cve-2017-15265. https://access.redhat.com/security/cve/cve-2017-15265.

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

[4] Docker. https://www.docker.com.

[5] Docker hub. https://hub.docker.com.

[6] Docker images and containers. https://linux.die.net/man/1/objdump.

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

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

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

[10] Json. http://www.json.org/.

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

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

[13] Radare2. http://rada.re/r/.

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

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

[16] Sqlite3. https://www.sqlite.org/.

[17] 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.

[18] 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.

[19] 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.

[20] docker slim. docker-slim. `https://github.com/docker-slim/docker-slim`.

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

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

[23] A. Grattafiori. Understanding and hardening linux containers. `https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-10pdf/`.

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

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

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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 2: 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 3: 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.

| Image Name | DockerGate Policy | Endpoint Execution | Common Commands | Base OS |
|---|---|---|---|---|
| abienvenu/kyela | 229 | Success | Success | Debian GNU/Linux 8 (jessie) |
| abnerzheng/catnip | 236 | Success | Success | Debian GNU/Linux 8 (jessie) |
| alpine | 152 | Failed (Created) | Failed (Created) | Alpine |
| andresriancho/django-uwsgi-nginx-ssh | 242 | Success | Success | Ubuntu 14.04.2 LTS |
| automatikdonn/krakenceph | 169 | Success | Success | Fedora 26 (Twenty Six) |
| banterability/env-test-ops | 235 | Success | Success | Ubuntu 14.04.4 LTS |
| bowlingx/docker-nginx-php | 238 | Success | Success | Ubuntu 14.04.3 LTS |
| centos | 184 | Success | Success | CentOS Linux 7 (Core) |
| cmathre/clay-test | 236 | Success | Success | Ubuntu 14.04.2 LTS |
| coralproject/elkhorn | 234 | Success | Success | Debian GNU/Linux 8 (jessie) |
| daveisrising/squad | 239 | Success | Success | Ubuntu 14.04.2 LTS |
| debian | 226 | Success | Success | Debian GNU/Linux 9 (stretch) |
| derrickh/0419-2 | 242 | Success | Success | Ubuntu 14.04.1 LTS |
| fergalmoran/dss.radio | 239 | | | |
| garrettrayj/mapnik-docker | 245 | Success | Success | CentOS Linux 7 (Core) |
| gesellix/morgue | 242 | Success | Success | Ubuntu 14.04.4 LTS |
| idgis/awstats | 236 | Success | Success | Ubuntu 14.04.5 LTS |
| jangorec/kidrd-pkg | 242 | Success | Success | Debian GNU/Linux buster/sid |
| jasonhcwong_erlang. | 240 | Success | Success | Ubuntu 14.04.3 LTS |
| joshdover/kafka | 211 | Success | Success | Alpine |
| kakakikikeke/m2i | 237 | Success | Success | Debian GNU/Linux 8 (jessie) |
| mohamedamjad/nginx-session-keeper. | 220 | Success | Success | Debian GNU/Linux 7 (wheezy) |
| moritanosuke/wallabag-docker | 228 | Success | Success | Debian GNU/Linux 8 (jessie) |
| muzili/bugzilla | 227 | Success | Success | CentOS Linux 7 (Core) |
| ness2u/cockpit | 238 | Success | Success | Ubuntu 14.04.3 LTS |
| nginx | 221 | Success | Success | Debian GNU/Linux 9 (stretch) |
| perspica/perspica-collector | 215 | Success | Success | |
| perspicaio/testapp | 138 | Failed | Failed | Unknown |
| programlabbet/mini-ruby. | 150 | Failed | Failed | Scratch |
| projectfalcon/gradle-cache-resource | 232 | Success | Success | Debian GNU/Linux 8 (jessie) |
| raulmz/adempiere_380 | 240 | Success | Success | Ubuntu 16.04 LTS |
| satanas78/docker-whale | 235 | Success | Success | Ubuntu 14.04.2 LTS |
| sherifmedhat_node | 239 | Success | Success | Debian GNU/Linux 8 (jessie) |
| sirile/scala-boot-test | 138 | Failed | Failed | Unknown |
| smdion/docker-upstatsboard | 238 | Success | Success | Ubuntu 14.04.1 LTS |
| spaziodati/neologism | 242 | Success | Success | Ubuntu 14.04.1 LTS |
| spunjabi_docker-whale | 236 | Success | Success | Ubuntu 14.04.2 LTS |
| subodhdharma/dockertest | 227 | Success | Success | Ubuntu 16.04.2 LTS |
| tectoro/golang-nested-docker | 239 | Success | Success | Debian GNU/Linux 8 (jessie) |
| thongdong7/docker-selenium | 239 | Success | Success | Ubuntu 14.04.3 LTS |
| tomcat | 229 | Success | Success | Debian GNU/Linux 9 (stretch) |
| universalcore/unicore-cms-fflangola | 238 | Success | Success | Debian GNU/Linux 8 (jessie) |
| z3ntu/android-n-build-env | 234 | Success | Success | Ubuntu 16.04.3 LTS |

Table 4: Evaluation Results for DockerGate