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 200 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 [31], OpenVZ [16], Docker[38] etc.). 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 [30]. The ability to virtualize an operating system into a single process without any additional virtual kernel makes the containers speed up the boot process [39] and makes the application easily available.

Docker [2] is one of the technologies that leverages the use of Linux containers. The containers are based on LXC[31] and provide a virtualization framework for software level as well as hardware level [29]. 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 [4] published by various vendors and community on Dockerhub [3].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 [34]. Moreover, there is research being done to introduce Docker to High Performance Computing [35]. Docker has also been considered as a solution to the problem of not having access to reproducible research, especially in Computer Systems Research [26, 25].

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. Common attacks like escalation-of-privilege attacks [32] 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 [5] is also active in mitigating new threats on a regular basis. As of April 2017, there have been only 14 CVEs registered since 2014 [21]. The attack surface of docker is mainly on runC process, docker daemon and containerd process which are the building blocks of Docker containers

Docker containers use runC [33] to start a container, which takes care of managing all the components required for a container which consist of cgroups, namespaces and capabilities [7]. In the most recent release of docker v1.17, docker introduced a feature to support Seccomp [6] 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 [6]. 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 [18]. 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**, 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 200 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 [20].* By using various Linux-based Binary Analysis tools [14, 13, 15], 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 20 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 [2] 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 [4]. 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 [4], 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 [36].

Docker Hub [3] and Docker Store are central repositories that maintain several official and community Docker images. Using these repositories, a Docker user 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 [12] 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 [12] was developed as an extension to Seccomp that could be used with a configurable policy to filter system calls using Berkeley packet filter [42] rules.

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 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. We had several decisions to make before going about implementing DockerGate. We decided to perform small experiments before going ahead with the actual development of the prototype.

2.3.1 Dynamic Analysis or Static Analysis ?

The first decision to take was to decide whether dynamic analysis or static analysis would be well suited for this solution. While dynamic analysis by logging all system calls made to the kernel would have been easier to implement, triggering all the executable code for every separate image would have been an almost unfeasible task. We still decided to experiment with a tool called strace [19] to figure out what system calls were required

We conducted a very naive experiment to determine the system calls used by a container. We used the strace utility in Linux to trace the system calls used by a container. We experimented with the hello-world image of docker and 'straced' the docker container for the image.

In this experiment, the strace identified 32 unique system calls for running this container. To verify that these 32 system calls were sufficient to successfully spawn the container, with default deny policy and allowing only 32 system calls identified we attempted to create and run a container, but it would fail to run. So, we decided to define Seccomp policy which would contain the 32 system calls and all the rest of 304 system calls. We followed an iterative approach where we would remove a system call and check if the container would run successfully. If container failed, we considered that system call as required system call for running a containers. If container ran successfully, we removed the system call from the list. With this iterative approach we shortlisted a total of 88 system calls.

Thus, concluding that strace utility is not capable of capturing all the system calls, we decided in performing static analysis on the docker image would provide better results as it would be all-inclusive of the executable code in the image.

2.3.2 Files to Analyze

We then proceeded to implement a proof-of-concept prototype for DockerGate. Because of the limitation of time, we decided to focus only on executable binaries that were present inside the image. This decision was supported by an initial analysis of the file composition of a random sample of 100 Docker images. It was found that Docker Images generally contain very little Source code and are mostly comprised of executable binaries in the form of ELF executables or JAR executable files. We decided to focus on ELF-executable files as they were the more prevalent types of files in the random sample.

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.

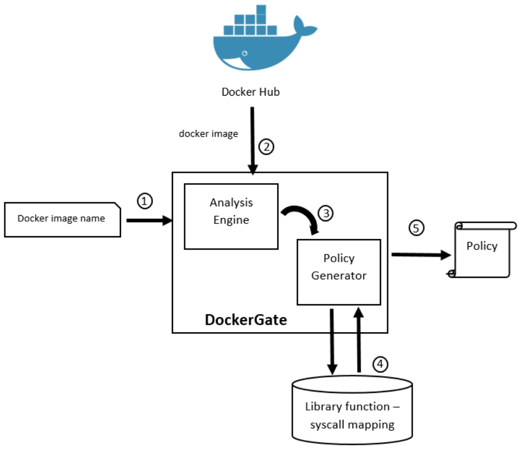


Figure 1: A high-level architecture of our approach

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.

In DockerGate, we analyze the docker image and identify the binary packages which can make various system calls to the host. Performing the 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

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 [41] in developing and downloading data set for their experiments. We used banyanops/collector [23] framework 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 in json format, as specified in docker docs [6].

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 frameworks ability to pull the docker image from Docker hub. The banyanops framework is configured to execute scripts to perform static analysis within the container. In this stage, banyanops iteratively creates a new containers 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 [10], pthread library functions. 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 library function call database consists of the records for standard libraries. The framework also supports analysis of any unconventional libraries found within a image/container. After the analysis of the library the database is updated accordingly.

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. The following describes its 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 [23] 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.

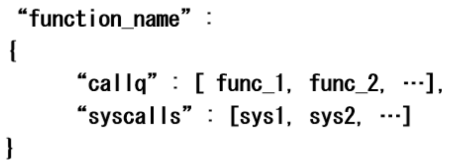
The custom scripts used to perform analysis have the following objectives:

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 [13] 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 [14]. 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 [15] to analyze and perform the mapping of library functions and system calls.



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

Figure 2: System Call Database Structure

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 containing function call to system call mapping. 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 [11] 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 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.

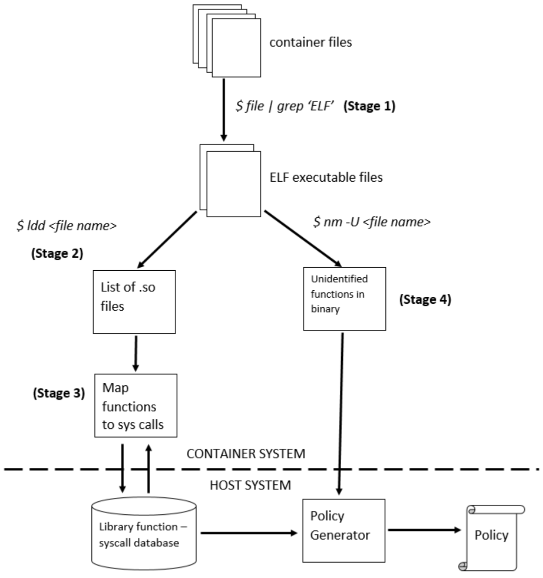


Figure 3: A detailed design

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 [8]. 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 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 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 [40], 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 [17], 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 200 Docker image names from those 96,000 names. We conduct our further experiments on this sample of 200 Docker images.

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.

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

As can be observed from Table 5.1, out of total 437 Dockerfiles, 118 images have openjdk base images. Further we deep dived to analyse the original base image of openjdk. It was found that all these images were based on either ubuntu, alpine or debian images as their most lowest level base image distribute almost equally. The majority of the images were based on ubuntu and debian. A complete list of the distribution of images can found in the Appendix.

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 base image of ubuntu and debian. While about 100 images are based on alpine and rest are based on images of fedora, centos, scratch.

Since the number of 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.

5.2 DockerGate Policy Evaluation

To evaluate the DockerGate we have considered only docker images with ubuntu or debian as the base images. We use the DockerGate framework by providing the name of a docker image as input to DockerGate, further it performs the analysis and updates the global database of system calls if required and generates a seccomp policy.

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 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 loading the generated policy and attempt to create and execute a simple container. If the status of the container is Up or it exits with exit code 0, we assume that container was created successfully and executed the application within the container or the command line provided as the input to it. If the container

Container Status	Number of docker images
Exit Status 0	7
Up (live container)	2
Created	2
Exit Status != 0	9
Total	**20**

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

resulted to land in any other states that is considered as a failure.

From a set of Ubuntu based docker images, we performed ran DockerGate on randomly selected 20 community images. The images contained applications ranging from simple binary files to complete Databases and Web Servers.

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

From the Table 5.2 we can see that 9 out of 20 containers exited in successful state. While 2 of them remained in Created state. And the rest of the 9 containers failed for various reasons.

To determine which system calls were used during the execution of the container we used a tool called SystemTap [28]. 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 20 images 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. 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 Android Applications[cite] 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 [24] 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 focused on the container hardening, which exploits the kernel capabilities and features to protect and secure the containers as well as limit the blast radius in case of any breach. 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 [27], 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 [37] 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 [22] 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] Default docker seccomp policy. https://github.com/moby/moby/blob/master/profiles/seccomp/default.json.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] Openvz. https://openvz.org.

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

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

[19] *strace(1) Linux's manual page*.

[20] Ubuntu. `https://www.ubuntu.com`.

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

[22] 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.

[23] banyanops. banyanops/collector. `https://github.com/banyanops/collector`.

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

[25] 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.

[26] 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.

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

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

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

[30] 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/`.

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

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

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

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

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

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.