Container Hardening Through Automated Seccomp Profiling

Nuno Lopes
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2020

Orientador
Rolando da Silva Martins, Faculdade de Ciências da Universidade do Porto

Coorientador
Manuel Correia, Faculdade de Ciências da Universidade do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas
pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/________

Acknowledgements

I would like to thank my family for their continuous support and for their counsel. With a very special mention to my sister that has always been by my side.

A very special thanks to all my friends, especially to the ones in Merendeiros. Also to everyone at the Information Security Master's Degree, in particularly to André Cirne and Fábio Freitas who always had by back and helped me become a Information Security professional.

Thanks for all your encouragement and support!

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto

Departamento de Ciência de Computadores

MSc. Information Security

Container Hardening Through Automated Seccomp Profiling

by Nuno Lopes

Currently, the use of container technologies is ubiquitous and thus the need to make them secure arises. Container technologies such as Docker provide several options to better improve container security, one of those is the use of a Seccomp profile. A major problem with these profiles is that they are hard to maintain because of two different factors: they need to be updated quite often and present a complex and time consuming task to determine exactly what to update, therefore not many people use them.

The research goal of this paper is to make Seccomp profiles a viable technique in a production environment by proposing a reliable method to generate custom Seccomp profiles for arbitrary containerized applications. This research focused on developing a solution for generating new Seccomp profiles for any containerized application. Some of the requirements were: have a solution with little dependencies, being capable of integrating with a vast number of environments and work with no human intervention.

Results showed that using a custom Seccomp profile can mitigate several attacks and even some zero day vulnerabilities on containerized applications. This represents a big step forward on using Seccomp in a production environment, which would benefit users worldwide.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

Mestrado em Segurança Informática

Container Hardening Through Automated Seccomp Profiling

por Nuno Lopes

O uso de containers está presente num vasto número de sistemas, e como tal, surge a necessidade de tornar estes sistemas seguros. Tecnologias de containers como o Docker fornecem várias opções para melhorar a segurança de um container, uma destas é o uso de um perfil Seccomp. Um grande problema com estes perfis é que eles são difíceis de manter devido a dois fatores: precisam de ser atualizados com bastante frequência e apresentam uma tarefa complexa e demorada para determinar exatamente o que atualizar. Resultando em não serem utilizados pela comunidade.

O objetivo desta dissertação é tornar os perfis Seccomp uma técnica viável em um ambiente de produção, propondo um método confiável de gerar perfis para um container arbitrário. Esta pesquisa concentrou-se no desenvolvimento de uma solução com poucos requisitos, permitindo uma fácil integração em qualquer ambiente e num funcionamento sem intervenção humana.

Resultados mostram que o uso de um perfil Seccomp pode mitigar vários ataques e até algumas vulnerabilidades de dia zero em containers. Representando um grande avanço no uso de Seccomp em containers sendo algo que beneficia toda a comunidade.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Context

Containers are used worldwide in several production environments. They help developers reduce the time needed to configure the required application dependencies and libraries. Given their worldwide usage the need to secure these containerized applications increases. Docker [1] is currently the standard for the container technology industry and thanks to its large community new projects to better secure containers arise very frequently.

The trend to increase security is to develop a custom container runtime to substitute RunC [2] (default container runtime) and better isolate containers in a lower level manner this is the case in projects such as: GVisor [2], Kata Containers [3] among others. Another technique commonly used to secure a container is to do so before it is deployed, for example by performing image scanning which compares the version of the several technologies against a vulnerability database. This is similar to security auditing tools, which ensures that a container is running with the industry best practices and is properly configured.

Docker also provides other security options which have not been fully embraced to their full potential, one example is the Seccomp profile [4]. It is possible to create a profile consisting of several syscalls that either will or will not be allowed to execute in the system. One of the biggest challenges is that creating and maintaining an up to date Seccomp profile is a complex process that presents multiple difficulties. The first one is knowing which syscalls [5] the containers actually needs. These might change when a new update is available, requiring a periodic check to ensure that all the features work properly. The absence of periodic checks could cause a self inflicted Denial of Service (DoS).

Seccomp can be a very efficient way when trying to reduce the attack vector on a container surface since the syscalls that it can make are only those whitelisted. In other words, in case a malicious actor gets access to the container, by a Remote Code Execution (RCE) vulnerability for example, he/she will be limited to the syscalls that are whitelisted in the profile. If the malicious actor tries to escalate privileges or attempts a container breakout through a syscall that was not specified in the profile, the attack will not go through. This might also be helpful when dealing with some zero day vulnerabilities that need syscalls which are not in the Seccomp profile.

This thesis aims to create a reliable method to use Dockers Seccomp feature in a production environment. To accomplish this, Seccomp needs to know what syscalls will be whitelisted therefore we need to create a tracing solution capable of logging all the syscalls that a given container generates. The process of gathering the generated syscalls will be automated by making use of the Continuous Integration/Continuous Development (CI/CD) pipeline. This way, every time there is an update to the code, the unit/integration tests will trigger and our tracing solution will capture all the syscalls generated by these tests.

Having this implemented the pipeline can finish by deploying a container with a custom Seccomp profile. Since there is a possibility that the tests will not explore all the needed syscalls we will introduce fuzzing containers. These containers will interact with the containerized application and explore new code paths by fuzzing the available functionalities. However, fuzzing is only recommended in cases where the unit/integration tests do not present a full coverage of the application functionalities, since they might explore unwanted code paths.

1.2 Contributions

This thesis contributes to the advancement of scientific knowledge in four ways:

- Development and implementation of an autonomous way of capturing all the syscalls that a container performs during the unit/integration testing phase. Automatically generate a new custom Seccomp profile from the previously described captured syscalls.
- Combine the Seccomp security feature with the Continuous Integration/Continuous Development (CI/CD) pipeline therefore solving the challenge of maintaining Seccomp profiles up to date.
- Fuzzing containers to generate new syscalls. Custom containers will be created in order to fuzz the target container and generate new syscalls, therefore having a more complete Seccomp profile.
- A comparison between vulnerable containers that use the achieve solution against plain vulnerable containers. A demonstration will be conducted on how our solution can mitigate high severity vulnerabilities and have a substantial impact on hardening containerized applications.

1.3 Document Structure

This thesis is structured in chapters. Each chapter is separated in sections, in order to help structure the thought of the reader.

Chapter 1 introduces this thesis, giving context about the problem as well as a brief explanation on the proposed solution and how it may impact production pipelines.

Chapter 2 presents the necessary background for the reader to fully understand concepts that will later be addressed.

Chapter 3 demonstrates similar solutions in order for the reader to understand the knowledge gap and fully grasp this thesis contribution to the community.

Chapter 4 presents the main contribution of the thesis. Here a detailed explanation about the development of the proposed solution will take place.

Chapter 5 is where the security posture of our solution is evaluated by pointing out some of the drawbacks, an overhead analyses and how effective it is when it comes to vulnerability mitigation.

Chapter 6 potential avenues of future work for this research are discussed.

Chapter 2

Background

This chapter gives an overview of the most important concepts and notions the reader needs to grasp in order to understand subsequent chapters. It introduces some lower level concepts such as: the Kernel and syscalls which are the base on an operating system and also our solution. We follow with a solid introduction about containers, what are they, understanding several container runtimes and a comparison between container technologies. Afterwards we explain some software development practices such as: Continuous Integration and Continuous Delivery and how these practices alongside with the Least Privilege Principle fit into our work, we also detail some key technologies used during the development of our solution that helped us achieve a least privilege architecture.

2.1 Kernel

The kernel is the heart of a computer's operating system, when a computer is booted up and the bootloader has finished then it's up to the kernel to handle the rest of the start-up. It provides a commutation between the user interface, the CPU, memory and any other I/O device. Since the kernel is an extremely sensitive component, it uses a separate area of memory called Kernel Space, this area is not accessible by any other application in the system. Furthermore the memory used by other application in the system is called user space. The communication between the kernel space and user space is done through the use of syscalls as seen in figure 2.1.

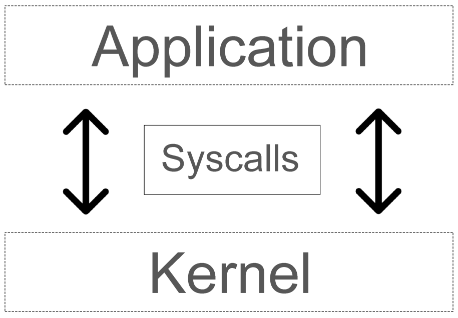


FIGURE 2.1: Application and Kernel Interaction.

2.1.1 Syscalls

Syscalls are used by every application in the system, they provide a mechanism interface between the application process and the operating system (kernel). They are used whenever an application requires the kernel to perform any given task, this workflow is depicted in figure 2.2, an important note is that syscalls are the only entrypoint for the kernel. There are several types of syscalls: process control, file management, device management, information maintenance and communications.

- **Process Control**
  Responsible for process creation, as well as, allocating and freeing memory for such process and managing the process throughout its life span.

- **File Management**
  File management syscalls manage any request related to file manipulation, this includes: creating, reading and writing to a file. Furthermore operation such as: set and get of attributes related to any file in the system are also dealt by these type of syscalls.

- **Device Management**
  These syscalls are responsible for managing any resource related to a device attached in the system. If the requested resource is available these syscalls will grant control over the resource and will also free it, when execution is over.

- **Information Maintenance**

  Information maintenance syscalls ensure that any requested data is accessible between the application and the operating system.

- **Communications**

  Interprocess communication syscalls are responsible creating or deleting any communication connection usually performed by: sockets, pipes, shared memory among other.

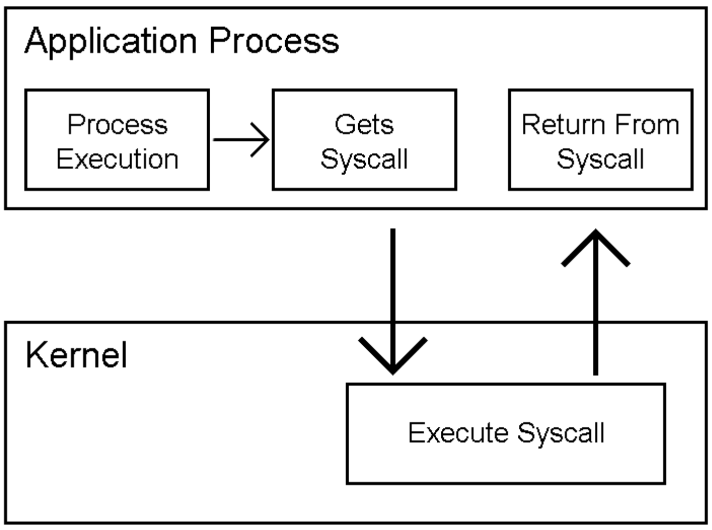


FIGURE 2.2: Syscall Workflow.

2.1.2 Namespaces

Kernel Namespaces [6] provide a layer of isolation between containers. This means that a given process within a container cannot see processes from other containers or from the host system, in other words the container will run in that separate namespace and all it has access is limited to that specific namespace. One important note is that even though a process in the child namespace has no idea if a parent namespace exists, any process in the parent namespace is capable of viewing the entire child namespace just like any other process in the system.

There are several kinds of namespaces: mount, process ID, Network, Interprocess Communication, UNIX Timesharing System, user ID and time as seen in figure 2.3. The mount namespace is used to isolate mounting points resulting in each namespace having different single-directory hierarchies. Process ID is what makes each process ID in a namespace isolated from other namespaces therefore it can result in processes having the same PID as long as they are in separate namespaces. Network namespaces provide each namespace with a private set of IP addresses, routing table, socket listing, connection tracking table, firewall rules among other resources. Inter-process Communication namespace, also known as IPC namespace, will only affect mechanisms such as: shared folders, pipes, signals and message queues, constraining their ability to communicate only within their namespace. UNIX Timesharing System (UTS) is responsible for changing the hostname on a given namespace, it is used to identify a machine both locally and remotely. USER ID is responsible for user segregation across namespaces, one major advantage is having a user with ID zero inside a namespace while it is mapped to an unprivileged user outside that same namespace. At last, the Time namespace results in processes from different namespaces seeing different system times.

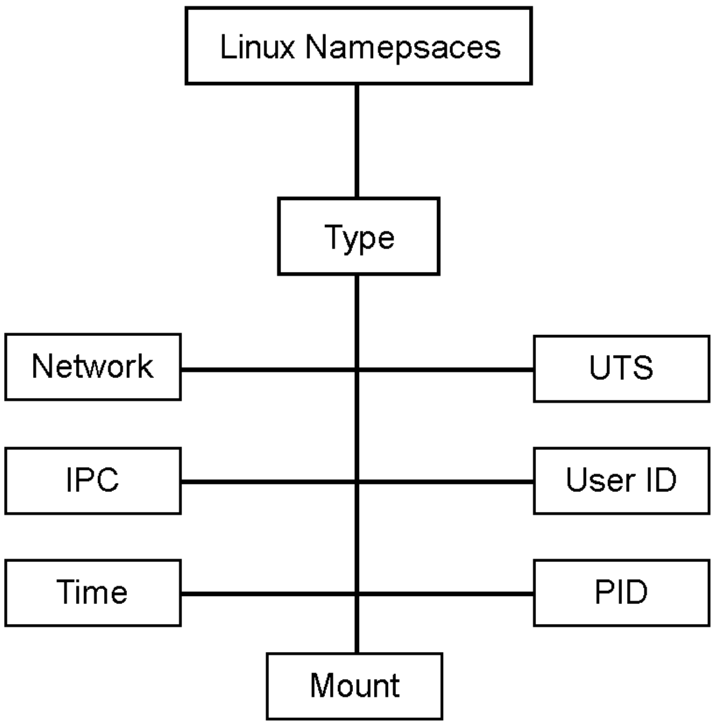


FIGURE 2.3: Linux Namespaces

2.1.3 Capabilities

Modern Linux kernels have a security feature called capabilities [7]. In essence, the idea is to split the kernel syscalls into groups of similar functionalities and assign a specific group of syscalls to a given process. By doing this, a process will only have the capabilities it needs rather than having unrestricted access. Besides assigning a set of capabilities to a process it is also useful to assign them to a user. If there is an unprivileged user in our system and he/she needs some extra privileges instead of granting root level permissions, it is recommended to use capabilities and assign only the required permissions.

If implemented correctly, by assigning only the required privileges to both processes and users, a least privilege architecture may be achieved, thereby reducing the attack vector on our system. Also if an attacker gains access to the system by compromising a user, he/she will be restricted to the limited capabilities of the user, making it harder to escalate his privileges.

2.2 Containers

A different approach using virtualization technology since it differs from the traditional hypervisor based virtual machines. Containers are lightweight applications that allow us to package both the code and all of the dependencies in order for that application to be executed. A container shares the hosts kernel and run as a resource-isolated processes which means it has stable deployments despite of the environment configuration. This also makes containers superior when it comes to distributing the application.

2.2.1 Container Runtime

When talking about container runtimes it is important to distinguish high level runtimes from low level runtimes even though these are completely different they often are misunderstood. Starting with an overview, a container runtime should be able to get an image and then run launch a container from that image.

We can clearly separate the both level, a high level runtime is responsible for image management, this includes building an image or downloading it. Also it needs to be able to display all images in the system (docker images, docker rm), manage instances of containers (docker ps) and finally run a container, this is where the high level runtime passes to the low level. In order for a container to be spawned the low level container runtime is responsible for creating custom namespaces for that specific container, set up the cgroups policies and run commands inside the container.

Containerd

Default high level runtime used by the Docker engine, which is responsible for managing the container life cycle from the moment the image is transferred until it is destroyed. It also makes sure the container is being executed calling RunC. It provides a higher level of abstraction that manages syscalls and OS specific functionalities.

RunC

The standard low level container runtime which provides a command line interface. Although it can be used as a stand-alone tool RunC is usually used by a higher level container runtime such as Containerd and is responsible for spawning and running containers, creating a custom namespace for a specific container, setting the control groups parameters enforcing rules such as capabilities or seccomp profiles and mounting the container file system, this workflow is depicted in figure 2.4. At the moment is the default runtime when using Docker.

Kata Containers

Kata provides a more secure container runtime alternative to RunC hence it builds a fast, lightweight virtual machine around the container as seen in figure 2.5, this provides a stronger isolation than a stand-alone container. It is capable of achieving this with the help of Intel Clear Containers that provide a hardware virtualization solution as another isolation technique.

Since each container has its own virtual machine the kernel is not being shared with the host system or any other containers running in the same machine. The benefit to this is

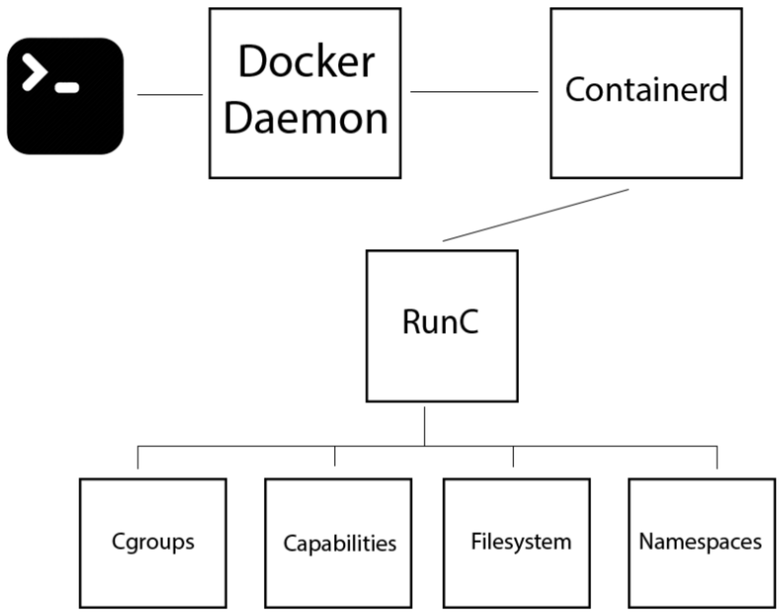


FIGURE 2.4: Docker Workflow.

that any type of malicious code can no longer exploit the shared kernel therefore reducing the attack vectors. The drawback is the overhead created by having a container inside a virtual machine, we found that tasks that need significant memory access are much slower.

gVisor

Google's container runtime, where the objective is to create a sandbox around each container providing strong isolation and not being as resource hungry as a VM. The runtime is called Runsc, which runs containers in a separate user-space kernel. When talking about virtualization technologies, there are usually two spaces, the kernel space and the user space. In the kernel space only privilege tasks are allowed to execute such as kernel extensions and the operating system kernel.

On the other hand the user space is for application software. Not only it runs the container on a separate user-space kernel, but also limits the syscalls that the container can make to the host kernel. This is done by having a custom implementation for some syscalls hence only the syscall that do not have a custom implementation will go through to the host

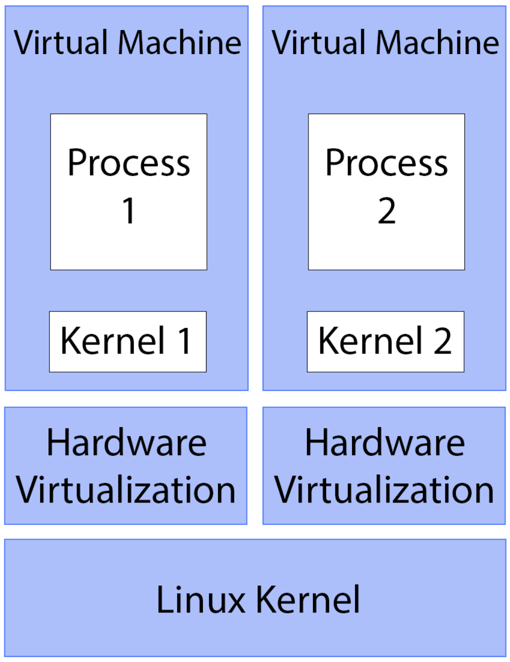


FIGURE 2.5: Kata Containers Architecture.

kernel.

2.2.2 Docker

Currently the industry standard technology to build, manage and deploy containerized applications. When talking about the Docker environment there are three main parts: the client, the Daemon and the registry each of them is crucial in order to have steady deployments.

One of the main reasons why Docker is so popular is because of its huge community and of how easy it is to deploy a container, nowadays there are several container technologies with a friendly ease of use but Docker was the pioneer. Therefore creating a huge community which continues to bring new users to the technology.

Docker Image

A file used to execute an application in a Docker Container, from the moment an image runs it becomes an instance of that Container. Docker images come from the dockerfile,

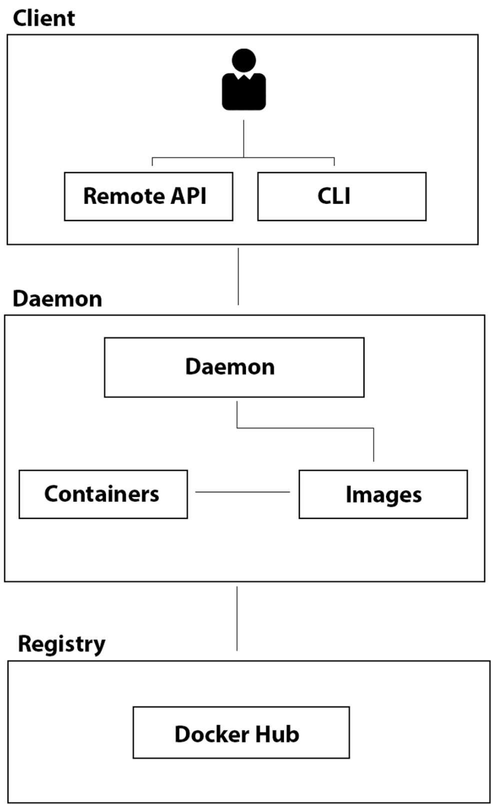


FIGURE 2.6: Docker Architecture

this is where the user specifies all the software and dependencies that the application requires. Besides in the dockerfile a list of commands will be executed in order to install such software and dependencies. Once the dockerfile is built it can be shared as a docker image and uploaded to docker hub.

Docker Client

Component responsible for providing a command line interface that allows the user to communicate with the Docker Daemon and successfully create, manage and delete containers. In order to use the docker client the user must be belong to the docker group or have elevated privileges in the system. This component is very sensitive and if there is a misconfiguration, the docker client is definitely a target for privileges escalation attacks.

Docker Daemon

Component responsible for provisioning a container. The Docker Client will send a command to the Docker Daemon, the Daemon will evaluate and respond to the request usually calling another technology such as Containerd. Since it must run with root privileges, before we start deploying containers we need to make sure that only trusted users have access to the Docker Daemon. An untrusted user with access to the Daemon can not only create containers as he pleases but also could spawn a new container and bind the whole host file system to the newly created container, having also the ability to read and write to files as he pleases. In other words he has root access on the machine. Because of this the Daemon is a critical component within the Docker architecture and must be protected. Extra precautions should be made when using an API to spawn containers, such as using HTTPS on any API endpoints and perform runtime parameter checking.

Docker Content Trust Signature Verification

Responsible for ensuring that a user can only pull and run images from a signed repository in order not to run an image from an untrustworthy source. This requires a configuration in the daemon.json file to enforce the signature verification. It works for both Official Docker images and User-Signed images. To enforce this configuration add the following listing 2.1 to the file in /etc/docker/daemon.json

```
{
    "content-trust": {
        "mode": "enforced"
    }
}
```

LISTING 2.1: Enforcing Docker Content Trust Verification

Docker Trusted Registry

Docker Trusted Registry (DTR) is docker's solution to image storage which provides some additional features such as image scanning, for DTR to perform scans in an image this option must be enabled after that we can perform periodic scans. There is also a functionality for Role Based Access Control (RBAC) in order to control users privileges, this can be synchronized from Active Directory or LDAP which most organization have.

Privileged Container

Privileged containers are used in a debug environment and should never be deployed in production. Some of the features that are used to isolate containers are not used when deploying a privileged container, the most significant are full capabilities and lifts the limitations enforced by cgroups. If an attacker gains access to a privileged container they are basically root on the host.

These containers are ultimately created by the root user and also are running as root. There may be some restrictions, especially if using AppArmor or Seccomp profiles, but nonetheless the processes are running as root which requires extra caution.

There are some solutions that in theory you can run a privileged container in a secure way, Sysbox is an implementation of this where they make use of Linux user namespaces in a similar way as userns-remap, they also virtualize portions of the host such as procfs and sysfs and lastly they do syscall emulation. This is a new technology and even though at the moment privileged containers are used in a debug environment there is work to be done so they can also work in production.

2.2.3 Docker Security

There are several layers when talking about Docker security[8], we can start with the bottom layer being the kernel security. Here we have features such as namespaces, capabilities and cgroups. The second layer is the Docker Daemon, a critical component in the architecture which must be correctly configured. Lastly, we have hardening solutions in order for a container to perform only the required actions, this way achieving a least privileged architecture.

Userns-remap

Userns-remap is a good practice when deploying any containerized application since it helps to mitigate some privilege escalation attacks. If an application does not require root access then we should create an unprivileged user for this purpose on the other hand if the application requires root and we don't specify any configuration then the root user inside the container will be mapped to the root user outside the container. This is a big problem and may result in a privilege escalation attack, using userns-remap means that we will re-map the root user inside the container to a low privileged user on the host system. If an attacker manages to perform a container breakout we will be stuck with a low privileged user making it more challenging to escalate its privileges.

Docker Daemon Socket
The Daemon socket is used as a way to communicate with the Daemon itself. There are times when we may need a container to create new containers usually called dind (docker inside docker), obviously this container will have extra privileges for this purpose. There are two alternatives: the first is to make this a privilege container granting root capabilities on the host. The second is to bind the hosts Daemon Socket into the container, this way our container will send messages to the host Daemon and create sibling containers which will reduce the privileges needed comparing to the first option.

Another option is to have the Socket available through the network for this we should be using TLS and generate a certificate only for this purpose. This way only connections from clients that can be authenticated will go through.

Rootless mode
A solution where the Docker Daemon does not need to run as root, very helpful when we need to mitigate potential attack vectors (Docker Daemon). This is made possible since the Docker Daemon which is the component that needs root access is running in a separate user namespace. Since Rootless mode is still experimental there are some known limitation when working with it such as: AppArmor and Cgroups are not supported, exposing SCTP ports and exposing privileged TCP/UDP ports.

2.2.4 Linux Containers

First container technology to provide a stable environment with the purpose of running containers. Originally docker used Linux Containers [9] as a low level container runtime however it was then replaced in docker version v1.10. Even though Docker stopped using Linux containers, they are still widely used in production environments. Linux Containers have several projects under active development, we will take a look on LXC and LXD in more detail.

LXC

LXC is the low level container runtime for Linux Containers it provides an interface to directly create, manage and terminate containers. LXC goal is to create an environment as close to the Linux system but without a separate kernel and no overhead. Just like others low level container runtimes it also makes use of several features that aim to increase isolation between each container in the system and between the host itself.

LXD

This project brings a Daemon to the LXC meaning that instead of using LXC command line interface you interact with LXD that will then forward its requests onto LXC. The focus of this project is to improve user experience when managing containers. There is also the possibility of enabling a REST API in order to remotely access the Daemon.

2.2.5 Linux Containers Security

In this section we will examine the security from the perspective of a user using LXD since the attack vector is wider (LXD + LXC). LXD offers several security mechanisms, some of them are similar to docker we will explain a subset of those which are only related to LXD.

LXD Daemon

Just like Docker the LXD Daemon is running as root therefore the precautions mentioned previously also apply here. There is also the possibility of enabling communications with the Daemon through the network therefore creating a new attack vector. In this case we have two options, TLS and Candid based authentication.

When using TLS, the first communication between the client and the server will be with the purpose of generating a keypair that will be used in the following HTTPS connections to the LXD Daemon. On the other hand when using Candid based authentication we have more control on who access what, meaning that there is the ability to grant or revoke access to specific users. We can set this up using a Candid server and configuring the LXD client privileges, once that is done we can connect our Candid server to as many LXD Daemons as we require and finally set up persistent storage within the Candid server. Once all of this is configured there is the possibility of adding an extra layer of security with Role Based Access Control (RBAC) creating roles to a project or a specific LXD instance. Each role has four levels :

- **Auditor**: Lowest privileged user, only has read access within the project;
- **User**: Execute commands in the instances, attach to console;
- **Operator**: Create, re-configure and delete instances and images;
- **Admin**: Ability to reconfigure the project and perform any action possible.

Even though this requires more configuration, provides a more organize environment which is ideally in production.

Linux Containers Isolation

LXD containers have and extra step when it comes to isolation from the host. When deploying containers that do not need to share data with others then you can enable the option *security.idmap.isolated* which will use non-overlapping User and Groups identifiers maps for each container. This is important because if two containers share these identifiers they also share limits, meaning that a user in a first container can DoS the same user in another container.

2.2.6 Podman

Podman [10] is an upcoming container technology with some interesting architectural changes, the most important being it does not have a Daemon. Which reduces the attack vector when using this technology. Containers also have the option to be run as root or in rootless mode. Having the possibility of a fully rootless architecture.

Podman Images

The location where the images are stored is different from Docker, Podman stores its images in */.local/share/containers* meaning that a different users cannot access other users images. Podman Images are built with the OCI standard therefore any container technology that supports the standard is fully compatible.

Podman also has the ability of pulling images from other registries, for example you can built and push an image into the Docker registry and later pull that same image and run it with Podman, as long as the registry is specified in the *registries.conf* file then Podman will look for the image there. An extra functionality is that Podman helps transitioning into the orchestration world by generating a Kubernetes YAML file from a container image. Besides creating YAML files Podman also helps to debug running Kubernetes containers with the command in listing 2.2

```
podman  pod
```

LISTING 2.2: Podman Image to Kubernetes

Podman Auditing

In order to better understand the different approach that Podman took which impacts auditing we will compare it with the Docker approach.

- Docker:
  Looking at the Docker architecture, the user interacts with the command line interface where his requests will be forwarded onto the Docker Daemon and finally the Daemon is responsible for responding to the request. Whenever a new container is created there are a few syscalls that need to be called, let's focus on fork and exec. When these syscalls are called the newly created container process will be a child of the Docker Daemon process. This implies that the UID (User Identifier) of any actions that the container performs will be the same UID that the Docker Daemon has. In a complex architecture since all the containers will have the same UID this will not be a viable option on a debugging scenario where there is the need to know what container did what, and whom created a specific container.

- Podman:

  The daemonless container architecture comes in handy here. Podman also calls the same syscalls, fork and exec, the difference is that since there is no Daemon the command line interface calls directly Podmans container runtime, which on time of writing is RunC, and when a new container is created the container UID will be the same as the user. Therefore containers created by the same user will have the same UID but containers created by different users will have different UIDs.

Podman Limitations

As we have seen Podman is a well establish technology with a small but growing community nonetheless Podman still has some limitations.

For starters it can only run on Linux based distributions, there is no wrapper for Windows or MacOS this happens because in order to create a container there is a need to use some Linux kernel functionalities such as namespaces. The reason why other container technologies work on Windows and MacOS is because when building or creating a container this is done inside a Virtual Machine. At the moment Podman does not have such a feature but in the future is totally plausible that this will be implemented.

Secondly, with Podman we can only manage one container at a time, meaning that there is no solution for something like Docker Compose which has a way of managing multiple containers locally. This is something that we might see implement by the Podman community growth since when deploying complex containers that need to talk with each other having a way to manage them all is a great advantage.

2.3 Continuous Integration/Continuous Delivery

Continuous Integration/Continuous Delivery (CI/CD) is a software development best practice, where teams use this technique during the life cycle of any given application, in order to ensure code quality and frequent releases. For this purpose a pipeline must be defined and a series of consecutive events where the output of one event is the input of the next one detail the whole automation process, as can be seen in figure 2.7.

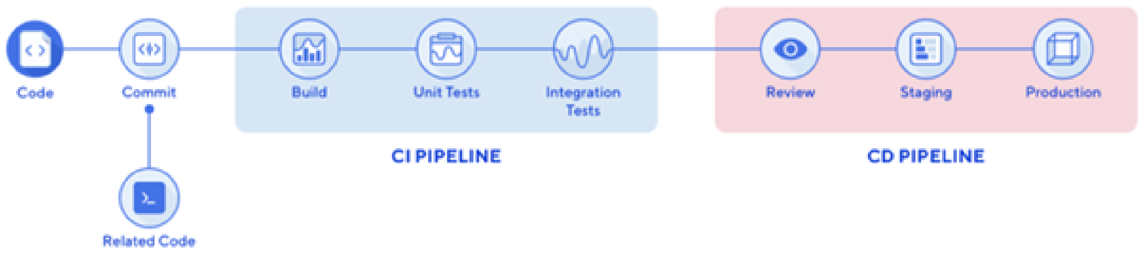


FIGURE 2.7: CI/CD Pipeline.

2.3.1 Continuous Integration

Continuous Integration [11] (CI) is a software development practice where members of a team merge their work (code) as frequently as possible. This is a key feature when it comes to industry best practices as it enables reliable releases of new features. During each integration process the software updates are built into a new release, which will later be validated by the unit tests. This ensures that the new code is working properly and is followed by the integration tests which will assure that the new code is compatible and performs correctly with existing external services. In practical terms continuous integration results in during any given day software developers to have shorter releases and keep improving software quality, additionally, having small but reachable goals can result in developers being more productive.

2.3.2 Continuous Delivery

Just like the previously explained CI, Continuous Delivery [12] (CD) is also a software development best practice in which after the CI phase the new release will be automatically be deployed to the production environment. Most of the time CD consists of small but frequent updates to the production environment. An important note is that the decision to deploy the new code to production falls under the developers who commit the new code to the delivery pipeline.

2.4 Least Privilege Principle

This principle [13] states that, only the minimum required privileges should be given to a computer system for it to execute all required processes. This means that the software should only be capable of performing the tasks it is required to do, and should only access the necessary information for such tasks. By working with this principle it is possible to shrink the attack vector that an attacker can use and essentially limit damages caused by security incidents. For this principle to be successful it requires the Administrator to define a few rules, such as: what each user has access to, when can a user be granted extra privileges and how long those extra privileges will be assigned to the user.

Besides assigning privileges to users it is also important to assign them to processes. This represents a continuous and extensive work though, with the appearance of containers this principle became a lot easier to achieve.

Our solution is based on this principle, we intend to give a container only the extremely necessary syscalls. In order to achieve this we must leverage some tools such as: eBPF, gRPC and Seccomp.

2.4.1 Extended Berkeley Packet Filter

The extended Berkeley Packet Filter (eBPF) [14] is an enhancement of the Berkeley Packet Filter [15] that was initially mostly used to trace network packets, with this new version eBPF is now capable of tracing several components from the whole system. In order to start tracing with eBPF every program must pass through the eBPF verifier to ensure that the code is safe. This verification checks that the code does not have any backward jumps (loops), does not exceed 4096 instructions and the code does not generate a runtime error. All of these precautions are due to the eBPF program being attached into the kernel. Within the kernel it is possible to attach an eBPF program to several entrypoints, such as: file system, sockets, scheduler, system call interface among others. Once any of these entrypoints gets called the eBPF program will run.

A popular solution to create and manage eBPF programs is to use BPF Compiler Collection [16] (BCC), which is a compiler for eBPF and can currently be used either with Python or Lua. BCC presents a way of instrumenting kernel hooks and their entrypoints in a more intuitive way. Usually, the first step in a BCC program is to define the entrypoint, then write a kernel hook function detailing what the program should do once the hook is called. The last step is to handle any output that comes from the hook and present it to the user.

2.4.2 gRPC

The gRPC [17] project was developed by Google with the aim of achieving a fast and efficient Remote Procedure Call (RPC) solution. The whole protocol is developed with http/2 [18], one major benefit is the latency reduction due to compression of HTTP header fields, request prioritization and timeout contracts between client and server, therefore minimizing the protocol overhead.

Within gRPC there are different types of communication: unary and stream. Unary is the simplest one where the client sends a synchronous request to the server and waits for a response, which is similar to a common function call. The second is a stream where the client sends a message to the server and the server returns a stream with a sequence of messages, which the client must read until there are no new messages. The stream call is a very powerful tool as the client can also initiate the gRPC call by sending a stream or in case of a bidirectional call, both the client and the server can send a stream of data to the same function call.

The gRPC protocol by default makes use of Protobuf as a serialization format, which is faster than earlier solutions such as: JSON or XML. Protobuf was designed to have fast encoding/decoding as well as small encoded messages. Protobuf and gRPC are totally independent and a custom serialization format can be used for a more tailored solution.

2.4.3 Seccomp

Secure Computing Mode also known as Seccomp [19], is a Linux kernel feature that sends a process into a secure state where it has a list of syscalls, which the process can perform (whitelist). Seccomp works at a syscall level allowing for a complete control on which syscalls are permitted in the profile.

The benefit of using a Seccomp profile is that when deployed, it decreases the attack surface of a container, so if an attacker manages to get inside the container he/she will be limited to only pursuing attacks allowed within the custom profile. Using this method, any exploit that requires a syscall that is not in the profile will be automatically mitigated.

Given Docker's popularity coupled with the increase need to ensure security, a built-in Seccomp profile has been added to all containers. Consequently, when a Docker container is deployed a Seccomp profile is inherently attached to it. Such profile must be quite generous allowing for a range of actions to be performed. Although, this is considered a best practice as it reduces the attack vector, it is quite challenging to configure and maintain, since the profile must be updated every time there is a change in the containerized application, resulting into a slow and daunting process.

To verify if your kernel support the seccomp feature use the command in listing 2.3, you should see a 'y' on each line.

```
grep SECCOMP /boot/config-$(uname -r)

CONFIG_HAVE_ARCH_SECCOMP_FILTER=y
CONFIG_SECCOMP_FILTER=y
CONFIG_SECCOMP=y
```

LISTING 2.3: Verify Kernel Support

In order to further demonstrate how seccomp can limit the attack surface of a container, a sample seccomp profile was created. In this profile all the syscalls are allowed except for chmod(), just like the command this syscall is used to change access permissions on any file in the system. The sample seccomp profile can be viewed in listing 2.4.

```
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "syscalls": [
        {
            "names": [
                "chmod"
            ],
            "action": "SCMP_ACT_ERRNO"
        }
    ]
}
```

LISTING 2.4: Sample Seccomp Profile

With this profile created we ran an alpine docker image with no profile attached. As expected since no profile is being enforced the user root was able to create a file and change the permissions this is demonstrated in listing 2.5.

```
docker run -ti alpine

/ # whoami
    root

/ # ls
    bin     dev     etc     home    lib     media   mnt     opt     proc    root    run
      sbin    srv     sys     tmp     usr     var

/ # touch example

/ # ls -la example
    -rw-r--r--    1 root     root             0 Oct 24 16:16 example

/ # chmod 777 example

/ # ls -la example
    -rwxrwxrwx    1 root     root             0 Oct 24 16:16 example
```

LISTING 2.5: Container without Seccomp Profile

The listing 2.6 demonstrated what happens the previously showed profile is enforced to a container. Even though the user root executed chmod() he got the output of Operation not permitted. When this is done at scale and several syscalls are not allowed an attacker will have a much smaller attack vector.

```
docker run -ti --security-opt seccomp=./profile.json alpine

/ # whoami
    root

/ # ls
    bin     dev     etc     home    lib     media   mnt     opt     proc    root    run
      sbin    srv     sys     tmp     usr     var

/ # touch example

/ # ls -la example
    -rw-r--r--    1 root     root             0 Oct 24 16:21 example

/ # chmod 777 example
    chmod: example: Operation not permitted

/ # ls -la example
    -rw-r--r--    1 root     root             0 Oct 24 16:21 example
```

LISTING 2.6: Container with Seccomp Profile

Chapter 3

State-of-the-art

In order to have a better grasp on the knowledge gap about container security we will cover some previous work done in this field. We will start by explaining some concepts about container hardening. While looking only at the features available in docker we will detail how users can increase security in their containers and some common security misconfigurations that may lead to compromises.

Looking at other projects that improve on Docker's features, research has been conducted with the aim of trying to reduce the attack vector of containerized applications we will explain how Container Fusion does it and how we expanded on their work. Additionally we will describe how AppArmor compares to our solution and some projects that were inspired by it such as: Docker-sec and Lic-sec. Furthermore, we will analyze previous work where container were monitored in order to warn the user of any suspicious behaviour occurring within the container. The last project relates to the various dangers that a docker image might be vulnerable to and how to prevent it from being tampered with.

3.1 Container Hardening

In this section we will explain some techniques and best practices to make sure containers are as harden as possible in a production environment. Even though we will focus in the Docker environment some of the concepts apply to other container technologies.

The big down side of containers is the fact that they are poorly isolated between: host and container as well as between containers themselves. Therefore preserving and using techniques to increase isolation is an important aspect in ensuring that containers stay secure. When using containers the user is susceptible to certain attacks we have compiled a list of some common attack scenarios:

- **Kernel Exploits**
  Due to Docker's architecture the kernel is shared among all the active containers as well as the host operating system. Because of this the kernel presents itself as a shared attack vector across the system. The typical attacking scenario happens when one container causes the kernel to panic and ultimately causing a disruption of service to any siblings containers as well as the host system.

- **Compromising Secrets**
  Very commonly containers require access to a database or some kind of service running outside of its scope, to accomplish this containers use API keys or username and passwords as a mean of authentication. These secrets are highly sensitive since an attacker with access to them will also be able to access the service. This could ultimately lead to a loss of availability, confidentiality and integrity of any data that was previously stored.

- **Poisoned Images**
  This attack vector requires some sort of social engineering since an attacker must convince the victim that they have a trusted and useful docker image. If the user believes in the attacker and blindly runs the poisoned image in his system all his data (including the one in the host operating system) is as risk of being compromised. The attacker will be able to read, modify and delete an arbitrary file in the victims computer.

- **Container Breakouts**
  Container breakout may easily occur if the user misconfigures a important setting when deploying the container. Other causes for this attack scenario are a vulnerable docker runtime (CVE-2019-5736) or granting a container with excessive privileges.

- **Denial-of-Service-Attacks**
  As previously stated containers share the kernel, however they also share hardware

to conduct any required computation. A malicious container may attempt to have full access to a certain resource and essentially deny that any other container use the resource in question. This will result in a Denial-of-Service on sibling containers and in some cases even the host system will be affected.

3.1.1 Kernel Exploits

Kernel Exploits present a dangerous attack vector no matter the environment and can result in the whole system being compromised. For example, a kernel exploit such as Dirty COW results in a privilege escalation even when exploited inside a container. It may be difficult to fully prevent a kernel exploit however there are some techniques used to perform system hardening and may successfully prevent an exploit, these techniques include: memory corruption defenses, filesystem hardening, miscellaneous protections, RBAC, GCC Plugins most of these techniques are enforced by grsecurity [20] a leader when it comes to kernel security.

Memory Corruption Defenses
Several measure can be implemented when it comes to prevent a memory corruption attack, the first one that comes to mind is use of Address space layout randomization (ASLR) which will randomize the addresses space of a given executable. This way the attacker does not know in what address a required function is located. Other mechanisms ensure a higher isolation between the kernel and userland, for example bound checks should be performed when the kernel is copying from and to the userland additionally the kernel should prevented from executing code in the userland. These mechanisms and other may prevent some memory corruption attacks.

Filesystem Hardening
One of the oldest techniques to harden the filesystem is to implement trusted path execution, the basic concept is to prevent unprivileged users from executing binaries that they created. Another interesting concept is to hide users processes from unprivileged users this type of isolation can be even greater if private network related information and sensitive information located in /proc is also isolated between users.

Role Base Access Control

Ideally there should be Role Base Access Control policies for the whole system following the idea of the least privileged. Resulting in restricting system access to unauthorized users. By implemented this only allowed users can access a certain resource, on some cases it might also provide a way of determining accountability.

3.1.2 Compromising Secrets

Secrets being compromise may lead to loss of integrity, confidentiality and availability on other adjacent systems. Therefore it is recommended not to use hard coded credentials, properly use encryption when these secrets are in transit and make sure the container is properly configured [21].

Docker Bench Security

An open source tool that automatically checks if your deployed containers are configured with a set of industry best practices. All of the tests were design according to the CIS (Center for Internet Security) Docker Benchmark v1.2.0. Listing 3.1 present the output returned from running Docker bench Security in the host system,

```
[INFO] 2 - Docker daemon configuration
[WARN] 2.1  - Ensure network traffic is restricted between containers on the
    default bridge
[PASS] 2.2  - Ensure the logging level is set to 'info'
[PASS] 2.3  - Ensure Docker is allowed to make changes to iptables
[PASS] 2.4  - Ensure insecure registries are not used
[PASS] 2.5  - Ensure aufs storage driver is not used
[INFO] 2.6  - Ensure TLS authentication for Docker daemon is configured
[INFO]      * Docker daemon not listening on TCP
[INFO] 2.7  - Ensure the default ulimit is configured appropriately
[INFO]      * Default ulimit doesn't appear to be set
[WARN] 2.8  - Enable user namespace support
[PASS] 2.9  - Ensure the default cgroup usage has been confirmed
[PASS] 2.10  - Ensure base device size is not changed until needed
[WARN] 2.11  - Ensure that authorization for Docker client commands is enabled
[WARN] 2.12  - Ensure centralized and remote logging is configured
[INFO] 2.13  - Ensure operations on legacy registry (v1) are Disabled (
    Deprecated)
```

```
[WARN] 2.14  - Ensure live restore is Enabled
[WARN] 2.15  - Ensure Userland Proxy is Disabled
[PASS] 2.16  - Ensure daemon-wide custom seccomp profile is applied, if needed
[PASS] 2.17  - Ensure experimental features are avoided in production
[WARN] 2.18  - Ensure containers are restricted from acquiring new privileges

[INFO] 4 - Container Images and Build File
[WARN] 4.1  - Ensure a user for the container has been created
[WARN]      * Running as root: pensive_pascal
[NOTE] 4.2  - Ensure that containers use trusted base images
[NOTE] 4.3  - Ensure unnecessary packages are not installed in the container
[NOTE] 4.4  - Ensure images are scanned and rebuilt to include security patches
[WARN] 4.5  - Ensure Content trust for Docker is Enabled
[WARN] 4.6  - Ensure HEALTHCHECK instructions have been added to the container
    image
[PASS] 4.7  - Ensure update instructions are not use alone in the Dockerfile
[NOTE] 4.8  - Ensure setuid and setgid permissions are removed in the images
[INFO] 4.9  - Ensure COPY is used instead of ADD in Dockerfile

[INFO] 7 - Docker Swarm Configuration
[PASS] 7.1  - Ensure swarm mode is not Enabled, if not needed
[PASS] 7.2  - Ensure the minimum number of manager nodes have been created in a
     swarm (Swarm mode not enabled)
[PASS] 7.3  - Ensure swarm services are binded to a specific host interface (
    Swarm mode not enabled)
[PASS] 7.4  - Ensure data exchanged between containers are encrypted on
    different nodes on the overlay network
[PASS] 7.5  - Ensure Docker's secret management commands are used for managing
    secrets in a Swarm cluster (Swarm mode not enabled)
[PASS] 7.6  - Ensure swarm manager is run in auto-lock mode (Swarm mode not
    enabled)
[PASS] 7.7  - Ensure swarm manager auto-lock key is rotated periodically (Swarm
     mode not enabled)

[INFO] Checks: 105
[INFO] Score: 19
```

LISTING 3.1: Docker Bench Security

FIPS Mode

FIPS stand for Federal Information Processing Standards [22] and is part of the United States government. Their responsibility is to approve cryptographic modules that provide enough security requirements for the United States government to use. Since Dockers cryptographic module has been validated by the National Institute of Standards and Technology (NIST) which is the institute in charge of the FIPS standard we now have the option to use FIPS encryption especially recommended during transmit or storage of sensitive information. To enable FIPS Mode you must manually add a systmed file to configure the mode.

3.1.3 Poisoned Images

To prevent such attack the user must be educated in common social engineering engagements as well as making good decisions when it comes to building his own dockerfile. Another important aspect is that outdated dockerfiles may have vulnerabilities due to using outdated software. To prevent an exploit on such vulnerabilities is it recommended to perform regular image scanning checks as well as a code review of the dockerfile before blindly running it.

Base Image

When choosing a base image one should consider the underlying image that is used to build it, for example when using a simple *node* image we are also using a Debian image beneath all the node dependencies. For this reason there should be an extra step before deploying a container and if there is no need on having general system libraries then it is recommended to opt for a minimal base image, therefore introducing as little OS dependent vulnerabilities as possible.

Trusted Images

When it comes to choosing an image one should always try to use a Docker Certified image, these are built and vetted by the Docker team. If you are building your one image make sure you sign it and then verify the signature before running it, Docker Content Trust makes this step very easy, simply configure it then users can publish signed images and later before running the image, it will automatically be checked for its integrity and origin. The image 3.1 shows an example of how a user can determine if an image can be trusted.

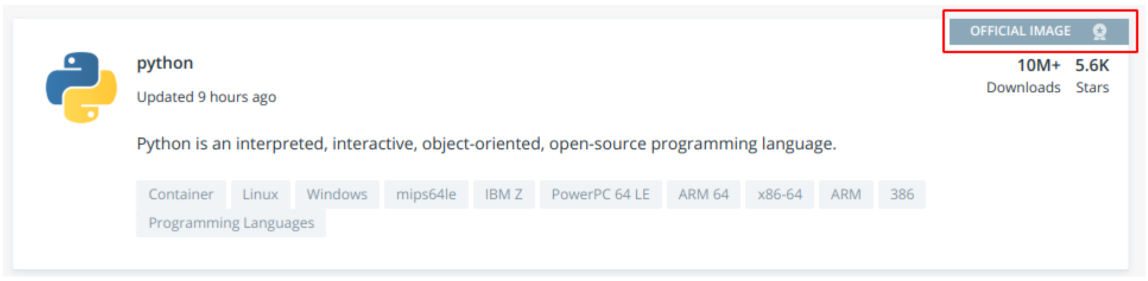


FIGURE 3.1: Docker Trusted Image.

Image scanning

Image scanning refers to the action of performing a static analysis on a containerized application. It takes the image as input and list all the software name and versions which is present in the container. With this information it checks against databases with known vulnerabilities. In case any of the software in the container is known to be vulnerable the image scanning tool will return an alert for the user to be aware of potentials risks.

Continuous Monitoring

New vulnerabilities are being discovered all the time, for this reason it is important to perform regular updates on the dependencies the container is using. This can be coupled with a previously explained image scanning technique, however it is important to perform periodic checks on the container being used.

3.1.4 Container Breakouts

One of the most critical scenarios where isolation might get bypassed is through a container breakout vulnerability, granting the attacker immediate access to the host and maybe even other containers.

Privileges

Whenever deploying a container we should give as little privileges as we can to the container making sure we are working on a least privilege environment. This can be accomplish by using rootless mode if possible also whenever writing a Dockerfile it is very important to specify a user because if a user is not specified then the default user is the root user.

Each container should have a dedicated user and a dedicated group this makes it harder for a privilege escalation onto the host.

Capabilities

The user has the ability to give certain capabilities to a container, if these are not properly configured may result in an attacker taking advantage of the extra capabilities and compromise the host operating system. We present a list of highly sensitive capabilities [23] that if an attacker has access to them can result in a container breakout.

- **SYS_ADMIN**
  SYS_ADMIN is the most sensitive capability in the linux kernel, if granted it allows the user to performs administrative type action on the whole system and pretty much removes all of the isolation between container and host. If grated an attacker could mount any given filesystem, create new namespaces, perform administrative operations on device drivers among other high privilege operations. A skilled attacker with such capability will easily escalate his privileges and get root on the host operating system.

- **SETUID and SETGID**
  SETUID and SETGID are two very similar capabilities that are able to manipulate a process Unique Identifier (UID) or Group Identifier (GID). Being able to manipulate these parameters maybe extremely dangerous as they are capable of modifying the value of any process and set their UID to one running in the host, this will result in being able to interact with a process running outside of the containers boundaries. Alternatively an attacker could change a process UID and GID to zero and start interacting with root level processes which would greatly expose the host environment to the attacker.

- **SYS_CHROOT**
  SYS_CHROOT allows an attacker to change the path of the root filesystem. These

capabilities proves very useful when it comes to privilege escalation attacks, an attacker can create a malicious binary but leave it in the same path as a binary commonly used after waiting for a privilege level process to execute it, the attacker will have privilege level permissions in the system.

- **SYS_PTRACE**
  SYS_PTRACE is a capability commonly used when the ptrace() syscall is required, this is used to attach to a given process and examine it, by monitoring his state, changing values and perform debugging style functionalities. In case the attacker has knowledge about a process identification number running outside of the boundaries of the container, he can use ptrace() to gather information about such process running in the host or even other containers. Ultimately the attacker will have a better knowledge on the whole system environment and might even be able to steal secrets that are in memory of a given process.

Running as Root
When it comes to running stuff as root the user must be educated about the dangers and why most of the time is not recommended. Even though there are some container who actually require root to operate this is not standard in most cases therefore if possible running as a low privilege user is always advisable.

A common misconception is that when the users see root inside the container they believe that this root user is restricted to only the container however if user namespace are not enabled the root user inside the container is mapped to a root user outside of the container. Since user namespaces or userns-remap as referred in the Docker documentation is not enabled by the default the user must enforce this policy manually, we provide a list of commands in listing 3.2 to enforce such policy.

```
#!/bin/bash
sudo cp /lib/systemd/system/docker.service /etc/systemd/system/

sudo sed -i '/ExecStart/ s/-H fd/--userns-remap=default -H fd/' /etc/systemd/
    system/docker.service

sudo systemctl daemon-reload

sudo systemctl restart docker
```

LISTING 3.2: Enabling userns-remap

In order to check if the above command successfully enabled usern-remap, you can head over to /etc/subuid and /etc/subgid and a new user called 'dockremap' will appear as seen in listing 3.3.

```
cat /etc/subuid
    ubuntu:100000:65536
    dockremap:165536:65536

cat /etc/subgid
    ubuntu:100000:65536
    dockremap:165536:65536
```

LISTING 3.3: Checking dockremap user

Namespaces

The user also has the possibility of lowering some isolation mechanisms between container and host by leveraging namespaces such as: pid, net, uts, ipc, usern. When using any of these options the container will use the host namespace and will have read and write permissions.

- **–pid=host**
  When the flag Process Identification Number (PID) is used will result in the container being able to view any process running in the host operating system. An attacker is capable of gathering information about the host and even other running containers.

- **–net=host**

  The network (net) flag is used when the user need to access the host's network from inside the container. An attacker can take advantage of this and become fully aware of any traffic running through the host's network, he will also be able to interact with any port that are binded.

- **–uts=host**

  This flag if fairly harmless when compared to other, using it grants the attacker the possibility of changing the hostname and domain information on the host system even from inside the container.

- **–ipc=host**

  The Interprocess Communications (ipc) flag results in a the container using the IPCs resources on the host system, such as: file, sockets, message queue, pipes, shared memory among others. An attacker can take advantage of this by using and interacting with host related IPCs.

- **–userns=host**

  The flag userns is related to the user namespace as discussed previously, when is enabled the userns-remap configuration will no longer take place as this flag will make sure the container is using the same users as the host. This presents a real danger as the user root inside the container is mapped to root on the host.

Docker Daemon

The Docker Daemon is a crucial part on docker's architecture and is particularly sensitive since it usually is running with root level privileges. The following attack show how an attacker who is a member of the docker group might take advantage of this.

- **Docker exec**

  Docker exec presents an easy and effective way of running a command inside the container, if a shell is required a call to '/bin/bash' or '/bin/sh' will suffice. However this command has a inherently dangerous feature, using the '-u' it is possible to execute such command under any user. An attacker might take advantage of this an attempt to execute a command as root as demonstrated in listing 3.4. In cases where the root user inside the container is mapped to root on the host this might present a serious security issue.

```
docker exec -u=0 a0b1 whoami
root

```

LISTING 3.4: Docker Exec

- **Docker Group access**
  With the previously explained issue in mind, it is possible to determine that any user who belong to the docker group has the ability of running any command as root, in any running container. Since the Docker Daemon does not enforce admin segregation on their users, anyone can manage all the containers in the system. For example, different users that belong to the docker group may have their own images, however they both have admin right on each other images. This presents an issue since there is no accountability when an unlawful action is taken. In order to demonstrate this, we created two temporary users in the system, both of them belong to the docker group however one of them created a docker alpine image and the other stopped this image execution. From listing 3.5 it is possible to see that docker does not distinguish the action taken by these different users.

```
docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
        STATUS                      PORTS
a0b1c4f8d080        alpine              "/bin/sh"           7 minutes
ago         Exited (0) 20 seconds ago
bcee8fc5d40b        alpine              "/bin/bash"         7 minutes
ago         Created


```

LISTING 3.5: Docker Accountability

3.1.5 Denial-of-Service-Attacks

In an environment with multiple containers it is important to define the resources that each container can use. If this step is left unattended then the possibility of one container using all the resources available and essentially DOSing [24] sibling containers is very real. In order to define exactly the system resources that each container has access to, it is recommended to implement control groups policies.

Control Groups

Control Groups [25] are a Linux kernel feature that is responsible for the container resource allocation. It has the ability to limit a specific resource, meaning that a group can be defined in order to not exceed the specified memory limit or amount of processors. Other functionality is the ability to prioritize different groups, meaning that a group can either have more or fewer CPUs and disk I/O throughput. While these restrictions are in place, all the groups will be monitored and if something is not according to the user configuration, it is possible to select a group of processes and stop or restart them.

Limiting the CPU usage, memory, disk I/O between containers allows hardware to be shared with rules in place and giving more control to the user. This plays a significant part when dealing with a Denial-of-service attack [26] since it will limit the compromised container resources so the system can keep running smoothly.

3.2 Container Fusion

In previous work [27], a new idea has been presented, called container fusion, which happens when there are two different containers and one needs some privileged access on the sibling container. The access is given based on the least privilege principle meaning that it will only have the required permissions to perform the needed actions therefore limiting the visibility and accessibility of the core container.

The containers are isolated using a number of known techniques such as: granting access to the core container with the help of namespaces, capabilities, a custom Seccomp profile and hardware restrictions using control groups.

In order to successfully define the capabilities and syscalls required to the custom Seccomp profile, they manually determine the capabilities needed by studying the container. Afterwards they review the syscalls that each capability would allow and removed the ones that were not required, and finally could deploy the container with these custom profiles. Looking at the conclusion they propose an automated way of determining capabilities in a sandboxed environment as future work, this is where we expand on their research by automatically generating a custom Seccomp profile.

3.3 AppArmor

AppArmor [28] makes use of a kernel module, which binds its profile to a program. The profile consists of several capabilities (whitelisting of capabilities) and some access control policies that the program needs in order to run successfully. The administrator must create an AppArmor profile and bind it to a specific program. To facilitate this AppArmor has two methods for profiling the first is called genprof, where AppArmor conducts an initial static analysis on the application and generates a general profile for it.

The second method is called learning, which as the name goes it used to generate a profile by learning from the application. This method will grant the application access to all capabilities, however it will log everything. When the work is done AppArmor can review which capabilities were called and generate a custom profile dynamically created and tailored to a specific application. Unless the administrator knows exactly the capabilities and access control needed for the application it is always recommended to use the learning method for profiling.

This research paper follows the same steps as the learning feature in AppArmor: allowing all the syscalls to be performed, logging and reviewing them and finally generating a new profile. In the case of AppArmor, one major disadvantage is that the whole process is not scalable and big application that require user input will result in a slow and hard process to maintain. Additionally AppArmor profiles are not as easy to read as Seccomp profiles.

3.4 Docker-Sec

Docker-sec [29] presents itself as a fully automated container security enhancement mechanism. Docker-sec protects Docker containers by enforcing access policies to the containers. It uses AppArmor to enforce policies on each component of the Docker architecture. In practical terms Docker-sec is a wrapper using bash script between Docker and AppArmor, it uses the same familiar parameters as Docker, and automatically starts AppArmor to begin creating a profile. Profiles related to container are firstly created statically leveraging the user parameters when running the container as previously explained this method is called genprof. Once the container begin Docker-sec makes use of the AppArmor learning feature and dynamically adjust the initial profile to better fit the container by restricting unnecessary capabilities.

Additionally Docker-sec also generates a custom profile for both the Docker Daemon and for Runc. Regarding the Daemon Docker-sec uses a modified version of the AppArmor profile publicly available in Docker Github repository where restrictions are enforced in order for the Daemon to only perform the necessary tasks. When it comes to Runc Docker-sec creates a custom profile enforced from the moment the container is initialized until the container is ready to start. Resulting in a protection of the entire process of the Docker environment.

Listing 3.6 shows the sample command for executing a container using Docker-sec. The first command will launch a nginx server the user must interacts with the server to generate new capabilities, once all the required functionalities are explored execute the second command (same as the first) to stop AppArmor tracing and generate the profile. Finally when running the nginx image using Docker-sec the previously generated profile will be enforced.

```
docker-sec train-stop safe-nginx

docker-sec train-stop safe-nginx

docker-sec run --name safe-nginx -p 80:80 nginx
```

LISTING 3.6: Docker-sec sample

3.5 Lic-Sec

Lic-Sec [30] expands on the previous explained work Docker-sec. As previously explained Docker-sec create a custom AppArmor profile and binds it to the several components in the Docker architecture. Lic-sec also does this however it divides the container life cycle into different phases. Once the different phases are defined it attaches a custom AppArmor profiles for each one.

This is a great improvement on Docker-sec research since most container require more syscalls during the initialization phase. By changing the profile of a container during its life cycle Lic-sec manages to reduce even more the surface that an attacker might try to exploit.

Lic-sec has several modes of functioning the first one is designed to generate a profile for the Docker Daemon and Runc therefore all the tracing must occur before these components start their execution. The second mode is used to trace the container initialization process, tracing begins before the container launching and terminates immediately once the container finishes initialization, therefore only gathering information related to that phase. The third mode uses AppArmor learning and traces the necessary information during the container's execution.

3.6 Security Monitoring of Containerized Environments

Previous work [31] has been done in hardening container environments by taking advantage of rule-based security monitoring. The goals is to identify any behaviour that might indicate a malicious event, for example: a hostile workload running inside the container, any attempt on a process/user gaining extra privileges or attacks that aim to reduce isolation between container and host.

Monitoring of container is achieved by collecting containers information during runtime, if the collected data triggers an alert the user will be notified about it, or the execution will be halted. This approach will collect information such as: specific system calls, file name patterns and network connection endpoints. With the collected data a set of policies are enforced and continuously checked for any suspicious behaviour. This is implemented by using two open source tools: Sysdig and Falco.

Sysdig

Sysdig [32] is used to collect the previously mentioned container information. It does this by leveraging a kernel module named sysdig_probe which is capable of collecting containerized process information as events. In other words, syscalls are translated into events which include information regarding the context in which the syscall was performed such as:

- Process performing the syscall;
- Process parents;
- IP address to which the process is communicating;
- Path of the file being read/written;
- Memory usage.

Falco

Open source tool from the Cloud Native Computing Foundation [33] project, capable of alerting when something suspicious is happening within the containers. The user can define what type of activities are considered suspicious and Falco will alert accordingly. Some examples are alerting when a shell is spawned, a container running in privileged mode, reading a sensitive file and many others.

In listing 3.7 we provide an example of a Falco configuration file for alerting when a shell is spawned inside a container. If all the conditions are evaluated to true then the parameter ”output” will be displayed to the user.

```
- macro: container
  condition: container.id != host

- macro: spawned_process
  condition: evt.type = execve and evt.dir=<

- rule: run_shell_in_container
  desc: a shell was spawned by a non-shell program in a container. Container
    entrypoints are excluded.
  condition: container and proc.name = bash and spawned_process and proc.pname
    exists and not proc.pname in (bash, docker)
  output: "Shell spawned in a container other than entrypoint (user=%user.name
    container_id=%container.id container_name=%container.name shell=%proc.name
    parent=%proc.pname cmdline=%proc.cmdline)"
  priority: WARNING
```

LISTING 3.7: Falco Sample

Falco makes use of Sysdig captures and performs checks with the collected container data to detect suspicious activities based on a preconfigured set of rules. In order for Falco to successfully stop malicious behaviour the user must configure all the necessary conditions for that specific event. At the moment Falco works on single event bases and does not provide detection of a collection of events.

Implementation

In figure 3.2 we present the architecture for the security monitoring solution. The solution integrates both Sysdig and Falco in order to prevent malicious activity of happening in the containers. First Sysdig is deployed and the sysdig_probe attached to the kernel, here all the syscalls related to the running containers will be captured as events. Then Falco will examine the captures and if the rules that were created in advance are evaluated to true, the anomaly detector will spawn a notification to the user and if intended the process will be halted.

The previously described solution was capable of mitigating several vulnerabilities within a containerized environment, Falco presents a very flexible rule creation which enables the user to configure specific rules for known attacks or common misconfigurations.

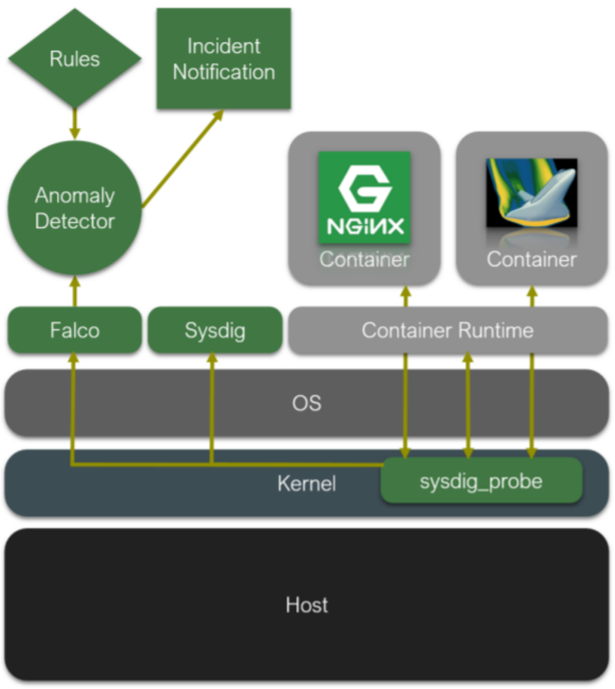


FIGURE 3.2: Secure Monitoring Architecture.

3.7 Container hardening based on trusted computing

Previous work [34] has been done in order to secure the docker image by leveraging cryptography algorithms, integrity measurements and real time monitoring. The process of downloading and running an arbitrary image in a system presents several attack vectors. An initial compromised may occur if the image is not trustworthy, in other words, an attacker placed a malicious image in docker hub and waited for a victim to use it, in order to compromise the victim system. Other scenarios present vulnerabilities using outdated software present in an image. Another possibility is the user downloaded a trustworthy image, however during the download this image has been tampered with by an attacker, the user will blindly execute the image in his system, believing it is trustworthy and ultimately be compromised.

To resolve the previously explained problem a trusted computing approach has been developed. It all starts before the docker image is downloaded, the user must ensure that the publisher of the image is verified. This can be done in a number of ways, as detailed in previous sections. Once the possibility of an image being malicious to begin with is excluded, the user can download the image from docker hub.

Regarding the second scenario where a trustworthy image is downloaded but has outdated software which introduces vulnerabilities to the system. The paper uses an image scanning technique, in the example provided the software Trivy was chosen. Trivy will scan all the version of the dockerfile component and check them against a vulnerability database. If the image has known vulnerabilities Trivy will alert the user, after the known vulnerabilities have been assessed the user can accept the risk and execute the image or halt the whole process.

After the image publisher is verified and no known vulnerabilities are reported, the final step is to ensure that during the download process the image has not been tampered with. Once the image is in the system an integrity verification is performed using sha-256 the user can compare this information against the publisher docker hub dashboard. Additionally every time this image is executed a new re-measure will occur, by comparing the sha-256 output to previous metric value calculations which will be stored under the lib directory. During the image active life cycle the paper also uses container monitoring tools to view the resources that the container is using, this ensures that if a container is malicious and using too much resources the user will noticed it and terminate its execution.

Once all the previous explained step have been done, the user can safely execute the image in his system. The entire workflow is demonstrated in figure 3.3.

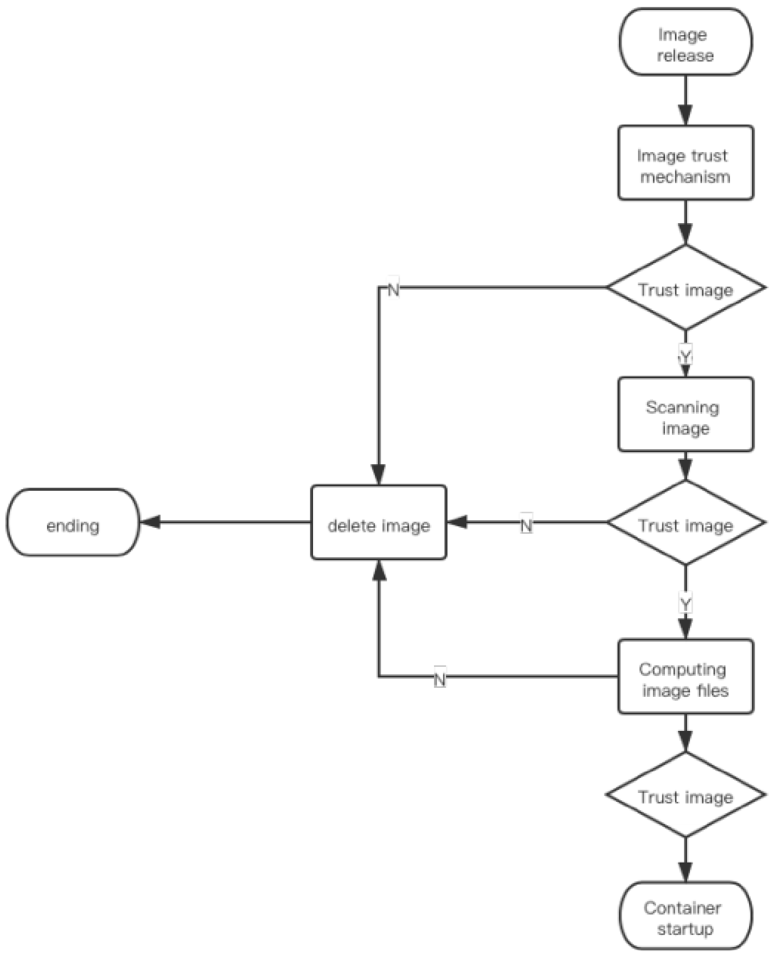


FIGURE 3.3: Workflow Pipeline.

Chapter 4

Implementation

In this section we will explain in detail how our proposal is implemented. We start by describing how our tracing solution works, in other words, the process of capturing syscalls from a container and generate a new Seccomp profile based on them. We follow by explaining how it is possible to integrate the previously described solution with the CI pipeline and how this removes the hassle and complexity of configuring each profile. We finish with a break down about how fuzzing integrates with this whole process and when it is recommended to use.

4.1 Architecture Overview

Our proposal is to help the community embrace the benefits of having reliable Seccomp profiles. We aim to remove Seccomp's main hurdles: the complexity to configure profiles and the difficulties to keep them up to date. Seccomp profiles require knowledge about the syscalls needed by a container, therefore the process starts by implementing our solution to trace every syscall performed by a given container.

In order to easily maintain the Seccomp profiles our tracing solution will be integrated with the Continuous Integration/Continuous Development (CI/CD) pipeline, as it is considered a best practice and a lot of companies, and more importantly Talkdesk, are migrating to this environment where it is possible to consistently build and test if the application works as expected. In our approach, we decided to create a new step in the pipeline.

Whenever a new feature is added, our tracing solution will capture all the syscalls that are generated by the unit/integration tests and create an updated Seccomp profile as previously described. In order for us to successful integrate with a CI/CD pipeline we require the tracing solution to already be running in the system. Whenever new code gets pushed into the repository, this will trigger the continuous integration pipeline where Docker will build a container with that application and run the unit/integration tests, this represents the first three stages from figure 4.1. Since our tracing solution is already running, all the syscalls generated by the unit/integration tests will be captured and a Seccomp profile will be generated based on them. The final stage in figure 4.1 represents the deployment of the container with the custom Seccomp profile. This workflow is illustrated in figure 4.1.

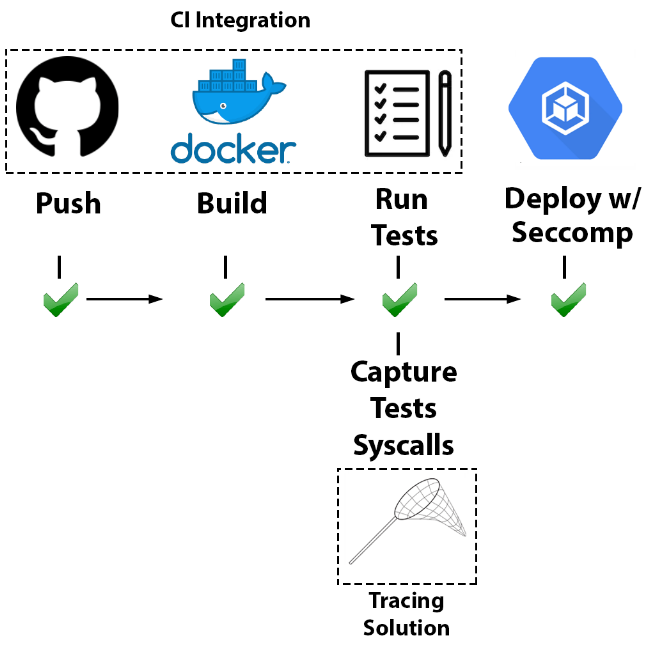


FIGURE 4.1: Proposal Workflow

The base architecture of the tracing solution consists of two components: the Container Tracing and the Container Exit Check. Obviously in order to capture the syscalls, a Testing Container needs to exist. To note that this is decoupled from our tracing solution as seen in figure 4.2. The Testing Container represents the containerized application that will be traced and later deployed with a custom Seccomp profile.

This workflow will start as soon as a new Testing Container is spawned, since it will start to generate syscalls from a different namespace thereby alerting the Container Tracing. The latter is responsible for capturing syscalls from containers, and writing them into an output file. In order to know when the container has exited, the Container Tracing performs a unary gRPC call to the last component, the Container Exit Check, to monitor its state. This last component must check when a specific container has finished its execution and then respond to the gRPC call. This check is not done inside the Container Tracing since it creates significant overhead. For this reason we decided to have a separate component responsible for this task. When the container ends its execution, the Container Tracing will take the capture files and generate a JSON Seccomp file that whitelists the syscalls that were traced. Upon completion, it is possible to move onto the last stage of our workflow and deploy the container assigning it the custom Seccomp profile. This architecture is illustrated in figure 4.2.

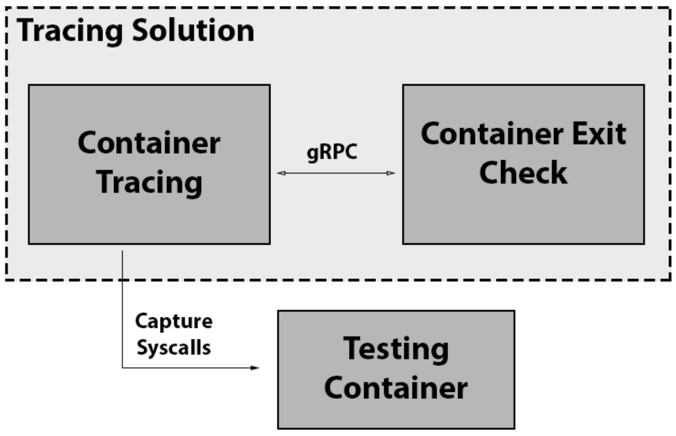


FIGURE 4.2: Tracing Solution Architecture

It may be difficult to fully cover the needed syscalls in more complex containers as there is always the possibility of a false negative occurring (a syscall that is not in the profile but should be). To respond to this challenge, a fuzzing solution is proposed in an attempt to increase the coverage of needed syscalls. The solution will deploy custom fuzzing containers which will interact with the target containerized application and generate new code paths. The user has total control on how each fuzzing container will operate, by choosing/creating a container capable of communicating with the target application and specifying it in a configuration file. This file will then create a number of sibling containers which will fuzz the target application. Considering that fuzzing is usually used to discover the presence of vulnerabilities it should be done with caution, and seen as totally optional. After all, fuzzing could explore unwanted code paths grating an attacker a bigger attack surface. However if the unit/integration tests do not provide sufficient coverage on all of the functionalities that the container has, fuzzing is recommended, as long as the user accepts the risks.

In order to facilitate the tracing process we have built a dockerfile containing all the dependencies needed to run our tracing solution. The dockerfile is based on an Ubuntu image and it automatically installs gnupg, grpcio, python3, python3-bcc and bcc-tools. We provide the dockerfile in listing 4.1.

```
FROM ubuntu

RUN apt-get update \
    && apt-get install -y gnupg

RUN apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 4052245BD4284CDD

RUN apt-get install lsb-core -y

RUN echo "deb https://repo.iovisor.org/apt/$(lsb_release -cs) $(lsb_release -cs
    ) main" | tee /etc/apt/sources.list.d/iovisor.list

RUN apt-get update

RUN apt-get install -y bcc-tools libbcc-examples linux-headers-$(uname -r)
    python3-bcc python3-pip

RUN pip3 install grpcio

RUN pip3 install grpcio-tools
```

LISTING 4.1: Dockerfile

4.2 Capture syscalls and generate a Seccomp profile

One of the first problems we encountered, was having a reliable method of capturing the syscalls. The solution here proposed is to make use of eBPF tracing capabilities which creates a kernel hook on a given syscall. Tracing syscalls used by the container happens when a syscall is called, causing our eBPF program to also be called. Listing 4.2 represents a general eBPF module, in this case for the syscall write. The following code snipped, is a BPF function that will log what syscall was called by the kernel and which namespace came from, to accomplish this the function bpf_trace_printk() is used to send both these parameters to the BCC program as output.

```
int syscall_write(void *ctx) {
    struct task_struct *task;
    task = (struct task_struct *)bpf_get_current_task();
    char * uts_name = get_task_uts_name(task);

    if (uts_name){
        bpf_trace_printk("%s:write\n", get_task_uts_name(task));
    }
    return 0;
}
```

LISTING 4.2: Sample eBPF Module

Understandably different operating systems have different syscalls, therefore we created a program capable of automatically generating all the necessary kernel hooks based on a syscall wordlist. To facilitate this process we provide a list of 407 different syscalls, the user is free to add or remove any given syscall and has full control on what syscalls will be traced, a sample of this wordlist can be seen in listing 4.3. To generate new kernel hooks functions simply execute generator.py, this program will iterate through the wordlist and create a new kernel function for each syscall. This can be easily accomplished by having a general kernel function and then replace the needed values with the syscall name, as is demonstrated in the listing 4.4. An important note is that in case the user desires to change any of the kernel hooks behaviour he/she can do so by changing the generalModule contents.

```
arch_prctl
rt_sigreturn
iopl
ioperm
modify_ldt
mmap
set_thread_area
get_thread_area
set_tid_address
fork
vfork
clone
clone3
unshare
```

LISTING 4.3: Sample Syscall Wordlist

```
Syscalllpath="syscalls.txt"

header="""
#include <uapi/linux/utsname.h>
#include <linux/pid_namespace.h>
struct uts_namespace {
    struct kref kref;
    struct new_utsname name;
};
static __always_inline char * get_task_uts_name(struct task_struct *task){
    return task->nsproxy->uts_ns->name.nodename;
}
"""

generalModule="""
int syscall_$(void *ctx) {
    struct task_struct *task;
    task = (struct task_struct *)bpf_get_current_task();
    char * uts_name = get_task_uts_name(task);
    if (uts_name){
        bpf_trace_printk("%s:$\\n", get_task_uts_name(task));
    }
    return 0;
}
"""
```

```python


def main():
    fd=open("modules.c", "w")
    fd.write(header)
    with open(Syscalllpath , "r") as f:
        syscalls=f.readlines()
        for syscall in syscalls:
            syscall=syscall.strip()
            module=generalModule.replace('$',syscall)
            fd.write(module)


if __name__== "__main__":
    main()
```

LISTING 4.4: Kernel Hook Generator

Once all the kernel hooks are generated we proceed to match each one with a different entrypoint. This is where we define that when the syscall write gets called by the kernel, the BPF program in listing 4.2 is executed. The most common way of defining new entrypoint is either using the function attach_kprobe() or attach_kretprobe(). The difference being that, when using attach_kprobe() the described BPF kernel hook will execute before the syscall starts its execution, however when using attach_kretprobe() the kernel hook will execute only when syscalls exits. In our case we could use either functions since we are interested in knowing only which syscalls were called. In listing 4.5 it is visible that we are using attach_kprobe(), in the first argument we detail the entrypoint in this case we use the function get_syscall_fnname() to define the syscall to be traced, the second argument is related to the kernel hook function name, therefore mapping one to another. This process will occur for every syscall in the previously mentioned wordlist.

```
logf = open("logTracer.log", "w")
prog=load_modules()

b = BPF(text=prog)
syscalls=load_syscalls()

for syscall in syscalls:

    syscall=syscall.strip()
    try:

        b.attach_kprobe(event=b.get_syscall_fnname(syscall), fn_name="
syscall_"+syscall)
        logf.write("Tracing "+syscall+'\n')
    except:

        logf.write("Failed to trace "+syscall+'\n')

logf.close()
```

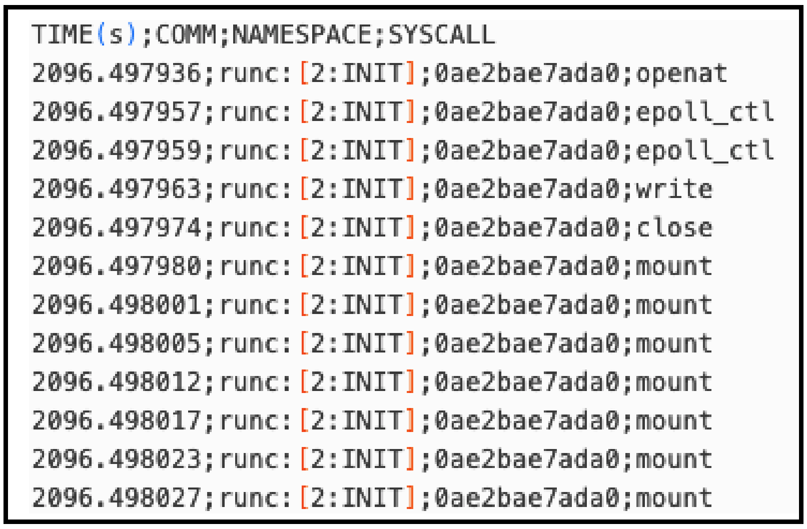
LISTING 4.5: Hooking

Finally there must be a check to determine from what namespace the syscall came from. Since we are only interested in capturing the Testing Container syscalls, in order not to clutter the output all the syscalls performed from the host and by the Container Tracing itself will be ignored. The first step to accomplish this is to determine the host and container namespace, this is defined in the variables: hostnameContainer and hostnameHost. Secondly, the function trace_fields() is used in order to retrieve the output from the kernel hook (function bpf_trace_printk()) after some string manipulation we check and if this comes from a different namespace, meaning that the syscall was called from a container, if it passes the verification we log the syscall as well as some parameters related to the environment.

```

hostnameContainer = socket.gethostname()
hostnameHost= os.environ['HOST_HOSTNAME']

while 1:

    try:

            (task, pid, cpu, flags, ts, msg) = b.trace_fields()

        except KeyboardInterrupt:

            print("Exit")
            exit(0)

        msg=msg.decode("utf-8")
        task=task.decode("utf-8")
        msg=msg.split(':')
        uts=msg[0]
        syscall=msg[1]

        if (uts!=hostnameHost and uts!=hostnameContainer):

            if uts not in fileDesc:

                fd = open("Captures/"+uts+".cap", "w")
                fd.write("%s;%s;%s;%s" % ("TIME(s)", "COMM", "NAMESPACE", "
    SYSCALL\n"))
                fileDesc[uts] = fd
                x = threading.Thread(target=sendMessage, args=([channel,uts]))
                x.start()
            else:

                fd=fileDesc[uts]
            try:

                fd.write("%f;%s;%s;%s\n" % (ts, task, uts, syscall))

            except Exception:

                print("Error on "+uts+" "+ task+ " "+syscall)
```

LISTING 4.6: Namespace Check

Our BCC program will create a kernel hook for each syscall in the system and then write the syscall called to the specific output file. The file with the captured data has several parameters: the time on which the syscall was called, the process that called it, the namespace and finally the syscall itself. There is the possibility of adding more parameters, these will provide more information about the environment in order to facilitate any type of data analysis. Figure 4.3 is a sample of what a capture file might look like.



```
TIME(s);COMM;NAMESPACE;SYSCALL
2096.497936;runc:[2:INIT];0ae2bae7ada0;openat
2096.497957;runc:[2:INIT];0ae2bae7ada0;epoll_ctl
2096.497959;runc:[2:INIT];0ae2bae7ada0;epoll_ctl
2096.497963;runc:[2:INIT];0ae2bae7ada0;write
2096.497974;runc:[2:INIT];0ae2bae7ada0;close
2096.497980;runc:[2:INIT];0ae2bae7ada0;mount
2096.498001;runc:[2:INIT];0ae2bae7ada0;mount
2096.498005;runc:[2:INIT];0ae2bae7ada0;mount
2096.498012;runc:[2:INIT];0ae2bae7ada0;mount
2096.498017;runc:[2:INIT];0ae2bae7ada0;mount
2096.498023;runc:[2:INIT];0ae2bae7ada0;mount
2096.498027;runc:[2:INIT];0ae2bae7ada0;mount
```

FIGURE 4.3: Sample Capture File

Additionally, our tracing solution outputs another file, named logTracer.log. This file is responsible for letting the user know which syscalls are being traced. Some operating systems may have syscalls with different names or some may simply not exits. In listing 4.7 we can see a sample logTracer.log file, every time our tracing solution successfully hooks onto a syscall will create a new entry stating that is tracing the given syscall. However if our tracing solution fails to hook onto the syscall will create a new entry stating that it failed to trace the syscall.

```

Tracing set_thread_area
Tracing get_thread_area
Tracing set_tid_address
Tracing fork
Tracing vfork
Tracing clone
Tracing clone3
Tracing unshare
Tracing personality
Tracing waitid
Tracing exit
Tracing exit_group
Tracing wait4
Tracing waitpid
Failed to trace sysctl
Tracing capget
Tracing capset
Tracing ptrace
```

LISTING 4.7: Sample Log File

One of the advantages of using eBPF is that it provides flexibility on which container technology to use. Since the syscalls are being captured at a kernel level this approach works on Docker, Podman and even other container technologies that have not been yet developed. Nonetheless, there is the possibility of having any method do this and in case another tracing technology is preferred the output should be a file with one syscall per line. This can easily be accomplished with a tool such as Strace [5] as long as the arguments used by the syscall are erased. The last part to successfully capture all the syscalls from a container is to know when to stop. For this, the Container Tracing sends a unary gRPC call to the Container Exit Check. This communication is done using a Protobuf where, listing 4.8 represents the used gRPC service.

```
syntax = "proto3";

message Uts {
    string uts = 1;
}
message Confirmation {
    int32 confirm = 1;
}

service Communication {
    rpc AddUuts(Uts) returns (Confirmation) {}
}
```

LISTING 4.8: gRPC Service File

The string 'uts' is passed in the 'Communication' service and is used to specifying the Docker container identifier. Using this information the Container Exit Check can make use of the Docker Software Development Kit (SDK) for Python and check the status for that specific container, once the status is "exited" it can respond to the gRPC call, by sending the 'confirm' message. Once the call is answered, the Container Tracing will no longer expect syscalls from that namespace and can start generating the custom Seccomp profile. This is demonstrated in listing 4.9 which represents the Container Exit Check. A new channel is created in port 50051, used to receive gRPC calls, when a new container is spawned the function AddUuts will add the container's namespace to the variable request.uts and check every second if the container has exited.

```

# open a gRPC channel
channel = grpc.insecure_channel('localhost:50051')

# create a stub (client)
stub = service_pb2_grpc.ComunicationStub(channel)

client = docker.from_env()

class ComunicationServicer(service_pb2_grpc.ComunicationServicer):

    def AddUuts(self, request, context):

        response = service_pb2.Cofirmation()
        print("Waiting for "+request.uts+ " to exit")

        container = client.containers.get(request.uts)
        containerCheck=container.status.strip()

        while containerCheck!="exited":

            sleep(1)
            container = client.containers.get(request.uts)
            containerCheck=container.status.strip()
            #print(containerCheck)
        try:

            response.confirm = 1
        except:

            print("Error on gRPC.\n\tPlease don't use flag --rm on the container to be traced.")
        return response
```

LISTING 4.9: Container Exit

To generate a Seccomp profile we take the capture file and compile a list of syscalls with no duplicates, once this is done we simply whitelist those and set a default action to block all the syscalls that were not mentioned. The listing 4.10 represents a sample seccomp profile generated using our previously described solution.

```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "syscalls": [
        {
            "names": [
              "openat" ,
              "epoll_ctl" ,
              "write" ,
              "close" ,
              "mount" ,
              "prctl" ,
              "read" ,
              "nanosleep" ,
              "gettid" ,
              "futex" ,
              "sched_yield" ,
              "seccomp" ,
              "brk" ,
              "epoll_pwait" ,
              "getdents64" ,
              "fstatat" ,
              "newfstatat" ,
              "fcntl" ,
              "capget" ,
              "socket" ,
              "connect" ,
              "lseek" ,
              "getpgrp" ,
              "getrlimit" ,
              "ioctl" ,
                "shutdown"
    ],
              "action": "SCMP_ACT_ALLOW"
        }
    ]
}
```

LISTING 4.10: Sample Seccomp Profile

Once all has been implemented, it is possible to start to trace containers and generate a custom Seccomp profiles for our needs. These profiles can be used when running a new container, simply specifying the profile location in the *–security-opt seccomp* flag using the Docker command line interface. As a proof of concept we took a vulnerable Apache container with an RCE vulnerability in Apache-Struts, CVE-2018-11776 [35], we traced the syscalls and generated a custom profile. Once the container with the custom profile was deployed, it was concluded that it was no longer possible to exploit the RCE vulnerability.

Requirements

In order to set up this environment, a few requirements must be installed in the system. The Container Tracing is the easiest to set up, since it only requires Docker to be installed. We provide a Dockerfile with all the dependencies needed, an important note is that in case you are using eBPF as a tracing method Linux 4.1 or above is necessary.

Regarding the Container Exit Check you need Python3 and some custom Python packages such as: Docker SDK and gRPC are necessary. The Docker SDK is used to check the status of a container and the gRPC is needed to answer the gRPC call from the Container Tracing.

4.3 Pipeline Integration

Setting up the environment

Taking into account the architectural requirements we made the following choices for the CI/CD environment setup:

- **Software Versioning Control System** - We will be using Gitlab to hook on to our Tracing server. Whenever new code gets pushed onto the Gitlab repository the hook will trigger on the Tracing server a new build of the application will start.
- **Tracing Server** - This component is responsible for hosting the Gitlab CI/CD Runner for the automated build process, along with all the complementary tools for tracing syscalls and generating new seccomp profiles.

To setup our Tracing Server, we begin by creating an EC2 Instance in Amazon AWS running Ubuntu 20 with the following hardware specifications: type t2.micro, vCPUs 1, memory 1 GiB, architecture x86_64.

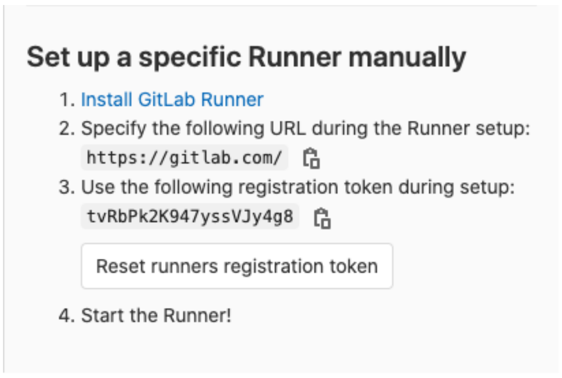
In order to connect to the instance, an Elastic IP was configured this represents a static public IP address in AWS terms. Secondly, we configured a Security Groups detailing that all traffic related to SSH is allowed from any IP. The security group was defined as follows: type SSH, protocol TCP, flow 22, port range Inbound, source 0.0.0.0/0, description ssh-inbound-all.

With the ec2 instance configuration in complete, we can follow with the installation of Gitlab CI/CD Runner. After connecting to the server we can run the following commands available in listing 4.11.

```

sudo curl -L --output /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64

sudo chmod +x /usr/local/bin/gitlab-runner

sudo useradd --comment "GitLab Runner" --create-home gitlab-runner --shell /bin/bash

sudo gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner

sudo gitlab-runner start
```

LISTING 4.11: Runner Installation

At last, we can register our local Gitlab CI/CD runner to the GitLab repository. This can be achieved by accessing the GitLab interface and getting the GitLabCI-token, which is located in CI/CD under the Settings tab. We provide a sample GitLabCI-token in image 4.4. With this token in possession register the runner and follow the instruction as seen in listing 4.12.



Set up a specific Runner manually

1. Install GitLab Runner
2. Specify the following URL during the Runner setup:
   `https://gitlab.com/`
3. Use the following registration token during setup:
   `tvRbPk2K947yssVJy4g8`

   Reset runners registration token
4. Start the Runner!

FIGURE 4.4: GitLab CI Token.

```
sudo gitlab-runner register

Runtime platform arch=amd64 os=linux pid=2457 revision=1b659122 version=12.8.0

Running in system-mode.
Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com/):
https://gitlab.com/

Please enter the gitlab-ci token for this runner:
tvRbPk2K947yssVJy4g8

Please enter the gitlab-ci description for this runner:
[ip-172-39-53-184]: Runner

Please enter the gitlab-ci tags for this runner (comma separated):
tracing
Registering runner... succeeded runner=tvRbPk2

Please enter the executor: docker, docker-ssh, parallels, shell, ssh,
    virtualbox, docker+
machine, kubernetes, custom, docker-ssh+machine:
shell

```

```
Runner registered successfully. Feel free to start it, but if it's running
    already the config
should be automatically reloaded!
```

LISTING 4.12: Runner Registration

This proof of concept was done using GitLab CI/CD pipeline [36]. The first step is to configure GitLab runner. This is responsible for creating a container, running the unit/integration tests and sending the results back to GitLab. In order to set up the GitLab Runner there are two options: locally and remotely. The remote option makes use of GitLabs shared runner hence, all of the process will happen on their servers, since our solution requires the capture of syscalls generated this is not a feasible option. Therefore the pipeline integration must be done locally. Start by installing *GitLab-Runner* on the system, setting up the runner using the flag *register* and supplying the *gitlab-ci* token for the project. The last step would be to create a file named *.gitlab-ci.yml* in the root of the repository. This file is where the pipeline is defined.

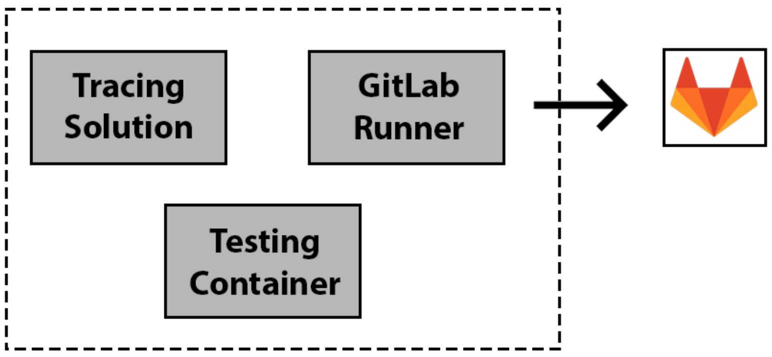


FIGURE 4.5: CI/CD Integration

From figure 4.5 it is clear that GitLab Runner and our tracing solution are running simultaneously, representing long term containers in the architecture, while the testing container is completely mutable and can be created and destroyed according to the pipeline.

This process results in a fully automatic environment that traces syscalls on new commits and generates a Seccomp profile based on those syscalls.

Requirements

One underlying requirement here is that whichever CI/CD technology is chosen it must have an option to perform the unit/integration tests inside a container. If this option is absent, all the syscalls will be generated by the host and will not be traced. Most of the CI/CD technologies have what is called a shell executor, which executes commands in a shell such as:*Docker build* and *Docker run*. Below is a generic example on what a *.gitlab-ci.yml* file would look like in order to build and run a container.

```
build_image:
  script:
    - Docker build -t container .
    - Docker run container
```

LISTING 4.13: CI/CD Sample

4.4 Fuzzing

As previously stated fuzzing should only be done as a last resort when the unit/integration tests alone are unable to cover all the functionalities of the application.

The implementation is based on a configuration file named *conf.json* where the user can configure a custom fuzzing container, define domains/IPs to fuzz and use a file with known inputs that will be mutated in order to see how the application responds to malformed inputs. Our solution provides some general fuzzing containers, which have several entry points depending on the type of technology you want to fuzz. Since the fuzzing is done on a wordlist base it is important that it is rich enough in order to generate new syscall paths, the user can even create a custom grammar to achieve a more complete fuzzing solution. All the containers defined in the *conf.json* file will be executed sequentially.

The workflow depicted in figure 4.6, shows how the fuzzing solutions works. An initial configuration is provided, which will fuzz a general webserver as well as some Transmission Control Protocol (TCP) connections such as sockets. The user can blindly deploy the fuzzing solution and in case any of defined vectors are available, it will fuzz the container.

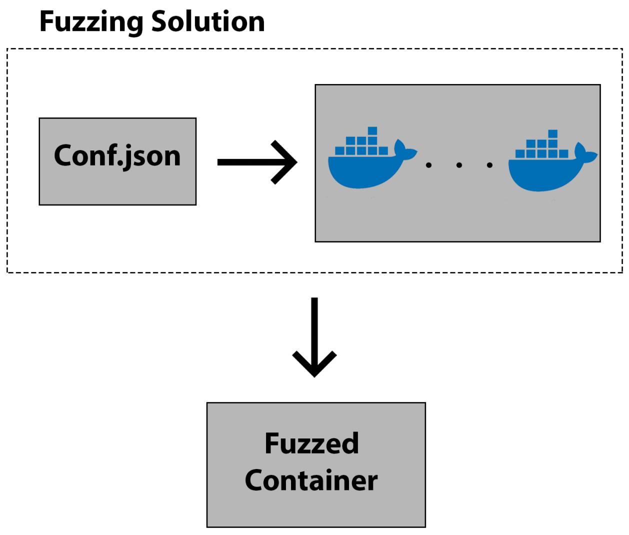


FIGURE 4.6: Fuzzing Solution Workflow

It is recommended that the user provides a custom wordlist, either one for that specific technology or a more general one (good examples can be found in the SecLists repository). Besides the default fuzzing containers the user can add custom containers to the configuration file thereby enabling new entrypoints for other technologies and protocols.

Fuzzing should be conducted with caution to make sure unwanted code paths are not explored. This entails understanding the container and protocols that are being fuzzed as well as the type of wordlist that is being used. Listing 4.14 demonstrates a sample configuration file used to fuzz both a web server and a socket. The parameter 'container' specifies the containers name which is currently in the file system. The parameter 'command' defines any flags needed by the fuzzing container, here the user must configure where the target containerized application is, (IP and port) as well as, the wordlist to be used during the fuzzing process.

More complex containers might require access to a volume, in this case the user can utilize the parameter 'mounthost' describing the path of a directory in the host which will later be bind to the parameter 'mountContainer' inside the container.

```
{
    "fuzz": [
        {
            "container": "nikto",
            "command": "-h http://localhost:8080/"
        },
        {
            "container":"socketclient",
            "command":"python3 client.py localhost 8081 wordlist.txt",
            "mounthost": "/home/ubuntu/Desktop/Share/Runtime/Fuzzer/Socket/
    Client",
            "mountContainer": "/app"
        }
    ]
}
```

LISTING 4.14: Fuzzing configuration file

Requirements

Since several containers are being deployed to perform fuzzing it is only required that Docker and the Docker SDK for Python are installed in order to communicate with the Docker Daemon.

Chapter 5

Evaluation

In this section, the security posture of our proposed solution is evaluated. Starting by identifying some of the drawbacks, followed with an overhead analyses during the continuous integration phase. Finally understanding what steps to take in order to determine how a current Seccomp profile behaves to a specific vulnerability.

5.1 Drawbacks

In order to successfully synchronize the Seccomp profile with the application, extensive and reliable unit/integration tests are required. In the best case scenario they will cover all of the application functionalities including how the application responds to malformed inputs. If this is not the case the user might get overconfident about the profile therefore having a false sense of security. In the case of having an incomplete profile the consequences will be that some functionalities of the application will not work as expected or in the worst case, not work at all resulting in a self-inflicted DoS, hence the need for strong unit/integration tests.

5.2 Fuzzing

When the application does not have a complete set of unit/integration tests a fuzzing solution is proposed. However fuzzing does not guarantee that the full length of needed syscalls will be covered. Therefore if the user relies completely on the fuzzing solution and does not try to implement effective unit/integration tests it can result in a self-inflicted DoS. Fuzzing should be seen as a complement to the tests where high entropy inputs are supplied to the several functionalities.

5.3 Overhead

During the continuous integration phase, all syscalls performed by the containers will be traced, thereby creating some overhead. Container benchmark tests took place in order to better understand how much this phase will be affected. These are divided into CPU intensive operations, network throughput and disk I/O.

These benchmarks were performed thirty times for each type, all the tests were conducted on google cloud with Ubuntu OS and Docker version 19.03.8. The machines used range from: a n1-standard-1 (1vCPU and 3,75 GB of memory), n1-standard-2 (2 vCPUs, 7,5 GB of memory) and n1-standard-4 (4 vCPUs, 15 GB of memory). The following graphs present the average, as well as the standard deviation, of the collected results and a comparison between how long these benchmarks took when our tracing solution is running versus when it is not. The bars coloured in gray represent a system where our tracing solution is running, on the other hand, bars coloured in black represents a system without our tracing solution.

Figure 5.1 presents the overhead analysis of CPU intensive task, when our tracing solution is running the tasks are in average 2x slower.

Figure 5.2 presents the overhead analysis of disk I/O intensive tasks. Here there is a more even playing field, since our tracing solution was only 1.2x slower.

Figure 5.3 presents the overhead analysis of network throughput, when our tracing solution is running the tasks are in average 2x slower.

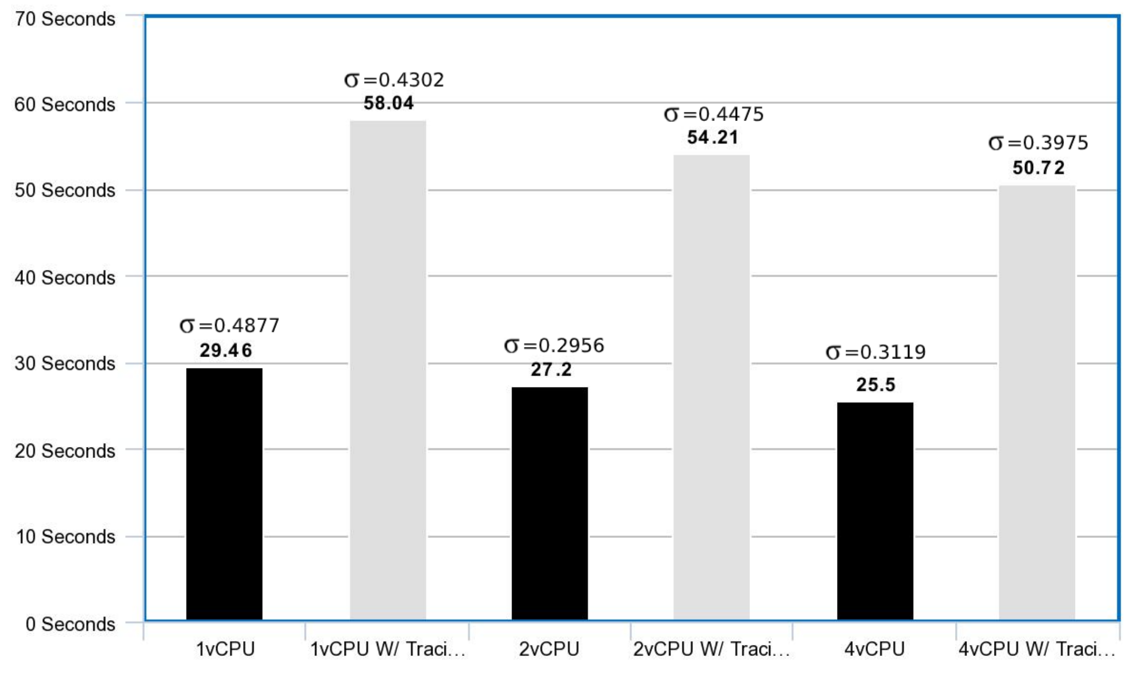


FIGURE 5.1: CPU Overhead graph

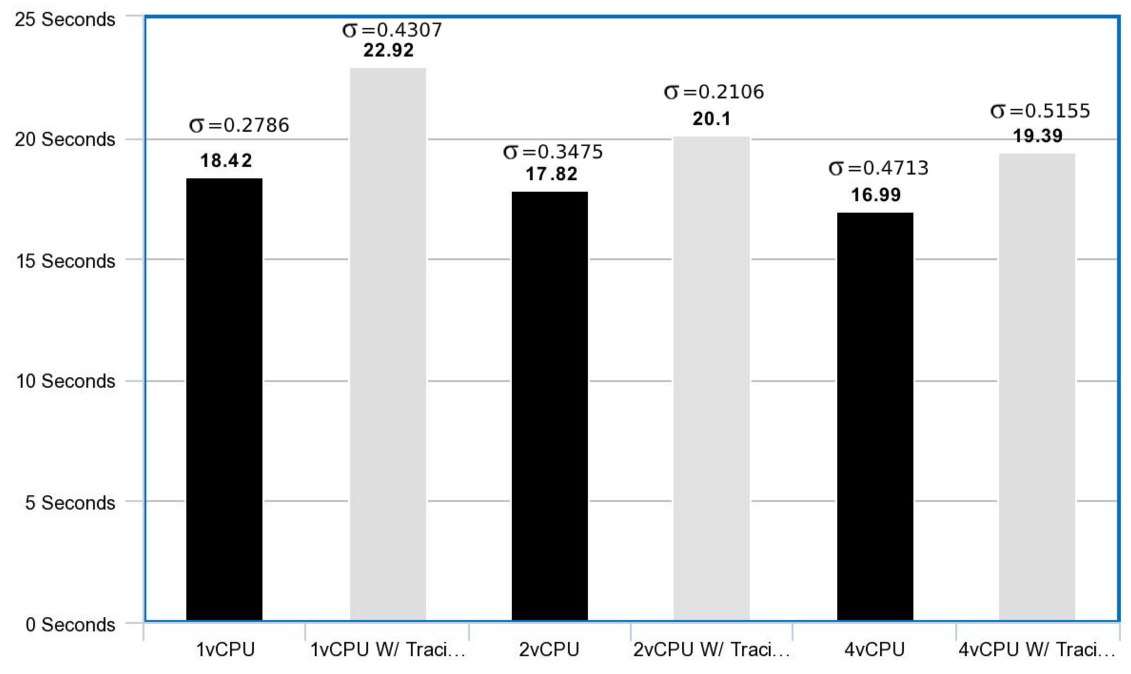


FIGURE 5.2: Network Overhead graph

It can be concluded that while tracing syscalls, both the CPU and network intensive task

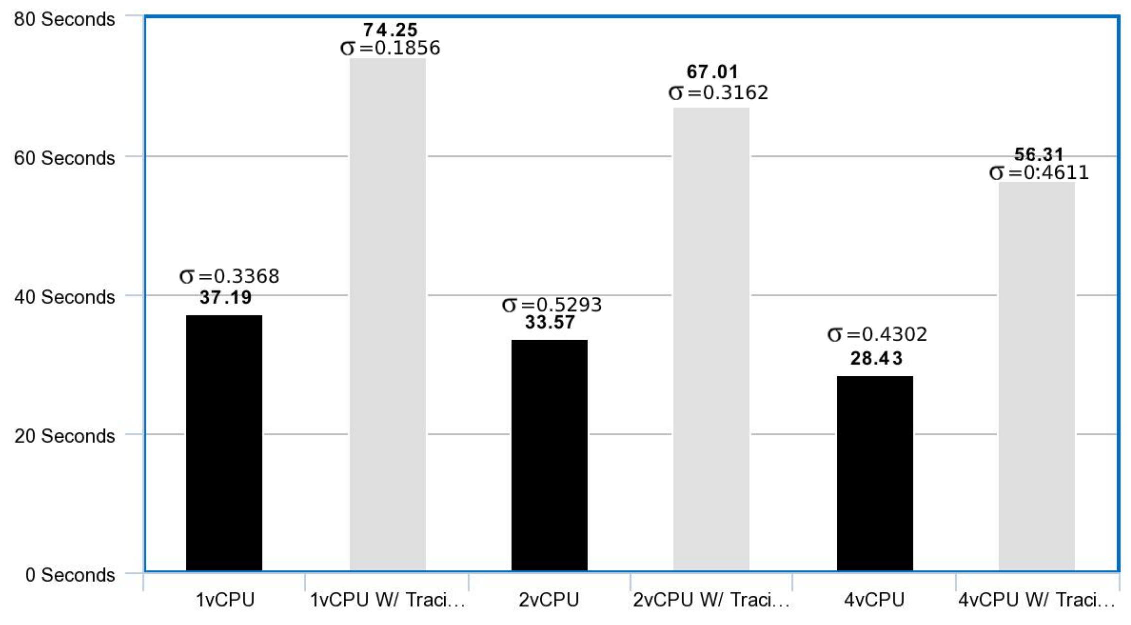


FIGURE 5.3: File I/O Overhead graph

will be approximately 2x slower, on the other hand, disk I/O does not present significant overhead.

5.4 Mitigation

In the case of our current Seccomp profile having syscalls that can be used to exploit a container, the currently proposed solution identifies the dangerous syscalls and offers the option to remove them from the Seccomp profile. Removing a syscall from a Seccomp profile should be done with extra caution since it may damage some functionalities. In order to prevent this, all the functionalities should be tested beforehand to ensure that everything is working as expected.

In order to successfully mitigate a vulnerable Seccomp profile, firstly it is necessary to trace the syscalls from a working proof of concept of the vulnerability. Using our tracing solution described in section 4.1, it is possible to trace and generate a Seccomp profile based on the syscalls used by the proof of concept. Secondly it is necessary for the user to determine which syscalls are needed by the container, which can be done with the help of the unit/integration tests or by the user manually interacting with each needed functionality. This results in two temporary profiles, one that maps the syscalls used by the exploit and the other maps syscalls needed by the application. It is then possible to use our program to check which syscalls are in the profile of the proof of concept and which are not in the other profile. The syscalls returned will be the ones used by the exploit, removing these syscalls from our profile is a temporary mitigation measure to the vulnerability and should be a last resort solution.

Using the previously described Apache container with an RCE vulnerability (CVE-2018-11776) the exploit was analysed and a group of dangerous syscalls was identified. Removing these syscalls from the Seccomp profile resulted in mitigating the vulnerability. Several tests using containers with different vulnerabilities were conducted, such as: CVE-2019-11043 [37] and CVE-2016-10033 [38] which resulted in not only mitigating the vulnerability but also having a better understanding of the exploit. We provide an example on how this works in practice. Where poc.json represents a Seccomp profile mapping the syscalls from the proof of concept container and usecase.json maps the syscalls required for the container to run normally. This is demonstrated in listing 5.1

```
python3 mit.py --poc poc.json --useC usecase.json
    Vulnerable syscalls:
      statfs
      vfork
      select
```

LISTING 5.1: Mitigation script

Here we can determine that the exploit uses the syscalls: statfs, vfork and select. After running the program 'mit.py' a new profile will appear in the same directory where these vulnerable syscalls are removed.

Chapter 6

Conclusion and Future Work

Correctly using a custom Seccomp profile is arguably a more secure approach as it limits the attack surface of a container. We have demonstrated that this can be achieved and incorporated within the CI/CD pipeline thereby removing the hassle of maintaining these profiles up to date. Results have showed that using a custom Seccomp profile can mitigate several vulnerabilities that require syscalls which are not present in the profile.

We believe that bringing attention to a new way of using these profiles is the first step to having a more secure environment by making Seccomp a viable technique. At the moment the cost of using our solution is approximately 2x slower. We recommend everyone to perform an analysis on their own pipeline to understand if having an overhead of twice the time is feasible for their context. We should note that the overhead is only relevant in the unit/integration tests therefore only this phase will be affected.

Future work can be done regarding the arguments of the syscalls, therefore not only whitelisting the syscalls but also the arguments. Furthermore, an interesting approach based on the work done from Lic-Sec is to have multiple profiles depending on the container's execution phase. Since the initialization process usually requires more syscalls than normal execution of the container, it would be interesting to change the attached seccomp profile during the container life cycle. This would result into an even smaller attack vector.

Other projects may analyze the capture file which includes several information about the syscall and its environment. Performing some sort of data analysis on this file could lead into interesting conclusions such as exploit detection. Additionally, having a complete understanding on which syscalls are used on a given exploit, would be helpful in vulnerability research and could result into understanding the risk level on each syscall.

Another interesting approach would be to monitor the blocked syscalls. This could help to understand when a profile is incomplete, and identify attack campaigns by seeing a group of syscalls being blocked across multiple services. Other projects may intend to create a defensive mechanism, start by collecting a list of the most common blocked syscalls and create a honeypot where these are whitelisted, when attacking nodes are identified, custom policies can created such as: block traffic; redirect traffic to a different service; analyse traffic and infer exploits or attack campaigns. This would remove the processing load from the application and improve Computer Security Incident Response Team (CSIRT) intelligence.

A different solution to the problem of creating reliable Seccomp profiles is to instead of tracing syscalls during the CI/CD phase is to do it in a staging environment. The application can be deployed normally and while users/testers are interacting with it, all the syscalls generated should be caught and logged. In order to achieve this we believe that a different tracing solution should be used since the current one presents an overhead that might damage user experience.

Bibliography

[1] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014. [Cited on page 1.]

[2] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gvisor case study," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019. [Cited on page 1.]

[3] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling mec services in fast and secure way," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 2019, pp. 209–214. [Cited on page 1.]

[4] J. Corbet, "Seccomp and sandboxing," *LWN. net, May*, vol. 25, 2009. [Cited on page 1.]

[5] M. Caceres, "Syscall proxying-simulating remote execution," *Core Security Technologies*, 2002. [Cited on pages 2 and 58.]

[6] E. W. Biederman and L. Networx, "Multiple instances of the global linux namespaces," in *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2006, pp. 101–112. [Cited on page 7.]

[7] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A permission check analysis framework for linux kernel," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1205–1220. [Cited on page 9.]

[8] T. Bui, "Analysis of docker security," *arXiv preprint arXiv:1501.02967*, 2015. [Cited on page 15.]

[9] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Advanced Science and Technology Letters*, vol. 66, no. 105-111, p. 2, 2014. [Cited on page 16.]

[10] L. Heinrich, T. Maeno, A. Forti, and P. Nilsson, "Continuous analysis–user containers on the grid," ATL-COM-SOFT-2019-015, Tech. Rep., 2019. [Cited on page 18.]

[11] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017. [Cited on page 21.]

[12] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at facebook and oanda," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 21–30. [Cited on page 21.]

[13] T. E. Levin, C. E. Irvine, and T. D. Nguyen, "Least privilege in separation kernels," in *International Conference on E-Business and Telecommunication Networks*. Springer, 2006, pp. 146–157. [Cited on page 21.]

[14] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8. [Cited on page 22.]

[15] D. Calavera and L. Fontana, *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O'Reilly Media, 2019. [Cited on page 22.]

[16] S. Goldshtein, "The next linux superpower: Ebpf primer," *Dublin: USENIX Association*, 2016. [Cited on page 22.]

[17] X. Wang, H. Zhao, and J. Zhu, "Grpc: A communication cooperation mechanism in distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 3, pp. 75–86, 1993. [Cited on page 23.]

[18] M. Belshe, R. Peon, and M. Thomson, "Hypertext transfer protocol version 2 (http/2)," 2015. [Cited on page 23.]

[19] Z. Jian and L. Chen, “A defense method against docker escape attack,” in *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, 2017, pp. 142–146. [Cited on page 23.]

[20] M. Fox, J. Giordano, L. Stotler, and A. Thomas, “Selinux and grsecurity: A case study comparing linux security kernel enhancements,” 2009. [Cited on page 29.]

[21] R. Yasrab, “Mitigating docker security issues,” *arXiv preprint arXiv:1804.05039*, 2018. [Cited on page 30.]

[22] K. McKay, L. Bassham, M. Sönmez Turan, and N. Mouha, “Report on lightweight cryptography,” National Institute of Standards and Technology, Tech. Rep., 2016. [Cited on page 32.]

[23] S. A. Mokhov, M.-A. Laverdiere, and D. Benredjem, “Taxonomy of linux kernel vulnerability solutions,” in *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*. Springer, 2008, pp. 485–493. [Cited on page 34.]

[24] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar, “Securing docker containers from denial of service (dos) attacks,” in *2016 IEEE International Conference on Services Computing (SCC)*. IEEE, 2016, pp. 856–859. [Cited on page 39.]

[25] R. Rosen, “Resource management: Linux kernel namespaces and cgroups,” *Haifux, May*, vol. 186, 2013. [Cited on page 39.]

[26] F. Lau, S. H. Rubin, M. H. Smith, and L. Trajkovic, “Distributed denial of service attacks,” in *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0*, vol. 3. IEEE, 2000, pp. 2275–2280. [Cited on page 39.]

[27] S. Suneja, A. Kanso, and C. Isci, “Can container fusion be securely achieved?” in *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, 2019, pp. 31–36. [Cited on page 39.]

[28] P. *Δήμου*, “Automatic security hardening of docker containers using mandatory access control, specialized in defending isolation,” 2019. [Cited on page 40.]

[29] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris, "Docker-sec: a fully automated container security enhancement mechanism," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1561–1564. [Cited on page 41.]

[30] H. Zhu and C. Gehrmann, "Lic-sec: an enhanced apparmor docker security profile generator," *arXiv preprint arXiv:2009.11572*, 2020. [Cited on page 42.]

[31] H. Gantikow, C. Reich, M. Knahl, and N. Clarke, "Rule-based security monitoring of containerized environments," in *International Conference on Cloud Computing and Services Science*. Springer, 2019, pp. 66–86. [Cited on page 42.]

[32] N. Sabharwal and P. Pandey, "Container application monitoring using sysdig," in *Monitoring Microservices and Containerized Applications*. Springer, 2020, pp. 235–269. [Cited on page 43.]

[33] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017. [Cited on page 43.]

[34] Y. Shen and X. Yu, "Docker container hardening method based on trusted computing," in *Journal of Physics: Conference Series*, vol. 1619, no. 1. IOP Publishing, 2020, p. 012014. [Cited on page 45.]

[35] "Cve-2018-11776," https://nvd.nist.gov/vuln/detail/CVE-2018-11776, (Last accessed: 14.04.2020). [Cited on page 61.]

[36] J. M. Hethey, *GitLab Repository Management*. Packt Publishing Ltd, 2013. [Cited on page 64.]

[37] "Cve-2019-11043," https://nvd.nist.gov/vuln/detail/CVE-2019-11043, (Last accessed: 14.04.2020). [Cited on page 72.]

[38] "Cve-2016-10033," https://nvd.nist.gov/vuln/detail/CVE-2016-10033, (Last accessed: 14.04.2020). [Cited on page 72.]