· U C ·

Universidade de Coimbra

Master in Informatics Engineering

2017-2018

Security Assessment and Analysis in Docker Environments

Final Dissertation

Student:
Ana Filipa Seco Duarte
afduarte@student.dei.uc.pt

Supervisor:
Prof. Dr. Nuno Antunes
DEI–UC

Jury:
Prof. Dr. Paulo Simões
DEI–UC

Jury:
Prof. Dr. Bruno Cabral
DEI–UC

3rd September, 2018

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

This work is within the software engineering specialization area and was carried out in the Software and Systems Engineering (SSE) Group of the Centre for Informatics and Systems of the University of Coimbra (CISUC).

This work was partially supported by the project ATMOSPHERE, funded by the Brazilian Ministry of Science, Technology and Innovation (51119 - MCTI/RNP 4th Coordinated Call) and by the European Commission under the Cooperation Programme, H2020 grant agreement no `777154`.
It was also partially supported by the MobiWise project: from mobile sensing tomobility advising (`P2020 SAICTPAC/0011/2015`), funded by Fundos Europeus Estruturais e de Investimento (FEEI), by the project METRICS (`POCI-01-0145-FEDER-032504`), both funded by the Portuguese FCT through Programa Operacional Competitividade e Internacionalização - COMPETE 2020.

This work has been supervised by Professor Nuno Manuel dos Santos Antunes, Assistant Professor at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

Cofinanciado por:

COMPETE 2020 PORTUGAL 2020 UNIÃO EUROPEIA Fundo Europeu de Desenvolvimento Regional

This page is intentionally left blank.

Acknowledgements

First of all, I want to thank my parents and my brother for all the support during my life. Without you, I certainly wouldn't be the person that I am today.

In addition, I would like to thank my advisor Nuno Antunes for all the patience and support given during this work, but also, for all the life lessons learned and for being there with an advice in moments that I needed a friend.

During the academic course, I met really awesome people. I would like to thank Luis Ventura, Noé Godinho, Pedro Oliveira, Ivo Gouveia, and Nino Matos for being there for me in hard moments, for being the responsibility for not letting me quit some moments of stress, but specially for being who you are. In the last year, I was lucky to share the same workspace with João R. Campos, Ines Valentim, and João Lopes. You certainly taught me a lot in the last year.

I would also like to thank Kevin, for being so supportive, and to my friends, who do not understand a thing about what I do, but pretended to be interested when I had some problem with my work.

Finally, but not less important I would like to thank not only Filipe Sequeira, but also some previously mentioned people, Noé Godinho, and specially to Luis Ventura, and João R. Campos for reviewing my thesis in the last days.

To all of you a big thank you.

Abstract

Containers are a lighter solution to traditional virtualization, avoiding the overhead of starting and configuring the Virtual Machines (VMs). Since Docker was announced in 2013, it has become the most popular containerization solution, due to its portability, ease of deployment, and ease of configuration. These attributes allow companies to save time in configurations and have led them to migrate some services from VMs when considering these features. However, the security problems that may exist in these environments are still not completely understood.

The goal of this work is to better understand the security of the Docker platform, and what could have been done to prevent its vulnerabilities. To this end, a detailed analysis of the security reports available to the community and the history of security issues was performed. Then, the available information about vulnerabilities was collected to systematize them according to causes, effects, and consequences. This showed that *bypass* and *gain privileges* were the most predominant consequences. Afterwards, a study on the static code analysis tools available for Docker codebase was conducted. The results were analyzed in order to understand the differences between the code with vulnerabilities and the respective corrections. Despite the various reported problems, the results suggest they are not suitable to find the considered vulnerabilities. Finally, a study was performed on some available exploits and correspondent patched code. Through this analysis, it was possible to better understand the cause of the vulnerabilities and their impact on the system. It was also possible to observe that some vulnerabilities could have been prevented if testing techniques, such as robustness and penetration testing, had been employed.

Keywords

Docker, Security, Security Assessment, Static Code Analysis

This page is intentionally left blank.

Resumo

Os *containers* são uma solução mais leve quando comparados com as máquinas virtuais, pois evitam a sobrecarga da inicialização da máquina virtual. Desde que o Docker foi anunciado em 2013, tornou-se na solução de gestão de *containers* mais famosa devido à sua portabilidade, mas também devido ao fácil lançamento e configuração de *containers*. Estes atributos do Docker permitiram que as empresas poupassem tempo em configurações, o que levou a que migrassem alguns dos seus serviços das máquinas virtuais. No entanto ainda não é totalmente conhecido quais são os problemas de segurança nestes ambientes.

O objectivo deste trabalho passa por perceber melhor a segurança na plataforma Docker e como é que as vulnerabilidades poderiam ter sido prevenidas. Para isto, começámos com uma análise detalhada dos relatórios de segurança disponíveis para a comunidade e o histórico dos problemas de segurança. Toda a informação obtida acerca das vulnerabilidades foi recolhida e utilizada para fazer uma sistematização tendo em conta causas, efeitos e consequências. O resultado desta análise demonstrou o escapar do sistema e ganhar privilégios são as consequências mais predominantes. Depois desta análise foi feito um estudo às ferramentas de análise estática disponíveis para o código do Docker, as quais foram aplicadas em código com vulnerabilidades e sem vulnerabilidades. Apesar destas reportarem vários problemas, os resultados sugerem que não são indicadas para encontrar as vulnerabilidades que estão a ser analisadas neste trabalho. Por fim, foi realizado um estudo a alguns ataques e respectivo código de correcção da vulnerabilidade. Através deste estudo foi possível ter uma melhor percepção da causa da vulnerabilidade. Também foi possível perceber que em alguns casos técnicas como testes de robustez e de intrusão podem evitar algumas vulnerabilidades.

Palavras-Chave

Docker, Segurança, Avaliação de Segurança, Análise Estática

This page is intentionally left blank.

Contents

Acronyms

V_P patched version. 32, 33

V_V vulnerable version. 32, 33, 36

CNCF Cloud Native Computing Foundation. 7

CVE Common Vulnerabilities and Exposure. 13, 23–26, 29, 41, 42

CVSS Common Vulnerability Scoring System. 13, 23, 28

CWE Common Weakness Enumeration. 13, 14, 23

DoS Denial of Service. 25, 28–30

FAV First Affected Version. xv, 2, 26, 27

LoC Lines of Code. xiii, 34–38

LXC LinuX Containers. 7, 11, 15

NIST National Institute of Standards and Technology. 13

NVD National Vulnerability Database. 13

OCI Open Container Initiative. 7, 11

OS Operating System. xiii, 1, 2, 5, 6, 8, 10, 12, 15, 51

SCA Static Code Analyzers. 2–4, 31, 33, 36, 49–51

VM Virtual Machine. v, 1, 2, 5–9, 16

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

List of Publications

This dissertation is partially based on the work presented in the following publication:

- **A. Duarte**, N. Antunes,"An Empirical Study of Docker Vulnerabilities and of Static Code Analysis Applicability", *8th Latin-American Symposium on Dependable Computing (LADC 2018)*, Foz do Iguaçu, Brazil, October 8-10, 2018.
  - ***Abstract:*** *Containers are a lighter solution to traditional virtualization, avoiding the overhead of starting and configuring the virtual machines. Docker is very popular due to its portability, ease of deployment and configuration. However, the security problems that it may have are still not completely understood. This paper aims at understanding Docker security vulnerabilities and what could have been done to avoid them. For this, we performed a detailed analysis of the security reports and respective vulnerabilities, systematizing them according to causes, effects, and consequences. Then, we analyzed the applicability of static code analyzers in Docker codebase, trying to understand, in hindsight, the usefulness of tools reports. For a deeper understanding, we analyzed concrete exploits for some vulnerabilities. The results show a prevalence of bypass and gain privileges, and that the used tools are rather ineffective, not helping to identify the analyzed vulnerabilities. We also observed that some vulnerabilities would be easy to find using robustness or penetration testing, while others would be really challenging.*

Part of the results obtained contributed for a project deliverable:

- N. Antunes, M. Vieira, I. Elia, L. Ventura, **A. Duarte**, J. Lopes,"Methodologies for trustworthiness estimation", EUBra-BIGSEA Deliverable 6.4.
  - ***Abstract:*** *This report presents the results of the security assessment of the components of the EUBra-BIGSEA infrastructure. Such assessments have been conducted based on the techniques described in D6.3 and are used for studying trustworthiness aspects. In practice, the deliverable presents the results of the application of the techniques and tools described in D6.3, and discusses the major trustworthiness observations. The results presented contribute to the other work packages of the project by providing security-related evidences that can be used for improvement.*

This page is intentionally left blank.

Chapter 1

Introduction

Cloud computing is built upon the sharing of services and resources with heterogeneous sources. Supporting the cloud with scaling storage improvement is difficult, from both a hardware and maintenance perspective. To improve the ease of scaling and maintaining clusters, virtualization techniques were created.

Over time, the cloud's massive growth has increased the importance of scalability, elasticity and shareability. In order to ensure these attributes with the increasing usage of the cloud, it is necessary to either spend large amounts of money on hardware, or improve resource management [4]. The emergence of **containers has further improved the management of resources**, however it was only in 2014 with the appearance of Docker that their adoption has increased in companies [5].

Containers are not a recent concept. In fact, they were first introduced decades ago when the BSD Operating System (OS) was launched in 1982. However, it was only in the past few years that their popularity has exploded, partially due to **Docker** [6]. Docker is an open source container platform that reduces the complexity of managing and deploying containers, compared to previous platforms [7]. Other platforms were launched before Docker, but they were never capable of reaching its level of popularity [8].

Also known as Operating-System-level virtualization, containers are a lighter option to traditional virtualization that avoids the overhead of starting and maintaining Virtual Machines (VMs) [1]. In particular, Docker provides abstraction and automation of such technologies and its containers wrap up software in a filesystem that contains everything it needs to run: code, runtime, system tools, and system libraries. Although Docker promises security and reliability, it is clear that privilege escalation or code execution attacks, when successful, could be devastating for the infrastructure provider.

Container solutions can be classified in two main types: **OS containers** which are similar to VMs, and **application containers**, which run one application per container and are suitable for microservice architectures. OS containers are useful to run identical or different distribution OSes, creating containers with identical environments. Application containers, such as Docker containers, are ideal to run different applications independently, isolated from each other, and with the resources they need.

Microservice architectures have many services connected to each other, which makes them hard to manage [9]. To handle the management of these types of architecture, with a large number of containers, users have begun to use orchestration tools. **Container orchestration** tools became popular because of the complexity in managing clusters of containers. These tools manage the multiple containers created, ensuring the cluster's scalability and availability. Many alternatives are available such as Kubernetes [10], Mesos [11], and Docker Swam [12], each having its own features suitable for different purposes.

Docker popularity has grown since it was launched, especially in 2014 when version 1.0 was released as ready-to-use for companies. The growth of its adoption has been mentioned in several studies, such as the one by the monitoring service Datadog [13]. Docker is an open source container platform which promises the standard container characteristics, such as being lightweight and portable in most OSes and infrastructures. Additionally, it also offers its own characteristics such as agility, improved portability, security, and cost saving [14]. In addition to the general container advantages, like low resource usage when compared to VMs, many companies have adopted this platform because of its aid to complexity management [8]. The adoption of this platform not only provides easy building, running, and deployment of containers, but also portability, since the containers can be executed in any other machine running Docker.

Due to Docker's increased adoption alongside its advantages, many companies have migrated their services to Docker containers. Some companies which have done this migration even have *business-critical* services running in containers. Therefore, security in Docker is an important aspect to make sure that those companies do not have their services compromised. Sometimes these companies have their containers running in platforms or machines that also have other containers, which could be owned by different organizations. This is another reason for Docker's security being important, since a compromised container can affect other containers running in the same machine.

Security is a crucial aspect to software systems, which must be built to reliably tolerate errors, malicious users, or accidents [15]. When a system has been developed it can be riddled with design flaws and implementation bugs, resulting in defects in the system. Defects can be exploited and used by attackers which results in vulnerabilities that compromise the security of the system. **In the case of Docker, any vulnerability that causes code execution, privilege escalation, or information violation attacks can be devastating to the services running in a platform with shared containers.**

As such, the **main goal of this work is to better understand the security of the Docker platform, and what could have been done to prevent its vulnerabilities**.

To achieve this, it was necessary to study the repositories with Docker's security vulnerabilities in order to understand their causes, effects, and consequences on the system. This analysis systematized the vulnerabilities and was important to understand the history of Docker's security issues since it was launched. To perform the systematization and understand the cause of the vulnerabilities, it was necessary to identify the First Affected Version (FAV). Using the FAV, it was possible to get the **patched and unpatched versions** of each vulnerability.

Afterwards, Static Code Analyzers (SCA) tools were applied to Docker's code, in order to understand if there were significant differences before and after the patches. For this, it was necessary to obtain those that could be applied to the Docker codebase language. An analysis was performed by applying the collected tools to the vulnerable and patched versions of the code, and comparing the alerts between them. Despite the tools being ineffective for identifying the security vulnerabilities, they detected several code-style problems.

Additionally, an analysis of known exploits was performed to understand how they were created and how they affected the system. While some studies focus evaluating the impact of these vulnerabilities, this research aimed at comparing the exploit of the vulnerability and its patch in order to understand how those could have been avoided. This analysis allows to better understand, using the exploits, the causes of the vulnerabilities, and it showed that some of them could have been detected by studying the behaviour of the system under unexpected conditions.

1.1. Contributions

The goal of this work was to assess the security of Docker environments, and understand if its security vulnerabilities could have been avoided. The main research contributions are as follows:

- **The proposal of a systematization of the vulnerabilities affecting Docker** this analysis systematized the cause, effect, and consequences of the vulnerabilities. This provides valuable information to both Docker's developers so they can try to prevent similar problems in the future, and users to understand some of the possible vulnerabilities in Docker. The analysis shows that the most common consequences are bypass and gain privileges, which are key issues in the cloud, where multi-tenancy is a common practice. The high prevalence of bypass and gain privilege is explained by the fact that Docker is a system running at a low-level, and that when exploited it can easily be used to breach and control other components of the system.
- **A study on the effectiveness of SCA tools in identifying security vulnerabilities** despite SCA tools not being effective in detecting the analyzed vulnerabilities, they were capable of detecting other problems in the codebase. These problems could lead to other vulnerabilities being introduced in the future, therefore it is important to remind developers to use this type of tools to improve the quality of the applications. Despite SCA tools were unable to find the security vulnerabilities, their usage should not be neglected, as several problems were identified by them.
- **In-depth analysis of exploits** – this analysis provides some of the known exploits and the respective patched code. By performing this manual analysis it was possible to notice that some of the vulnerabilities could have been detected before they were activated, if some techniques such as robustness and penetration testing had been used. SCA tools can improve the quality of the code, and different testing techniques can identify problems that would be difficult to find if testing manually.

1.2. Thesis Structure

The document is divided in chapters, described as follows:

Chapter 2 presents the background and related work. It introduces the virtualization and the progress of containers in the last decades, and the basic concepts of security. It also presents the architecture and components of Docker, as well as the studies related to Docker's security.

Chapter 3 presents the main research objectives to accomplish this study, and the approach taken to complete these objectives. The approach introduces the methodology to perform the security analysis on Docker platform.

Chapter 4 presents the systematization of the vulnerabilities. This chapter contains the security analysis and the data collected for the characterization. It also provide a time analysis of the vulnerabilities analyzed.

Chapter 5 describes the available tools for static analysis. It begins with a description of the used SCA tools and identification of the code analyzed. This also contains the results applied to the code.

Chapter 6 demonstrate the analysis of security patches and respective exploits. In the analysis is explained why the exploit could not be detected and, in some cases, what could help.

Chapter 7 presents a discussion and the lessons learned about the work developed, and the interpretation of the obtained results.

Chapter 8 contains the main conclusions and the future work. The conclusion makes a statement of the security of Docker and this work. The future work present the steps to continue this research.

Chapter 2

Background and Related Work

This chapter introduces the basic concepts and technologies used in this work, as well as the relevant work already performed in Docker Security. In the beginning of this chapter it will be introduced the concepts of virtualization and containers, followed by Docker architecture and its features. Finally, some concepts of security are introduced as well as related studies with the work developed.

2.1. Virtualization

Cloud computing gives the possibility of having shared computer resources on the Internet, without the need to purchase and maintain physical equipment. This concept has many applications, such as having documents readily-available on any device with an Internet connection, or running a company's services on some server with scalable resources.

Virtualization is a concept which was most likely introduced by IBM somewhere between late 1960's and early 1970's. The objective of IBM was to create a robust time-sharing solution in order to give its users efficiency in the sharing of computer resources [16]. At the time, resources were expensive and limited, which made this a historical mark in computer history.

At its core, virtualization is the creation of a virtual environment that simulates or replicates a real one. This environment can be slightly different from the original one, but it tries to maximize the similarities in order to be capable of performing the same tasks. When using virtualization, a physical machine can run multiple Virtual Machines (VMs), with different Operating Systems (OSes) without conflicts, as they are isolated from each other. The responsible component for making this possible is the **virtual machine monitor (VMM)** more commonly known as **hypervisor**. This component is responsible for creating a virtual environment between the physical machine's hardware and the virtual machines.

Hypervisors can run in bare-metal environments or on top of a host OS, and it is on top of the hypervisors that the guest OSes run. The hypervisor can run multiple guest OSes with different natures thanks to the emulation of hardware and the kernel.

A team of researchers within the Xen project studied a lightweight design to increase the performance, and improve the memory management of VMs [17] named unikernels. Unikernels have a hypervisor in the base of the system like VMs, however, instead of a full OS virtualization, it only visualize the kernel's libraries necessary to run the target application. However, until the present date this technology is not so popular as VMs.

With the ease of worldwide access to the Internet, virtualization has become extremely popular in different types of services, which also brings the need to scale machines to have more resources.

2.2. Containers

Containers and VMs are both virtualization tools, but containers have appeared as a more lightweight solution. VMs virtualize the entire OS, as well as the resources of each VM, such as RAM, storage, and CPU. Containers do not create this abstraction layer between the hardware and software, because they share the same kernel space with the host machine. This makes it possible for containers to virtualize the OS in a similar way to VMs, but in a lighter fashion.

2.2.1. History and Evolution

The original idea of containers goes back to 1979, when the `chroot` system call was introduced in UNIX. The `chroot` allows users to change the root folder temporarily, effectively isolating an application from the rest of the system, preventing it from accessing files outside of the defined `chroot` directory. In 1982 `chroot` was added to BSD which led to the `jail` concept being launched in 2000 with FreeBSD [18]. ***Jails*** in FreeBSD are built upon `chroot` but as an expansion, since not only is the file system isolated, but also the set of users and the networking subsystem. Other systems came up later such as Linux VServer [19] in 2001, Oracle Solaris release Zones [20] in 2004, and OpenVZ [21] which was released in 2005.

By that time, the Linux kernel already had an important feature for process isolation called `namespaces`, whose goal was to support the implementation of containers. `Namespaces` are a specific set of processes isolated by `namespace`, where those inside a group can only see each other, and cannot see processes out of that scope. Currently there are 6 types of `namespaces` provided by Linux kernel [22]:

- `mnt` - mount points
- `pid` - process identification
- `net` - network devices, ports, stacks, etc.
- `ipc` - System V IPC, POSIX message queue
- `uts` - Hostname and NIS domain name
- `user` - User and group IDs

An important feature for containerization was the implementation of process containers in 2006 (which were renamed to control groups in 2007) usually referred to as `cgroups`, and integrated on the Linux Kernel. `Cgroups` are responsible for monitoring and limiting resources for a group of processes. The controlled resources are CPU, RAM, disk, and network.

In 2008, Linux launched `LinuX Containers (LXC)` [23], which were implemented using `cgroups` and `namespaces`. This was the most complete implementation of the Linux container manager. Later, in 2015 *Canonical Ltd* started a new open source project called `LXD` [24], which was built on top of `LXC`.

Despite existing for over a decade, containers have only recently become popular. Actually, it was in 2013 with the release of the Docker container platform that container popularity increased. Docker is a platform where users are able to build, run, and deploy applications in a simple way, which is explained with detail in Section 2.3.

Due to the growing number of containers and their applicability in cloud services, there was also the need to manage them, which led to the appearance of **container orchestrators**. Their purpose was to manage containers in a container platform and allow users to deploy, monitor, scale, and dynamically control the configuration of resources [25]. Many orchestration tools have been developed over the past years. Docker has its own orchestration tool which is called Docker Swarm, but there are others such as Kubernetes developed by Google which is integrated with Docker since version 18.01.0 [26], Marathon from Mesos, and Nomad from HashiCorp.

In 2015 a group of companies created the Cloud Native Computing Foundation (CNCF) and the Open Container Initiative (OCI), both housed in the Linux Foundation. OCI was launched by Docker, CoreOS, and other leaders of the container industry (21 members to be exact) and focus on container formats and runtimes. CNCF was created by Google and 18 other companies and focus on cloud architectures to serve modern applications like containers. Both have the objective of standardizing different aspects of containers. Editor Brandon Butler, from Network News separates their utilities in a clear explanation, referring to the purpose of CNCF as [27]:

"... sort of like creating instructions to build a Lego set, but saying you can use whatever colored pieces you want to actually construct it."

and differing it with OCI as:

"... the Open Container Initiative (OCI) is getting everyone to agree on what size the Lego blocks are, while the CNCF is creating the instructions of how to build the Lego set."

The members present in OCI and CNCF are different, but both have important roles in the create of mature services and platforms for containers, which benefit companies and users.

2.2.2. OS Containers vs Application Containers

Despite Docker's popularity, there are different containerization technologies that can be used in containerization. Containers are used as a lightweight alternative to VMs. A VM emulates a complete OS and loads the kernel into its own memory region, whereas containers run an operating system inside the host's OS, which means that the OS that runs inside the container shares the same kernel space with the host OS, reducing performance overhead.

The usage of containers depends on the user's needs, there are two kinds of containers, as shown in Figure 2.1: OS containers and application containers.

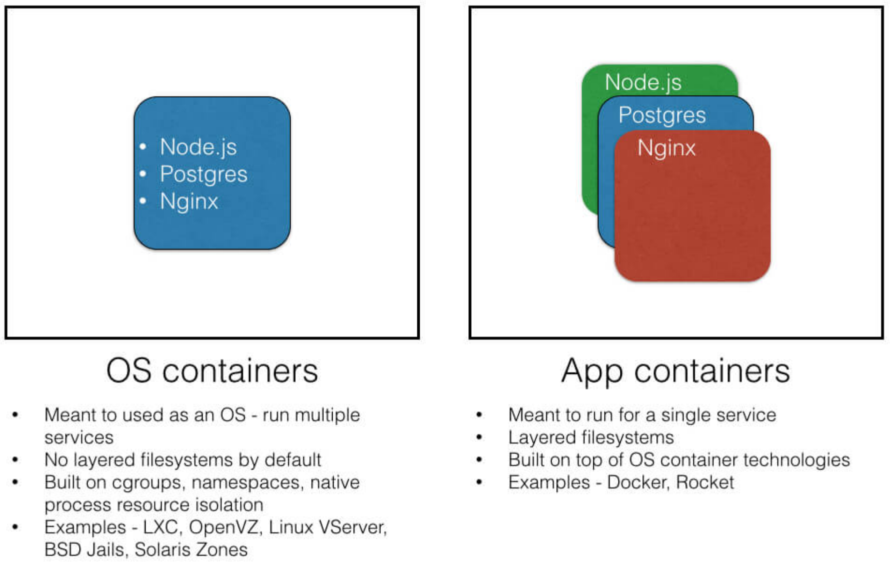

• Node.js
• Postgres
• Nginx
Node.js
Postgres
Nginx
OS containers
• Meant to used as an OS - run multiple services
• No layered filesystems by default
• Built on cgroups, namespaces, native process resource isolation
• Examples - LXC, OpenVZ, Linux VServer, BSD Jails, Solaris Zones
App containers
• Meant to run for a single service
• Layered filesystems
• Built on top of OS container technologies
• Examples - Docker, Rocket

Figure 2.1: Application containers vs OS containers (from [1]).

OS containers are virtual environments that combine `namespaces` and `cgroups` to provide isolation for applications. This type of container is useful to simulate an OS inside a container, using technologies such as `LXC`, `LXD`, or OpenVZ. This is useful to build an environment similar to a VM but with faster start up and better performance. One of its drawbacks is that it is not possible for a Linux system to host a Windows or a Mac OS container.

Application containers are useful to run and isolate a single service. A user can run multiple containers isolated from the system, with a service in each, without having to load the complete OS, focusing only on libraries which are required by the service. This type of container can be used inside OS containers. Docker is one of the most famous example of this type of container, but there are others such as `rkt` [28].

2.3. Docker

Docker is the most popular container platform in the cloud community [8]. Originally, Docker was being developed as *dotCloud* which was a Platform as a Service (PaaS), but it was later released as the Docker project. As mentioned before, Docker was initially released in 2013 but its first stable version only came out in June 2014, when many companies and developers were already using it [29].

Despite Docker not being the first container platform, it was the first one adopted by companies in large scale [13]. The reason for Docker to be adopted in such short time is mainly related to the general features of containers and the fact that they are more lightweight and faster to deploy than VMs. With other existing alternatives, the reason Docker became the most popular container platform was that it reduces the complexity of the deployment of containers and the management of resources available compared to the previous platforms.

Docker is an open source software, with the source code available in a Git repository [30]. In the beginning Docker was open to the community in its GitHub repository, but recently Docker created the Moby project to break the monolithic architecture into components and try to create a place where "container lovers" can exchange ideas [31]. Moby has the objective of enabling and accelerating software containerization, and contains the framework and the toolkit components, which can be assembled into container-based systems.

Docker is being **developed using the Go programming language**, which is claimed to be a great partner due to some of its advantages such as, fast construction, concurrency naturally built-in, and join the low level communication language with the garbage collection to free allocated variables [32, 33].

The Docker platform is composed by Docker Hub [34] and the Docker Engine [35]. Docker Engine is a client-server application composed by the Docker Client, Docker Daemon, and Docker Registry. Docker Client is where the commands are executed by the users. After receiving their commands, the Client communicates to the Daemon which handles the container's management. Both of these components can run in the same system, or be connected remotely communicating via a RESTful API over UNIX sockets or network interface. Docker Registry is the Docker Image storage available to users (explained with more detail in subsection 2.3.1).

Docker containers are created using files with instructions called *images*. An image is a set of instructions given by a file (*Dockerfile*), where each instruction creates a layer on it. This cluster of layers allows a faster image rebuild because only the layers which were changed will be replaced. An image can be based on other ones, and usually the most simple image have a required system libraries as a base image.

This Chapter presents products and tools, execution driver, and security components disposed by Docker. This background was necessary to understand the vulnerabilities and what was compromised.

2.3.1. Products and Tools

Docker has a set of products and tools available for the community to simplify, manage, and secure containerized applications [36, 37]. Those that are most relevant for this work are:

Docker Hub *- Docker containers work based on images. Docker images are the base of containers, in a simple way, an image is composed by the application, the required* *libraries to run it, and the configuration files.* Due to the community around the creation and sharing of images, Docker created Docker Hub, which is a cloud-repository registry service linked to Docker Cloud to deploy images. Through this service, Docker provides its users with a store, into which they can push their images, or build directly from. Docker Hub [34] is divided in two types of public repositories, official and community. Official repositories are the repositories that belong to official companies, like Docker, Postgres, Nginx, and others where they manage their own images and validate them with the community. On the other hand, the community repository is where any individual user or organization can store their images and share them with the Docker community.

Docker Swarm - Sometimes, to launch a complex application, it is necessary to have a cluster of hosts and a manager to control all the containers in the cluster. Docker has integrated Docker Swarm, which is Docker's orchestration solution, to manage clusters and treat multiple hosts as if they were single host, where it is possible to scale them seamlessly [38].

Kubernetes - Similar to *Docker Swarm*, Kubernetes, or K8s, is a tool to orchestrate clusters of containers, scale them if needed, and automatically deploy them. This tool is within Docker recently, and is an alternative to *Docker Swarm* for Docker users.

Docker Compose is a tool with a purpose similar to Docker Swarm. Both are used in orchestration, but Compose is responsible for orchestrating containers instead of Docker hosts. Compose can manage the lifecycle of the application by starting, stopping, or rebuilding the service, and monitoring the services running.

Docker Machine is a virtualization level used to create and manage Docker hosts, allowing the installation of the Docker Engine. Docker Machine can be installed in local machines connected to the network or on cloud providers, supporting a way to provision Docker hosts. Prior to Docker Machine, the Windows OSes users only could run Docker by a tool named ***Boot2Docker*** [39] which was a light distribution of *Tiny Core Linux* crafted to run Docker, now deprecated [40]. Before Docker version `1.12`, Docker Machine was the only way to run Docker on Mac or Windows OSes, but now Docker for Mac and Docker for Windows are available. However, Docker Machine is still used in machines that do not have the minimum requirements to run Docker for Mac and Docker for Windows.

Docker Notary is a tool that supports the Content trust Docker function, to use digital signatures in the data sent and received from the Registry. Notary manages and publishes trusted collections of content in order to ensure the content's integrity.

Docker Registry is the tool responsible for hosting and distributing Docker images. Registry acts as a repository with various Docker images that are labeled, and allow users to pull the most recent image available with that label. This mechanism interacts with the Docker Daemon to give the intended image to the host.

2.3.2. Execution Driver

Some vulnerabilities affect the execution driver and in turn Docker became compromised. Then, it is important to understand how it works and what were the changes since Docker was released.

Docker suffered some changes since it was first released, especially in the default execution driver, which is the responsible for isolating each container's execution environment. Versions of Docker before `0.9` had LXC as the default driver, but subsequent versions changed until the current components:

Libcontainer is written using the Go programming language and authorizes the manipulation of namespaces, `cgroups`, Linux Capabilities, *AppArmor* profiles, network interfaces, and firewalling rules. This package is not dependent on LXC and gave Docker an stability improvement [41]. Later in 2015, Docker contributed with the libcontainer project and other code to OCI and version `1.11` was released with `runC` and `containerd`.

RunC is a container runtime, used to spawn and run containers according to the OCI, making them available everywhere. It includes libcontainer to set the operating system in order to construct containers.

Containerd is the daemon responsible for controlling `runC` and handle the complete container by managing the resources, downloading images and calling `runC` with the appropriate parameters to run containers as show in Figure 2.2.

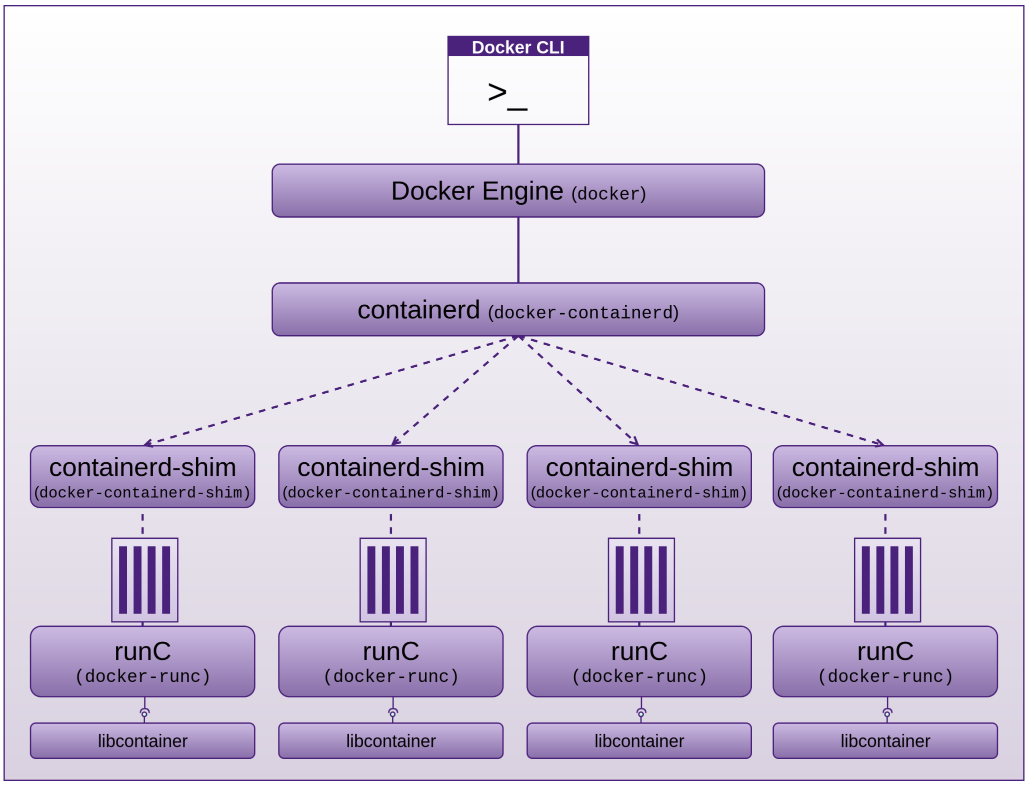

Docker CLI
>_
Docker Engine (docker)
containerd (docker-containerd)
containerd-shim (docker-containerd-shim)
containerd-shim (docker-containerd-shim)
containerd-shim (docker-containerd-shim)
containerd-shim (docker-containerd-shim)
runC (docker-runc)
runC (docker-runc)
runC (docker-runc)
runC (docker-runc)
libcontainer
libcontainer
libcontainer
libcontainer

Figure 2.2: Relation between runC and containerd in Docker engine (from [2]).

2.3.3. Security Components

With security in mind, Docker has various security mechanisms, such as the kernel security and its namesapaces and cgroups, protecting the possible attack surface of the Docker daemon, container configurations, and features of the kernel. Both namespaces and cgroups were already described in Section 2.2.1, and this subsection describes Docker's other mechanisms that ensure container security [42]: AppArmor, Linux Capabilities, and Docker

Daemon.

AppArmor is a component from Linux kernel. This is a security module which protect the OS from application's security threats. AppArmor allows the creation of security profiles for each program needed, where it is possible to define the system resources that each application can access and with what privileges it can do so. Docker has a default profile named `docker-default`, however, it allows users to create custom AppArmor profiles for containers.

Linux Capabilities [43] are distinct units associated with superuser privileges. Before kernel `2.2` permissions were divided into two categories, which was privileged (superuser or root when user ID is zero) and unprivileged (user ID different of zero) users. After version `2.2`, the privileges can be independently enabled or disabled using Capabilities. Most of the container's tasks are handled by the host. In most cases containers would not need the root privileges, only permissions for some Capabilities. Therefore, Docker permits the independent configuration, allowing the addition and the removal of capabilities as needed by its users.

Docker daemon is the process responsible for the management of containers and getting the images prepared to build, run, load from the disk, or pulling them from the registry. This daemon runs with root privileges, and for this reason is important that it is ran by trusted users only. Docker daemon suffered some changes since its first version, in order to become more secure and less exposed to malicious users who want to create arbitrary containers. One of the changes was the replacement of the RESTfull API endpoint with UNIX sockets. Docker's documentation refers the daemon as "potentially vulnerable" when loading inputs like images. In order to turn this functionality less vulnerable, since version `1.10.0` Docker stores all the images with cryptographic checksums on their content to prevent collision attacks with existing images.

2.4. Software Security Concepts

Nowadays it is common to hear about cyberattacks on systems, and the protection of users. System insecurities are provoked by technical issues, such as the large number of lines of code, exposure to the Internet, or even the security mechanisms of different applications adversely affecting one another. The protection of these insecurities can be summarized in one word, security. **Security** is one of the most important aspects to consider during the systems development, and it includes several quality attributes [44]:

- **Confidentiality** - preventing the access to undisclosed information to unauthorized entities.
- **Integrity** - protection of the system and its information against unauthorized changes.
- **Availability** - the system or the information must be readily accessible and protected from malicious denial of service.
- **Authenticity** - the identity of an entity or a system must be validated.

Cyberattacks are usually done by malicious users, hackers, or even users who violate one of the security attributes unintentionally. These violations can happen due to a vulnerability in the system. A **vulnerability** is a weakness or a type of software fault in a system, that can be explored by internal or external threats in order to lead to a security failure [15]. When a user explores a vulnerability, trying to perform a malicious action, by violating the security properties, it is called an **attack**. As shown in Figure 2.3, an attack can have consequences in a system and lead to an **intrusion**, which is a successful activation of a vulnerability. This activation is accomplished by using an **exploit**, which is a piece of code with a sequence of commands that activates the vulnerability. After the successful intrusion from the attacker, the target system can suffer a state deviation which generates an **error** followed by a **failure** that represents an incorrect behaviour of the system.

When a vulnerability is found, the correct thing to do is to report it to the entity responsible for the software. This makes it easier for the company to verify and correct the vulnerability, by developing a patch. The **patch** keeps the system safe by fixing the vulnerabilities, and it is usually released to costumers via updates.

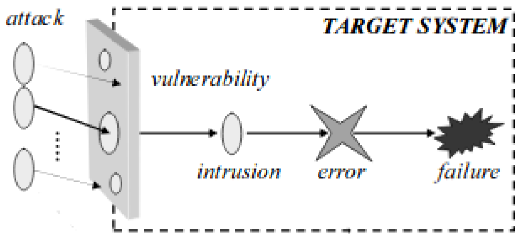


Figure 2.3: Overview an attack and related concepts (from [3]).

2.4.1. Security Bugs Repositories

Due to the number of vulnerabilities reported in the last years, the MITRE Corporation has created the Common Weakness Enumeration (CWE) which is responsible for classifying the class of the vulnerabilities and assigning an identifier to them. The MITRE Corporation is also the owner of Common Vulnerabilities and Exposure (CVE), which is a list of the found vulnerabilities with an identifier, a description, and public references associated to each one. To refer a vulnerability, it is usual to use the ID assigned to it instead of its description, where the first four numbers of the ID refer the year when the vulnerability was documented and the other numbers are the number associated with it. To classify the impact and the severity of the vulnerabilities, which helps organizations prioritize them, they use the Common Vulnerability Scoring System (CVSS), an algorithm provided by the initiative of Forum of Incident Response and Security Teams (FIRST), that uses a collection of metrics such as the time evolution and the impact in the system.

National Vulnerability Database (NVD) is a repository from National Institute of Standards and Technology (NIST) that performs analysis in CVEs and collects all the information related to them, providing the CWE, CVSS, and other important data. This work used the data from cvedetails.com which gets the data from the NVD repository, and some other additional sources such as exploit-db.com which is an exploit database maintained by Offensive Security. This company provides Information Security Certifications and high end penetration testing services [45].

2.4.2. Vulnerability Types

Different vulnerabilities have different consequences in software, which allow attackers to have control, steal data, or change the behavior of it when the vulnerability is exploited. There are no standard classification of vulnerabilities, however some organizations and researchers have some results of exploits documented and listed according to the year.

The CWE is responsible for categorize the class of a vulnerability, which can also be called the vulnerability type. The CWE is a list available with the description of software weakness, in order to help organizations understand the types of security flaws in software products. MITRE with the collaboration of SANS Institute and other software security experts, made a TOP 25 list of the most dangerous software errors of the vulnerabilities in 2011 [46].

(OWASP) is a foundation with the objective of helping developers and organizations with application security. As similar to CWE list, OWASP released a list in 2017 with the TOP 10 application security risks [47].

Although both lists have different classifications for the vulnerabilities, they have some types in common. OWASP has a general classification which is more related to the analysis developed in this work, however the most suitable types in the list are:

- **Injection** - source of data that can be used as an injection vector, which can result in data loss, corruption, disclosure of information, or denial of access. These flaws occur when an attacker send malicious code to an interpreter.
- **Sensitive Data Exposure** - unprotected data may let an attacker steal keys, execute man-in-the-middle attacks, or steal clear text of data, which allows to gain access to information.
- **Broken Access Control** - not properly enforcing the restrictions of authenticated users allows an attacker to gain privileges and bypass access control to the system, and modify sensitive files.
- **Security Misconfiguration** - security misconfiguration or the lack of upgrade allows attackers to gain unauthorized access and privileges to the system.
- **Insecure Deserialization** - deserialization exploitation leads to remote code execution, or attack injection and privilege escalation.
- **Using Components with Known Vulnerabilities** - usage of components with vulnerabilities facilitates the exploitation of those vulnerabilities in the system and enables various attacks and impacts.

2.4.3. Detection of Software Vulnerabilities

Developers gain habits through time with their experience. Sometimes they are good habits, but other times not so much. To improve the quality and security of the developed code, it should be analyzed before being sent to production, in order to reduce the bugs in it. This method requires the presence of expert developers to make a code review. **Static Code Analysis** is the same method but in an automated way using a software tool. Similarly to manual code reviews, these tools are a method to discover defects during the production of software [48].

Static analysis is performed before the program execution, and it is applied in the source code, binary code, or byte code. Nowadays there are Integrated Development Environments (IDEs) that contain static code analyzers to warn the developer of errors such as the use of a variable without it being initialized, or about the syntactic of the code. These tools are very useful to automatically verify the presence of vulnerabilities in the code, to find the source of security flaws instead of their effects. New vulnerabilities and rules can be added to the tool database, and besides vulnerabilities it can also detect bugs.

Static code analysis tools also have some limitations: false negatives and false positives [44]. **False negatives** happen because these tools cannot find all the vulnerabilities. They have a restricted scope for the vulnerability detection because they only know what is stored in their database. Additionally, the analysis is constrained because the thoughtless addition of conditions would make the testing everything take a long time. **False positives** manifest due to code aspects that are difficult to understand if they are, or not, vulnerabilities (i.e. fprintf, strcpy).

Docker is developed in `Go` language, thus the static code analysis tools have to be valid for this language. Like Docker, `Go` language is also an open source project and was initially created by a Google employee [33]. All the static analyzers developed for the `Go` language are also open source and the tools available for this language are meager in diversity. Several tools can be found online, but they are usually combinations and variations of the same static analyzers.

2.5. Docker Security Studies

Docker is a platform with many available tools and components, which offers a rich set of features to manage containers. The fact that it makes using containers much simpler has made it very popular. Being widely used, even to deploy critical applications, has led to some studies which try to understand the level of security it provides.

The study [49] did an overview about the security of some of Docker's components like containers, repositories, and orchestrators, and analyzed the most common security problems. The authors proposed a solution which involved higher levels of abstractions like orchestrators, although it was not tested by them. Mohallel, Bass and Dehghantaha [50] proved through an experiment between a Docker server machine (with LXC as the execution driver) and a bare metal server machine running the same application, that there was an increase in the attack surface exposed in the host of the Docker server. Despite the usage of official repositories, they concluded that the exposure is due to the vulnerabilities exposed by the OS images inside the container. In [51] the authors performed an thorough survey on related work developed in Docker security and analyzed the Docker ecosystem by listing security issues related, and running some experiments. The authors also applied some use cases to Docker usages, which led to the conclusion that many vulnerabilities are a result of casting containers as VMs, which its inappropriate due to the different architectures.

Other security studies targeted specific components of Docker like Docker Hub. The study [52] is about the state of security vulnerabilities in Docker Hub images. The authors created a framework to automatically analyze the images (official and community) from the repository. The study reveals that each image had 70 vulnerabilities on average, in those, one has high severity level. The authors suggest regular security updates in Docker Hub images would reduce the general number of vulnerabilities, since all inherit the father vulnerability.

As Docker shares the kernel with the host, some security studies try to take advantage of kernel vulnerabilities. Privilege escalation or modification of the shared memory can lead to an escape from the containerization, compromising the host [53]. This work shows that it is possible to escape from containerization through Kernel methods, and proposes a defense method to prevent this behavior. Another kernel issue revealed leakage channels that exposed host system information, due to an incomplete implementation of a system resource partition mechanism in the Linux Kernel [54]. The authors proved that this vulnerability could lead to a power attack that could compromise the dependability of power systems in data centers. The authors also proposed and validated a solution to these problems.

2.6. Analysis of Dependability and Security

One of the means to do security evaluations is through empirical studies. Empirical studies are based on data that is observed, and by analyzing this data it is possible to improve the observed systems. The following studies were important to complete the background knowledge required to do the empirical study present in this work. An example is the study realized by Zviran and Haga [55], where it was realized that users do not give the necessary importance to the passwords used to protect their systems. This study was based on data from a group of user's passwords using exploration hypothesis and password characteristics: password length, composition, change frequency and selection method. This study showed that the security of passwords is low because users tend to violate the secure practices resulting in passwords that are easy to guess, because they are easy to remember.

A study about software [56] describes a process to do an Orthogonal Defect Classification (ODC), which classifies the software faults and provides the knowledge to understand their nature of them considering the cause-effect relation. This classification helps to reduce the number of defects in the code, changes made to it, and also provides feedback to the developers.

Another study by Tan and Croft [57] is on the interaction between the Android native language and the Java Development Kit. The authors applied static analysis tools and manual inspection to the code, where they found undiscovered bugs. This led them to conclude that the native code (C/C++) is unsafe, goes against Java's security model, should be kept to a minimum and be ported to safer languages such as Java.

More related with virtualization, Ormandy [58] from Google, does a research on the virtual machine implementation of *x86* systems, in order to assess the security exposure of the host. The security test was made through a tool that exposes security flaws and proved that virtualization is no security panacea.

Milenkoski et al. [59] suggest an improvement to hypercall interfaces by performing an analysis of hypercall's vulnerabilities and characterizing their attack surface through vulnerabilities of the hypercall handler. The authors also trigger vulnerabilities and analyze their effects.

A different study by Elia [60] is an analysis of five years of OpenStack vulnerabilities to identify the frequency and trends of the vulnerabilities found in the platform. This study was based on existing security reports and showed that most of the attacks are exploited by inside attackers, trivial vulnerabilities remain undetected for a long period of time, and the majority of these are easy to detect and correct.

This page is intentionally left blank.

Chapter 3

Research Objectives and Approach

Docker's popularity is leading organizations to integrate and deploy their applications in containerized environments. This raises concerns about Docker's security and any existing paths that can lead to the possibility to activate existing vulnerabilities in the platform.

This chapter describes the main research objectives for this work, as well as the approach developed to reach them. Section 3.1 presents the objectives as well as their interpretations. Section 3.2 proposes and explains the approach developed to reach the research objectives.

3.1. Research Objectives

The **main goal of this work is to better understand the security of the Docker platform, and what could have been done to prevent its vulnerabilities**. This allows to understand the problems of availability, confidentiality, integrity, and isolation in the platform.

For this, it was necessary to analyze and systematize the vulnerabilities in order to understand them. Additionally, static analysis tools were applied to the code, to verify if they could have prevented the existence of vulnerabilities. Finally, to further analyze and understand the vulnerabilities some exploits were studied in detail.

The following concrete objectives were defined:

- **Perform an Analysis and Systematization of the Vulnerabilities**

  To get a better understanding about Docker's security it is necessary to start with an analysis on existing online security reports, as they constitute an important source of information to obtain insights about the history of security problems. This provides the necessary information about previous vulnerabilities and the impact they had in the security of Docker. It also makes it possible to understand which are the most common causes, their effects, and what are the consequences of their exploitation.

- **Study the Applicability of Static Code Analysis in the Docker code**

  Static code analysis is one of the techniques traditionally used for the detection of vulnerabilities. Static code analysis tools are useful in order to obtain information from the code without executing it, however, they are also usually associated with many false positives.

This objective focuses on understanding how useful these tools can be when analyzing Docker's code, by understanding how they work and in which ways they can aid developers.

- **Analysis of Vulnerabilities Exploits**

  Based on the information about vulnerabilities, it is possible to analyze both the exploits and the patched code, in order to demonstrate the vulnerabilities, i.e. vulnerability exploits. The patch contains the code changed to correct a vulnerability, as well as the exploit, allows a study of the impact caused by the vulnerabilities in Docker environment. The objective of this analysis is to discover some techniques that could be used to prevent the vulnerabilities. This allows to understand which are the suitable techniques for some of the types of vulnerabilities analyzed in the previous objectives and how they could improve the code.

3.2. Approach

The diagram illustrated in Figure 3.1 presents an overview of the approach to accomplish the research objectives of this work. As can be observed, the it consists of a set of steps which are grouped in three main activities that map with the goals, as explained next.

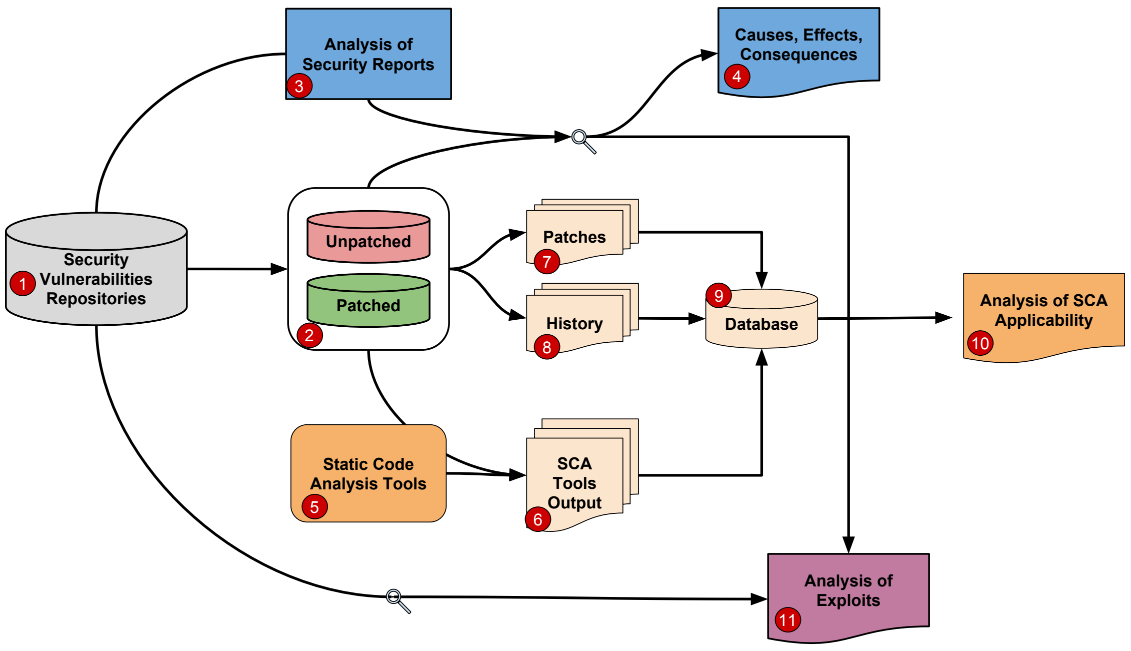

Analysis of Security Reports 3
Causes, Effects, Consequences 4
Security Vulnerabilities Repositories 1
Unpatched
Patched 2
Patches 7
History 8
Database 9
Analysis of SCA Applicability 10
Static Code Analysis Tools 5
SCA Tools Output 6
Analysis of Exploits 11

Figure 3.1: Overview of the designed research approach.

The initial information source for this work were the **security vulnerabilities repositories (1)**, in which the history of vulnerabilities that have affected Docker, including the respective reports and patches, are organized. Each vulnerability has a description about the bug and what effect it had in the system. This information is available mostly in the security reports at `cvedetails.com` and `cve.mitre.com` (presented in Section 2.4.1).

From the security vulnerabilities repositories, it is possible to obtain the **patched and unpatched (2)** versions of the code associated with the vulnerability. To do this, it was necessary to search through the `Git` repository history, searching for the issues associated with the collected vulnerabilities. In these issues, developers usually discuss the solution to the problems, and it is possible to obtain the unpatched and patched versions of the code pertaining to the vulnerabilities.

Vulnerabilities systematization and analysis The security reports were used in the **Analysis of Security Reports (3)**, which consisted in collecting the information of each vulnerability found and performing a thorough manual analysis in order to understand each one: what were the effects of them being exploited and how they were corrected. This resulted in a systematization of the vulnerabilities according to their **cause, effect, and consequence (4)**.

The advantage of determining the cause of the security bugs based on *post mortem* analysis is that it reduces the subjectivity of the classification. Chapter 4 provides the analysis and the obtained results during this activity.

Static code analysis The phase of **static code analysis tools (5)** uses tools to analyze the code of the patched version, as well as the code of the version before the patch (i.e. the last one affected by the vulnerability), in order to understand which kinds of symptoms are present. The used tools are static code analyzers available for the Go language (programming language used by Docker). The objective is to study the output of these analyzers and understand if there are significant differences between the unpatched and patched codes and if any of the vulnerabilities could have been avoided.

For this, the code of these versions was analyzed using the selected tools, and the **Output of the Tools (6)** was collected and uniformized, to be added to a relational database in a subsequent step.

For all of the **Patches (7)**, the lines of code related to the vulnerabilities were identified, as well as the corrected code. To understand the **History (8)** of the vulnerabilities, the lines related to them were tracked across the older versions of the codebase.

After these steps, it becomes necessary to understand the vulnerability with the help of the **security reports (3)**, or with the information from the vulnerability **patch (7)**, which shows the lines that were corrected.

Afterwards, this data was inserted into a **Relational Database (9)**, with the proper relations to make it possible to easily analyze the information. The resulting database was then subject of **Analysis (10)** trying to unveil differences between the patched segments of code and the remaining code. This can be seen in Chapter 5.

Analysis of vulnerabilities exploits After validating the vulnerabilities, an **Analysis of Exploits (11)** was made. To perform this step, it was necessary to find the exploits associated with each vulnerability. To obtain these exploits a thorough analysis of the issues was required, focusing on the details behind the vulnerability exploitation and the patch. The results are presented in Chapter 6.

Chapter 4

Security Analysis of Container Platforms

This Chapter focuses on the vulnerability reports published since Docker's launch. Although Docker's first version is from 2013, the first stable release was on June 2014 [29]. Therefore, this work considers version `1.0` as the minimum version affected by vulnerabilities and considers vulnerabilities until the end of 2017.

The approach starts with collecting as many vulnerabilities from Docker as possible. Several vulnerabilities are listed in the Common Vulnerabilities and Exposure (CVE) database system, which provides a dictionary with vulnerability details and its affected systems. Docker is a relatively recent software, and although its adoption has grown exponentially, there is a reduced number of vulnerabilities listed, as presented in the next section. For each one, cvedetails.com provides Common Vulnerability Scoring System (CVSS) scores, Common Weakness Enumeration (CWE) IDs, and affected products. It also provides detailed information about the vulnerabilities and their impact on a system. An overview of these repositories is presented in Section 2.4.1.

This study consists in a set of steps, which are repeated for each of the vulnerabilities available, as portrayed in Figure 4.1 and detailed next.

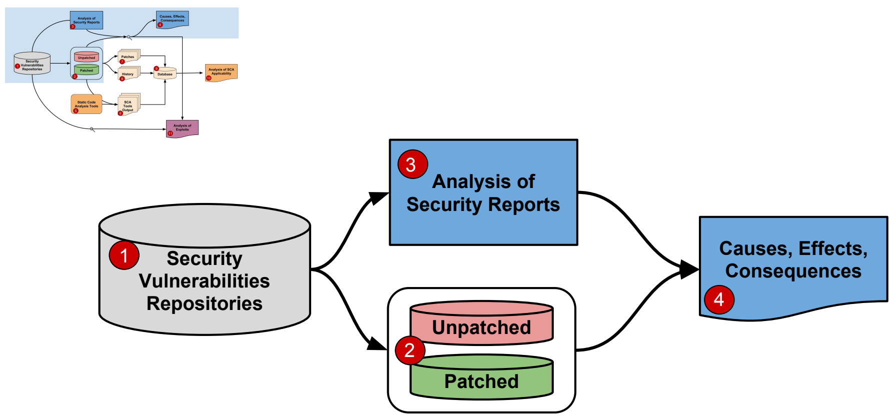

1
Security Vulnerabilities Repositories
3
Analysis of Security Reports
2
Unpatched
Patched
Causes, Effects, Consequences
4

Figure 4.1: Approach followed for the analysis of Docker vulnerabilities.

After collecting the vulnerabilities from the repositories, it was necessary to analyze the data manually in order to solve data inconsistencies and to gain better increased insight about them. This manual analysis allowed us to understand the typical exposure times of the vulnerabilities in Docker, and how these vulnerabilities distribute along the platform's development (see Section 4.2).

Finally, a thorough analysis allowed us to systematize the vulnerabilities considering their main characteristics, mainly in terms of their root *causes*, their *effects* and the *consequences* that these effects lead to.

4.1. Vulnerabilities Overview

To collect the vulnerabilities existent was used http://cve.mitre.org and by searching on its database it is possible to find the *40* CVE entries that match with "Docker" as a keyword. Table 4.1 provides an overview of these vulnerability reports. Although, not all the information of these CVE is available in the repositories. This occurs with 3 of the vulnerabilities, that were marked as "RESERVED". That means they were reserved for use by a security research or CVE Numbering Authority Organization, responsible for the distribution of CVE IDs to researchers and information technology vendors [61]. These vulnerabilities were considered because the additional information was available in issues from the Docker's repository.

As can be observed, the listed vulnerabilities are quite diverse, both in terms of type, impact, and even the venue that is used for exploitation. Therefore, it was necessary to have a thorough view. A preliminary overview allowed us to further focus the scope of our analysis. As Figure 4.2 shows, of the 40 vulnerabilities, 11 were problems of software components that used Docker, and it was found that these vulnerabilities did not affect Docker's security, and therefore were left outside of the scope of the analysis. The remaining ones were considered in our in-depth analysis, this is 29 were really an issue in Docker.

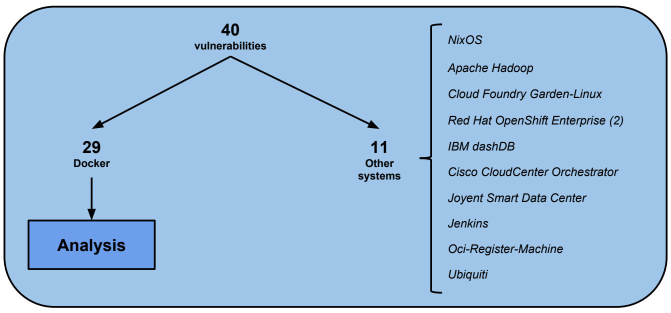


Figure 4.2: Docker vulnerabilities preliminary distribution.

Table 4.1: Description for each vulnerability

CVE / CVSS		Software	Description
2014-0047	4.6	Docker	Due to a number of unsafe usages of the "/tmp" folder, an attacker with local access can overwrite arbitrary files or perform symbolic link attacks.
2014-3499	7.2	Docker	A user can escalate privileges because the socket used to manage Docker is world-readable and world-writable.
2014-5277	5.0	Docker Registry	When users attempt to contact the registry, if the connection fails it drops from HTTPS to HTTP. This allows man-in-the-middle attacks.
2014-5279	10	boot2docker	The Docker daemon managed by boot2docker, improperly enables unauthenticated TCP connections by default, which allows a remote attacker gain privileges or execute arbitrary code from children containers.
2014-5280	9.3	boot2docker	Docker daemons enabling TCP connections without TLS authentication, which allows attackers to conduct the user execute unauthorized commands.
2014-5282	5.5	Docker	The improper validation of image IDs allows remote attackers to redirect to another image through the loading of untrusted images via 'docker load'.
2014-6407	7.5	Docker	Malicious images with symlink and hardlink traversal can lead to users extracting files to arbitrary locations, and leverage remote execution of code and privilege escalation.
2014-6408	5.0	Docker	Security options can be applied to images, allowing them to change the default run profile of the container's images.
2014-8178	ND	Docker	The non-global unique identifier from Docker images layer is vulnerable during docker pull and push. This might poison host's image cache and allows maliciously crafted images to poison subsequently pulled images.
2014-8179	ND	Docker	During Docker pulls validation, is possible to inject new attributes for the *json* file from the image. This allows the corruption of the verified content at *json* deserialization, leading to pulling unverified layers.
2014-9356	5.4	Docker	When using absolute symlinks, through archive extraction or volume mounts, it is possible to write files on the host system and/or escape containerization leading to escalated privileges.
2014-9357	10	Docker	Malicious images or builds can escalate privileges and execute arbitrary code by making use of an LZMA (.xz) archive.
2014-9358	6.4	Docker	The lack of image ID validation allows users to get a malicious image from docker load or registry communications. This makes it possible for attackers to make path traversal and repository spoofing attacks.
2015-1843	4.3	Docker	The fix to CVE-2014-5277 was incomplete, therefore, this CVE is a return to of that vulnerability.
2015-3627	7.2	Docker	Before performing chroot, the file-descriptor is passed to the pid-1 process of container. This makes the file-descriptor vulnerable to insecure openings and symlink traversals.
2015-3629	7.2	Docker	In a container respawn it is possible to mount a namespace breakout, allowing escaping from containerization and privilege escalation.
2015-3630	7.2	Docker	The files "/proc/asoud", "/proc/timer_stats", "/proc/latency_stats", and "/proc/fs", should not have write permissions. This allows users to bypass security restrictions.
2015-3631	3.6	Docker	A user can create a volume in the "/proc" and "/" folders, which allows users to override files and specify arbitrary policies for Linux Security Modules.
2015-9258	5	Docker Notary	A vulnerability in a security algorithm not matched to a key allows an attacker to control the field from the signature algorithm. This might lead forge a signature by forcing misinterpretation.
2015-9259	7.5	Docker Notary	The function *checkRoot* does not check *json* files, which can lead an attacker to produce update files referring to an old *root.json* file, even if the user creates a new one.
2016-0761	10	Cloud Foundry Garden-Linux	Cloud Foundry Garden-Linux contains a flaw in managing container files during Docker image preparation, which allows changes to host files and directories.
2016-3697	2.1	Docker	Docker does not treat numeric UIDs properly, allowing attackers to gain privileges.
2016-3708	5.5	Red Hat OpenShift Enterprise	Red Hat Openshift Enterprise 3.2 uses Docker. If multi-tenant SDNs are active and a build is executed on a namespace, a container can gain access to network resources from other namespace's containers.
2016-3738	6.5	Red Hat OpenShift Enterprise	Red Hat Openshift Enterprise does not restrict source to image builds as it should, allowing unauthenticated users to access Docker's socket and to gain privileges.
2016-6349	2.1	oci-register-machine	Oci-register-machine allows the use of Docker containers.Through the usage of the machinectl command, it is possible to list all containers from all users, unlike in Docker which lists only each user's containers. This makes it possible for unintended users to obtain sensitive information.
2016-6595	4.0	Docker Swarmkit	An user can exhaust resources causing a Denial of Service (DoS) by repeatedly joining and exiting a swarm cluster. This vulnerability is disputed because some users argue that resources are allocated to that container and others say that it is necessary to free allocated resources.
2016-8867	5.0	Docker	Due to a misconfiguration in the ambient capabilities of runc, an attacker can bypass and gain privileged access.
2016-8954	7.5	IBM dashDB	IBM DashDB local has hard-coded credentials, which can lead to attackers gaining access to Docker's containers and databases.
2016-9223	10	Cisco CCO	Cisco Cloud Center Orchestrator uses Docker and because of a misconfiguration (deployments using port 2375), it can lead to remote attackers gaining privileges.
2016-9962	4.4	Docker	Coming from runc this vulnerability allows, through a namespace ptrace, after container execution, a user to gain access to file-descriptors and escape from containerization. This can happen when a user has root privileges inside the container.
2017-0913	ND	Ubiquiti	Ubiquiti UCRM has local file system isolated in a Docker container, however this vulnerability from version 2.3.0 to 2.7.7 allows that an authenticated user to read arbitrary files in it. A successful exploitation requires the credentials to an account with "Edit" access to "System Customization".
2017-6074	7.2	Linux Kernel	A function in the linux kernel mishandles a packet data structure in the Listen state, which allows local users to obtain root privileges or cause a denial of service (double free) via an application.
2017-6507	4.3	Docker	A vulnerability in AppArmor discards profiles that are not present inside "/etc/apparmor.d". This affects LXD and Docker, making containers unconfined, leading to bypasses and unauthorized actions.
2017-7412	7.2	NixOS	NixOS does not protect Docker socket, meaning it is world-writable, allowing users to gain privileges with Docker commands.
2017-7669	8.5	Apache Hadoop	Apache Hadoop uses Docker in Linux Container Executor, which runs Docker commands with root privileges, lacking input validation. A remote attacker can execute arbitrary code with root privileges.
2017-11468	5.0	Docker Registry	Docker Registry does not limit the amount of data that can be accepted by a user. This may lead to a denial of service.
2017-10940	9.0	Joyent Smart Data Center	Joyent's Triton Cloud uses the Docker API, and its process does not validate uploaded files from users. This allows remote attackers to gain privilege escalation and to execute arbitrary code with root privileges.
2017-14992	4.3	tar-split	Docker's lack of content verification allows a remote attacker cause a DoS via crafted image layer payload, which means the creation of a gzip bombing.
2017-16539	4.3	Docker	Docker does not mask the path "/proc/scsi", which allows an attacker to write to the "/proc/scsi/scsi" file and use it to remove devices.
2017-1000094	4.0	Jenkins	Jenkins has a Docker common plugin which provides an ID list that lets users configure the jobs they want to authenticate in Docker registry without verifying the user's credentials. This allows users without permission to get a list of valid IDs and steal valid credentials from other users.

4.2. Exposure Time Analysis

To understand the exposure time of the vulnerabilities, our analysis started with the data available in `CVE Mitre` and in `CVEDetails`. However, this data has some inconsistency such as, missing vulnerability information (e.g. first system version).

After a manual check of the descriptions, it was found that many specified the affected version, but most of them only mentioned that the vulnerability was present before a given version. In these cases, and in cases where the version was earlier than 1.0, it was assumed that 1.0 was the first affected version, as there usually was not sufficient information to know with certainty.

Another shortcoming was the vulnerabilities in modules that affect Docker such as, RunC, Docker Registry, or Notary. The versions of Docker that have been affected by those are not clear, making it necessary to get the patch commit added to the Docker source code, and then manually verifying the date of the integration. Table 4.2 shows the versions and dates obtained.

Table 4.2: Vulnerabilities First Affected Version (FAV) and Patched version dates.

CVE	**First Affected Version**		**Patched Version**	
2014-0047	1.0.0	09-06-2014	1.5.0	22-01-2015
2014-3499	1.0.0	09-06-2014	1.2.0	20-08-2014
2014-5277	1.0.0	09-06-2014	1.3.1	28-10-2014
2014-5279	1.0.0	09-06-2014	1.3.0	14-10-2014
2014-5280	1.0.0	09-06-2014	1.3.0	14-10-2014
2014-5282	1.0.0	09-06-2014	1.3.0	14-10-2014
2014-6407	1.0.0	09-06-2014	1.3.2	20-11-2014
2014-6408	1.3.0	14-10-2014	1.3.2	20-11-2014
2014-8178	1.0.0	09-06-2014	1.8.3	12-10-2015
2014-8179	1.0.0	09-06-2014	1.8.3	12-10-2015
2014-9356	1.3.0	14-10-2014	1.3.3	11-12-2014
2014-9357	1.3.2	20-11-2014	1.3.3	11-12-2014
2014-9358	1.0.0	09-06-2014	1.3.3	11-12-2014
2015-1843	1.0.0	09-06-2014	1.5.0	10-02-2015
2015-3627	1.0.0	09-06-2014	1.6.1	07-05-2015
2015-3629	1.0.0	09-06-2014	1.6.1	07-05-2015
2015-3630	1.0.0	09-06-2014	1.6.1	07-05-2015
2015-3631	1.0.0	09-06-2014	1.6.1	07-05-2015
2015-9258	1.0.0	09-06-2014	1.8.0	11-08-2015
2015-9259	1.0.0	09-06-2014	1.8.0	11-08-2015
2016-3697	1.1.2	23-07-2014	1.11.2	31-05-2016
2016-6595	1.12.0	28-07-2016	1.12.1	18-08-2016
2016-8867	1.12.2	11-10-2016	1.12.3	26-10-2016
2016-9962	1.0.0	09-06-2014	1.12.6	10-01-2017
2017-6074	1.0.0	09-06-2014	17.03.0	01-03-2017
2017-6507	1.0.0	09-06-2014	1.13.0	18-01-2017
2017-11468	1.6.0	07-04-2015	17.03.2	29-05-2017
2017-14992	1.0.0	09-06-2014	17.09.1	07-12-2017
2017-16539	1.0.0	09-06-2014	17.11.0	20-11-2017

Based on the dates from the FAV release and the patch version release, an analysis was made to understand the exposure time of a vulnerability. Figure 4.3 presents a summary of the vulnerabilities grouped according to the time they were exposed. As it is possible to observe, in 4 of the 29 cases the period of exposure to attacks was one month or less. However, it is also possible to observe that more than half of the vulnerabilities were exposed to exploitation at least 11 months, with a maximum time of exposure of 43 months in two cases. In fact, the average exposure time in these vulnerabilities is about 13 months.

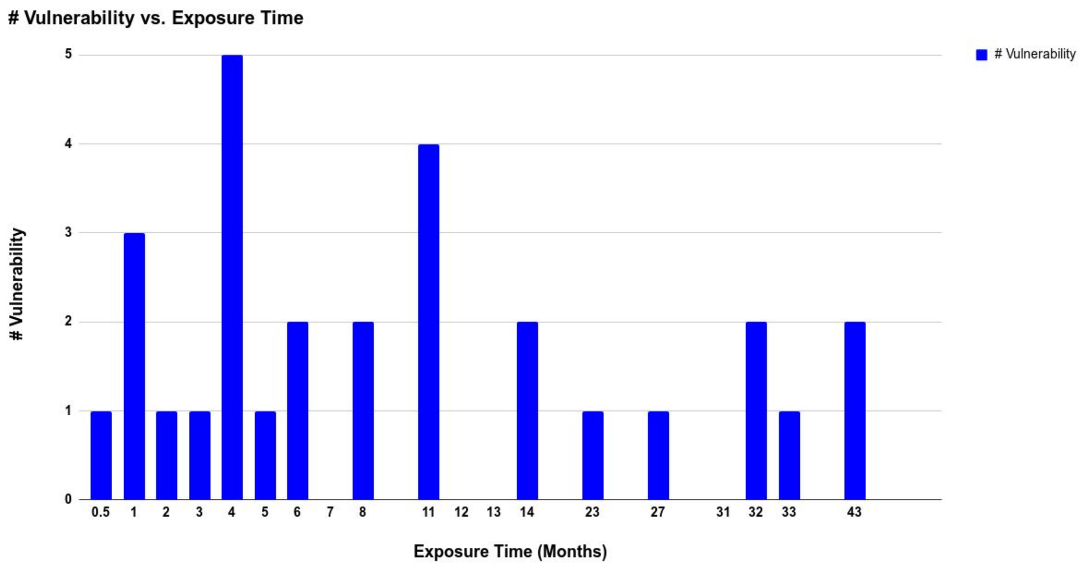


Figure 4.3: Number of vulnerabilities by exposure time in months.

A different analysis is presented in Figure 4.4. In this case, the exposure time of vulnerabilities is represented along the timeline of Docker's existence, since the FAV until the patch release. This representation allows us to observe which were the periods of the system's life that had more security issues, and how these issues are distributed overtime.

As we can observe, most of the vulnerabilities are concentrated in the first year of the lifecycle of the project. Still, in 2015 and most of 2016 the software contained at least 7 vulnerabilities at the same time. As this number reduces overtime it suggests that overall the quality of the software in terms of security is improving as Docker is reaching a higher maturity level.

It is important to note, however, that although 2017 already ended, it is possible that new vulnerabilities can appear which affect earlier versions. In fact, this is highlighted by one of the vulnerabilities that was discovered in the end of 2017 and patched at the time, and affected all the previous versions of the software (according to the report).

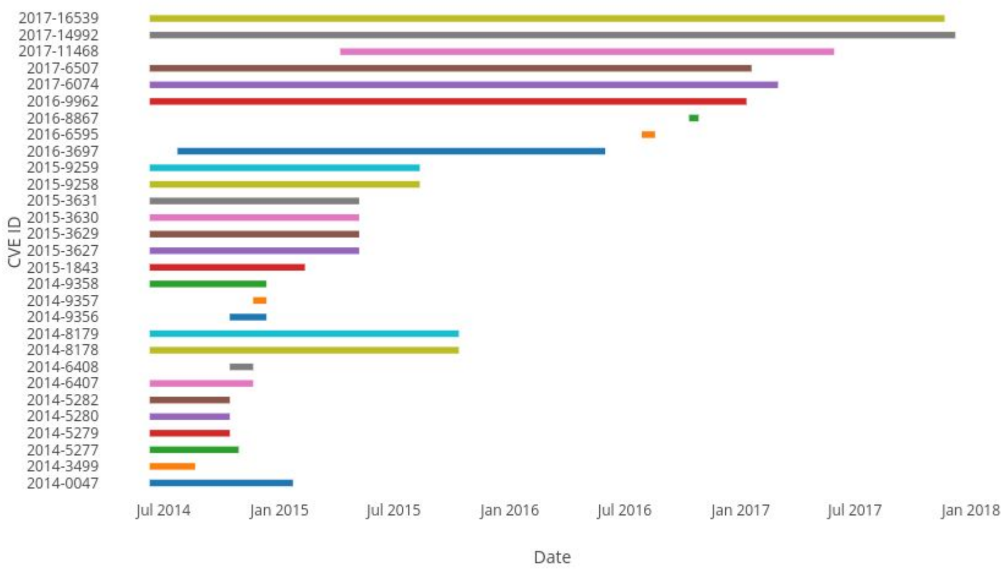


Figure 4.4: Vulnerabilities by date

4.3. Vulnerabilities Characterization

The next step was to understand the vulnerabilities, systematizing them according to their causes, effects and consequences.

- **Cause** – determined by the changes in the code to correct it, and the reason for the vulnerability's existence, which includes: incorrect permission management, unprotected resources, improper security validation, umoderated resources, incorrect configurations, and incorrect recovery mechanisms.
- **Effect** – impact in the system which leads to the consequence of the attack, including Write Arbitrary files, Resource Exhaustion, Exposed System, and Security Restriction Violation.
- **Consequence** – the possible result of an attack, which includes DoS, Gain privileges, Execute code, Bypass, and Gain information.

The objective is to group vulnerabilities by similar characteristics, trying to understand the most frequent mistakes, venues of attack, and impact of vulnerabilities.

Using these three factors, it is possible to obtain a more orthogonal and comprehensive classification than in the cases of using a single classification, as is the case of CVSS.

The consequence of an attack do not have a standard classification. CVEdetails.com has been doing a collection of types of vulnerabilities consequences, by frequency, since 1999. Even though the page lists various types of vulnerabilities, but only the most relevant are used. These consequences are (higher to the lower impact):

- **Gain Privileges** - an attacker gains unauthorized privileges, which can give access to restricted parts of the system.
- **Execute Code** - an attacker can trigger malicious code execution.
- **DoS** - this attack can lead to an exhaustion of the system's resources, reducing its availability.
- **Bypass** - the attacker can bypass the security restrictions and gain unauthorized access to the system.
- **Gain Information** - an attacker gains unauthorized access to private information, which can be used to exploit other vulnerabilities.

As the consequence of the vulnerability is often clearly stated in its description, a `Top-Down` approach was used to find its effect and cause.

All the vulnerabilities considered were analyzed and characterized individually. When some vulnerabilities had more than one potential classification for consequence, the option was to consider the consequence with **higher impact in the system**. Table 4.3 shows the results the classification performed based on the security reports.

Table 4.3: Docker vulnerabilities characterization

CVE	Cause	Effect	Consequence
2014-0047	Unprotected Resources	Exposed System	Gain Information
2014-3499	Incorrect Permission Management	Exposed System	Gain Privileges
2014-5277	Incorrect Recovery Mechanism	Restriction Violation	Gain Information
2014-5279	Unprotected Resources	Exposed System	Gain Privileges
2014-5280	Incorrect Permission Management	Restriction Violation	Execute code
2014-5282	Improper Validation	Exposed System	Bypass
2014-6407	Incorrect Permission Management	Restriction Violation	Execute Code
2014-6408	Incorrect Permission Management	Restriction Violation	Bypass
2014-8178	Incorrect Configuration	Exposed System	Execute code
2014-8179	Improper Validation	Restriction Violation	Execute code
2014-9356	Unprotected Resources	Write Arbitrary Files	Gain Privileges
2014-9357	Incorrect Permission Management	Restriction Violation	Execute Code
2014-9358	Improper Validation	Exposed System	Bypass
2015-1843	Incorrect Recovery Mechanism	Restriction Violation	Gain Information
2015-3627	Unprotected Resources	Write Arbitrary Files	Gain Privileges
2015-3629	Unprotected Resources	Write Arbitrary Files	Bypass
2015-3630	Incorrect Permission Management	Exposed System	Bypass
2015-3631	Unprotected Resources	Write Arbitrary Files	Bypass
2015-9258	Improper Validation	Exposed System	Bypass
2015-9259	Incorrect Permission Management	Restriction Violation	DoS
2016-3697	Improper Validation	Restriction Violation	Gain Privileges
2016-6595	Unmoderated Resources	Resource Exhaustion	DoS
2016-8867	Incorrect Configuration	Restriction Violation	Gain Privileges
2016-9962	Unprotected Resources	Exposed System	Gain Information
2017-6074	Unmoderated Resources	Resource Exhaustion	Gain Privileges
2017-6507	Unprotected Resources	Exposed System	Bypass
2017-11468	Unmoderated Resources	Resource Exhaustion	DoS
2017-14992	Unprotected Resources	Resource Exhaustion	DoS
2017-16539	Unprotected Resources	Exposed System	DoS

Based on this analysis, it is possible to create a branching diagram that groups the most frequent classifications and represents the relation between them. This representation is depicted in Figure 4.5. The figure represents the number of vulnerabilities that match each classification. For example, from the `5` vulnerabilities that were classified as a DoS consequence:

- `3` were due the effect of resource exhaustion;
- `1` due to the violation of the security restrictions;
- `1` because of the exposed system.

In particular, the effect of resource exhaustion is because of the unmoderated and unprotected resources in the code.

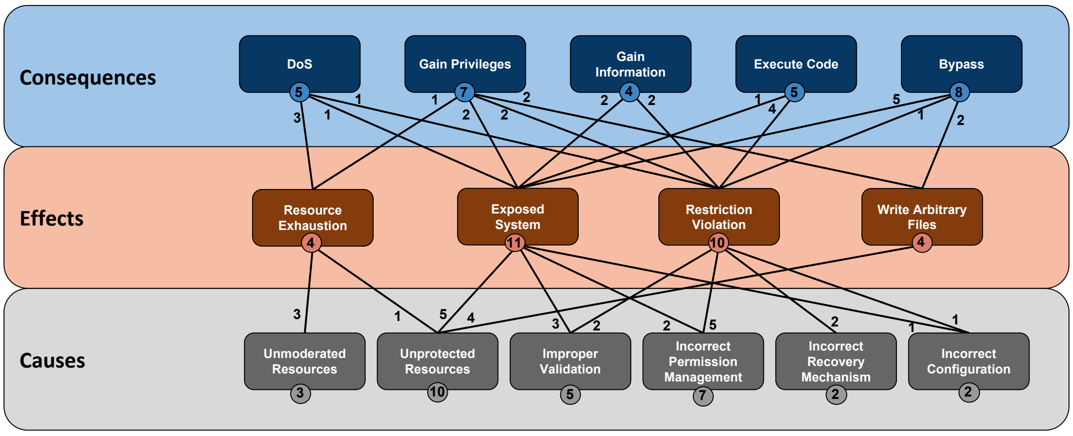


Figure 4.5: Docker vulnerabilities classification with the total vulnerabilities for each classification in each level and the relation between.

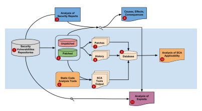
As can be observed, the most common causes for security issues are the unprotected resources and the incorrect management of permissions. The most common effects are an exposed system (which means that a part of the system is left unprotected) and restriction violation (which means that there is a restriction in place, but the vulnerability makes it possible to go around it). Finally, the most common consequences are bypass (8), gain privileges (7) and execute code (5), which are sensitive problems in cloud environments, where multi-tenancy is a common practice.

It is possible to observe a higher prevalence of **bypass** and **gain privileges** attacks, which could be explained by the fact that Docker is a system running at a lower level, and that, when exploited can easily be used to breach and control other components of the system.

Chapter 5

Static Code Analysis Applicability

The objective of this analysis is to verify if developers could have avoided some of the reported vulnerabilities if they had used Static Code Analyzers (SCA) tools. Figure 5.1 presents an overview of the approach, which is described in detail below.



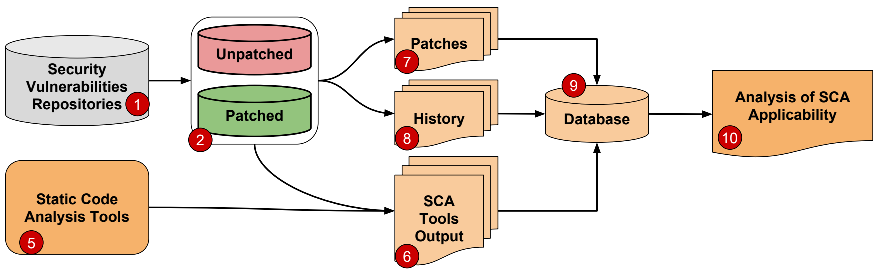

Security Vulnerabilities Repositories
1
Unpatched
Patched
2
Patches
7
History
8
9
Database
Analysis of SCA Applicability
10
Static Code Analysis Tools
5
SCA Tools Output
6

Figure 5.1: Overview of the approach to collect the data

5.1. Selection of Static Code Analysis Tools

Static code analysis can help developers to minimize the time and the effort of code review. It can find bugs and security vulnerabilities in the code, and identify patterns which are prone to leading to errors, without having to run the code. Its usage helps reduce the number of the vulnerabilities in the code before moving to the testing phase.

All the available tools for Go language are open source and these were analyzed. The selected tools, were those that worked properly when applied on Docker code, and those which gave output without errors from the tool being deprecated. However the excluded tools have similar features of others selected for this analysis. Therefore, of all tools **Go Meta Linter** [62] and **Go Reporter** [63] were selected. These two were chosen due to being the most mature, and having the most important features needed for the analysis. They combine the static analyzers described in Table 5.1.

Table 5.1: Static Code Analyzers

Analyzer	Description
deadcode	identifies dead (unused) code.
errcheck	verifies if all error return values are used.
gas	inspects the source code using the AST, looking for security problems.
goconst	searches for and counts the number of repeated strings in the code that could be replaced by a variable.
gocyclo	calculates the cyclomatic complexity of functions.
golint	identifies coding style mistakes in the source code.
gosimple	detects complex code that could be simpler.
gotype	performs an analysis similar to the compiler, finding invalid operations, operands, and undeclared operations.
gotypex	performs the same analysis as *gotype* but only for external test files in a directory.
ineffassign	detects ineffectual assignments to the existing variables in the code.
interfacer	suggests narrower interfaces that can be used.
maligned	detects structs that would take less memory if they were sorted, undeclared variables, and invalid operations.
megacheck	runs *staticcheck* , *gosimple*, and *unused.*
spellcheck	detects misspelled English words.
staticcheck	detects bugs and inefficiencies in the code.
structcheck	finds unused struct fields, however it is not capable of handling embedded structs yet.
unconvert	identifies unnecessary type conversions such as re-declarations and unused imports.
unused	detects unused constants, vars, functions and types.
varcheck	finds unused variables or constants.
vet	identifies suspicious code and potential errors.
vetshadow	identifies variables that may have been unintentionally declared more than one time, one in a certain scope, and declare the same variable out of scope.

5.2. Identification of Patched Segments and their History

To analyze the differences between the vulnerable code and the code without known vulnerabilities, it was necessary to analyze the vulnerability patches. Every patch represents the difference between the version immediately before the patch, hereinafter called **vulnerable version ($\mathbf{V_V}$)**, and the first corrected version, hereinafter referred to as **patched version ($\mathbf{V_P}$)**.

With this, it was possible to determine the segments of code that have been corrected, and the ones that have resulted from the corrections. Every patch has 3 types of modifications: deleted (`d`), changed (`c`), and added (`a`). Figure 5.2 depicts the relations between the different segments of code.

As can be observed, the `d` segments only exist in the V_V and the `a` segments only make sense in the V_P. The `c` segments exist in both versions but with different contents, and we differentiate them using the names `c_old` and `c_new`.

As some of the considered versions (V_i) precede the V_V (and therefore precede the patch), they also contain the vulnerable code, this means those that do not precede the first affected version (as shown in Table 4.2). These code segments are called **history of the vulnerable code**, and are represented in the figure using `d_history` and `c_history`. The `a` segments do not have a history, as they first appear in the patches.

The segments of the code **not affected by the known vulnerabilities** are represented in white in Figure 5.2.

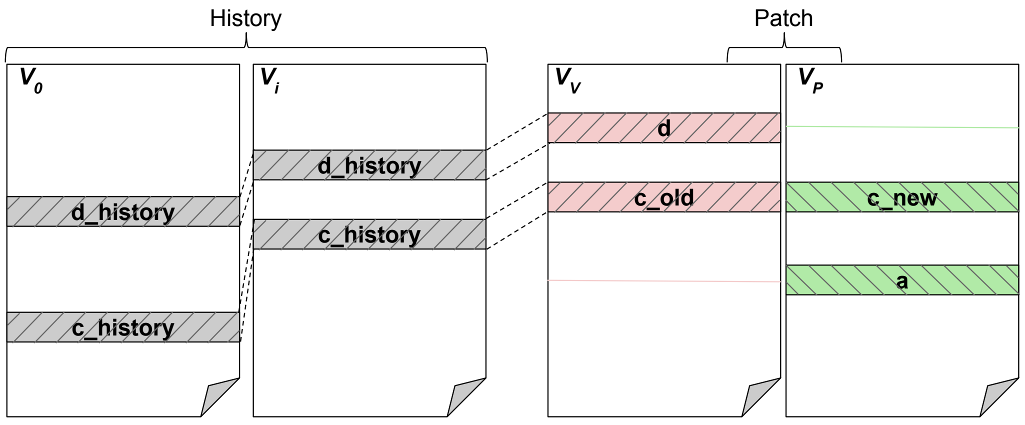


Figure 5.2: Relation between the different segments of code considered.

To obtain this information it was necessary to perform a manual analysis in which these segments were identified, according to code version, file and lines of code. A detailed search in Docker *GitHub* issues was necessary to identify the `pull request (PR)` released to correct each vulnerability, which in practice corresponds to the patch for the vulnerability. In most cases the `PR` mentions the CVE, and in a few cases its description matches the one of the vulnerability.

Using this `PR` as the patch, It was possible to determine the versions $\mathbf{V_V}$ and the $\mathbf{V_P}$. Thus, 2 snapshots were obtained of the codebase for each vulnerability analyzed: one from the respective $\mathrm{V_V}$ and the other from the $\mathrm{V_P}$. Using the `diff` tool, it is easy to extract the segments `d`, `c_old`, `c_new`, and `a` defined by files and respective lines of code. This information was stored in the aforementioned database under a table named `patches`, together with a reference to the CVE and commit.

Finally, it was necessary to perform a tracing of the segments `d` and `c_old` to determine their history. This information was stored in the same database under a table named `history`, with a reference to the respective patch.

5.3. Overall Results for Static Code Analysis

Using the code versions obtained in Section 5.2, it was possible to apply the SCA tools selected in Section 5.1, and add the obtained results to the database, identifying each alert by type, tool, line, file and version.

With this, it was possible to perform an analysis for each threat reported by the SCA tools, and understand if the problem was present from the beginning or if it was something introduced recently, which also allows one to know if the usage of SCA tools could have detected the vulnerability in earlier stages.

Figure 5.3 presents the number of alerts of each SCA tool. As we can observe, the number of alerts reported by *golint* is much higher than any other tool. This tool is focused on coding style mistakes, and the high number of alerts is due to the fact that Docker's source code is developed by an open source community, which makes it very hard to follow a single code convention.

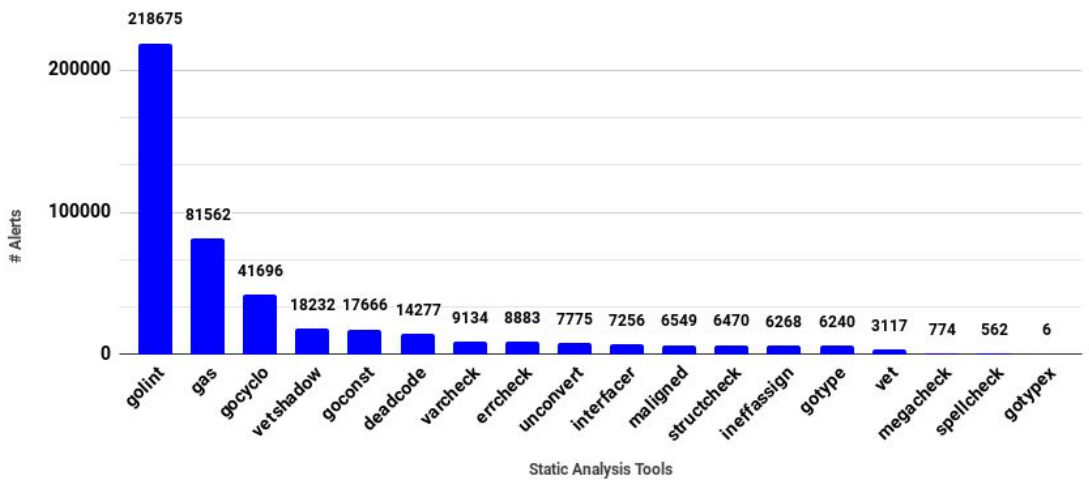


Figure 5.3: Total of alerts by static code analysis tool.

The second value that stands out is from *gas*, which reports a large number of alerts due to it being a security tool that identifies several errors. However, it is known to report a considerable number of false positives [64].

The third highest value in the chart is relative to *gocyclo*, which throws alerts when the cyclomatic complexity is high than *10* in any function. With this value it is possible to understand that there are many functions in the code with high cyclomatic complexity. This means there are many pathways through the code, which in turn turns it more complex to understand and more difficult for tests.

5.3.1. Comparing Patched and Vulnerable Segments

In this section, it was performed an analysis to study the differences between patched and vulnerable code by using the data obtained in Section 5.2. The summary of the results is presented in Table 5.2, including the number of segments, alerts, and Lines of Code (LoC) for each of the considered code groups.

Table 5.2: Number of alerts and size of segments.

	Segments	**Alerts**	**LoC**	**Alerts/LoC**
Patch	703	87	5516	0.016
Vulnerable	565	74	2443	0.030
Vulnerable History	1818	316	13624	0.023
Not affected	-	454674	3275409	0.139

As can be observed, the parts of the code that were not affected by security vulnerabilities present a much larger number of alerts both in absolute terms (Alerts) as in relative terms (Alerts/LoC). The former is expected, as the size of the respective codebase is much larger, the discrepancy in relative terms was not expected.

This may be an indication that the different parts of the code have very different purposes and characteristics, which influences the type of alerts reported. It also shows that the alerts reported by the tools are not very useful and that the tools are not focused on security related concerns. Furthermore, the vulnerable parts are a rather small sample size, which may cause a large variance in the results.

When comparing the number of Alerts/LoC of the patched code with the one of vulnerable code, it is also possible to see that the number is much lower. In the context of the previous observation, it is not possible to draw conclusions related to security. However, in this case, both sets of code have approximately the same functional purposes. Thus, the comparison is more useful than the previous:

1) it is possible to notice that the size of the patched code is 2 times larger than the unpatched code. This shows that the predominant activity when patching is to insert new code.

2) the ratio of Alerts/LoC is lower in the patched code. Considering the type of alerts reported, this is an indicator that the code is, on average, more carefully written than the original.

3) the segments of the vulnerable history are bigger than the ones for vulnerable code because in practice it represents copies of the vulnerable segments through more than one version. The absolute numbers are greater because each vulnerable segment can have `0+` depending on the number of versions affected. However, the relative numbers (Alerts/LoC) are moderated lower, which can be justified for some of the changes of the code in the previous version. As an example, a segment of code can became greater from version to version until the precede version of the patch.

It is important to clarify that the goal of separating the vulnerability history is not to learn from it. The main goal was to separate the data, so as avoid tainting the code that is considered as "not affected" by the vulnerabilities.

A more specific analysis was performed to understand the importance between the modification in the segments in Table 5.3. The correspondent alerts to each modification segment allows a knowledge about the alerts that appear, and the alerts that remained between the versions.

Table 5.3: Number of alerts and size of segments in code changes.

	Alerts	**Matched Alerts**	**LoC**	**Alerts/LoC**
changed new (c_new)	22	14	466	0.047
added (a)	65	-	5050	0.013
c_old	19	14	465	0.041
d	55	-	1978	0.028

Taking a closer look, `c_new` and `c_old` are the segments that have more Alerts/LoC, where in those alerts, 14 are common between them. In these alerts, there are three types: the most repeated problem is about cyclomatic complexity of the function, related to the tool *gocyclo*; style problems about missing comments in go methods by tool *golint*; redeclaration of the variable used to store the error messages by tool *vetshadow*.

Added and deleted segments, have an inferior number of Alerts/LoC than the changed ones, however, the deleted segments have more than added files. This is an evidence that developers have more attention to the code when they correct it.

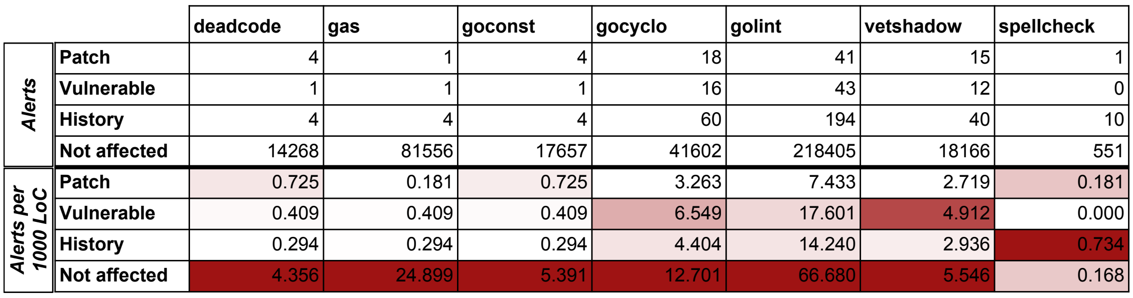
5.3.2. Analysis per Static Analyzers

The code changes between the patched and vulnerable version are important to understand the code's evolution. These changes usually improve the quality of the code, however some features can have less attention paid to them by the developers, which makes the code worse.

This section analyzes the alerts of the SCA tools reported in the *patch*, *vulnerable*, and *history* code. To focus on the more frequent alerts, only the tools that reported more than 5 alerts in all three segments are being analyzed.

As shown before, the number of alerts in the code is not a reliable measure, so in this analysis the focus is in the number of alerts per 1000 LoC. The table represented in Figure 5.4 shows the alerts in the each segment by tool, and the respective number of alerts per 1000 LoC.

In certain tools the *vulnerable* code is the second highest value, however this is not reliable due to the small sample size for this code, where a single alert can highly influence the values.



		deadcode	gas	goconst	gocyclo	golint	vetshadow	spellcheck
Alerts	Patch	4	1	4	18	41	15	1
	Vulnerable	1	1	1	16	43	12	0
	History	4	4	4	60	194	40	10
	Not affected	14268	81556	17657	41602	218405	18166	551
Alerts per 1000 LoC	Patch	0.725	0.181	0.725	3.263	7.433	2.719	0.181
	Vulnerable	0.409	0.409	0.409	6.549	17.601	4.912	0.000
	History	0.294	0.294	0.294	4.404	14.240	2.936	0.734
	Not affected	4.356	24.899	5.391	12.701	66.680	5.546	0.168

Figure 5.4: Alerts per tool, and Alerts per 1000 LoC

As can be observed, two of the tools have different distributions from the others between the *not affected* and the rest of the code, which are *vetshadow* and *spellcheck*.

Vetshadow stands out for having a similar number of alerts per 1000 LoC in the *vulnerable* and the *not affected* code. This indicates that the *vulnerable* code almost has the same number of problems as the *not affected* code, when related to shadow variables declared.

Spellcheck stands out for the number of alerts per 1000 LoC being higher in the *history* code. This peculiar case is due to a vulnerability that has 10 history versions. This was due to the comments related to the corrected function, which were modified before the V_V correcting the misspelled word.

Not affected is the code with the most alerts and alerts per 1000 LoC, in the versions analyzed with SCA tools. The tools with the most distributed values in the four types of code are *gocyclo* and *golint*. In both, the larger number belongs to the *not affected* code, and the smaller value to the *patch* code, which was expected.

In *gocyclo*, the *vulnerable* code is almost two times smaller than the *not affected* code, which indicates that code which was not patched has more functions with high cyclomatic complexity. The *patch* code has much lower cyclomatic complexity, which suggests that it was more carefully written.

Golint is the tool with the largest values in the alerts per 1000 LoC across all code. Comparing the *not affected* and *vulnerable* codes, the number is five times smaller in the vulnerable one. The data suggests that the code has many style mistakes, but also that it has improved with the patches.

The *gas* tool has the second largest values in *not vulnerable* code. This tool's alerts are sixty times higher in this code than in the *vulnerable* code.

In this analysis, there are some unexpected values, since the *vulnerable* code should have more alerts than the other code analyzed. Nonetheless this analysis is useful to understand that these tools are useful, specially for style corrections and cyclomatic complexity warnings.

5.3.3. Analysis per Types

The analyzed static tools have different purposes, however, most of them give similar corrections with different words. To have a clear perspective of their output, an analysis of the keywords from the alerts was performed. After looking at a general overview of the reported alerts, it was possible to extract some patterns and group them in **types**. This division was made into six types, which are:

Presentation - suggestion of variable names, comments before methods, and method names.

Bad usage of variables - variables with wrong types, bad conversion, syntax, and shadow declarations.

Unused code - variables not found in the packages, unhandled, dead code, and unreachable code.

Optimization suggestions - functions with high cyclomatic complexity, redundant code, and structs using more memory than necessary.

Function calls - missing arguments in function calls, bad return values, wrong number of arguments, and formatting directive (e.g. usage of *Println* instead of a *Printf*) functions.

Spell errors - misspelled words and verbs.

These types were defined due to being the most common in the *patch*, *vulnerable*, *history*, and *not affected code*. Figure 5.5 shows the alerts in each type.

Similar to the analysis performed in Section 5.3.2, two of the types have only one alert in the *patch* code, which is a small number to make a conclusion. Thus, the following comparison is more useful than the tool analysis:

		Presentation	Bad usage of variables	Unused code	Optimization suggestions	Function calls	Spell errors
Alerts	**Patch**	44	18	5	18	1	1
	Vulnerable	40	12	6	16	0	0
	History	198	40	8	60	0	10
	Not affected	234762	28345	21517	43459	86270	566
Alerts per 1000 LoC	**Patch**	7.977	3.263	0.906	3.263	0.181	0.181
	Vulnerable	16.373	4.912	2.456	6.549	0.000	0.000
	History	14.533	2.936	0.587	4.404	0.000	0.734
	Not affected	71.674	8.654	6.569	13.268	26.339	0.173

Figure 5.5: Alerts per output type, and Alerts per 1000 LoC

1) the *vulnerable* code has a larger number of alerts per 1000 LoC than the *patch*, as expected. This suggests that the *patch* code corrected many problems existent in the *vulnerable* code.

2) most of the types suggest that the number of alerts per 1000 LoC increased from the *history* versions to the *vulnerable* code, which suggests that the code tends to became worse across versions which are not patches.

3) the difference between the *not affected* and the rest of the code, still has a significant difference. However, in this analysis the difference to the *vulnerable* code is never greater than 5 times, unlike what happens in the analysis by tool.

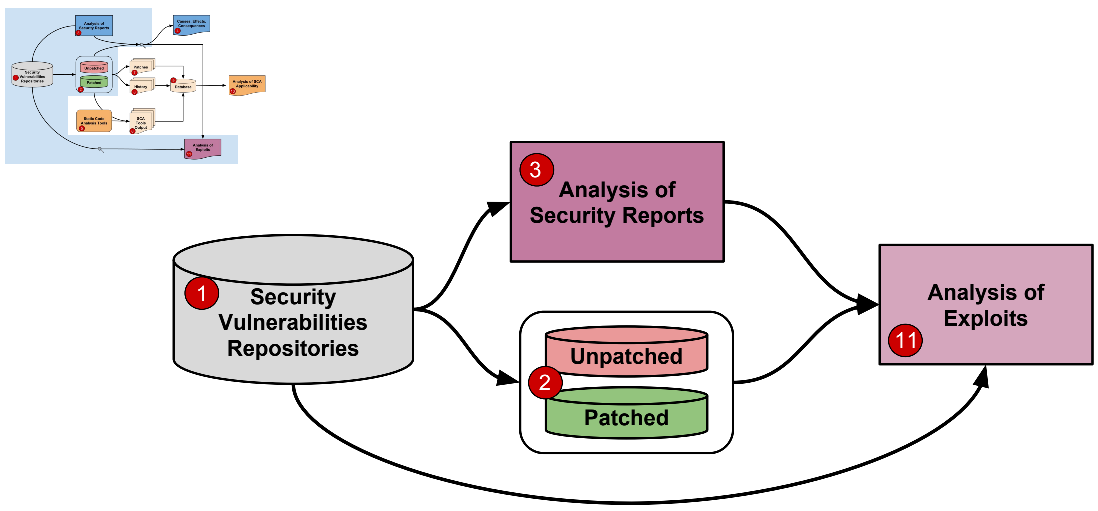
4) function calls do not have alerts in the *vulnerable* code compared to *not affected* code. Observing the numbers, this type of alert, along with *Presentation*, have the largest number of alerts. These types are the reason why the *not affected* code has such a large number of alerts.

5) spell errors gives the same number of alerts for the *history* code as the *spellcheck* tool. These alerts are almost the same, however when observing the *not affected* code the number is superior. This is due to the fact that some tools give spell errors as well.

Chapter 6

Analysis of Security Patches and Exploits

To better understand the observed results, a closer look was taken at most peculiar cases. For this, the code of the patches was analyzed. In some cases it was even necessary to develop or locate security exploits for the vulnerabilities. Figure 6.1 presents an overview of the followed methodology for the analysis, as described below.



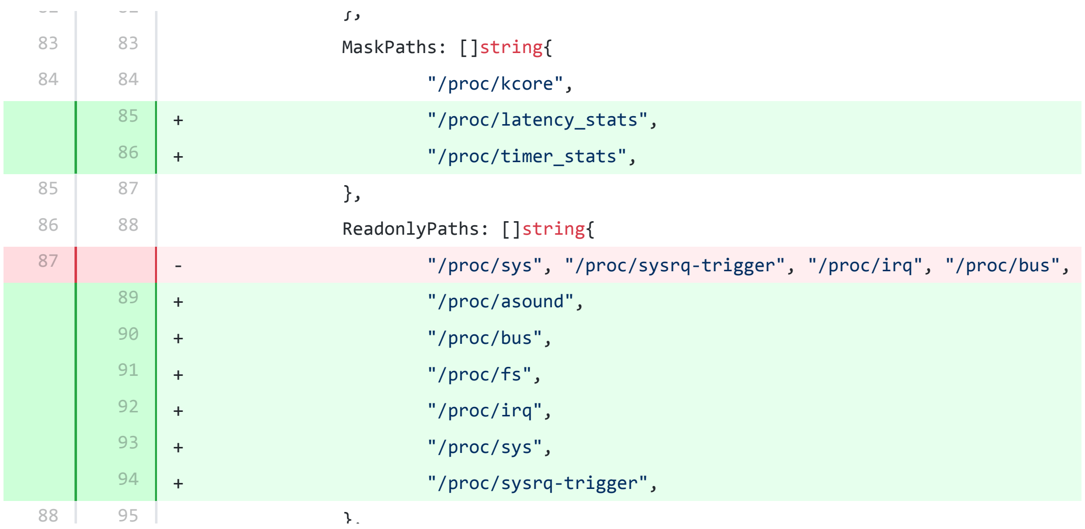
3
Analysis of Security Reports
1
Security Vulnerabilities Repositories
Unpatched
2
Patched
Analysis of Exploits
11

Figure 6.1: Approach followed for Patch analysis and Exploit development.

As can be observed, the approach begins with an initial collection of information from the **security vulnerabilities repositories (1)**. In this case it was possible to perform the analysis for 5 discussed below.

The **analysis of the exploits (11)** allows analyze the impact to the system in practice. These interpretation combined with the patch helps the interpretations of the results of the static analysis and the systematization of the vulnerabilities.

Figure 6.2 presents the patch for `CVE-2015-3630` in which the problem was that the files in the `/proc/` folder had write permissions, which allows users to bypass security restrictions. The patch updated those files to mask the paths and to only give read permissions to the files. An example of an attack is accessing a file from the `/proc` folder and reading or change sensitive information.



```
                },
                MaskPaths: []string{
                        "/proc/kcore",
+                       "/proc/latency_stats",
+                       "/proc/timer_stats",
                },
                ReadonlyPaths: []string{
-                       "/proc/sys", "/proc/sysrq-trigger", "/proc/irq", "/proc/bus",
+                       "/proc/asound",
+                       "/proc/bus",
+                       "/proc/fs",
+                       "/proc/irq",
+                       "/proc/sys",
+                       "/proc/sysrq-trigger",
                },
```

file: daemon/execdriver/native/template/default_template.go https://github.com/moby/moby/pull/13073/files

Figure 6.2: Code snippet from the patch for `CVE-2015-3630`.

Listing 6.1 is an example of an exploit, where it can be seen that it is very easy to read the content of the host file `/proc/timer_stats` from inside the `docker` container. Similarly, the second line shows that it is also very easy to modify the content of this file.

Listing 6.1: Exploit for `CVE-2015-3630`

```
docker exec -it xploit3630 cat /proc/timer_stats
docker exec -it xploit3630 sh -c "echo A >/proc/timer_stats"
```

This is the type of vulnerability that is ***very hard to validate with static analysis, testing techniques and also very hard to model****. Thus, this type of issue can only be effectively detected with systematic inspection of the system specification and implementation.*

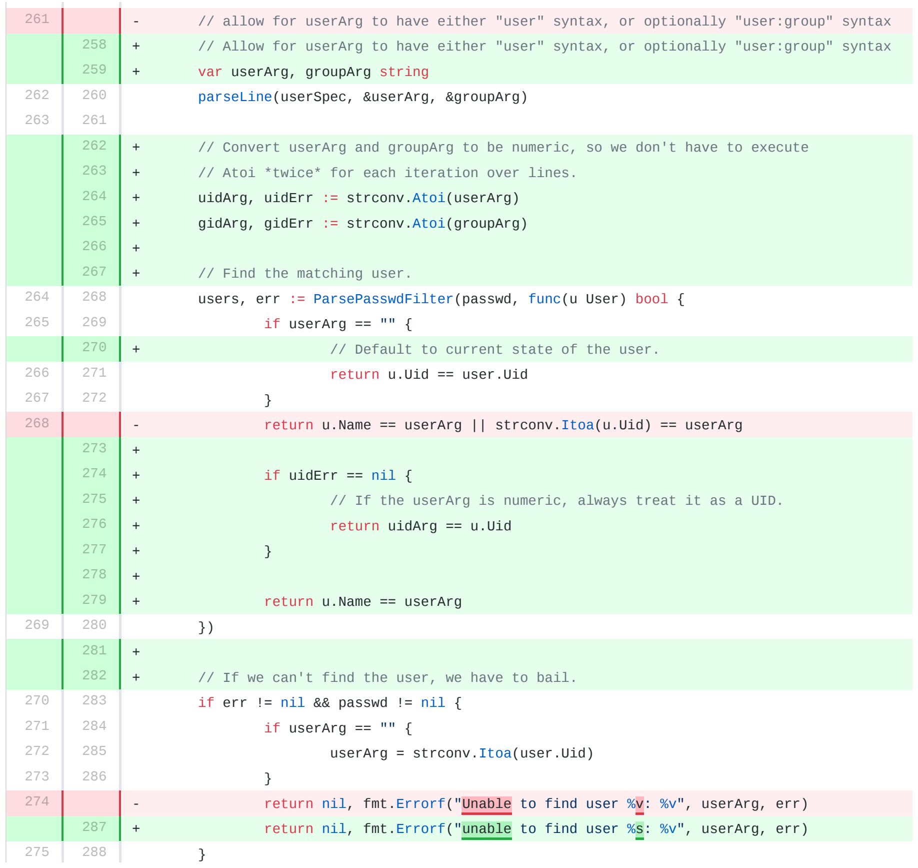
This vulnerability also shows that when dealing with containerization solutions, every piece of attack surface has substantial consequences in the host. This shows the particular importance of detailed verification and validation of these platforms, and the need to continuously understand how they can be improved.

Vulnerabilities `CVE-2017-16539` and `CVE-2015-3631` has a similar exploit and patch but related to different files.

Besides the vulnerability `CVE-2015-3630` allows a user obtain info and write in these files, `CVE-2017-16539` have different cause and effect because the affected files change the state of hardware, which does not have the same impact when affecting the host. `CVE-2015-3631` has different cause and effect because the files affected target the Linux security modules, so is expected more caution with these resources.

Analysis of CVE-2016-3697

`CVE-2016-3697`, presented in Figure 6.3, is an example of a vulnerability in which the validation of user ID (`UID`) was not treated properly, which allows attackers to gain privileges through malicious images. As can be observed in Figure 6.3, the correction was implemented by treating the `UID` as a numeric value.



```
-   // allow for userArg to have either "user" syntax, or optionally "user:group" syntax
+   // Allow for userArg to have either "user" syntax, or optionally "user:group" syntax
+   var userArg, groupArg string
    parseLine(userSpec, &userArg, &groupArg)

+   // Convert userArg and groupArg to be numeric, so we don't have to execute
+   // Atoi *twice* for each iteration over lines.
+   uidArg, uidErr := strconv.Atoi(userArg)
+   gidArg, gidErr := strconv.Atoi(groupArg)
+
+   // Find the matching user.
    users, err := ParsePasswdFilter(passwd, func(u User) bool {
            if userArg == "" {
+                   // Default to current state of the user.
                    return u.Uid == user.Uid
            }
-           return u.Name == userArg || strconv.Itoa(u.Uid) == userArg
+
+           if uidErr == nil {
+                   // If the userArg is numeric, always treat it as a UID.
+                   return uidArg == u.Uid
+           }
+
+           return u.Name == userArg
    })
+
+   // If we can't find the user, we have to bail.
    if err != nil && passwd != nil {
            if userArg == "" {
                    userArg = strconv.Itoa(user.Uid)
            }
-           return nil, fmt.Errorf("Unable to find user %v: %v", userArg, err)
+           return nil, fmt.Errorf("unable to find user %s: %v", userArg, err)
    }
```

file: vendor/src/github.com/opencontainers/runc/libcontainer/user/user.go `https://github.com/moby/moby/pull/22998/files`

Figure 6.3: Code snippet from the patch for `CVE-2016-3697`.

To produce the exploit for this Common Vulnerabilities and Exposure (CVE) it is necessary to run the container and inside of it introduce a specially crafted line into the `/etc/passwd` file, as shown in the first two lines of Listing 6.2. This allows one to create a user with root privileges. Afterwards, an attacker can access the container with that user ID to gain privileges (as shown on the third line).

Listing 6.2: Exploit for CVE-2016-3697

```
docker run -it --rm --name xploit2016_3697_1.11.1
echo '1234:x:0:0:root:/root:/bin/bash' >> /etc/passwd

docker exec -it --user 1234 xploit2016_3697_1.11.1 id
```

This bug is a ***classical example of a case in which robustness testing*** *would easily uncover that the* `UID` *was being incorrectly handled. From the point of view of the system this would be very hard to find, as the means to exploit are not through a simple interface. With systematic testing of these functions, using robustness testing to inject interface faults at operation level, it would be possible to uncover the corner cases.*

However, many times these type of protections and concerns are ***avoided for performance efficiency concerns****. In the case of Figure 6.3, is noticeable that the developers that developed the patch left a comment highlighting the need to minimize repetitive* `Atoi` *operations. This balance between efficiency and security is a very difficult trade-off to handle.*

Analysis of CVE-2014-9356

Figure 6.4 shows the patch of the vulnerability CVE-2014-9356. The description of this CVE is marked as "RESERVED", which means it has been reserved to be used by a security research or CVE Numbering Authority Organization, without details [61]. This CVE was published as a vulnerability that could overwrite arbitrary portions in the host file system and/or escape containerization, using absolute symbolic links through archive extraction or volume mounts.

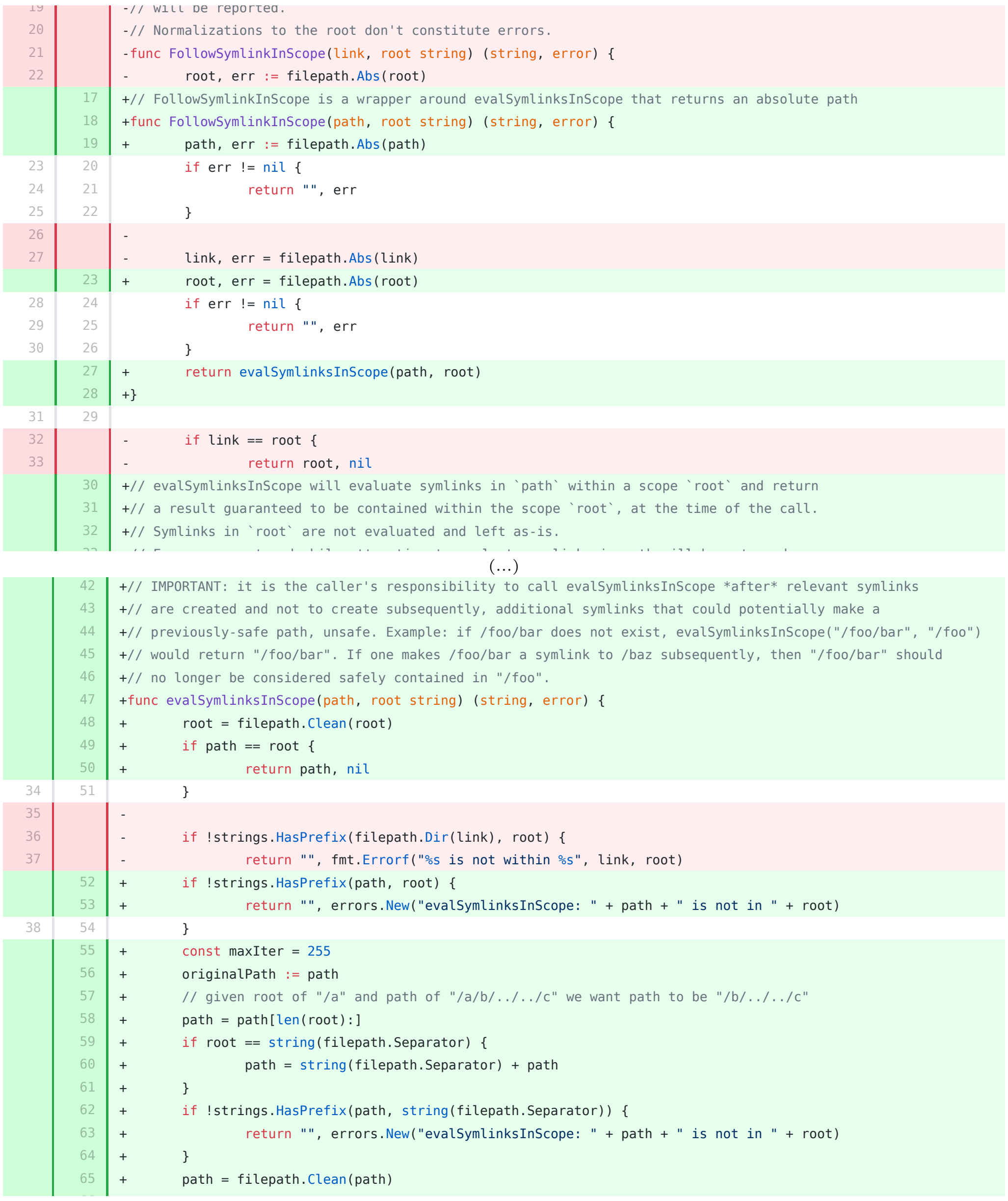
The patch of this vulnerability completely redefines the evaluation of the symbolic links before giving the result of the path to the directory, and prevents the creation in '/' path in order to prevent attacks to escape the root.

This can be exploited using an archive extraction, by creating a symbolic link with a long back path to escape the containerization and then compressing it as shown in Listing 6.3.

Listing 6.3: Exploit for CVE-2014-9356 - Create symlink

```
ln -s /../../../../../../../../../usr/bin/ symlink
tar -cvf symlink.tar symlink
```

On the creation of the container, it is necessary to extract the symlink in it, and then it is possible to add any file to the symlink folder. As an example, in Listing 6.4 the injected file is inserted into the symlink folder, which will overwrite a file with the same name present in */usr/bin*. To conclude the exploit, it is only necessary to build and run the container.



```
-// will be reported.
-// Normalizations to the root don't constitute errors.
-func FollowSymlinkInScope(link, root string) (string, error) {
-        root, err := filepath.Abs(root)
+// FollowSymlinkInScope is a wrapper around evalSymlinksInScope that returns an absolute path
+func FollowSymlinkInScope(path, root string) (string, error) {
+        path, err := filepath.Abs(path)
         if err != nil {
                 return "", err
         }
-
-        link, err = filepath.Abs(link)
+        root, err = filepath.Abs(root)
         if err != nil {
                 return "", err
         }
+        return evalSymlinksInScope(path, root)
+}

-        if link == root {
-                return root, nil
+// evalSymlinksInScope will evaluate symlinks in `path` within a scope `root` and return
+// a result guaranteed to be contained within the scope `root`, at the time of the call.
+// Symlinks in `root` are not evaluated and left as-is.
```

(...)

```
+// IMPORTANT: it is the caller's responsibility to call evalSymlinksInScope *after* relevant symlinks
+// are created and not to create subsequently, additional symlinks that could potentially make a
+// previously-safe path, unsafe. Example: if /foo/bar does not exist, evalSymlinksInScope("/foo/bar", "/foo")
+// would return "/foo/bar". If one makes /foo/bar a symlink to /baz subsequently, then "/foo/bar" should
+// no longer be considered safely contained in "/foo".
+func evalSymlinksInScope(path, root string) (string, error) {
+        root = filepath.Clean(root)
+        if path == root {
+                return path, nil
         }
-
-        if !strings.HasPrefix(filepath.Dir(link), root) {
-                return "", fmt.Errorf("%s is not within %s", link, root)
+        if !strings.HasPrefix(path, root) {
+                return "", errors.New("evalSymlinksInScope: " + path + " is not in " + root)
         }
+        const maxIter = 255
+        originalPath := path
+        // given root of "/a" and path of "/a/b/../../c" we want path to be "/b/../../c"
+        path = path[len(root):]
+        if root == string(filepath.Separator) {
+                path = string(filepath.Separator) + path
+        }
+        if !strings.HasPrefix(path, string(filepath.Separator)) {
+                return "", errors.New("evalSymlinksInScope: " + path + " is not in " + root)
+        }
+        path = filepath.Clean(path)
```

file: pkg/symlink/fs.go
https://github.com/moby/moby/pull/9617/files

Figure 6.4: Code snippet from the patch for CVE-2014-9356.

Listing 6.4: Exploit for `CVE-2014-9356` - Dockerfile

```
FROM busybox
ADD symlink.tar /
ADD inject /symlink/
```

The particularity that lead to this vulnerability was totally unforeseen by the development team, as the patch required the team to write a completely new way of handling the symbolic links.

Identifying this issue would require the use of systematic ***testing with invalid or maliciously crafted inputs****, i.e. with robustness or penetration testing. However, the interface is so complex, that this approach would have to be applied at function level to be effective.*

In fact, automated path traversal assessment techniques are used in other domains, and are able to generate the type of maliciously crafted inputs that would trigger this vulnerability, if applied at function level.

Analysis of CVE-2014-9358

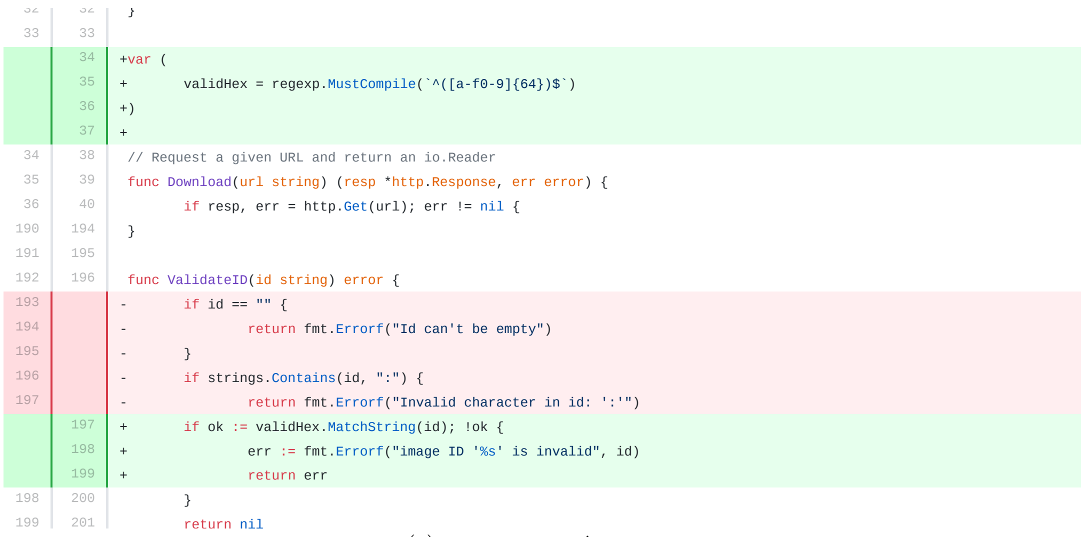
This vulnerability is due to the lack of validation of image `IDs` which allows users to get malicious images using `docker load` or from `Docker Registry`. Attackers can do path traversal and repository spoofing attacks. Figure 6.5 presents the three patched files, focusing on the most relevant segments in order to understand the vulnerability.

In Figure 6.5.(a) the function responsible for validating the ID of an image was updated because its previous implementation not considering the entire ID, only focusing on the invalid characters in it. This function was added to the file Figure 6.5(b) to validate the repository name of the image and to the one in Figure 6.5(c) to load the image to the repository.

This shows that there were two main problems behind the vulnerability: *i)* the validation of the `IDs` was implemented incorrectly, rejecting only the cases that had an invalid char, and *ii)* this validation was not used in two key points, in which it was assumed that the *ID* provided was trustworthy.

Similarly, this case requires the use of systematic testing with invalid or maliciously crafted inputs, such as systematic and automated path traversal assessment techniques. However, the issues behind this vulnerability are due to ***secure development principles****.*

The incorrect validation issue is due to the fact that the developers ***do not employ secure defaults****, as the validation strategy used follows a black-listing approach. The unforeseen invalid* `IDs` *default to accept, as they are not on the black-list. During the correction, the team opted for a more recommended white-listing approach, in which the accepted structure of the* `IDs` *is carefully expressed in a regular expression, and*

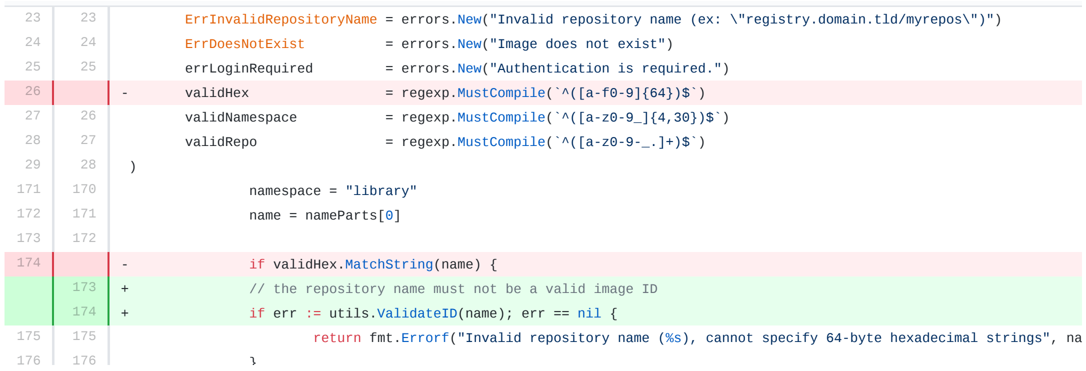


```
 }

+var (
+        validHex = regexp.MustCompile(`^([a-f0-9]{64})$`)
+)
+
 // Request a given URL and return an io.Reader
 func Download(url string) (resp *http.Response, err error) {
        if resp, err = http.Get(url); err != nil {
 }

 func ValidateID(id string) error {
-       if id == "" {
-               return fmt.Errorf("Id can't be empty")
-       }
-       if strings.Contains(id, ":") {
-               return fmt.Errorf("Invalid character in id: ':'")
+       if ok := validHex.MatchString(id); !ok {
+               err := fmt.Errorf("image ID '%s' is invalid", id)
+               return err
        }
        return nil
```

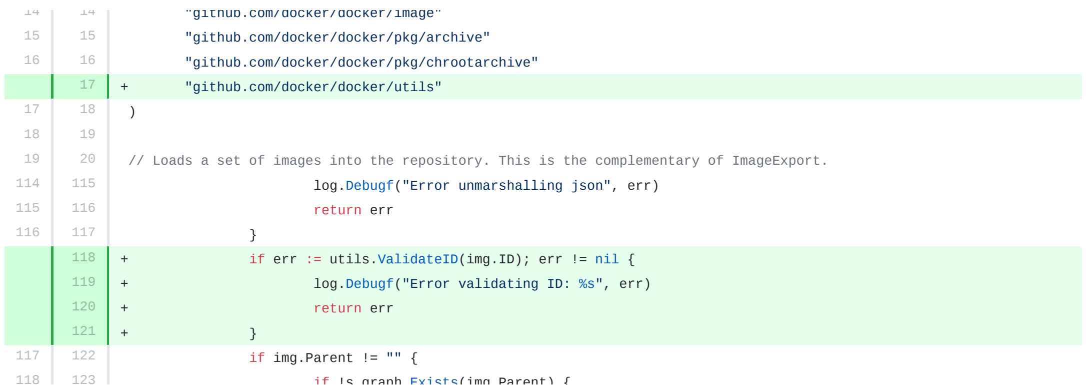
(a) file: `utils/utils.go`



```
        ErrInvalidRepositoryName = errors.New("Invalid repository name (ex: \"registry.domain.tld/myrepos\")")
        ErrDoesNotExist          = errors.New("Image does not exist")
        errLoginRequired         = errors.New("Authentication is required.")
-       validHex                 = regexp.MustCompile(`^([a-f0-9]{64})$`)
        validNamespace           = regexp.MustCompile(`^([a-z0-9_]{4,30})$`)
        validRepo                = regexp.MustCompile(`^([a-z0-9-_.]+)$`)
 )
                namespace = "library"
                name = nameParts[0]

-               if validHex.MatchString(name) {
+               // the repository name must not be a valid image ID
+               if err := utils.ValidateID(name); err == nil {
                        return fmt.Errorf("Invalid repository name (%s), cannot specify 64-byte hexadecimal strings", na
                }
```

(b) file: `registry/registry.go`



```
        "github.com/docker/docker/image"
        "github.com/docker/docker/pkg/archive"
        "github.com/docker/docker/pkg/chrootarchive"
+       "github.com/docker/docker/utils"
 )

 // Loads a set of images into the repository. This is the complementary of ImageExport.
                        log.Debugf("Error unmarshalling json", err)
                        return err
                }
+               if err := utils.ValidateID(img.ID); err != nil {
+                       log.Debugf("Error validating ID: %s", err)
+                       return err
+               }
                if img.Parent != "" {
                        if !s.graph.Exists(img.Parent) {
```

(c) file: `graph/load.go`
`https://github.com/moby/moby/pull/9620/files`

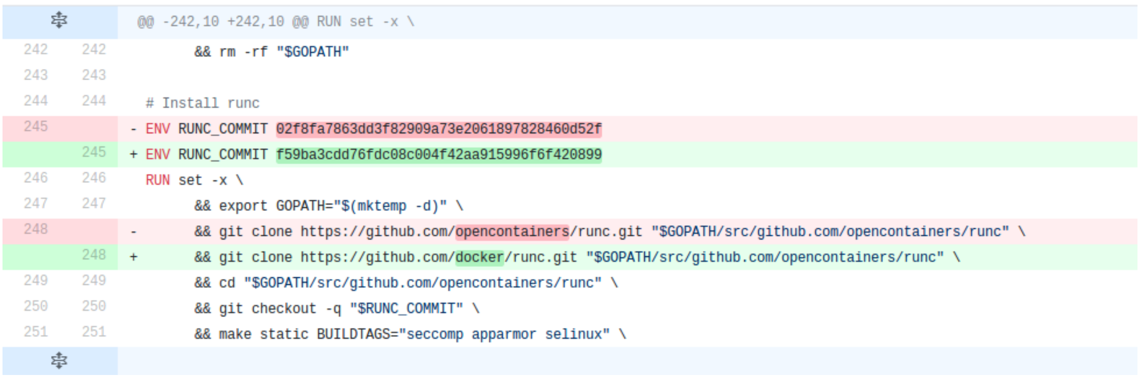
Figure 6.5: Code snippets from the patch for `CVE-2014-9358`.

> *only the ones that fit this structure are accepted. Unforeseen `ID` structures in this case default to reject, which is the safe behaviour.*
>
> *The missing usage of the validation is due to incorrectly trusting an input form another (sub-)system. It is necessary to always* ***consider all input as malicious until proven otherwise****, as should be assumed that external systems, sub-systems or even components are insecure.*

The characterization of the vulnerability `CVE-2014-5282` in the Table 4.3 is the same of `CVE-2014-9358` because the source problem was the same. The emergence of these vulnerabilities was because of the same cause, however the patch for `CVE-2014-5282` did not validate the ID properly and led to the `CVE-2014-9358`. 77some of the patched code is the same and was modified between these versions.

Analysis of CVE-2016-8867

The misconfiguration of the capability polices enable the incorrect application of ambient capabilities in Docker Engine. This vulnerability was brought to Docker by a *runC* commit and affects kernel 4.4. As shown in Figure 6.6, the patch for this vulnerability was a revert commit in order to preserve the capabilities as they were in the system.



```
@@ -242,10 +242,10 @@ RUN set -x \
         && rm -rf "$GOPATH"

 # Install runc
- ENV RUNC_COMMIT 02f8fa7863dd3f82909a73e2061897828460d52f
+ ENV RUNC_COMMIT f59ba3cdd76fdc08c004f42aa915996f6f420899
 RUN set -x \
         && export GOPATH="$(mktemp -d)" \
-        && git clone https://github.com/opencontainers/runc.git "$GOPATH/src/github.com/opencontainers/runc" \
+        && git clone https://github.com/docker/runc.git "$GOPATH/src/github.com/opencontainers/runc" \
         && cd "$GOPATH/src/github.com/opencontainers/runc" \
         && git checkout -q "$RUNC_COMMIT" \
         && make static BUILDTAGS="seccomp apparmor selinux" \
```

file: Dockerfile `https://github.com/moby/moby/pull/27610/files`

Figure 6.6: Code snippet from the patch for `CVE-2016-8867`.

Exploit this vulnerability allows attack through malicious images to bypass user permissions, and also allows an attacker to gain privileges to the system. An example of the exploit to gain privileges is composed by two steps, where one is the creation of Dockerfile as shown in Listing 6.5 and the steps to gain privileges to the user shown in Listing 6.6.

Listing 6.5: Exploit for `CVE-2016-8867` - Dockerfile

```
FROM debian
RUN useradd example
RUN id
USER example
RUN id
RUN cat /etc/shadow
CMD /bin/bash
```

Listing 6.6: Exploit for `CVE-2016-8867`

```
docker build --no-cache -t example .
docker run -u example -it example

chmod +s /bin/sh
/bin/sh
```

Dockerfile creates a container and add a new user inside of it, print the users system file, and execute the bash shell for the container. After the Dockerfile, build, and run the container is possible to run commands directly inside of it. Executing the commands *chmod +s* to user to the execution followed by the execution of file */bin/sh*, the created user in the Dockerfile automatically turn into a root user.

> *This vulnerability was introduced to the system by an external component, which was totally unexpected for the developer team, since the other system works.*
>
> *To identify this vulnerability would be required* ***integration tests before bump the components to the system****. However, Docker contains multiple components integrated in it, turning it in a complex process to perform.*
>
> *Static analysis could not detect this vulnerability, since the integration of components are the insertion of the commit ID in a Dockerfile.*

This page is intentionally left blank.

Chapter 7

Discussion

The analysis performed during this work began with the systematization of the vulnerabilities. This provides the knowledge of the most frequent causes, effects, and consequences in the already known vulnerabilities.

After the study of each vulnerability, an analysis of the patch, vulnerable, history, and not affected code was performed with Static Code Analyzers (SCA) tools, to detect problems that could be associated with the analysis. The alerts given by each tool were analyzed and grouped by type.

Finally, a thorough analysis of some exploits available in *Git* issues was performed. This study crossed an exploit with its corresponding patch, which provided insights about the incapability of the SCA tools in detecting the vulnerabilities. It also provided an analysis of other techniques that could have detected those vulnerabilities.

Considering the activities performed in this study, and the respective results, it is possible to systematize the following lessons:

1) **Key causes/consequences of Docker vulnerabilities**

   Results show that the main causes of the vulnerabilities are unprotected resources and incorrect permission management. These are the most common security issues from the developers. The most likely effect of these causes are exposed system and restriction violation. These effects are important, as their consequences are mostly **bypass** and **gain privileges**, which can let attackers access other containers or the host.

2) **Existing SCA tools are very ineffective**

   Most SCA tools for Go code are used for checking the language style formatting, and others give more or less the same information as the compiler when its necessary to build the code. Although good practices are important for software quality and maintainability, which contribute to security, in this study, the focus is on their ability to identify vulnerabilities.

3) Existing SCA tools do not help in analyzed issues

As can observed in the examples given in Chapter 6, most issues were not about the code itself, but rather unprotected resources or incorrect permissions. These kind of issues are hard to find with SCA tools.

4) Similar exploitation

In some of the analyzed exploits it is possible to notice some patterns. This means that if attackers can recognize those patterns and study the code, it could lead to more vulnerabilities being found. This kind of insight can compromise the security of the platform.

5) Testing techniques could have avoided some of the issues

Several of the analyzed vulnerabilities could have been avoided with the systematic application of techniques such as robustness and/or penetration testing. However, in the observed cases, this analysis should have been done at function level, which makes it a much harder activity.

6) Many issues have roots in efficiency/security trade-offs

As it is common in computing systems, many of the observed issues have their issues in the trade-off between security and efficiency. Sensitive operations are often performed without the complete validation of their inputs, and often the motivation is to avoid the performance penalty. In solutions with the characteristics that Docker possesses, this is yet one of the main challenges: *achieve performance and efficient resource management, while at the same time providing security guarantees.*

Chapter 8

Conclusions and Future Work

Docker is one of the most used platforms for application containers, providing abstraction and automation. Docker promises agility, portability, security, and cost saving. In this work a security analysis to the Docker platform was performed. Initially, a diversified array of techniques, including analysis of security reports, static code analysis tools, and analysis of exploits was applied.

Security analysis of containers platforms shows that most of the reported vulnerabilities are concentrated in the beginning of Docker's development still, many vulnerabilities were patched only after an extended period of time. In particular, **some vulnerabilities went undetected for more than one year since the first release**, increased the probability that some services were exploited and compromised where Docker was running. A systematization allows one to observe that **bypass and gain privileges are the most frequent flaws** due to Docker containers sharing the kernel space with the host Operating System (OS). Exploiting these vulnerabilities leads to an escape from the containerized environment to control the system components.

The Static Code Analyzers (SCA) tools proved to be important in maintaining the quality of the code, and identifies multiple issues with it. Despite the **output of these tools not having a direct association to the vulnerabilities**, good practices may avoid other vulnerabilities in the future.

Additionally, the analysis of the exploits suggests that some vulnerabilities could be avoided with the use of traditional testing techniques in some critical functions. Techniques such as robustness and penetration testing are some examples.

Future work includes focusing in the development of a static analysis tool that could be applied to Go programming language. This work would be more interesting with more vulnerabilities, in order to draw more concrete conclusions. Additionally, it is also advisable to apply the identified techniques to the Docker codebase (robustness and penetration testing), to validate if their use could indeed have avoided some of the vulnerabilities.

This page is intentionally left blank.

Bibliography

[1] Gabor Nagy. Operating system containers vs. application containers. `https://blog.risingstack.com/operating-system-containers-vs-application-containers/`. Accessed: 2017-11-14.

[2] Vps. `https://f1.holisticinfosecforwebdevelopers.com/chap03.html`. Accessed: 2018-01-20.

[3] Nuno Neves, Joao Antunes, Miguel Correia, Paulo Verissimo, and Rui Neves. Using attack injection to discover new vulnerabilities. In *International Conference on Dependable Systems and Networks, 2006. DSN 2006.*, pages 457–466. IEEE, 2006.

[4] Docker rocker: container technology usage doubles; serious money follows. `https://goo.gl/KrzHgb`. Accessed: 2018-01-20.

[5] Datadog. 8 surprising facts about real docker adoption. `https://www.datadoghq.com/docker-adoption-2015/`. Accessed: 2017-12-28.

[6] Docker. `https://www.docker.com/`. Accessed: 2017-08-31.

[7] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[8] DZone. What is docker and why is it so darn popular? `http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/`. Accessed : 2018-01-20.

[9] DZone. Docker...containers, microservices and orchestrating the whole symphony. `https://dzone.com/articles/dockercontainers-microservices`. Accessed: 2018-01-20.

[10] Kubernetes. `https://kubernetes.io/`. Accessed: 2018-01-20.

[11] Mesos. `http://mesos.apache.org/`. Accessed: 2018-01-20.

[12] Docker swarm overview. `https://docs.docker.com/engine/swarm/`. Accessed: 2018-01-20.

[13] Datadog. 8 surprising facts about real docker adoption. `https://www.datadoghq.com/docker-adoption/`. Accessed: 2017-12-28.

[14] What is docker. `https://www.docker.com/what-docker/`. Accessed: 2017-12-28.

[15] Ross J Anderson. Security engineering: A guide to building dependable distributed systems. chapter 1. Wiley Publishing, 2008.

[16] Oracle. Brief history of virtualization. https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html. Accessed: 2017-11-30.

[17] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30, 2013.

[18] Free *BSD*. https://www.freebsd.org/. Accessed: 2017-12-28.

[19] Linux VServer. http://linux-vserver.org/. Accessed: 2017-12-28.

[20] Oracle solaris zones. https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm#OPCUG426. Accessed: 2017-12-28.

[21] Open*VZ*. https://openvz.org/Main_Page. Accessed: 2017-12-28.

[22] Namespaces in operation, part 1: namespaces overview. https://lwn.net/Articles/531114/. Accessed: 2018-01-04.

[23] What's LXC? https://linuxcontainers.org/lxc/introduction/. Accessed: 2017-12-28.

[24] What's LXD? https://linuxcontainers.org/lxd/. Accessed: 2017-12-28.

[25] Emiliano Casalicchio. Autonomic orchestration of containers: Problem definition and research challenges. In *10th EAI International Conference on Performance Evaluation Methodologies and Tools. EAI*, 2016.

[26] docker. Docker v18.01.0-ce. https://github.com/docker/docker-ce/releases/tag/v18.01.0-ce. Accessed: 2018-08-12.

[27] What's behind Linux's new Cloud Native Computing Foundation? https://www.networkworld.com/article/2950489/cloud-computing/what-s-behind-linux-s-new-cloud-native-computing-foundation.html. Accessed: 2018-01-04.

[28] Coreos rkt. https://coreos.com/rkt/. Accessed: 2017-12-28.

[29] Julien Barbier. It's here: Docker 1.0. https://blog.docker.com/2014/06/its-here-docker-1-0/. Accessed: 2017-11-14.

[30] Docker. Github docker. https://github.com/docker/docker/. Accessed: 2017-11-21.

[31] Introducing moby project: a new open-source project to advance the software containerization movement. https://blog.docker.com/2017/04/introducing-the-moby-project/. Accessed: 2018-01-20.

[32] Mina Andrawos and Martin Helmich. *Cloud Native Programming with Golang: Develop microservice-based high performance web apps for the cloud with Go.* Packt Publishing Ltd, 2017.

[33] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[34] Docker hub. `https://hub.docker.com/`. Accessed: 2018-01-08.

[35] Docker engine. `https://docs.docker.com/engine/docker-overview/`. Accessed: 2018-01-08.

[36] Docker products. `https://https://www.docker.com/products/`. Accessed: 2018-08-08.

[37] Docker product and tool manuals. `https://docs.docker.com/manuals/`. Accessed: 2018-01-08.

[38] Docker swarm overview. `https://docs.docker.com/swarm/overview/`. Accessed: 2018-01-08.

[39] Boot2Docker. boot2docker. `http://boot2docker.io/`. Accessed: 2018-05-12.

[40] Tech Target. Boot2Docker. `https://searchitoperations.techtarget.com/definition/Boot2Docker`. Accessed: 2018-08-12.

[41] Docker 0.9: introducing execution drivers and libcontainer. `https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/`. Accessed: 2018-01-03.

[42] Docker security. `https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface`. Accessed: 2018-01-08.

[43] man pages. Capabilities(7). `http://man7.org/linux/man-pages/man7/capabilities.7.html`. Accessed: 2018-12-21.

[44] Miguel Pupo Correia and Paulo Jorge Sousa. Segurança no software. *Lisboa: FCA*, 2010.

[45] Exploit Database. About the exploit database. `https://www.exploit-db.com/about-exploit-db/`. Accessed: 2018-01-10.

[46] 2011 cwe/sans top 25 most dangerous software errors. `https://cwe.mitre.org/top25/index.html`. Accessed: 2018-08-20.

[47] Top 10 OWASP critical security risks. `https://www.owasp.org/index.php/Top_10-2017_Top_10`. Accessed: 2018-08-20.

[48] Panagiotis Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006.

[49] Theo Combe, Antony Martin, and Roberto Di Pietro. To docker or not to docker: A security perspective. volume 3, pages 54–62. IEEE, 2016.

[50] A. A. Mohallel, J. M. Bass, and A. Dehghantaha. Experimenting with docker: Linux container and base os attack surfaces. In *2016 International Conference on Information Society (i-Society)*, pages 17–21, Oct 2016.

[51] A. Martin, S. Raponi, T. Combe, and R. Di Pietro. Docker ecosystem – vulnerability analysis. *Computer Communications*, 122:30 – 43, 2018.

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

[53] Zhiqiang Jian and Long Chen. A defense method against docker escape attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, pages 142–146. ACM, 2017.

[54] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. Containerleaks: Emerging security threats of information leakages in container clouds. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pages 237–248. IEEE, 2017.

[55] Moshe Zviran and William J Haga. Password security: an empirical study. *Journal of Management Information Systems*, 15(4):161–185, 1999.

[56] Ram Chillarege, Inderpal S Bhandari, Jarir K Chaar, Michael J Halliday, Diane S Moebus, Bonnie K Ray, and M-Y Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on software Engineering*, (11):943–956, 1992.

[57] Gang Tan and Jason Croft. An empirical security study of the native code in the jdk. In *Usenix Security Symposium*, pages 365–378, 2008.

[58] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments, 2007.

[59] Aleksandar Milenkoski, Bryan D Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. Experience report: an analysis of hypercall handler vulnerabilities. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 100–111. IEEE, 2014.

[60] Ivano Alessandro Elia, Nuno Antunes, Nuno Laranjeiro, and Marco Vieira. An analysis of openstack vulnerabilities. In *2017 13th European Dependable Computing Conference (EDCC)*, pages 129–134. IEEE, 2017.

[61] CVE. Frequent asked questions. `https://cve.mitre.org/about/faqs.html#why_CVE_entry_marked_RESERVED_when_being_publicly_used`. Accessed: 2017-11-17.

[62] alecthomas. Go meta linter. `https://github.com/alecthomas/gometalinter`. Accessed: 2018-06-01.

[63] 360EntSecGroup-Skylar. Go reporter. `https://github.com/360EntSecGroup-Skylar/goreporter`. Accessed: 2018-06-01.

[64] Go AST Scanner.